UNIVERSIDADE DO SUL DE SANTA CATARINA

LUCAS MARCONDES DE MATTOS LINSMEYER

LUTIELO FERNANDES

UMA PROPOSTA BASEADA NA ARQUITETURA DE MICROSERVIÇOS PARA REAPROVEITAMENTO DE MÓDULOS DE SOLUÇÕES MONOLÍTICAS.

Florianópolis

2016

LUCAS MARCONDES DE MATTOS LINSMEYER
LUTIELO FERNANDES

UMA PROPOSTA BASEADA NA ARQUITETURA DE MICROSERVIÇOS PARA REAPROVEITAMENTO DE MÓDULOS DE SOLUÇÕES MONOLÍTICAS.

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação da Universidade do Sul de Santa Catarina, como requisito parcial à obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Flávio Ceci, Dr.

Florianópolis

2016

LUCAS MARCONDES DE MATTOS LINSMEYER
LUTIELO FERNANDES

UMA PROPOSTA BASEADA NA ARQUITETURA DE MICROSERVIÇOS PARA REAPROVEITAMENTO DE MÓDULOS DE SOLUÇÕES MONOLÍTICAS.

Este Trabalho de Conclusão de Curso foi julgado adequado à obtenção do título de Bacharel em Sistemas de Informação e aprovado em sua forma final pelo Curso de Graduação em Sistemas de Informação da Universidade do Sul de Santa Catarina.

Florianópolis, 21 de junho de 2016.

_____

Professor e orientador Flávio Ceci, Dr.
Universidade do Sul de Santa Catarina

_____

Prof. Luiz Otávio Botelho Lento, MSc.
Universidade do Sul de Santa Catarina

_____

Vinicius Fraccaroli Kopcheski, BSc.
Arquiteto de Software – Softplan

AGRADECIMENTOS

Agradecemos primeiramente a Deus por estar conosco dando força e sabedoria nessa caminhada.

Às nossas famílias pelo incentivo para não desistirmos das dificuldades encontradas no decorrer da faculdade.

A empresa Softplan Planejamento e Sistemas pela oportunidade de estudo, pelo espaço disponibilizado e pelo tutor cedido para realização deste trabalho.

Aos nossos amigos por todo apoio.

Ao Vinicius Fraccaroli Kopcheski, por seus ensinamentos, paciência e confiança ao longo do desenvolvimento deste trabalho. É um prazer tê-lo na banca examinadora.

Ao Dr. Luiz Otávio Botelho Lento, com quem partilharmos o que era o broto daquilo que veio a ser esse trabalho. Nossas conversas durante o desenvolvimento foram de grande valor para este projeto. Desejamos a sua participação na banca examinadora deste trabalho desde o princípio.

Ao Marcelo Teixeira Maluf e Diego Marafon, por todo apoio dado durante o desenvolvimento deste trabalho.

Agradecemos a todos os professores do curso por disseminarem seus conhecimentos conosco. Em especial ao orientador do trabalho de conclusão de curso, Dr. Flávio Ceci, que nos auxiliou com suas experiências e conhecimento em todos os momentos em que precisamos.

RESUMO

A tecnologia é uma das ciências que tem apresentado maior evolução nos últimos tempos. Gadgets, ferramentas, linguagens de programação e uma infinidade de áreas ligadas a tecnologia, estão em constante evolução. A arquitetura dos softwares não foge a essa regra e com o passar do tempo, conceitos que eram empregados em praticamente todos os sistemas desenvolvidos, utilizando o que é chamada de arquitetura monolítica, começaram a ficar obsoletos e devido à algumas características como alto acoplamento, grande base de código e não separação do contexto de negócio, se tornaram obstáculos para a manutenção e evolução desses sistemas.

Novas abordagens de arquitetura de software começaram a surgir e o termo microserviços começou a ganhar muita força no mercado de desenvolvimento de software. Essa recente abordagem tem características diferentes da arquitetura monolítica e ao invés de tratar o sistema como uma único bloco, trata de dividir o software em pequenos blocos que tratam de apenas um domínio de negócio.

Para sistemas com mais robustez essa abordagem tende a ser efetiva. Porém, é possível migrar uma solução monolítica para uma arquitetura de microserviços, respeitando o objetivo original da aplicação?

Esse trabalho trata de apresentar uma proposta de arquitetura para sistemas monolíticos, que tomaram proporções onde a aplicação da arquitetura de microserviço se faz conceitualmente mais efetiva que a arquitetura monolítica.

Palavras-chave: Microserviço. Arquitetura de Software. Monolito.

ABSTRACT

Technology is one of the many sciences that have evolved greatly since recent years. Gadgets, tools, programming languages and countless other areas technology related are in constant evolution. Software architecture is no different, as technology evolves, so do the concepts related to it. Concepts that were once the foundation of most software are now considered obsolete given it's coding approach; structured programming, low code coupling, massive blocks of code and a lack of subject segregation become obstacles when it comes to maintain and evolve the software.

New concepts and approaches has been used and the term microservices is currently a well accepted in the segment. This approach has a different way of handling the code, instead of dealing with the software as a single-block it segregates and splits the logic in much smaller segments that deal with a very specific matter.

To long lasting, robust software this approach tends to be more effective. Although, is it possible to update such an aged architecture to the microservices take keeping the purpose of the software intact?

This document presents a possible approach to deal with the maintenance of software that have outgrown its expectations and uses a conceptually ineffective architecture in which the microservices is more effective than the current.

Keywords: Microservice. Software Architecture. Monolith

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Cada vez é mais perceptível a evolução da tecnologia, fazendo com que conceitos se tornem defasados de forma muito mais rápida, devido ao surgimento de novos princípios que atendem melhor as necessidades atuais. No mercado de desenvolvimento de *software*, não poderia ser diferente. Segundo Hagler (2015, tradução nossa), "nos últimos anos, as arquiteturas *web* têm evoluído de maneira muito dinâmica e o resultado disso é que agora temos diversas abordagens para escolher na hora de construir uma nova arquitetura de *software*."

Atualmente, a arquitetura mais utilizada – devido ao código legado – é a monolítica. Segundo Fowler e Lewis (2014, tradução nossa), "Esse padrão funciona razoavelmente bem para pequenas aplicações, pois o desenvolvimento, testes e implantação de pequenas aplicações monolíticas são relativamente simples". Porém esta arquitetura pode se tornar um problema em aplicações robustas, devido ao fato de dificultar a utilização do conceito de entrega contínua e apresentar problemas como o "*Single point of failure*", que representa a indisponibilidade de todo um sistema, devido à inconsistência em apenas um dos módulos desse sistema – este é o problema mais recorrente nas aplicações monolíticas. Apesar dessa e outras desvantagens, desenvolver dessa forma não deve cair em total desuso, porém existem outras abordagens mais atuais que apresentam melhores recursos para o desenvolvimento de *software*, como a arquitetura baseada em microserviços.

"O termo Micro*services* ou Microserviços surgiu nos últimos anos para descrever uma maneira particular de conceber aplicações de *software* como uma suíte de serviços independentemente implementáveis". (FOWLER; LEWIS, 2014, tradução nossa).

Ao utilizar microserviços para desenvolver um sistema, há a garantia de que quando um dos módulos apresentar um problema crítico, os outros não serão diretamente afetados, além de trazer uma base de código menor, o que torna a manutenção mais simples. Em compensação, o processo de *deploy*[1] se torna mais complexo, pois há a necessidade de "subir" cada sistema de forma unitária. Também, verifica-se uma maior dificuldade na reutilização do código, já que os sistemas são tratados de forma individualizada. Esse último fator traz algumas duplicidades no código fonte de uma aplicação como um todo.

[1] Instalação de uma aplicação, ou seja, disponibilizar ela para seus usuários.

Tendo em vista a evolução tecnológica nas arquiteturas de desenvolvimento de *software* e as dificuldades enfrentadas com as arquiteturas legadas, este trabalho propõe apresentar uma proposta baseada na arquitetura de microserviços para reaproveitamento de módulos de soluções monolíticas e mostrar os seus benefícios. Além dos benefícios desse reaproveitamento, também são apresentadas as dificuldades de aplicar esta arquitetura e uma visão geral da diferença de comportamento das duas arquiteturas.

1.1 PROBLEMÁTICA

A arquitetura monolítica apresenta diversas facilidades, como desenvolver, testar, efetuar o *deploy* e escalar aplicações de pequeno porte. Porém, quando as aplicações se tornam de grande porte, aumentando o número de *commiters*[2], esta abordagem manifesta um número de problemas cada vez maior. Na empresa em que os presentes autores trabalham, a maioria das aplicações desenvolvidas estão baseadas nessa arquitetura e, como se tornaram aplicações de grande porte, sofremos com problemas que já são conhecidos dessa abordagem e diversos autores, como Fowler (2015B), Parmar (2014), Pember (2013) e Richardson (2014B), afirmam que:

- Aplicações monolíticas com o código fonte extenso intimida os desenvolvedores, principalmente aqueles que são novos no time. O entendimento e as alterações na aplicação podem se tornar tarefas difíceis. O resultado disso é a perda de produtividade no desenvolvimento do código. Também, como não há limites bem definidos entre os módulos, a modularização quebra constantemente. Além disso, pode se tornar difícil a compreensão de como efetuar uma manutenção corretamente, a qualidade do código tende a piorar.
- Sobrecarga da IDE[3] (*Integrated Development Environment*) – Uma aplicação de grande porte, pelo fato de possuir um código extenso, pode

[2] Indivíduos capazes de modificar o código-fonte de um *software*.

[3] Ambiente integrado de desenvolvimento, podem ser utilizados com diversos propósitos, desde a codificação até a concepção de modelos.

tornar a IDE lenta, fazendo com que a produtividade dos desenvolvedores diminua.

- Sobrecarga da *web container*[4] – Quanto maior for a aplicação, mais tempo ela levará para iniciar. Essa demora causa um enorme impacto na produtividade do desenvolvedor, pois o mesmo fica ocioso, esperando o *container* iniciar. Isso também impacta no *deployment* da aplicação.
- *Deploy* frequente é difícil – A realização de *deploys* frequentes em uma aplicação de grande porte se torna um obstáculo. Ao atualizar apenas um componente, é necessário realizar o *redeploy*[5] de toda a aplicação. Isso interrompe atividades em segundo plano, independente se essas atividades são impactadas pela mudança ou não. Há o risco de ao fazer o *redeploy* os componentes que não foram atualizados falharem durante o *redeployment*. Portanto, o risco aumenta a cada *redeploy*, desencorajando atualizações frequentes. Esse é um problema que os desenvolvedores de *interface* do usuário costumam enfrentar, pois precisam de um *feedback*[6] constante de suas alterações. Caso haja a necessidade de correção, o *redeploy* deve ser feito rapidamente.
- Escalabilidade da aplicação pode ser tornar algo difícil - A arquitetura monolítica só permite escalar a aplicação como um todo. Por um lado, pode-se escalar um volume grande de transações executando mais instâncias da aplicação. Algumas configurações em estrutura *cloud* podem ajustar o número de instâncias dinamicamente conforme a demanda. Mas, por outro lado, essa arquitetura não pode ser escalada com um número crescente de dados. Cada instância da aplicação acessará a toda base de dados, o que torna o armazenamento em cache[7] menos eficiente, aumenta o consumo de memória e o tráfego de entrada e saída. Além disso, cada componente da aplicação possui diferentes requisitos de recursos – um pode precisar de um grande processamento e outro componente pode

[4] Ambiente responsável por gerenciar os ciclos de vida das ações do sistema, mapeando URLs e garantindo direitos de acesso para estas ações.

[5] Realizar novamente a atividade descrita no item 1.

[6] Resposta a um determinado pedido ou acontecimento.

[7] Memória de rápido acesso

precisar de muita memória. Com uma arquitetura de microserviços, é possível escalar um componente de forma independente.

- Obstáculo ao desenvolvimento - Uma aplicação monolítica também é um obstáculo ao desenvolvimento escalável. Uma vez que a aplicação toma determinadas proporções, é mais produtivo que a equipe de desenvolvimento seja dividida em times menores, que focam em áreas específicas da aplicação. Por exemplo, pode-se ter uma equipe voltada para o desenvolvimento de *interfaces* para o usuário, outra equipe voltada para o módulo de contas a pagar e contas a receber, uma equipe para o módulo de estoque, e assim por diante. O problema das aplicações monolíticas é que essas impedem que times sejam dispostos a trabalhar de forma independente. As equipes devem coordenar seus esforços de desenvolvimento e *redeploy*. É muito mais difícil para uma equipe executar alterações e realizar atualizações em produção.
- A tecnologia escolhida precisa de um comprometimento a longo prazo – uma arquitetura monolítica acaba forçando o desenvolvedor à "se casar" com a linguagem escolhida (em alguns casos, a uma versão específica desta linguagem). Em uma aplicação monolítica, pode ser difícil, mesmo que incrementalmente, adotar uma nova tecnologia. Por exemplo, vamos supor que uma JVM[8] tenha sido escolhida para o projeto. Existem algumas linguagens que podem ser utilizadas, que se comunicam bem com o Java[9], como Groovy[10] e Scala[11]. Porém componentes não compatíveis com linguagens JVM não podem ser utilizados se a arquitetura utilizada for monolítica. Além disso, se a aplicação utilizar um *framework*[12] que eventualmente se tornar defasado, uma migração para um *framework* mais atual e melhor pode ser algo realmente complicado. É possível que a

[8] Java Virtual Machine, é um programa que carrega e executa os aplicativos Java, ou seja, um interpretador Java.

[9] Linguagem de programação orientada a objetos desenvolvida na década de 90.

[10] Linguagem de programação orientada a objetos desenvolvida para a plataforma Java como alternativa à linguagem de programação Java.

[11] Linguagem de programação de propósito geral, diga-se multiparadigma, projetada para expressar padrões de programação comuns de uma forma concisa, elegante e type-safe.

[12] Conjunto de classes que unidas servem para prover soluções específicas.

necessidade de adotar um novo *framework* faça com que seja necessário reescrever a aplicação por inteiro, o que é algo arriscado.

Além dos problemas conhecidos desta abordagem, citados anteriormente, também sofre-se com o um alto grau de acoplamento entre os diversos componentes que compõem o sistema. Porém este acoplamento não está apenas a nível de código, mas principalmente está presente no modelo de dados utilizado.

Esse alto grau de acoplamento tem gerado alto custo para o desenvolvimento/manutenção dos sistemas, uma vez que entender toda a regra de negócio envolvida em um sistema monolítico de grande porte acarreta em uma considerável curva de aprendizagem para os novos colaboradores, bem como, desenvolver testes e ter um *core* que atenda vários clientes, demanda um alto esforço. É possível migrar uma solução monolítica para uma arquitetura de microserviços, respeitando o objetivo original da aplicação?

1.2 OBJETIVOS

Apresentam-se os objetivos a serem alcançados através deste projeto.

1.2.1 Objetivo geral

Propor uma arquitetura baseada em microserviços para migrar uma aplicação previamente desenvolvida de maneira monolítica.

1.2.2 Objetivos específicos:

- Identificar os módulos da aplicação passiveis de migração para microserviços;

- definir um barramento de comunicação entre os microserviços;
- propor um protótipo funcional a partir dos microserviços construídos;
- implementar um protótipo funcional para atestar a proposta de solução;
- avaliar o protótipo de solução a partir de um experimento aplicado, utilizando um questionário.

1.3 JUSTIFICATIVA

Seguindo a nova tendência do mercado de desenvolvimento de *software* e visando, além de um avanço tecnológico, à aplicação dos conceitos de *Domain Drive Design* (DDD) – traduzindo para o português, Projeto Orientado à Domínio, que pode ser definido como uma visão e abordagem para lidar com domínios complexos, que buscam fazer do próprio domínio de negócio o foco principal do projeto, além de manter um modelo de *software* que reflete em um profundo entendimento do domínio (AVRAM, 2007) – procurando desacoplar os nossos sistemas, que, atualmente possuem um alto grau de acoplamento, não somente em nível de código, mas principalmente em nível de modelo de dados. Segundo Gupta (2015), a utilização de microserviços força que cada serviço seja visto como responsável por um domínio de negócio, construindo dessa forma aplicações usando DDD. Cada serviço é responsável por uma única parte da funcionalidade, e a faz bem (*Single Responsibility Principle*). As interações com os demais sistemas devem ser feitas, via *interfaces* (*Explicitly Published Interface*), sem a necessidade de utilização de *views* ou tabelas, apenas serviços.

Cada serviço é capaz de ser desenvolvido, atualizado, substituído e escalado de forma independente (*Independent DURS - Deploy, Update, Replace, Scale*). Além disso, microserviços geralmente são projetados para possuírem uma comunicação leve, baseada em REST[13], através de HTTP, STOMP [14]sobre *WebSocket*[15], esses e outros protocolos similares são utilizados para comunicação entre serviços (*Lightweight Communication*).

[13] Padrão de comunicação utilizado por *web services*.

[14] Protocolo de troca de mensagens simples orientado à texto.

[15] Tecnologia de comunicação *full-duplex* baseado no protocolo TCP.

Este trabalho visa a apresentar uma proposta baseada na arquitetura de microserviços para reaproveitamento de módulos de soluções monolíticas. Essa alteração da arquitetura facilita a construção de *software*, uma vez que desenvolver testes, refatorar código e treinar novos colaboradores, torna-se algo muito mais fácil quando se tem componentes (serviços) menores, com responsabilidades bem definidas e que respeitam o domínio de negócio de cada aplicação. Pelo fato de serem serviços específicos e com menores quantidades de código, a IDE trabalha mais leve, assim, aumentando a produtividade do desenvolvedor. Além de que utilizar essa nova arquitetura nos permite escalar independentemente de duas maneiras cada serviço: uma delas é clonar o serviço e aumentar o processamento da aplicação ou sua memória, a outra forma é aplicar o conceito *sharding*, que é "[...] uma técnica muito usada atualmente para lidar com escalabilidade de massas de dados, consiste basicamente em dividir os dados de uma aplicação entre vários bancos [...]" (JUNIOR, 2010). Essa configuração de escalabilidade é feita sempre conforme a necessidade da aplicação. Isso é bem diferente em aplicações monolíticas, que podem ter módulos com requisitos bem diferentes um dos outros, porém todos os módulos devem iniciar juntos no *deploy*, com as mesmas configurações.

Serviços podem ser implantados independente de outros. Qualquer alteração em um serviço pode ser facilmente feita por um desenvolvedor, sem a necessidade de envolvimento de outros times de desenvolvedores. Por exemplo, uma pequena alteração pode gerar uma nova versão, sendo o resultado disso a clara agilidade dos microserviços, como também a facilidade da integração contínua e entrega contínua, além de viabilizar testes em cenários de vários clientes, o que proporciona um único código que atenda vários deles.

Uma das facilidades de trabalhar com microserviços é que tipicamente cada serviço é construído por apenas uma equipe, sendo essa equipe responsável por tal serviço. Ou seja, todos os envolvidos com o domínio desse serviço trabalham juntos, próximos e no mesmo time. Isso melhora significativamente a comunicação entre os membros da equipe, pois todos estão envolvidos no mesmo assunto, focados no mesmo objetivo, compartilhando suas dificuldades a cada dia.

O uso da arquitetura de microserviços possui outras facilidades, uma delas é que um serviço que apresente mal comportamento, como, por exemplo, quando uma conexão com o banco de dados não é fechada, essa afetará somente ele mesmo, ao contrário da estrutura monolítica, a qual afetaria toda a aplicação. Isso melhora o isolamento de falhas e o quanto uma falha pode prejudicar a aplicação, estabelecendo assim uma aplicação com poucos *single*

points of failure. Outra facilidade, e acredita-se que uma das mais importantes, é que, para desenvolver um serviço, o desenvolvedor é livre para escolher a linguagem de programação mais adequada para fazê-lo. Isso permite que o serviço seja refeito com outra linguagem, tecnologia, ferramenta ou *framework*, caso alguma escolha do passado não tenha sido a mais adequada.

1.4 ESTRUTURA DA MONOGRAGIA

O capítulo 1 faz a introdução do trabalho, trazendo os objetivos, a justificativa, a problemática e a estrutura do trabalho.

O capítulo 2 apresenta o referencial teórico.

O capítulo 3 apresenta o método de pesquisa utilizada.

O capítulo 4 apresenta o projeto de solução, apresentando as metodologias utilizadas bem como a modelagem da arquitetura proposta.

O capítulo 5 apresenta a proposta de solução, bem como as ferramentas utilizadas, e avaliação do protótipo desenvolvido.

O capítulo 6 mostra as conclusões e trabalhos futuros.

2 DESENVOLVIMENTO DE *SOFTWARE*

Na década de 70, durante a crise do *software*, os sistemas criados geralmente eram de má qualidade devido à falta de estrutura e planejamento que os projetos de *software* eram concebidos. A partir desta crise a necessidade de tornar o desenvolvimento de *software* em um processo estruturado e padronizado se tornou irrevogável. Hoje em dia há um grande foco das organizações na análise dos processos de desenvolvimento de *software* e segundo Pressman (2006, p.17):

> Os processos de *software* formam a base para o controle gerencial de projetos de *software* e estabelecem o contexto no qual os métodos técnicos são aplicados, os produtos de trabalho (modelos, documentos, dados, relatórios, formulários etc.) são produzidos, os marcos são estabelecidos, a qualidade é assegurada e as modificações são adequadamente geridas.

Esse mesmo autor também constata que a engenharia de *software* é uma tecnologia que deve estar alinhada com compromisso organizacional e qualidade. Segundo ele, a base que sustenta essa tecnologia da engenharia de *software* é a camada de processo.

> O processo de engenharia de *software* é o adesivo que mantém unidas as camadas de tecnologia e permite o desenvolvimento racional e oportuno de *softwares* de computador. O processo define um arcabouço que deve ser estabelecido para a efetiva utilização da tecnologia de engenharia de *software*. (PRESSMAN, 2006, p.17)

O desenvolvimento de *software* é um processo que utiliza o conhecimento de forma intensiva, além de envolver muitas pessoas que executam tarefas em diferentes fases ou atividades. O conhecimento encontrado nas empresas desenvolvedoras de *software* é relacionado as mais diversas áreas, porém, identificar o conteúdo, localização e o uso deste conhecimento é um dos problemas mais recorrentes entre elas.

Cada um dos envolvidos no desenvolvimento de *software* toma decisões técnicas ou administrativas. Os desenvolvedores por exemplo, fazem suas tomadas de decisão baseadas em experiência, conhecimento pessoal ou conhecimento obtido a partir de contatos informais, o que é frequente em empresas de menor porte. Porém, se levar em consideração empresas que lidam com um maior número de informações, esse mesmo paradigma acaba não podendo ser aplicado. Em empresas grandes, que trabalham com informações mais robustas, não deve acontecer o repasse de informações pessoais de maneira informal. O conhecimento precisa ser compartilhado, porém de forma estruturada, sendo assim, o processo de compartilhamento de informações precisa ser bem definido.

Durante o desenvolvimento de *software* diversos artefatos e documentos são gerados, ou seja, há uma grande produção de conhecimento. Conhecimento esse que deve ser absorvido e disponibilizado entre toda a equipe de desenvolvimento, para que possa ser reutilizável em projetos futuros. No entanto, para que isso seja possível, o conhecimento individual precisa ser capturado no sentido de os demais integrantes da equipe tenham a oportunidade de aprendizado.

2.1 PROCESSO DE *SOFTWARE*

No mercado da tecnologia de informação, grande parte das pessoas têm ciência que desenvolvimento de *software* não é um processo tão simples como muitos pensam. Segundo Fuggetta (2000, tradução nossa), "O desenvolvimento de *software* é um processo coletivo, complexo e criativo. Sendo assim, a qualidade de um produto de *software* depende fortemente das pessoas, da organização e de procedimentos utilizados para criá-lo e disponibilizá-lo".

Além da complexidade do desenvolvimento de *software*, observa-se que, como citado por Fuggetta, a qualidade do *software* em produção depende dos procedimentos utilizados, ou seja, do modelo de desenvolvimento escolhido, das pessoas envolvidas no projeto e da organização. Completando o que o autor fala em seu artigo, Howard Baetjer Jr escreve o seguinte sobre processos de *software*:

> [...] pelo fato de *software*, como todo capital, ser conhecimento incorporado, e como esse conhecimento está inicialmente disperso, tácito, latente e incompleto na sua totalidade, o desenvolvimento de *software* é um processo de aprendizado social. O processo é um diálogo no qual o conhecimento, que deve se transformar em *software* é reunido e incorporado ao *software*. O processo fornece interação entre usuários e projetistas, entre usuários e ferramentas em desenvolvimento e entre projetistas e ferramentas em evolução (tecnologia). É um processo iterativo no qual a própria ferramenta serve como meio de comunicação, com cada nova rodada de dialogo explicitando mais conhecimento útil do pessoal envolvido. (BAETJER 1998, pág. 85, tradução nossa)

Visto a complexidade de desenvolver *softwares* e a grande quantidade existente de processos de *software*, é notório que criar processos do zero não é uma boa estratégia, apesar de existir organizações que desenvolvem seu próprio processo de desenvolvimento, ou

personalizam algum existente conforme suas necessidades, visto que não existe um processo ideal, Sommerville (2011, p. 19) complementa:

> Embora não exista um processo 'ideal' de *software*, há espaço, em muitas organizações, para melhorias no processo de *software*. Os processos podem incluir técnicas ultrapassadas ou não aproveitar as melhores práticas de engenharia de *software* da indústria. De fato, muitas empresas ainda não se aproveitam dos métodos da engenharia de *software* em seu desenvolvimento de *software*.

Resumidamente pode-se descrever que processo de *software* é um conjunto de atividades a serem aplicadas sistematicamente entre algumas fases. Cada uma das atividades é composta por participantes que podem ter responsabilidades. Toda atividade conta com diversas entradas e fornece diversas saídas, isto é, define quem faz o quê, como e quando para atingir um respectivo objetivo.

Tem-se na literatura técnica algumas definições para processo de *software*:

Jalote (2002, tradução nossa) define um processo de *software* como um conjunto de atividades, ligadas por padrões de relacionamento entre elas, pelas quais se as atividades operarem corretamente e de acordo com os padrões requeridos, o resultado desejado é produzido. O resultado desejado é um *software* de alta qualidade e baixo custo.

Para Pressman (2011, p. 52) "Processo de *software* é definido como uma metodologia para as atividades, ações e tarefas necessárias para desenvolver um *software* de alta qualidade. ".

Humphrey (1989) define processo de *software* como um conjunto de ferramentas, métodos e práticas necessárias para transformar os requisitos do usuário em *software*.

Sommerville (2011, p. 19) define que é "[...] um conjunto de atividades relacionadas que levam à produção de um produto de *software*[...]".

Florac e Carleton (1997) definem como uma organização lógica de pessoas, materiais, energia, equipamentos e procedimentos empregados na execução de atividades projetadas para produzir um resultado específico;

E, finalmente, para Fuggetta (2000), um processo de *software* é definido como um conjunto coerente de políticas, estruturas organizacionais, tecnologias, procedimentos e artefatos *que* são necessários para conceber, desenvolver, disponibilizar e manter um produto de *software*;

As atividades executas em cada fase do processo de *software* são definidas para atingir os objetivos propostos, naturalmente, essas formam um conjunto mínimo para alcançar o objeto final, que é o produto de *software*. Entre Sommerville (2011), Pressman (2011) e outros autores, realizando uma combinação dos conceitos, podemos identificar as seguintes atividades genéricas em um processo de *software*:

- Especificação: Definição de funcionalidades do *software* e suas restrições;
- Projeto e Implementação: Produção do *software* especificado;
- Validação de *software*: Validação do *software* para garantir que ele faça o que o cliente deseja;
- Manutenção e evolução: O *software* deve ser corrigido conforme erros encontrados e também deve evoluir conforme necessidades do cliente.

Essas atividades relacionadas ao processo de *software* estão vinculadas com o produto final, a produção de *software*. A discriminação das tarefas de cada atividade, quais serão executadas e qual sua ordem de execução é definida pelo modelo de desenvolvimento de *software*, no qual se tem um grande número de opções.

2.2 MODELOS DO PROCESSO DE *SOFTWARE*

Como citado na seção anterior, existem vários modelos de processo de *software* e todos têm como base a combinação de atividades genéricas do processo de *software* para alcançar o produto final, que é o produto de *software*. Se tem grande número de modelo de processos de *software* pois, cada modelo é uma abstração do processo genérico, o qual foi falado na seção anterior. Todo modelo de processo de *software* pode ser considerado um *framework* de processo, sendo capazes de serem ampliados e ou adaptados para serem processos mais específicos.

Mas o que é um modelo de processo de *software*? Segundo Sommervile pode-se definir que um modelo de processo de *software* é:

> [...] uma representação simplificada de um processo de *software*. Cada modelo representa uma perspectiva particular de um processo e, portanto, fornece informações parciais sobre ele. Por exemplo, um modelo de atividade do processo pode mostrar as atividades e sua sequência, mas não mostrar os papéis das pessoas envolvidas. (SOMMERVILLE, 2011, p. 19)

Os modelos de processo de *software* mais conhecidos são o modelo em cascata, o modelo evolucionário, o modelo iterativo, o modelo incremental e o modelo baseado em componentes. Além desses ainda existem outros, mas não entraremos em detalhes neste trabalho.

2.2.1 Modelo de desenvolvimento sequencial ou linear

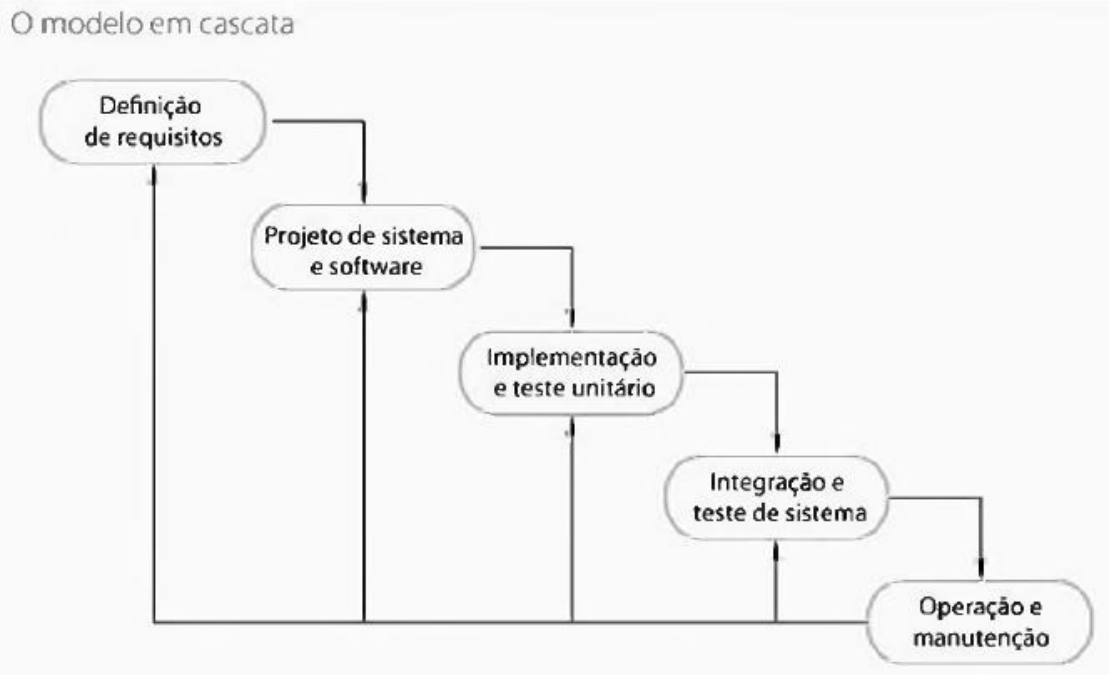
O modelo sequencial ou linear, inicia no nível de sistema e avança de forma sequencial, através de análise, projeto, implementação, testes e manutenção. Esse modelo abrange um conjunto distinto de atividades, ações, tarefas, marcos e produtos. Os produtos de do modelo (dados, documentos) são gerados através das atividades definidas pelo modelo.

2.2.1.1 Modelo cascata

Este modelo foi idealizado em 1970 por Royce e tem como característica principal a sequencialidade das atividades. Segundo Sommerville (2011) e Pressman (2006), esse modelo considera as atividades fundamentais do processo de especificação, desenvolvimento, validação e evolução, e representa cada uma delas como fases distintas, como: especificação de requisitos, projeto de *software*, implementação, testes, integração e manutenção. Devido o encadeamento entre uma fase e outra, esse modelo é conhecido como ‘modelo em cascata', apresentado na Figura 1. Um grande problema nesse modelo é que os impactos dos erros causados nas primeiras etapas apenas são identificados próximo ao fim do projeto. Conforme podemos ver na Figura 1, a proposta que esse modelo oferece é que as etapas de desenvolvimento sigam uma sequência, ou seja, o artefato gerado na primeira atividade é

encaminhado para a segunda atividade, assim como o artefato gerado na segunda atividade é encaminhado para a terceira atividade e assim por diante. O conceito é que as atividades que serão executadas nesse modelo só podem ser iniciadas quando a atividade anterior tiver sido finalizada, isto é, de forma sequencial.

Figura 1 - Modelo cascata.



Fonte: Sommerville (2011, p. 20)

Esse modelo possui a vantagem de só avançar para a próxima atividade quando o cliente analisar e aceitar os artefatos gerados no fim de cada uma. O modelo presume que o cliente esteja acompanhando de perto e ativamente o projeto. Além disso, parte do princípio que o cliente sabe qual seu desejo, sua ambição de produto final.

Uma das vantagens do modelo é a documentação produzida em cada fase e um dos seus problemas é a divisão inflexível do projeto em estágios distintos. Tudo que será feito deve ser definido no início do processo, o que torna difícil reagir às mudanças de requisitos. Para Pressman (2006) este modelo pode ser usado quando os requisitos forem bem compreendidos e houver pouca probabilidade de mudanças radicais durante o desenvolvimento do sistema.

Com a utilização desse modelo a expectativa é sempre alcançada, pois, o que é para ser gerado em certa atividade foi defino e produzido, sendo assim, não pode-se voltar

atrás e alterar os artefatos, assim como as conclusões das atividades anterior. Devido a esse fato, é comum que muitas ideias sejam descartadas e não aproveitadas no projeto.

2.2.2 Modelo de desenvolvimento evolucionário

Segundo Sommerville (2007) o modelo de desenvolvimento evolucionário intercala as atividades de especificação, desenvolvimento e validação. Um sistema inicial é desenvolvido rapidamente baseado em especificações abstratas. Este sistema é, então, refinado com as entradas do usuário para produzir um sistema que satisfaça suas necessidades. Esta abordagem é mais eficaz do que a abordagem em cascata na produção de sistemas que atendam às necessidades imediatas dos usuários. A vantagem de um processo de *software* baseado na abordagem evolucionária é que a especificação pode ser desenvolvida de forma incremental. À medida que os usuários compreendem melhor seu problema, esse conhecimento é repassado para o desenvolvimento do *software*.

Modelos evolucionários caracterizam-se por serem iterativos e apresentarem particularidades que possibilitam o desenvolvimento de versões completas do *software*. Os processos evolucionários se caracterizam por dois modelos comuns: Prototipação e Espiral.

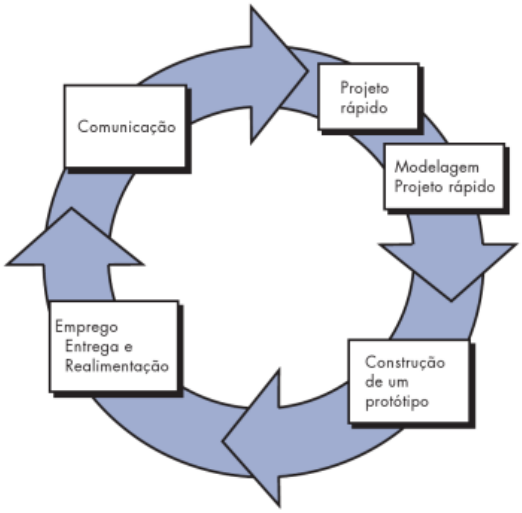
2.2.2.1 Prototipação

A prototipação é utilizada em situações na qual o cliente definiu objetivos para o *software* a ser produzido, contudo, não definiu em grandes detalhes os requisitos da funcionalidade, tais como: suas entradas, processamento e saídas. Logo, o desenvolvedor não sabe como o *software* deve interagir com o cliente. Quando uma situação dessas ocorre, o modelo de prototipação é uma excelente alternativa, visto que a prototipagem pode ser utilizada em qualquer processo de *software* e auxilia os interessados a entender melhor o que está para ser desenvolvido.

Assim, Bezerra (2007) destaca que a prototipagem (construção de protótipos) é uma técnica que serve de complemento à análise de requisitos. Além disso, no contexto do desenvolvimento de *software*, um protótipo é um esboço de alguma parte do sistema.

Esse modelo utiliza do conhecimento dos requisitos iniciais para montar um protótipo do *software* que obedeça às necessidades do cliente. A produção de *software* se deve a realização das atividades de: análise de requisitos, o projeto, a codificação e os testes. Essas atividades podem não ser realizadas de modo formal e evidente. Pode-se ver as atividades deste modelo na Figura 2.

Figura 2 - Modelo de prototipação.



Fonte: Pressman (2011, p. 63)

O modelo de prototipação, figura 2, pode ser apresentado da seguinte forma: onde cada uma das atividades é definida conforme a seguir:

1. Comunicação: Reuniões com as partes interessadas para definição dos objetos do projeto e levantamento dos requisitos já conhecidos.
2. Plano rápido: Planejamento do protótipo.
3. Modelagem e projeto rápido: Modelagem e projeto do protótipo.
4. Construção do protótipo: Codificação do protótipo.
5. Implantação, entrega e *feedback*: Entrega do protótipo para análise do cliente.

Nesse modelo, o protótipo é considerado uma versão inicial do *software*. O mesmo é usado para mostrar conceitos, testar opção de projeto e examinar sobre os possíveis problemas e solução no percurso de desenvolvimento. O essencial é que o desenvolvimento do protótipo seja rápido, para que os custos sejam controlados e também para que o cliente possa ensaiar com o protótipo logo no início da produção de *software*.

2.2.2.2 Espiral

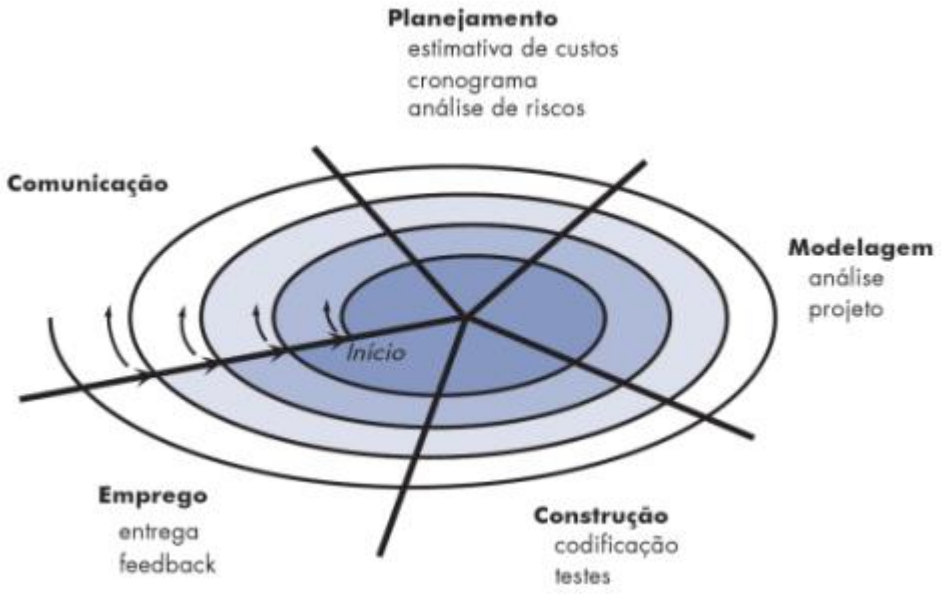
Segundo Sommerville (2011) o modelo espiral foi proposto por Boehm em 1988 como forma de integrar os diversos modelos existentes à época, eram eles o modelo linear e modelo de prototipação. O projeto contemplava eliminar as dificuldades e explorar os pontos fortes dos modelos já existentes. Esse modelo foi desenvolvido para abranger as melhores características do ciclo de vida clássico como da prototipação, acrescentando um novo recurso, a análise de riscos, recurso que nenhum dos outros modelos possuía.

Contudo, a criação do desse modelo não se dá através da simples incorporação de características dos modelos anteriores. Esse modelo apresenta o modelo de forma diferenciada, assumindo que o processo de desenvolvimento ocorre em ciclos, cada um contendo fases de avaliação e planejamento, onde a opção de abordagem para a próxima fase (ou ciclo) é determinada.

Segundo Pressman (1995) o modelo espiral, utilizado para engenharia de *software*, foi concebido para utilizar os melhores atributos do ciclo de vida clássico de desenvolvimento de *software* e da prototipação. Junto a isso, ele acrescenta o conceito de análise de riscos, que não estava presente nos outros modelos.

Como pode-se ver na Figura 3, aqui o modelo é representado como uma espiral, e não como uma sequência de atividades.

Figura 3 - Modelo espiral.



Fonte: Pressman (2011, p. 63)

Como pode-se ver na figura 3, o modelo define cinco importantes atividades representadas por cinco quadrantes:

1. Planejamento – Análise de risco.
2. Engenharia – Modelagem.
3. Construção e evolução – Codificação e testes.
4. Emprego – Avaliação do cliente.
5. Comunicação com o cliente.

O número de tarefas por regiões é decidido conforme o tamanho e complexidade do projeto.

Atualmente o modelo espiral trata de maneira realista o desenvolvimento de *software* de grande escala. Pelo fato da abordagem deste modelo ser cíclica, permite com quem a empresa que presta o serviço e o cliente possam tomar decisões para possíveis riscos de cada etapa evolutiva. Portanto, para isto, é pré-requisito que se tenha uma experiência na determinação de riscos, pois sem esta experiência será difícil ter sucesso com este modelo.

2.3 TÉCNICAS E PRÁTICAS UTILIZADAS

O desenvolvimento de *software* certamente é um dos processos mais discutidos no mundo da tecnologia. Com o passar do tempo essas discussões geraram novos conceitos que constituíram padrões de projetos que são utilizados na implementação do *software*. Para o desenvolvimento da proposta apresentada nesse projeto, foram adotados algumas técnicas e boas práticas de desenvolvimento que serão apresentadas a seguir.

2.3.1 *Test-Driven* Development

Test-Driven Development (TDD) nos últimos tempos passou a ser uma das práticas mais utilizadas pelos desenvolvedores de *software*. Segundo Aniche (2014), "A ideia é bem simples: escreva seus testes antes mesmo de escrever o código de produção".

Ao escrever testes antes de começar o desenvolvimento do código de produção, o desenvolvedor garante que boa parte ou todas as funcionalidades desenvolvidas estão de acordo com os requisitos e regras de negócio levantadas inicialmente.

Para Cardoso (2013),

> [..]os testes devem ser unitários. Isto implica em um teste automatizado certificar-se de apenas uma funcionalidade do código utilizando para isso quantos *asserts* forem necessários. Por serem testes automatizados são fácil e rapidamente executados eliminando testes manuais que encarecem o *software* em vários aspectos. [..] (CARDOSO, 2013).

A utilização de TDD reduz a adição de *bugs*[16] no código e ajuda a reduzir custos do projeto – devido à cortes com gastos de testes manuais e devido aos erros serem encontrados em um estágio inicial do desenvolvimento e não em estágios mais avançados.

[16] Falha no *software* que provoca mau funcionamento.

2.3.2 *Domain-Driven Design*

O termo *Domain-Driven Design*, em português Projeto Orientado a Domínio, surgiu do título do livro escrito por Eric Evan em parceria com Martin Fowler — *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Segundo Martin Fowler (2014), DDD trata modelos de larga escala dividindo-os em diferentes contextos limitados e explicitando seus relacionamento.

Basicamente DDD é a definição de um contexto que permite apenas a inserção de comportamentos que façam parte daquele domínio de negócio. Dentre os conceitos que envolvem uma arquitetura baseada em microserviços, o entendimento dos limites de um contexto certamente é uma das mais complicadas de se aplicar, já que esta característica precisa de um alto nível de entendimento do negócio para ser corretamente praticada.

2.3.3 Clean Code

Clean Code é o termo utilizado para o processo de codificar utilizando boas práticas de desenvolvimento. Segundo Vuorinen (2014), *Clean Code* é subjetivo e cada desenvolvedor tem sua própria visão do conceito. Existem algumas ideias na indústria de *software* e na comunidade de desenvolvedores das melhores práticas que tornam um código limpo, porém, não há um conceito definitivo e dificilmente haverá.

Dentre as muitas referências que existem para aplicação de boas práticas de codificação, o livro *Clean Code: A Handbook of Agile Software Craftsmanship* dos autores Robert C. Martin, Michael C. Feathers e Timothy R. Ottinger, é uma das obras mais conceituadas sobre o tema e grande parte dos conceitos apresentados neste livro foram utilizadas no desenvolvimento da proposta desse trabalho.

2.4 MICROSERVIÇOS

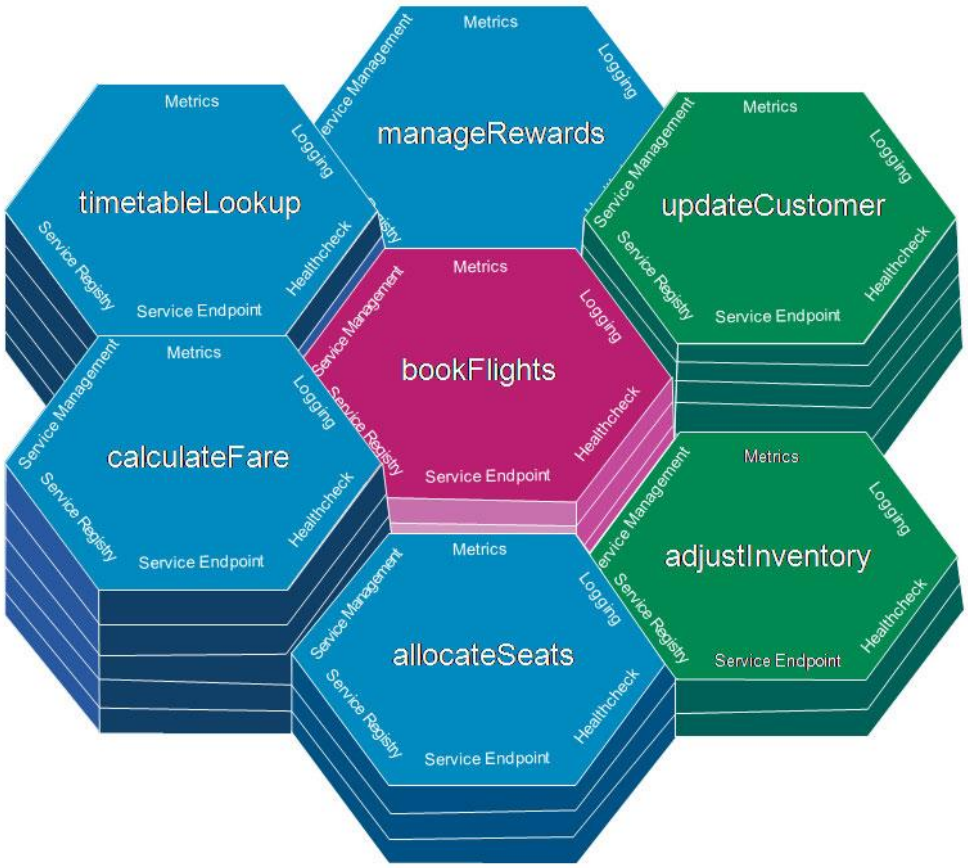
Microserviços é um estilo de arquitetura, que em aplicativos complexos são compostos por dois ou mais serviço. Para Newman (2015), microserviços são pequenos serviços autônomos e independentes que trabalham em conjunto. Microserviços também pode ser visto como uma evolução dos sistemas distribuídos, que surgiram a partir da necessidade de tornar as aplicações monolíticas de código pesado, para serviços que trabalham de forma independente. Cada microserviços foca em apenas uma tarefa, a qual executa de maneira eficiente. Esta tarefa executada por determinado microserviço sempre representa uma pequena parte do negócio.

2.4.1 Conceitos

Segundo Namiot (2014), microserviços é um termo relativamente novo que se refere à criação de componentes independentes e reutilizáveis, responsáveis por domínios de negócios bem definidos, além de utilizar uma comunicação leve. Já para LEWIS e FOWLER (2014), arquitetura de microserviços é uma abordagem de desenvolvimento para uma única aplicação em formato de pequenos serviços, cada um executando seu processo de forma independente e se comunicando utilizando mecanismos leves, como APIs que se comunicam via HTTP. Esses serviços são construídos em torno das necessidades do negócio e o *deploy* desses serviços deve ser feito de maneira independente por mecanismos de *deploy* totalmente automatizados. Há um mínimo gerenciamento centralizado destes serviços. Serviços estes que podem ser escritos em diferentes linguagens de programação e utilizar diferentes tecnologias de armazenamento.

A figura 4 exemplifica de maneira visual, como uma aplicação arquitetada em microserviços se dispõe. Neste exemplo todos os microserviços da arquitetura se comunicam com o microserviço “principal” (*bookFlights*) e entre si, caso necessário.

Figura 4 - Exemplo de aplicação utilizando microserviços.



Fonte: IBM (2015, p. 4)

A seguir estão listados os aspectos que segundo a IBM (2015) são considerados como chave na definição de microserviços.

2.4.1.1 Pequeno e focado

Microserviços precisam focar em apenas uma unidade de trabalho, sendo assim eles são pequenos. Não existem regras do quão pequenos os microserviços devem ser. A regra Two-Pizza Team, tipicamente é referenciada como uma linha a se seguir, que apresenta o seguinte: se você não conseguiu alimentar com duas pizzas uma equipe que está construindo um microserviço, então seu microserviço é muito grande. Se for desejo que seu microserviço

seja pequeno o suficiente, então você pode facilmente reescrever e manter todo o microserviço, se necessário dentro de uma equipe.

Um microserviço também precisa ser tratado como uma aplicação ou como um produto. Ele deve possuir seu próprio código fonte, além de sua própria forma de executar builds e *deployment*. Embora o dono do produto queira incentivar o reuso do microserviço, reuso não é a única motivação para utilização de microserviços. Existem diversas motivações, como otimizações pontuais visando melhorias nas *interfaces* responsivas utilizadas pelo usuário, tornando assim a resposta às necessidades do cliente mais rápida.

A granularidade de um microserviço também pode ser determinada baseando-se na necessidade do negócio. Rastreamento de pacotes, serviço de quota no seguro de automóveis e previsão do tempo, são todos exemplos de serviços entregues por terceiros, ou disponibilizados como competência central de um serviço necessário.

2.4.1.2 Baixo acoplamento

Baixo acoplamento é definitivamente uma característica essencial dos microserviços. O *deploy* de um microserviço deve poder ser feito sem a necessidade de terceiros. Não pode haver a necessidade de efetuar o *deploy* de outros microserviços, para que este funcione. Esse baixo acoplamento, permite *deploys* mais rápidos e frequentes, o que torna o desenvolvimento mais eficiente.

2.4.1.3 Linguagem neutra

Utilizar a ferramenta certa para o trabalho certo é muito importante. Microserviços precisam ser construídos utilizando a linguagem de programação e a tecnologia que é mais adequada para a tarefa. Microserviços são um conjunto que se completa para formar uma aplicação complexa, e eles não precisam ser escritos utilizando a mesma

linguagem. Em determinados casos, Java pode ser a linguagem mais apropriada, e em outros casos a linguagem mais adequada pode ser Python, por exemplo.

A comunicação com microserviços é através de APIs de linguagem neutra, tipicamente um recurso baseado em HTTP, como o REST. É necessário padronizar a integração e não a plataforma utilizada pelos microserviços.

2.4.1.4 Contexto limitado

O conceito de contexto limitado para microserviços, é o fato de não haver a necessidade de um microserviço saber sobre a implementação de outro microserviço utilizado dentro de uma mesma aplicação. Quando há a necessidade de comunicação entre dois microserviços, a divulgação das informações deve ser feito da maneira mais simples e eficiente possível.

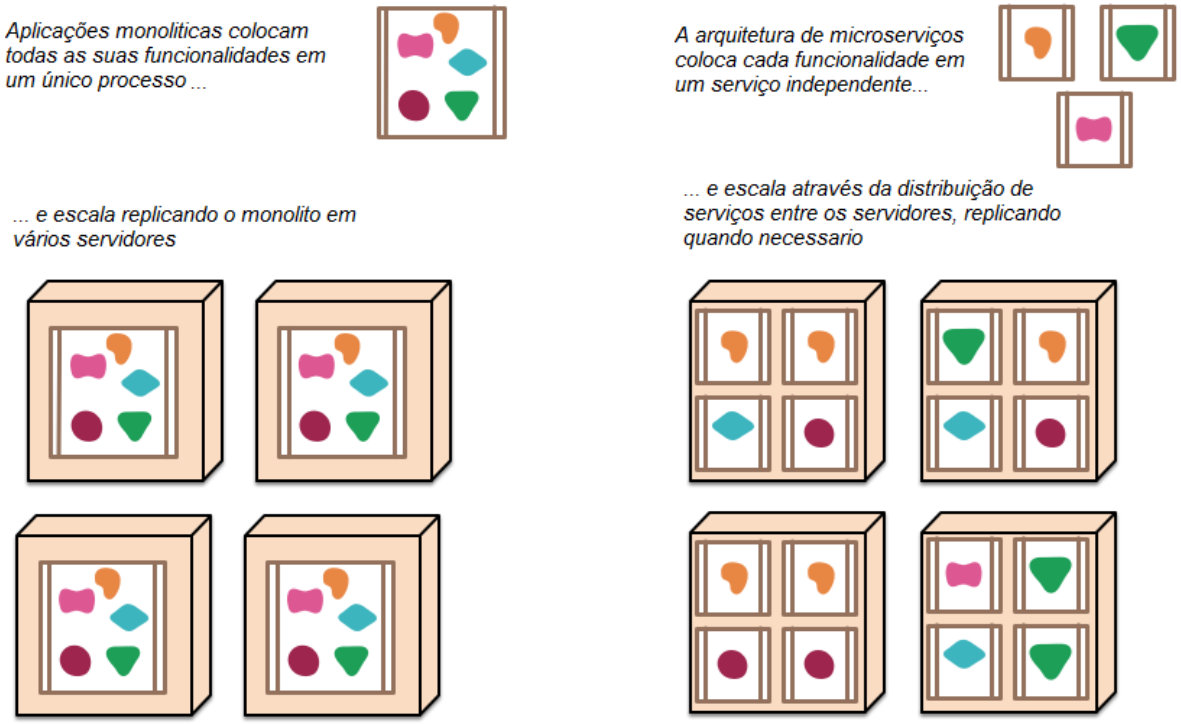
2.4.2 Modelo de arquitetura em microserviços

Quando se trata de microserviços não se deve utilizar paradigmas como por exemplo a arquitetura *web* tradicional, como parâmetro. Segundo Rosa (2015), as vantagens observadas em um modelo *web* tradicional, como a facilidade de instalação, no caso de uma aplicação utilizando a tecnologia Java, visto que é necessário apenas a implantação de um arquivo WAR[17] no contêiner *web*, escalabilidade que se torna uma tarefa mais simples, visto que replicar uma aplicação pode ser feito a partir da execução de cópias desta, que podem ser orquestradas por um balanceador de carga que efetuar o direcionamento das requisições oriundas dos clientes, não podem ser vistas como benefícios na arquitetura baseada em

[17] *Web* Application Archive - arquivo JAR usado para distribuir uma coleção de JavaServer Pages, Servlets Java, classes Java, arquivos XML, bibliotecas de tags, páginas *web* estáticas (arquivos HTML e relacionados) e outros recursos que, juntos, constituem uma aplicação *web*.

microserviços. Bases de código de sistemas monolíticos intimidam os desenvolvedores, já que a sua compreensão e modificabilidade no código fonte são muito mais complexas. A forma como a arquitetura *web* trata a escalabilidade, também não pode ser tratada como uma maneira eficiente de executar tal premissa, devido à escalabilidade poder ser executada em apenas uma dimensão, podendo a arquitetura não ser dimensionada da maneira mais adequada com o volume crescente de dados, além de não ser possível dimensionar cada componente de maneira interdependente, como podemos ver na figura 5.

Figura 5 - Monólitos e Microserviços.



Fonte: (FOWLER; LEWIS, 2014)

A figura 5 apresenta as diferenças citadas no texto de uma maneira mais clara, onde fica evidente a replicação dos monólitos baseados em uma arquitetura mais clássica. Também fica evidente que a escalabilidade a partir de uma arquitetura baseada em microserviços é muito mais eficiente e focada na necessidade que a aplicação apresenta.

2.4.2.1 Características da arquitetura em microserviços

Nesta seção é apresentado as características de uma arquitetura de microserviços descritas pela IBM (2015), Newman (2015) e outros autores.

2.4.2.1.1 Orientado ao negócio

Segundo a IBM (2015), quando um sistema é quebrado em serviços compostos, há um grande risco que está decomposição seja feita ao longo dos limites existentes na organização. Isso significa que o sistema pode ser mais frágil, devido a existência de mais partes independentes à serem gerenciadas, além do risco de haver serviços com integração falha, mesmo havendo comunicação entre os serviços resultantes.

Além disso, se os serviços que compõe o sistema, não forem projetados para o objetivo certo, eles não poderão ser reutilizados. Os custos de desenvolvimento e manutenção também podem ser maiores se os serviços forem projetados de acordo com os limites organizacionais ou tecnológicos.

É fundamental projetar os microserviços vislumbrando o objetivo final do negócio. Para tanto, pode ser necessário que as equipes dispostas ao longo dos limites organizacionais se juntem para projetar os serviços de forma conjunta, ao invés de usar microserviços para efetuar chamadas entre as equipes.

2.4.2.1.2 Projetado para falha

A IBM (2015) e Newman (2015) definem esta característica dos microserviços utilizando algumas analogias, como por exemplo:

Engenharia de *software* pode utilizar muitas ideias da engenharia civil, onde os engenheiros lidam com a concepção, construção e manutenção de estruturas físicas, tais como estradas, pontes, canais barragens e edifícios. Engenheiros civis projetam sistemas esperando a falha de componentes individuais e constroem camadas redundantes que garantem a segurança e a estabilidade das construções.

A mesma mentalidade pode ser aplicada na engenharia de *software*. A mudança de mentalidade é o que define a aceitação do fato de que falhas isoladas são inevitáveis, mas o objetivo do projeto é manter o sistema funcionando pelo maior tempo possível. Práticas de engenharia como modelo de falhas e injeção de falhas devem ser incluídas como parte do processo de entrega continua, para projetar sistemas mais confiáveis.

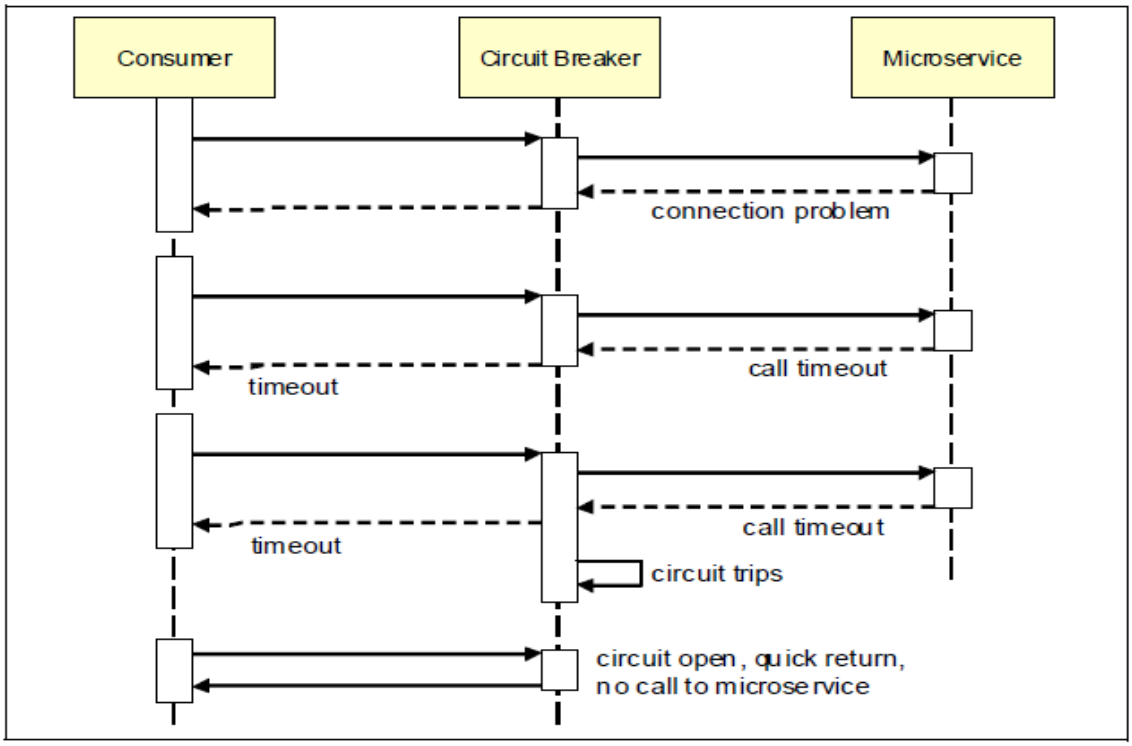
Há diversos padrões de projeto para projetos que visam a falha e a estabilidade. Segue a descrição de alguns destes padrões.

2.4.2.1.2.1 Circuit breaker (Disjuntor)

Um dos padrões que a IBM (2015) define como projetado para falha o padrão disjuntor que segundo a instituição, normalmente é usado para garantir que quando houver uma falha, o serviço que apresentou esta falha não afete o sistema inteiro. Isso aconteceria se a houvesse uma grande quantidade de chamadas para o serviço que está apresentando falha, e para cada chamada seria necessário aguardar um determinado tempo de espera para constatar que o serviço não está respondendo antes de prosseguir. Esperar este tempo de resposta de um serviço que falhou, utilizaria recursos e tornaria o sistema como um todo instável.

O padrão disjuntor, se comporta exatamente como um disjuntor utilizado no circuito elétrico de uma casa. Quando um serviço falhar, o objeto disjuntor que envolve este microserviço permite chamadas subsequentes ao serviço até um determinado limite de tentativas falhas é atingido. Nesse ponto, o disjuntor define que as chamadas ao microserviço que falhou, não serão mais executadas. Esta configuração poupa recursos do sistema e mantém a estabilidade do sistema como um todo. A Figura 6 apresenta o diagrama de sequência de um disjuntor aplicado à um microserviço.

Figura 6 - Diagrama de sequência de um disjuntor.



Fonte: IBM (2015, p. 22)

Quando o disjuntor desarma e o circuito é aberto, uma contingência pode ser iniciada no lugar. A lógica de contingência tipicamente executa um pequeno ou nenhum processamento, e retorno um valor. A lógica de contingência deve possuir chances mínimas de falha, devido esta rodar como resultado de uma falha inicial.

2.4.2.1.2.2 Bulkheads (Anteparas)

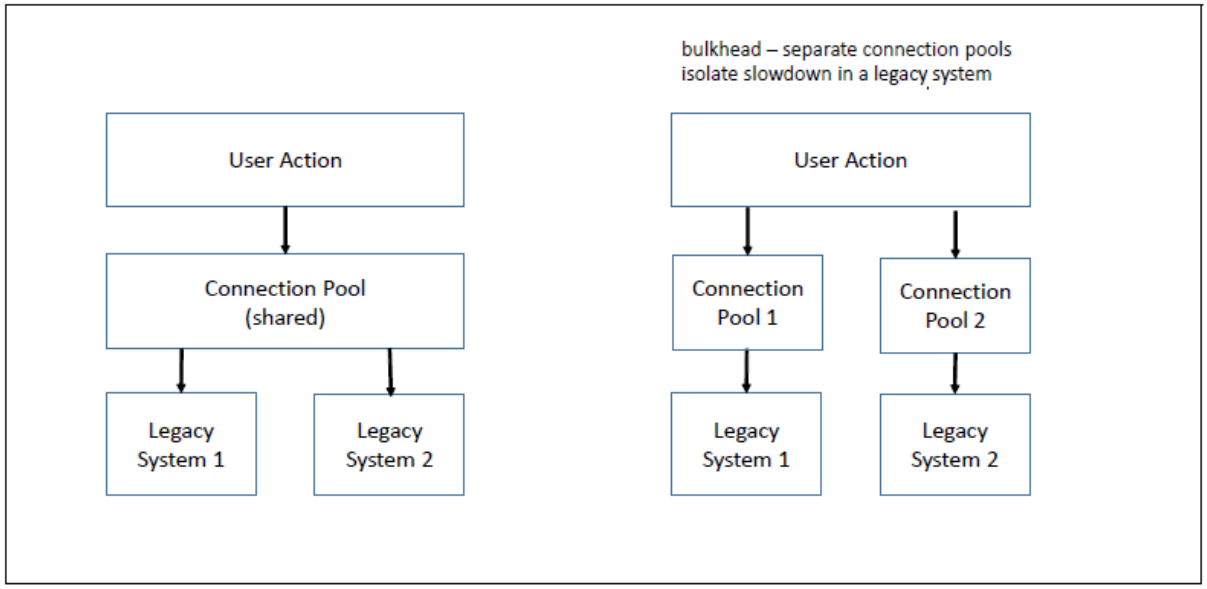
Este também é um padrão projetado para falha definido pela IBM (2015) utilizando a seguinte analogia: O casco de um navio é composto de diversas anteparas individuais e impermeáveis. A razão para isso é que se uma antepara é danificada, essa falha se limitará apenas à esta antepara, ao invés de afundar o navio inteiro.

Este tipo método de partição pode ser utilizado em *software*s também, para isolar a falha em pequenas porções do sistema. O limite do serviço (isto é, o microserviço em si) serve como uma antepara que isola qualquer falha. Romper com a funcionalidade (assim

como feito em uma arquitetura SOA) em microserviços separados, serve para isolar o efeito desta falha em apenas um microserviço.

O padrão Bulkhead também pode ser aplicado dentro dos próprios microserviços. Por exemplo, considere um pool de threads que é utilizada para chegar em dois sistemas existentes. Se um dos sistemas apresentar uma demora que afete a funcionalidade do pool de threads, o acesso ao outro sistema existente também será afetado. Havendo pools de threads separadas garantiria que essa demora em um dos sistemas existente, afetaria diretamente apenas sua própria pool de threads, e não afetaria o acesso a nenhum dos outros sistemas existentes. A figura 7 apresenta a diferença entre um *pool* de *threads* compartilhada para um *pool* de *threads* única para cada sistema.

Figura 7 - Exemplo do uso do padrão Bulkhead: Utilizando pools de thread separadas para isolar falhas.



Fonte: IBM (2015, p. 23)

2.4.2.1.3 Gerenciamento de dados descentralizados

Para a Richardson (2015A) e IBM (2015), em uma aplicação monolítica, lidar com transações é uma tarefa fácil devido ao fato de todos os componentes da aplicação fazerem parte do monólito. Quando pensamos na arquitetura em microserviços que é distribuída, as transações que ocorrerão podem acontecer em diversos serviços distintos. Na arquitetura de microserviços, a preferência é pela BASE (*Basically Available, Soft state, Eventual consistency*) acima do ACID (*Atomicity, Consistency, Isolation, Durability*). Transações distribuídas devem ser evitadas sempre que possível. O ideal é que cada microserviço gere seu própria banco de dados. Isto permite persistência em mais de uma linguagem de programação, utilizando bases de dados diferentes (por exemplo, *Cloud*ant versus MongoDB, ambos são NoSQL) e diferentes tipos de armazenamento de dados (como *Structured Query Language* (SQL), NoSQL, grafos). No entanto, mais de um microserviço pode utilizar o mesmo banco de dados e isto pode se dar devido à diversas razões, por exemplo, para preservar a natureza ACID de uma transação que seria distribuída através dos microserviços e dos bancos de dados.

Independente da razão, deve-se tomar cuidado ao compartilhar bases de dados através de microserviços. Os prós e contras devem ser considerados. Compartilhar bases de dados viola alguns dos princípios da arquitetura baseada em microserviços. Por exemplo, o contexto deixa de ser limitado, devido aos dois serviços compartilharem a base de dados, eles precisam se conhecer, e as mudanças realizadas no banco de dados devem ser controladas entre os dois serviços.

No geral, o nível de compartilhamento entre microserviços deve ser o mais limitado possível para que o baixo acoplamento ocorra.

2.4.2.1.4 Detectabilidade

Para a IBM (2015), como descrito anteriormente, a arquitetura em microserviços requer que os serviços construídos sejam confiáveis e tolerantes às falhas. Isso implica a construção de microserviços de maneira que a infraestrutura subjacente seja criada e destruída, garantindo que os serviços possam ser reconfigurados de acordo com a localização dos outros serviços que precisam se conectar a ele.

Com o uso da nuvem e contêineres para efetuar o *deploy* dos microserviços, estes serviços precisam ser reconfigurados de maneira dinâmica. Quando uma nova instância do serviço é criada, o resto da rede deve poder rapidamente encontrá-lo e começar a se comunicar com ele.

A maioria dos modelos de descoberta de serviços conta com um serviço de registro que permite que todos os serviços explicitamente se registrem e efetuem chamadas para ele, garantindo que os serviços possam se comunicar a partir deste modelo de descoberta de serviços. Existem diversos serviços de registro que possuem código aberto e que fornecem a possibilidade de descoberta de serviços, como por exemplo, Zookeeper, Consul e Eureka.

É importante salientar que nem todos os modelos de descoberta de serviços dependem de um serviço de registro. Por exemplo, *Cloud* Foundry, é uma plataforma *open source* que funciona como serviço (PaaS), cujo Bluemix – proposta de nuvem mais recente da IBM – é construído sobre e não depende de um serviço de registro. Serviços de *Cloud Foundry* são anunciados para o *Cloud Controller*, para torna-lo consciente do serviço e sua disponibilidade.

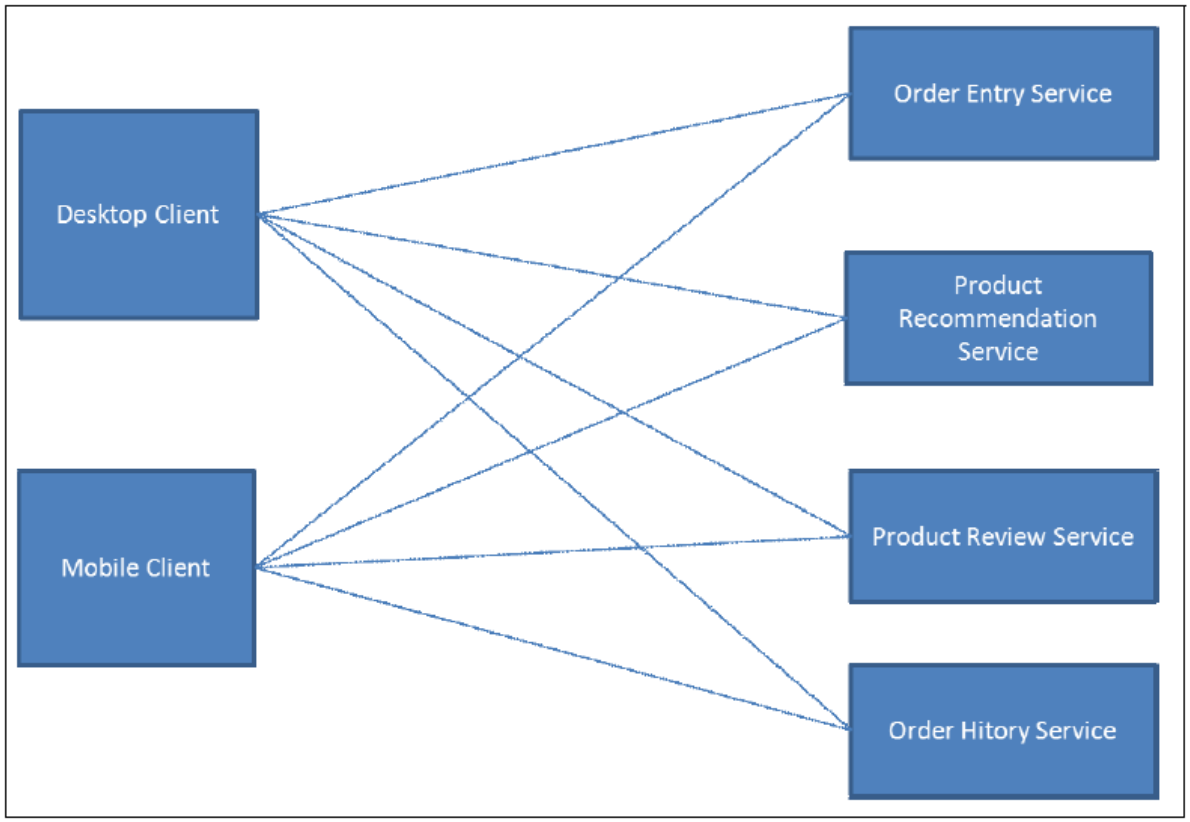
Bons serviços de descoberta são uma fonte rica de metadados, cujo registro de serviços pode prover e pode então ser usado na requisição de serviços para tomar decisões de como usar um serviço em específico. Por exemplo, um serviço requerente (*client*) pode perguntar se um serviço possui dependências especificas, suas implicações e se este é capaz de cumprir os requisitos específicos necessários. Alguns serviços de descoberta também possuem a capacidade de balanceamento de carga.

2.4.2.1.5 Design de comunicação entre serviços

Segundo a IBM (2015) e Newman (2015), quando os microserviços estão espalhados através de todo ambiente de implantação (servidores, maquinas virtuais (VMs) ou contêineres), como a comunicação é feita entre os microserviços? Para comunicações unidirecionais, filas de mensagens são o suficiente, mas esta implementação não se aplica para comunicações síncronas e bidirecionais.

Em aplicações monolíticas, clientes da aplicação, como navegares e aplicações nativas, fazem do *Hypertext Transfer Protocol* (HTTP) um balanceador de carga, que espalha os pedidos para várias instâncias idênticas do aplicativo. Mas na arquitetura de microserviços, o monólito foi substituído por uma coleção de serviços. A figura 8 apresenta um cenário onde os clientes do sistema podem fazer chamadas RESTful para as *interfaces* de programação da aplicação (API).

Figura 8 - Chamadas RESTful entre clientes e serviços.

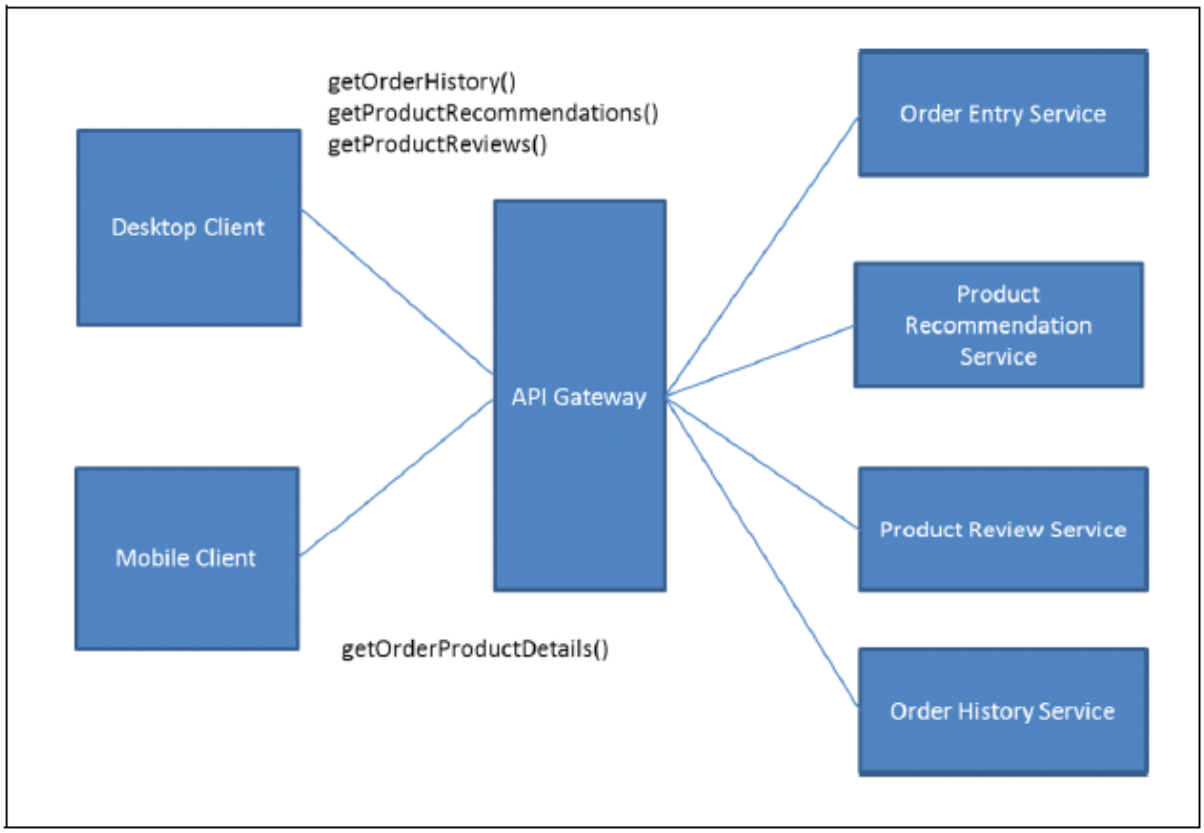


Fonte: IBM (2015, p. 25)

Na superfície, isto pode ser desejável. No entanto é provável que dependendo da granularidade dos serviços haja incompatibilidade nesta comunicação. Alguns clientes podem ser mais “faladores”, outros podem exigir muitas chamadas na obtenção dos dados necessários. Devido os serviços escalarem de maneira diferente, há outros desafios a enfrentar, como a manipulação de falhas parciais.

Uma melhor abordagem para os clientes efetuarem algumas conexões, talvez apenas uma, é a utilização de um front-end como uma API Gateway, como é demonstrado na figura 9.

Figura 9 - Utilizando um API Gateway para efetuar as chamadas dos microserviços.



Fonte: IBM (2015, p. 25)

O API *Gateway* encontra-se entre os clientes e os microserviços, e provê APIs que são adaptados para os clientes. O *Gateway* serve principalmente para ocultar a complexidade tecnológica (por exemplo, a conectividade a um mainframe) versus a complexidade da *interface* disponibilizada. Informalmente, um API *Gateway* é um *facade*. Entretanto, o padrão *facade* provê uma visão uniforme da complexidade interna para os clientes externos, onde uma API *Gateway* fornece uma visão uniforma dos recursos externo para os internos da aplicação.

Muito parecido com o padrão de projeto *facade*, o *Gateway* API fornece uma *interface* simplifica para os clientes, tornando os serviços mais fáceis de usar, entender e testar. Isto é porque o *Gateway* pode fornecer diferentes níveis de granularidade para clientes *desktop* e *web*. A API *Gateway* deve fornecer APIs de menor granularidade para clientes *mobile* e APIs de maior granularidade para clientes *desktop* que podem usar uma rede de alto desempenho.

Neste cenário, a API Gateway reduz o diálogo ao permitir que clientes reduzam múltiplas requisições em uma única requisição otimizada para um determinado cliente (como um cliente *mobile*). A vantagem é que o dispositivo que experimenta as consequências da latência da rede apenas uma vez, e se aproveita da conectividade de baixa latência e do *hardware* mais poderoso do servidor.

2.4.2.1.6 Lidando com a complexidade

Segundo a IBM (2015), uma arquitetura baseada em microserviços gera utilização de recursos adicionais para muitas operações. Isto é claramente um grande aumento das partes transitórias, assim como, a complexidade. Racionalizar o comportamento de cada componente individual, pode ser simples, mas o entendimento do comportamento do sistema como um todo é algo muito mais difícil.

Ajustar um sistema baseado em microserviços, que pode ser composto por diversos processos, balanceadores de carga e camadas de mensagens, requer monitoramento e operações de infraestrutura de alta qualidade. Promover a profusão de microserviços através da linha de desenvolvimento para a produção, requer um auto grau de automação na liberação de novas versões e na execução de *deploys*.

2.4.2.1.7 Design evolucionário

Para Stenberg (2015) e IBM (2015), a arquitetura baseada em microserviços permite a abordagem de *design* evolucionário. É possível introduzir novos recursos e capacidades à ideia central da aplicação, alterando apenas um microserviço. É necessário efetuar o *deploy* e liberar uma nova versão do microserviço que conterá a mudança. A mudança afeta apenas quem consome este microserviço e, se a *interface* não mudar, todos os consumidores do serviço continuaram funcionais.

Devido às características de baixo acoplamento, pequeno e focado, além da limitação do contexto dos microserviços, alterações podem ser feitas rapidamente e frequentemente com pequenos riscos à integridade da aplicação. Essa facilidade na atualização dos microserviços permite a evolução destes.

Pelo fato de um microserviços ser pequeno e geralmente projetado, desenvolvido e mantido por uma pequena equipe, é comum encontrar microserviços que ao invés de serem melhorados e mantidos, acabam sendo reescritos por completo. Uma compensação comum, está em quão pequeno o microserviço é. Quanto mais microserviços existem na decomposição de uma aplicação monolítica, menor é cada um deles. E quanto menor cada um deles é, o risco no *deploy* e na liberação de versão também diminui.

Entretanto, quanto menor um microserviços é, a necessidade de atualização que a aplicação apresenta para o microserviço é inversamente proporcional, sendo assim, o risco aumenta. Os princípios de um microserviço, como *deploy* independente e baixo acoplamento, precisa ser considerado quando determinada funcionalidade se torna um microserviço.

2.4.3 Barramento *web* baseado em REST

REST significa *Representational State Transfer* (ou Transferencia de Estado Representativo, em tradução livre), e segundo Saudate (2014, p.4),

> […] é um estilo de desenvolvimento de *web services* que teve origem na tese de doutorado de Roy Fielding. Este, por sua vez, é co-autor de um dos protocolos mais utilizados no mundo, o HTTP (HyperText Transfer Protocol). Assim, e notável que o protocolo REST e guiado (dentre outros preceitos) pelo que seriam as boas práticas de uso de HTTP:
>
> - Uso adequado dos métodos HTTP;
> - Uso adequado de URL's;
> - Uso de códigos de status padronizados para representação de sucessos ou falhas;
> - Uso adequado de cabeçalhos HTTP;
> - Interligações entre vários recursos diferentes.

Para a IBM (2015), REST é um estilo arquitetural para aplicações em rede, inicialmente usado para construir *web services* mais leves, de fácil manutenção e escaláveis. Um serviço baseado em REST é chamado de serviço RESTful. REST não depende de nenhum protocolo, mas quase todo serviço RESTful utiliza HTTP como protocolo subjacente.

REST é muitas vezes utilizado por aplicações *web* como forma de permitir que recursos se comuniquem a partir de troca de informações. Se for considerado a *web* como uma plataforma para aplicações, REST permite que estas aplicações possuam baixo acoplamento, podendo haver escalabilidade e provendo funcionalidades através de serviços.

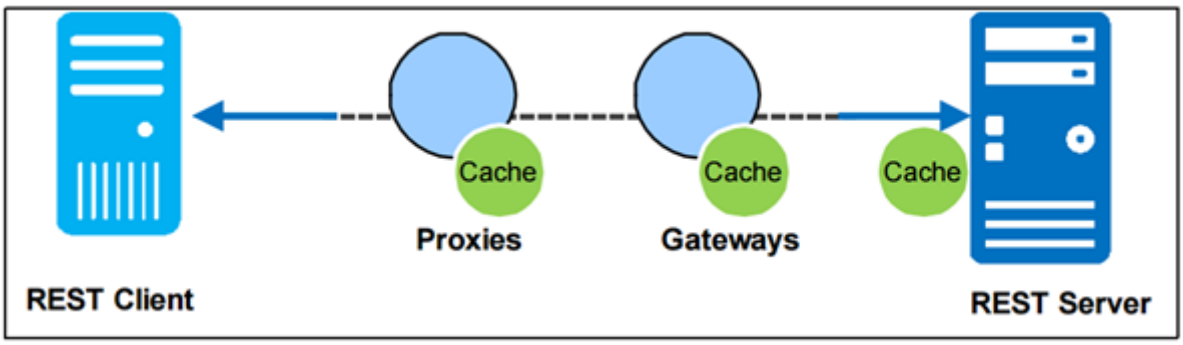
Rosa (2000 apud FIELDING, 2015), o autor defende a utilização do REST para interligar sistemas essencialmente heterogêneos, sendo possível realizar a troca de mensagens e informações mantendo a semântica dos dados entre os dispositivos envolvidos, e ainda garantir a segurança, integridade e consistência nos dados distribuídos.

Segundo a IBM (2015), na última década, REST surgiu como um modelo para projetar *web services* de forma predominante, e atualmente quase todas as principais linguagens de desenvolvimento possuem estruturas para construção *web services* RESTful. As APIs REST utilizam verbos em HTTP para executar ações em cima dos recursos. A API estabelece um mapeamento entre as operações CREATE, READ, UPDATE e DELETE e o POST corresponde com as ações HTTP GET, PUT e DELETE.

A *interface* RESTful foca em funções e recursos dos componentes, ignorando os detalhes da implementação interna. Requisições são feitas ao servidor, que tem conhecimento dos componentes, e que esconde os detalhes dos usuários. As interações devem ser *stateless*, devido às requisições poderem navegar entre as camadas intermediárias da topologia cliente servidor.

Estas camadas intermediárias podem ser *proxies* ou *gateways* e as vezes informações cacheadas. Uma restrição do REST é que ele necessita de comunicação *stateless*. Toda requisição deve conter toda informação necessária para a interpretação desta. Isso aumenta a visibilidade, confiabilidade e escalabilidade, porém também diminui a performance devido às mensagens *stateless* serem maiores.

Figura 10 – Arquitetura REST com caches intermediários



IBM (2015, p. 33)

Como visto na figura 10, entre um REST *Client* e um REST *Server* pode-se ter estruturas de proxies e gateways que podem cachear informações da comunicação, agilizando o processo de execução e resposta das chamadas REST.

2.4.4 Benefícios da utilização de microserviços

Muitas organizações estudaram, aplicaram e concluíram que a arquitetura de microserviços é uma abordagem superior a arquitetura monolítica. Porém, muitas outras, têm encontrado diversas dificuldades com essa nova arquitetura, dificuldades essas que tem prejudicado a produtividade de seus desenvolvedores. Para cada tipo de projeto, cada contexto específico a escolha do tipo de arquitetura deve ser bem planejada, sendo assim, entendendo os impactos de cada arquitetura e com sensatez escolher a mais aplicável no seu projeto

“Como qualquer estilo arquitetônico, microserviços proporcionam benefícios, mas eles vêm com custos. ” (FOWLER, 2015A, tradução nossa).

Os benefícios dos microserviços são vários e variados. Muitos desses benefícios são conhecidos e vêm de brinde pelo fato de ser uma forma de produzir sistemas distribuídos. No entanto, microserviços procuram alcançar esses benefícios de maneira ainda maior, pois, os conceitos de sistemas distribuídos e a arquitetura orientada a serviços são fortemente defendidos.

2.4.4.1 Modularização

Fowler (2015A) enxerga a definição de limites de cada módulo como a primeira grande vantagem de microserviços. Este é um benefício importante para novos desenvolvedores no projeto, pois em um microserviço os limites de cada módulo devem ser mais fortemente definidos do que um monólito. Mas, o que é limite entre módulos? A maioria das pessoas envolvidas no desenvolvimento de *software* concorda que é uma boa prática dividir *software* em módulos: pedaços de *software* desacoplados um do outro.

O desejo de todo arquiteto de *softwares* é que os módulos de sua aplicação trabalhem eficazmente para que se for preciso alterar apenas uma parte do sistema, na maioria das vezes seja preciso entender somente essa pequena parte do sistema para fazer a mudança, e que seja possível encontrar essa pequena parte com bastante facilidade. Para Newman (2015) boa estrutura modular é útil em qualquer programa, mas torna-se cada vez mais importante à medida em que o *software* cresce. Em outra abordagem, boa estrutura modular torna-se cada vez mais importante à medida em que a equipe de desenvolvimento aumenta.

É sabido que a estrutura de um sistema de *software* espelha a estrutura de comunicação da organização que construiu. Com o uso de microserviços é possível que cada equipe cuide de unidades independentes, evitando assim as comunicações entre diferentes equipes, fato que é bem frequente em organizações cujo *software* tem uma estrutura modular ruim. Segundo Fowler (2015A) um aspecto importante deste acoplamento é a persistência dos dados. Uma das principais características de microserviços é o gerenciamento de dados descentralizado. Cada serviço deve gerar seu próprio banco de dados e qualquer outro serviço deve passar por API do serviço para chegar a ele. Isso elimina qualquer tipo de banco de

dados integrado, que é uma prática desagradável bem comum de acoplamento em sistemas maiores.

Para tal modularização, é necessário ter disciplina, pois, quanto maior a equipe fica, mais difícil se torna manter esses limites. Esta vantagem pode se tornar uma desvantagem se você não manter os seus limites corretamente. Para que isso não ocorra o domínio deve ser bem compreendido. O que se pode concluir é que existem muitos casos de projetos que estão utilizando microserviços e, pelo fato de estarem usando, a manutenção nos módulos está significativamente mais fácil.

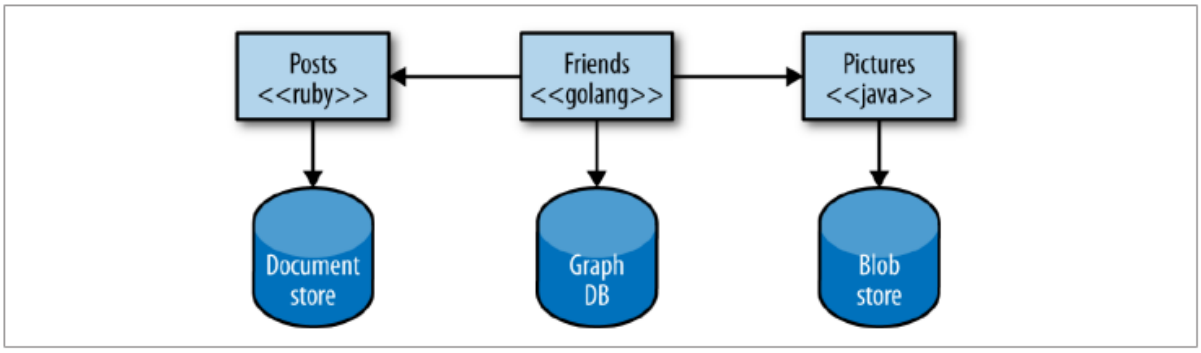
2.4.4.2 Desenvolvimento poliglota

Segundo Contino (2015), Gupta (2015), Newman (2015) e Richardson (2014A) com microserviços é possível misturar várias linguagens, estruturas de desenvolvimento e tecnologias de armazenamento de dados.

Uma vez que um sistema é composto de vários serviços independentes, os desenvolvedores têm liberdade para escolher com qual tecnologia querem trabalhar dentro de cada um. Microserviços podem ser escritos em linguagens diferentes, usam diferentes bibliotecas, e também diferentes armazenamentos de dados. Isso permite escolher uma ferramenta apropriada para cada serviço, usando a linguagem e bibliotecas mais adequadas para determinados tipos de problemas. Também é possível reescrever um serviço do zero usando melhores linguagens e tecnologias.

Segundo Newman (2015), se uma parte do sistema precisa melhorar o seu desempenho, pode-se decidir usar uma tecnologia diferente que é capaz de alcançar os níveis de desempenho exigidos. Também é possível decidir como armazenar os dados em diferentes partes de nosso sistema. Por exemplo, para uma rede social, podemos armazenar a interação dos usuários em um banco de dados orientado a grafos, mas talvez as mensagens que os usuários trocam entre si poderia ser armazenada em um banco de dados orientado a documentos, dando origem para uma arquitetura heterogênea como a que mostra a Figura 11.

Figura 11 - Microserviços permite que você adote facilmente diferentes tecnologias.



Fonte: Newman (2015, p. 4)

Com microserviços, somos capazes de adotar novas tecnologias mais rapidamente, e compreender como os novos avanços podem ajudar. Uma das maiores barreiras para experimentar e adotar novas tecnologia é o risco associado a mudança. Com uma aplicação monolítica, se houver uma tentativa de mudança de banco de dados ou linguagem de programação, ou *framework*, qualquer uma dessas terá impacto sobre grande parte do sistema. Com um sistema constituído por vários serviços, tem-se vários novos lugares para experimentar uma nova tecnologia. Fazer uma prova de conceito torna-se muito mais fácil, basta, pegar um serviço menor e testar a nova tecnologia nele, limitando qualquer impacto negativo. Muitas organizações consideram essa capacidade de absorver rapidamente novas tecnologias uma vantagem real. Portanto, não se deve subestimar a experimentação que os microserviços proporcionam. Em um sistema monolítico, as decisões iniciais sobre linguagens e *frameworks* são difíceis de serem revertidas.

2.4.4.3 Resiliência

Segundo Gupta (2015) um conceito fundamental na engenharia de resiliência é a distribuição. Caso um componente de um sistema falhar, que falhe, mas não em cascata. Com microserviços é possível isolar o problema, fazendo com que o resto do sistema continue a funcionar normalmente. Limites de memória ou processamento de certo serviço torna uma distribuição óbvias. No entanto, é preciso ter cuidado para garantir que os serviços são capazes de realizar corretamente esta acrescida resistência. Para tal, é necessário entender as

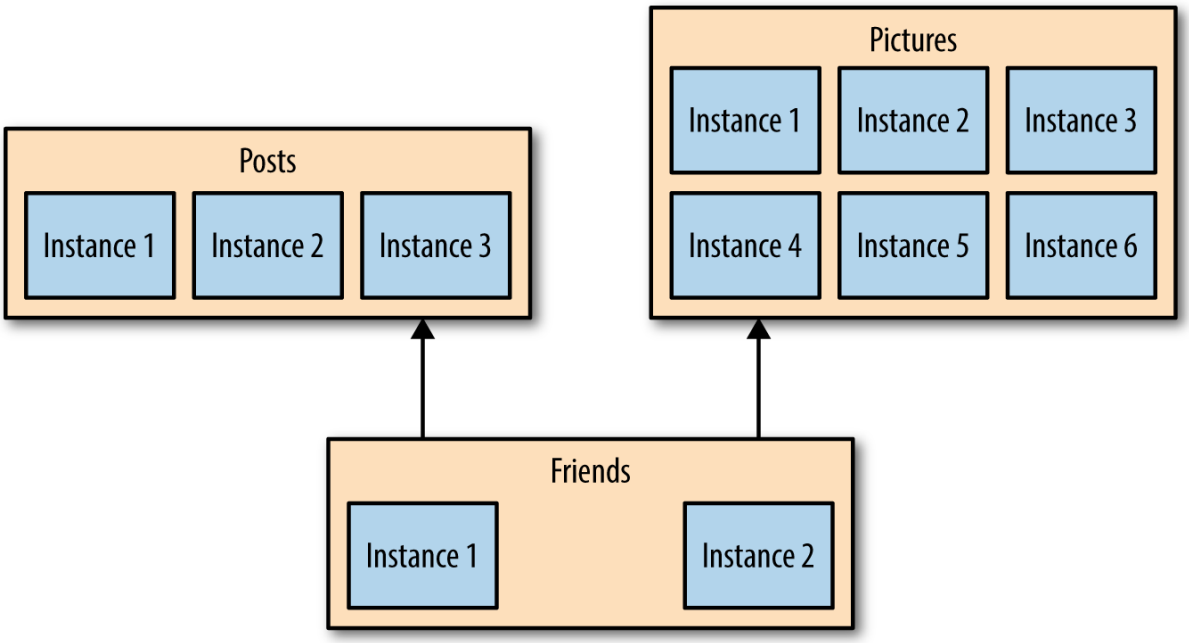
fontes de fracasso que os sistemas distribuídos possuem, e lidar com elas. A rede, por exemplo, pode e vai falhar em certo momento. Por isso é necessário saber como lidar com essas situações, tornando o impacto (se houver) o menor possível para o usuário final do *software*.

Se uma aplicação monolítica quebrar, em seguida, uma série de funcionalidades param de funcionar. No lado oposto, se um serviço quebra - apenas a pequena funcionalidade em particular parará de funcionar. É muito mais simples de construir alguma resistência em torno de serviços pequenos. Por exemplo, as falhas do sistema de processamento de pedidos podem ser mitigadas com filas de mensagens à serem revertidas.

2.4.4.4 Escalabilidade

Defensores de microserviços costumam dizer que os serviços são mais fáceis de escalar, uma vez que se um serviço recebe uma grande quantidade de carga que você pode escalar apenas este serviço, ao invés de toda uma aplicação. Segundo Newman (2015) e Richardson (2014A) microserviços estão bem adaptados a nuvem, onde pode-se escalar conforme a carga dos aplicativos. Em uma arquitetura monolítica, a tendência é dimensionar o monólito horizontalmente, adicionando novos nós. Os microserviços são muito mais flexíveis, e é possível escalar cada serviço independentemente clonando o serviço e aumentando o processamento da aplicação ou sua memória, com base em suas necessidades como na figura 12.

Figura 12 - É possível escalar apenas aqueles serviços que necessitam.



Fonte: Newman (2015, p. 6)

Se a aplicação estiver hospedada em um servidor na nuvem com possibilidade de configuração para escalabilidade automática pode-se aplicar esse escalonamento conforme demanda. Isto permite controlar os custos de forma mais eficaz. Não é sempre que uma abordagem de arquitetura pode ser tão estreitamente correlacionada com uma economia de custo.

2.4.4.5 Facilidade no *deploy*

Segundo Fowler (2015A) e Gupta (2015) a prática de entrega contínua teve um efeito profundo sobre a indústria de *software*, e esta pratica está profundamente entrelaçada com o movimento de microserviços. Vários esforços do movimento de microserviços foram provocados pela dificuldade de implantação de grandes monólitos, onde uma pequena mudança, que fosse de uma linha de código em uma aplicação monolítica poderia causar com que toda a implantação a falhasse.

Um princípio fundamental de microserviços é que os serviços são componentes e, portanto, são implantados independentemente. Com microserviços, caso você altere apenas um serviço, não vai derrubar toda a aplicação, pois cada serviço é implantado independente de outros serviços. Isto permite o implantar código novo mais rápido. Qualquer alteração local em um serviço pode ser facilmente feita por um desenvolvedor sem a necessidade de interação com outras equipes, tornando a entrega de novas funcionalidades aos clientes mais rápida. Como resultado, tem-se a agilidade de microserviços, sendo assim, um grande facilitador de entrega contínua e integração contínua. Esta relação é uma via de mão dupla.

Com muitos serviços a necessidade de implantar é frequente, portanto, é essencial a prática da entrega contínua. Para Avram (2015) e Contino (2015) a grande vantagem da entrega contínua é a redução do tempo entre uma ideia e *software* em execução. Organizações que fazem isso podem responder rapidamente as mudanças do mercado, introduzindo novas funcionalidades no *software* antes dos seus concorrentes.

Embora muitas pessoas citam a entrega contínua como uma razão para usar microserviços, é essencial mencionar que mesmo grandes monólitos podem ser entregues de forma contínua também.

2.4.4.6 Composição

Microserviços devem oferecer uma *interface* que é uniforme e projetada para suportar a sua composição. As *interfaces* dos serviços devem ser projetadas de forma que seja fácil sua identificação, devendo representar, a manipulação dos recursos recebidos, descrevendo o contexto da API e suas operações. As interações com os demais sistemas devem ser feitas via essas *interfaces*, sem a necessidade de utilização de *views* ou tabelas, apenas serviços. Este é um princípio de bom *design* a nível de código e de classe, mas também a nível sistêmico, pois faz com que os componentes do sistema, desde os mais simples aos mais complexos se comuniquem através dessas *interfaces* bem definidas.

2.4.4.7 Alinhamento organizacional

Muito dos problemas causados no desenvolvimento de *software* estão associados com as grandes equipes e grandes bases de código. Estes problemas podem ser agravados quando a equipe é distribuída em locais físicos distantes. Sabe-se também que as equipes menores, são uma tendência, pois ao trabalhar em bases de código menores tendem a ser mais produtivas.

Segundo Newman (2015) microserviços permite alinhar a arquitetura sistêmica com a organização, minimizando o número de pessoas que trabalham em qualquer projeto apenas para bater o ponto. Alterando a estrutura da organização para que cada serviço tenha o número de pessoas específico para ser mantido. Praticando assim a lei de Conway: organizações que desenvolvem *software* tendem a produzir sistemas que são cópias da estrutura organizacional.

2.4.4.8 Serviços otimizados e substituíveis

É comum que os desenvolvedores que trabalham em organizações de tamanho médio ou grande, desenvolvam para grandes sistemas legados, o que torna o trabalho um tanto quanto desagradável, pois, ninguém quer fazer grandes alterações no código. Porém, estas alterações que os desenvolvedores fogem de fazer são ponto vital para a organização manter seus clientes e continuar funcionando. Mas então, por que essa aplicação não é substituída? Sabe-se o porquê: é uma aplicação muito grande e qualquer alteração errada que quebre em produção coloca em risco seu emprego.

Segundo Gupta (2015) o uso de serviços pequenos e bem modularizados, o custo de substituição para uma tecnologia mais adequada, ou a definição de desuso do mesmo, se torna muito mais fácil. Ao excluir um número grande de linhas de código em um único dia de uma aplicação monolítica a preocupação é grande. Porém, com microserviços, pelo fato de a

maioria das vezes eles terem tamanho similar, as barreiras para reescrever ou remover serviços são poucas.

Segundo Newman (2015) as equipes que usam abordagens de microserviços são confiantes para reescrever completamente os seus serviços quando necessário, ou colocar em desuso um serviço quando o mesmo não for mais necessário. Quando temos um serviço pequeno com poucas linhas de código, torna-se difícil um apego pelo *software*, ou seja, nenhum desenvolvedor vai colocar empecilhos caso seja definido remover o serviço da aplicação, colocando-o em desuso.

2.4.5 Microserviços como componente

Conceitualmente, um componente é uma unidade de *software* independente que pode ser representada como substituível e atualizável, que pode ser utilizado com outros componentes para formar um sistema complexo. Segundo Fowler e Lewis (2014) serviços podem ser classificados como componentes porque estão fora do processo principal e as comunicações entre eles são realizadas utilizando chamadas de procedimento remoto, ao contrário das bibliotecas que são acopladas no processo principal.

Arquitetura de microserviços também utilizam bibliotecas, mas a sua principal forma de construir componentes é "quebrando" o seu próprio *software* em serviços. Para criação de cada serviço é necessário ter um planejamento prévio para se projetar e construir as *interfaces* do serviço em desenvolvimento. Uma das principais razões para a utilização de serviços como componente, é que os serviços são independentemente implementáveis. Sendo assim, toda a estrutura como suas dependências estão presentes dentro do próprio componente, isso garante o seu isolamento e diminui a quantidade de chamadas remotas para outros componentes, pois quase tudo que o serviço precisa está contido nele e caso os componentes precisem se comunicar, se comunicam através de chamadas remotas, comumente implementados em APIs.

Segundo Namiot e Sneppe (2014) o domínio tem que estar bem definido, diminuindo o acoplamento e garantido o isolamento do componente. O isolamento físico é

uma das principais forças de microserviços, e a chave para um serviço escalável com alta disponibilidade. Namiot e Sneppe também salientam que a componentização dos microserviços, deve utilizar as seguintes premissas para o desenvolvimento preciso de sistemas baseados em microserviços:

• Chamadas/Respostas devem utilizar dados arbitrariamente estruturados.

• Eventos assíncronos devem fluir em tempo real e em ambas direções.

• Pedidos e respostas podem fluir em qualquer direção.

• Pedidos e respostas podem ser arbitrariamente aninhados.

• O formato de serialização de mensagem deve ser conectável (JSON, XML).

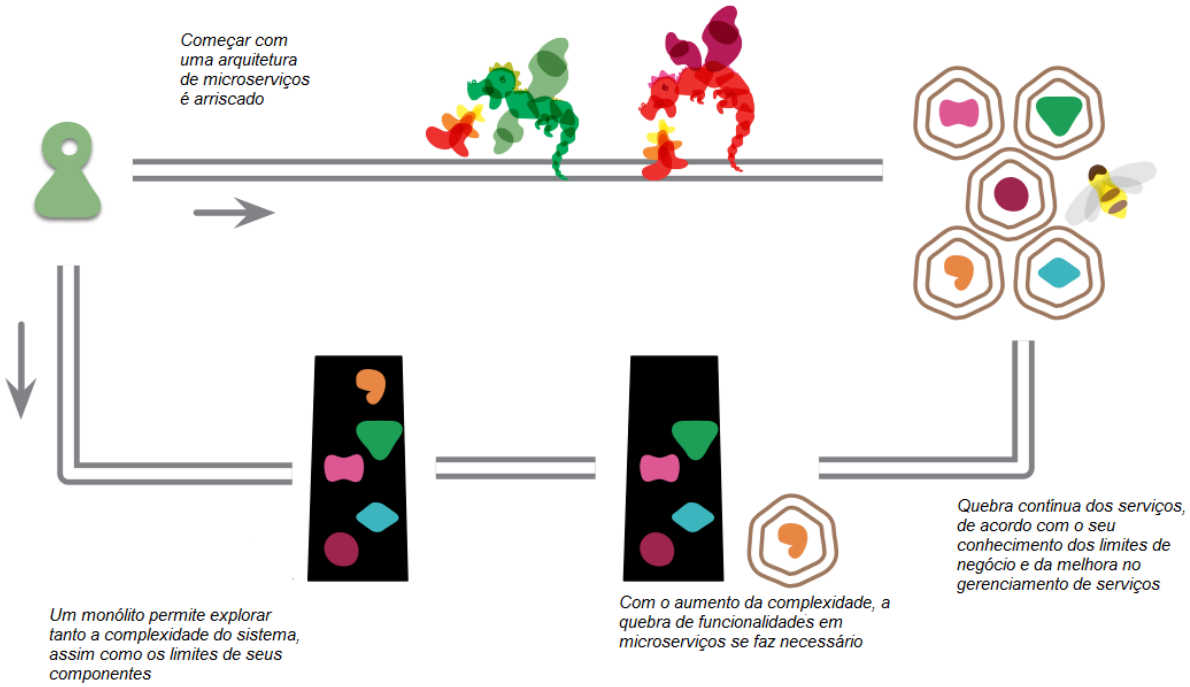
Ao seguir essas premissas o serviço implementado obedece ao conceito que todo serviço pode ser chamado por qualquer cliente, visto que, quem define o contexto da aplicação e as *interfaces* que serão disponibilizados para seus consumidores, é ele mesmo.

2.4.6 Alteração para microserviços

A maioria das empresas de *software web* de grande escala têm evoluído sua arquitetura de sistema a partir de uma aplicação e banco de dados monolítico a um conjunto de microserviços de baixo acoplamento. Netflix, Google, eBay e outros sites de grande escala são exemplos desta mudança de arquitetura, e, os mesmos sugerem que outras organizações devem considerar a migração para microserviços.

Segundo Fowler (2015B) a maioria das histórias de sucesso sobre equipes que usam uma arquitetura de microserviços, começaram com uma arquitetura monolítica na qual a aplicação ficou grande demais e foi quebrada, gerando vários serviços. Os casos em que foi construído um sistema a partir do zero com uma estrutura de microserviços, acabaram em sérios apuros. Defende-se que não se deve começar um novo projeto com microserviços, mesmo que você tenha certeza que sua aplicação vai ser grande o suficiente para valer a pena optar por esse tipo de arquitetura, este tipo de dificuldade pode ser visualizado na Figura 13.

Figura 13 – Começar com microserviços X quebrar monólito em microserviços.



Fonte: Fowler (2015B)

Como pode-se ver na Figura 13, microserviços é uma arquitetura útil, porém só é útil quando o sistema toma determinadas proporções e se torna mais complexo. Ou seja, cria-se uma estratégia onde inicialmente deve-se construir um sistema em forma de monólito, mesmo que seja provável que a arquitetura de microserviços seja a mais adequada posteriormente. Portanto, após a virada de chave para uma mudança arquitetônica, de monolítica para microserviços, o primeiro desafio é extrair os recursos do sistema monólito antigo. O conceito de contexto limitado chamado *Domain Driven Design* (DDD) deve ser atendido para definir os novos componentes de serviços, pois, dessa forma evita refatorações gigantes e garante que cada novo componente extraído do monólito se torne um novo serviço que pode e deve trabalhar normalmente com o sistema monólito antigo, até que todo o monólito se torne componentes de serviço.

Segundo Namiot e Sneppe (2014) a abordagem de microserviço possui vários desafios em sua implementação, além da complexidade de se administrar vários sistemas distribuídos, os testes são mais difíceis para essas aplicações. Outro desafio, é que essa abordagem também leva ao aumento do consumo de memória, devido ao próprio espaço de endereço para a cada serviço.

Um dos maiores desafios é quebrar o monólito e definir como dividir o sistema em serviços. Segundo Newman (2015), obter o domínio errado de um serviço pode ser caro, e componentes excessivamente acoplados podem ser piores do que ter um único sistema monolítico.

Segundo Fowler (2015B) não se deve começar um novo sistema com microserviços a menos que você tenha experiência razoável em construir sistemas com este tipo de arquitetura. A abordagem mais comum é começar com um sistema monólito e criar serviços gradualmente através dos módulos deste monólito, mas, para que esta seja uma tarefa fácil é preciso ter muita disciplina ao construir seu monólito, construindo-o de forma modular. Portanto, não tenha medo de construir um monólito que você pode vir a deixar de usar.

3 MÉTODO

Para chegar a proposta de arquitetura de microserviços para reaproveitamento de módulos de soluções monolíticas que é apresentada neste trabalho, foi preciso definir um caminho a ser traçado para alcançar o resultado esperado, este que foi estipulado pelo método. “Método” deriva do latim *methodus*, que significa “caminho”; a palavra, no entanto, tem origens gregas: *meta* (através, por meio de) *hodos* (caminho), donde *methodos*.

Segundo Ferrari (1982, p.19), pode-se definir método “como procedimento racional arbitrário de como atingir determinados resultados [...]. ”.

Já para Gil (2008, p.8) “pode-se definir método como caminho para se chegar a determinado fim”.

Em concordância com as definições citadas anteriormente, para Eva Maria Lakatos e Marina Marconi (1986, p.81), método, é o "conjunto das atividades sistemáticas e racionais que, com maior segurança e economia, permite alcançar o objetivo [proposto], traçando o caminho a ser seguido, detectando os erros e analisando as decisões do cientista”.

Com isso, nesta seção é apresentado os caminhos escolhidos pelo método a fim de executar os objetivos do trabalho, em sequencias esclarecer os tipos de pesquisa, as etapas metodológicas, a proposta e suas delimitações.

3.1 TIPOS DE PESQUISA ADOTADOS

Antes de caracterizar esta pesquisa, definindo os tipos de pesquisa foi necessário definirmos primeiramente o que é uma pesquisa. Segundo Gil (2008, p.26), a pesquisa tem um caráter pragmático, é um “processo formal e sistemático de desenvolvimento do método científico. O objetivo fundamental da pesquisa é descobrir respostas para problemas mediante o emprego de procedimentos científicos”. Já Demo (1996, p.34) insere a pesquisa como atividade cotidiana considerando-a como uma atitude, um “questionamento sistemático crítico

e criativo, mais a intervenção competente na realidade, ou o diálogo crítico permanente com a realidade em sentido teórico e prático".

A palavra pesquisa deriva do termo em latim *perquirere*, que significa "procurar com perseverança". Pesquisa é um conjunto de ações que visam a descoberta de novos conhecimentos em uma determinada área. A pesquisa é realizada quando se tem um problema e não se tem informações para solucioná-lo.

Com esta breve explicação de pesquisa, pode-se então agora caracterizar este trabalho. Quanto à natureza, este projeto se classifica como aplicado; quanto a abordagem, é classificada como qualitativo; quanto aos objetivos, se classifica como exploratório; e quanto aos procedimentos, como bibliográfico.

3.1.1 Pesquisa aplicada

Boaventura (2004) classifica a pesquisa aplicada como o tipo de pesquisa que busca aumentar o conhecimento científico através da descoberta de leis e efeitos.

Já segundo Gil a pesquisa **aplicada**

> Apresenta muitos pontos de contato com a pesquisa pura, pois depende de suas descobertas e se enriquece com o seu desenvolvimento; todavia, tem como característica fundamental o interesse na aplicação, utilização e consequências práticas dos conhecimentos. Sua preocupação está menos voltada para o desenvolvimento de teorias de valor universal que para a aplicação imediata numa realidade circunstancial. De modo geral é este o tipo de pesquisa a que mais se dedicam os psicólogos, sociólogos, economistas, assistentes sociais e outros pesquisadores sociais. (GIL, 2008, p. 27).

Nesse sentido, esta pesquisa pode ser considerada aplicada, pois objetiva a geração de conhecimento através do estudo de bibliografia específica, para aplicação prática com intuito de simular as diversas formas de aplicar a estrutura de microserviços no desenvolvimento de *software*.

3.1.2 Pesquisa qualitativa

Quanto à pesquisa **qualitativa** Richardson (1999) informa que esta tem aumentado sua credibilidade nas ciências sociais. "Essa legitimidade, porém, foi comprada ao preço de incorporar critérios positivistas de validade e generalização.". A pesquisa qualitativa busca explicar o porquê das coisas, exprimindo o que convém ser feito, não se preocupando com representatividade numérica, mas, sim, com o aprofundamento da compreensão de um grupo social, de uma organização, etc. Para Deslauriers (1991, p. 58) nesse tipo de pesquisa "o conhecimento do pesquisador é parcial e limitado. O objetivo da amostra é de produzir informações aprofundadas e ilustrativas: seja ela pequena ou grande, o que importa é que ela seja capaz de produzir novas informações.". Nesse sentido, esta pesquisa deve ser considerada qualitativa, pois foi realizada uma reflexão pela perspectiva dos avaliadores acerca da arquitetura de microserviços.

3.1.3 Pesquisa exploratória

Quanto à pesquisa **exploratória** Gil (2008, p.27) afirma que

> As pesquisas exploratórias têm como principal finalidade desenvolver, esclarecer e modificar conceitos e ideias, tendo em vista a formulação de problemas mais precisos ou hipóteses pesquisáveis para estudos posteriores. De todos os tipos de pesquisa, estas são as que apresentam menor rigidez no planejamento. Habitualmente envolvem levantamento bibliográfico e documental, entrevistas não padronizadas e estudos de caso. Procedimentos de amostragem e técnicas quantitativas de coleta de dados não são costumeiramente aplicados nestas pesquisas.

Este tipo de pesquisa geralmente é realizado nas áreas em que há pouco conhecimento acumulado e sistematizado. Por sua natureza de sondagem, não comporta hipóteses que, todavia, poderão surgir durante ou ao final da pesquisa. "Este tipo de pesquisa é realizado especialmente quando o tema escolhido é pouco explorado e torna-se difícil sobre ele formular hipóteses precisas e operacionalizáveis." (GIL, 2008, p. 27).

Nesse sentido, pode-se considerar esta pesquisa como exploratória pelo fato de que foi realizado inúmeros levantamento bibliográficos a respeito de microserviços, algumas entrevistas com pessoas que tiveram experiências práticas com o problema pesquisado e também analise de exemplos que ajudaram a compreender a arquitetura de microserviços.

3.1.4 Pesquisa bibliográfica

Quanto à pesquisa **bibliográfica** Fonseca (2002, p. 32) afirma que

> A pesquisa bibliográfica é feita a partir do levantamento de referências teóricas "já analisadas, e publicadas por meios escritos e eletrônicos, como livros, artigos científicos, página de *web* sites" (Matos e Lerche: 40) sobre o tema a estudar. Qualquer trabalho científico inicia-se com uma pesquisa bibliográfica, que permite ao pesquisador conhecer o que já se estudou sobre o assunto. Existem, porém, pesquisas científicas que se baseiam unicamente na pesquisa bibliográfica, procurando referências teóricas publicadas com o objetivo de recolher informações ou conhecimentos prévios sobre o problema a respeito do qual se procura a resposta. As conclusões não podem ser apenas um resumo. O pesquisador tem de ter o cuidado de selecionar e analisar cuidadosamente os documentos a pesquisar de modo a evitar comprometer a qualidade da pesquisa com erros resultantes de dados coletados ou processados de forma equívoca.

Ou seja, esse tipo de pesquisa busca a solução de um problema a partir de conhecimentos teóricos, que vão contribuir para um determinado assunto, e, em concordância com Fonseca (2002), Gil (2008, p. 50) conclui que

> A pesquisa bibliográfica é desenvolvida a partir de material já elaborado, constituído principalmente de livros e artigos científicos. Embora em quase todos os estudos seja exigido algum tipo de trabalho dessa natureza, há pesquisas desenvolvidas exclusivamente a partir de fontes bibliográficas. Parte dos estudos exploratórios podem ser definidos como pesquisas bibliográficas, assim como certo número de pesquisas desenvolvidas a partir da técnica de análise de conteúdo.

Nesse sentido, esta pesquisa deve ser considerada como pesquisa bibliográfica porque se utiliza de livros e artigos científicos para o aperfeiçoamento do tema, arquitetura de microserviços.

3.2

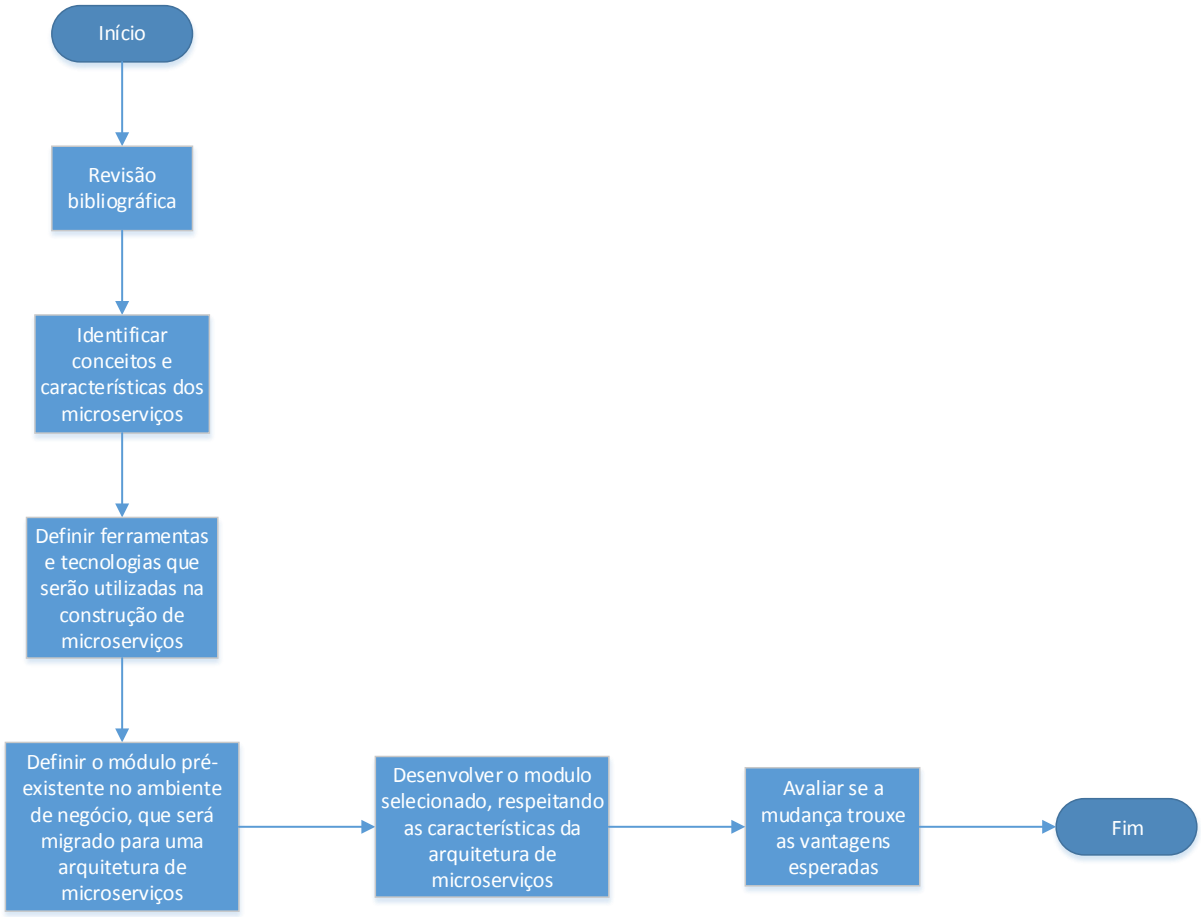
3.3 ETAPAS DA PESQUISA

Segundo Silva (2005, p. 29), “A pesquisa é um procedimento reflexivo e crítico de busca de respostas para problemas ainda não solucionados”. Para executar um procedimento de pesquisa é necessário planejamento, que define as etapas da pesquisa. Há diversas formas de estruturar estas etapas e a escolha destas etapas, passa pela definição de qual tipo de pesquisa será realizada.

Inicialmente é feita a revisão bibliográfica para identificar os modelos de desenvolvimento baseados na arquitetura em microserviços que já foram publicados. Os conceitos e características utilizados para desenvolver *software*s baseados em microserviços serão identificados. As ferramentas e tecnologias que oferecem suporte ao desenvolvimento baseado em microserviços serão avaliadas. A partir destas definições, um método para criação de uma aplicação baseada em microserviços e útil para o negócio da empresa que os integrantes da pesquisa trabalham. Por fim, é feita uma avaliação das vantagens e desvantagens apresentada pela aplicação construída, baseada na arquitetura de microserviços.

A figura 14 apresenta o fluxograma das atividades que serão executadas para alcançar o objetivo proposto neste trabalho.

Figura 14 – Fluxograma das atividades propostas.



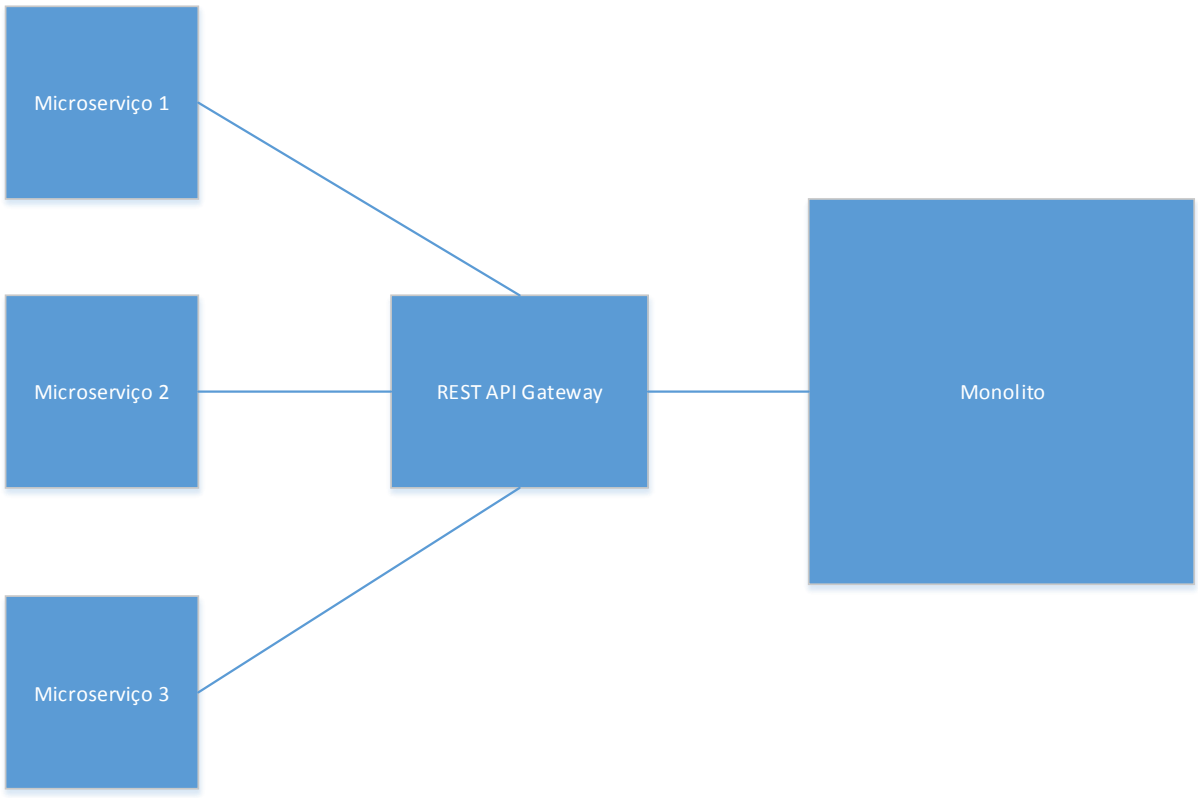
Fonte: Autoria própria.

Após a execução das atividades apresentadas na figura 14, é feita uma avaliação para atestar a viabilidade da arquitetura baseada em microserviços proposta.

3.4 ARQUITETURA DE PROPOSTA DE SOLUÇÃO

O objetivo deste trabalho consiste em elaborar uma proposta de arquitetura de microserviços para reaproveitamento de módulos de soluções monolíticas. A proposta visa fragmentar módulos de uma aplicação monolítica, fazendo com que eles se tornem microserviços que inicialmente gravitam em torno do monólito principal, como pode-se ver na figura 15.

Figura 15 – Arquitetura proposta.



Fonte: Autoria própria.

A arquitetura proposta é composta por microserviços que se comunicam com o monólito e também se comunicam entre si, através de uma API *Gateway* que utiliza REST como método de comunicação. O intuito é migrar módulos chave do monólito para a arquitetura de microserviços.

Para avaliar se a arquitetura de microserviços é satisfatória e pode-se adaptar ao modo de desenvolvimento atual, será realizado um experimento aplicado em um módulo passível de migração para microserviços.

Ao final do trabalho, é avaliado o novo paradigma de desenvolvimento proposto e sua viabilidade no contexto da empresa.

3.5 DELIMITAÇÕES

Como mencionado nos tópicos anteriores, este trabalho propõe apresentar uma proposta de arquitetura em microserviços, visando a quebra dos módulos de uma aplicação *web* já estruturada em um modelo mais tradicional de desenvolvimento. A ideia é utilizar um barramento de comunicação *web* já existente, utilizando o padrão RESTful. A estrutura de microserviços à ser escolhido, será uma estrutura já utilizada no mercado, sem a pretensão de criar uma. O microserviço que será criado através do módulo desacoplado do monólito servirá como um *web service* que estará em nível de aplicação (SaaS). Esta proposta não contemplará nenhum aspecto de segurança, como a segurança das informações que trafegam nas requisições REST. Também não é objetivo propor maneiras de garantir a disponibilidade dos serviços. O objetivo é de apenas experimentar esta nova arquitetura de microserviços no ambiente de desenvolvimento da empresa em que os presentes autores trabalham.

4 MODELAGEM

Antes do processo de desenvolvimento de *software*, é necessário entender as características e comportamentos que o sistema comportará. Os modelos disponíveis podem ser utilizados durante a construção do *software* para identificar as características e funcionalidades providas por este, além de ajudar no planejamento da construção do sistema.

4.1 METODOLOGIAS E TÉCNICAS

O processo de modelagem envolve a escolhe de métodos e técnicas que melhor se encaixem na construção do modelo de *software* proposto. Atualmente o mercado de *software* oferece diversas ferramentas que implementam algumas destas técnicas e padrões e que facilitam a construção do modelo de *software* proposto.

4.1.1 Orientação à objeto

No mercado de desenvolvimento de *software* existem diversas linguagens de programação e essas linguagens utilizam diferentes paradigmas.

Atualmente o paradigma mais utilizado é a Orientação a Objetos. Isso ocorre pelo fato deste padrão ter evoluído muito nos últimos tempos.

Sommerville (2007, p. 208) afirma que “análise orientada a objetos se concentra no desenvolvimento de um modelo orientado a objetos do domínio da aplicação. Os objetos nesse modelo refletem as entidades e as operações associadas ao problema a ser resolvido”.

Este modelo descreve como o *software* funciona para satisfazer uma série de requisitos definido pelo cliente. (PRESSMAN, 2002, p.560)

A Programação Orientada a Objetos é sustentada por quatro pilares que são:

4.1.1.1 Abstração

Consiste na definição de um objeto do mundo real para o mundo da programação, se preocupando com os aspectos essenciais de acordo com um contexto. Em outras palavras, abstração é trazer objetos reais para o mundo da computação, respeitando suas principais características e comportamentos.

4.1.1.2 Encapsulamento

A ideia de encapsulamento na programação orientada a objetos é a de segregar o *software* em partes isoladas. Essa segregação tende a deixar o *software* mais flexível, tornando assim, eventuais novas implementações e correções mais fáceis de serem executadas. Além de proteger os dados manipulados dentro de determinadas classes, o que define onde e por quem essa classe poderá ser manipulada.

4.1.1.3 Herança

A herança é um recurso da orientação à objeto que permite que classes herdem atributos e métodos de outras classes. Este mecanismo promove o reuso e reaproveitamento de código já existente ou comportamentos generalizados, além de permitir especializar operações ou atributos.

4.1.1.4 Polimorfismo

Polimorfismo é o preceito de duas ou mais classes que derivam da mesma superclasse ou implementam a mesma *interface* - quando a linguagem permite o uso de *interfaces* - podem chamar métodos com a mesma assinatura, porém comportamentos distintos, onde cada classe derivada especializa este método, de acordo com o propósito da classe. Uma maneira de implementar o polimorfismo é utilizando uma classe abstrata, onde os métodos são declarados, mas não implementados, deixando a implementação para as classes que herdam os métodos desta classe abstrata.

4.1.2 UML – Linguagem de Modelagem Unificada

UML – *Unified Modeling Language* – traduzido para o português significa Linguagem de Modelagem Unificada. Para Ramos (2006) UML é uma linguagem para modelagem de dados orientado a objetos, usada para especificar, construir, visualizar e documentar um sistema de *software*. Essa linguagem foi desenvolvida na metade da década de 1990 por Grady Booch, James Rumbaugh e Ivar Jacobson.

> Eles possuem um extenso conhecimento na área de modelagem orientado a objetos já que as três mais conceituadas metodologias de modelagem orientado a objetos foram eles que desenvolveram e a UML é a junção do que havia de melhor nestas três metodologias adicionado novos conceitos e visões da linguagem. (HAFEMANN, 2000, p. 10).

Segundo Ribeiro (2007 apud ANDRADE, 2012), a UML define uma notação que é uma união de diversas notações preexistentes, removendo alguns elementos e adicionando outros com o objetivo de transformá-la numa notação mais expressiva. A UML foi aprovada pelo OMG (*Object Management Group*) em 1997 e desde então, tem tido grande aceitação pela comunidade de desenvolvedores de sistemas.

Além desses três citados anteriormente definirem uma notação e um conjunto de símbolos padrões para esta linguagem, eles também se propuseram a especificar em detalhes toda a semântica desejável de um modelo de sistema. Demonstrando através de diagramas

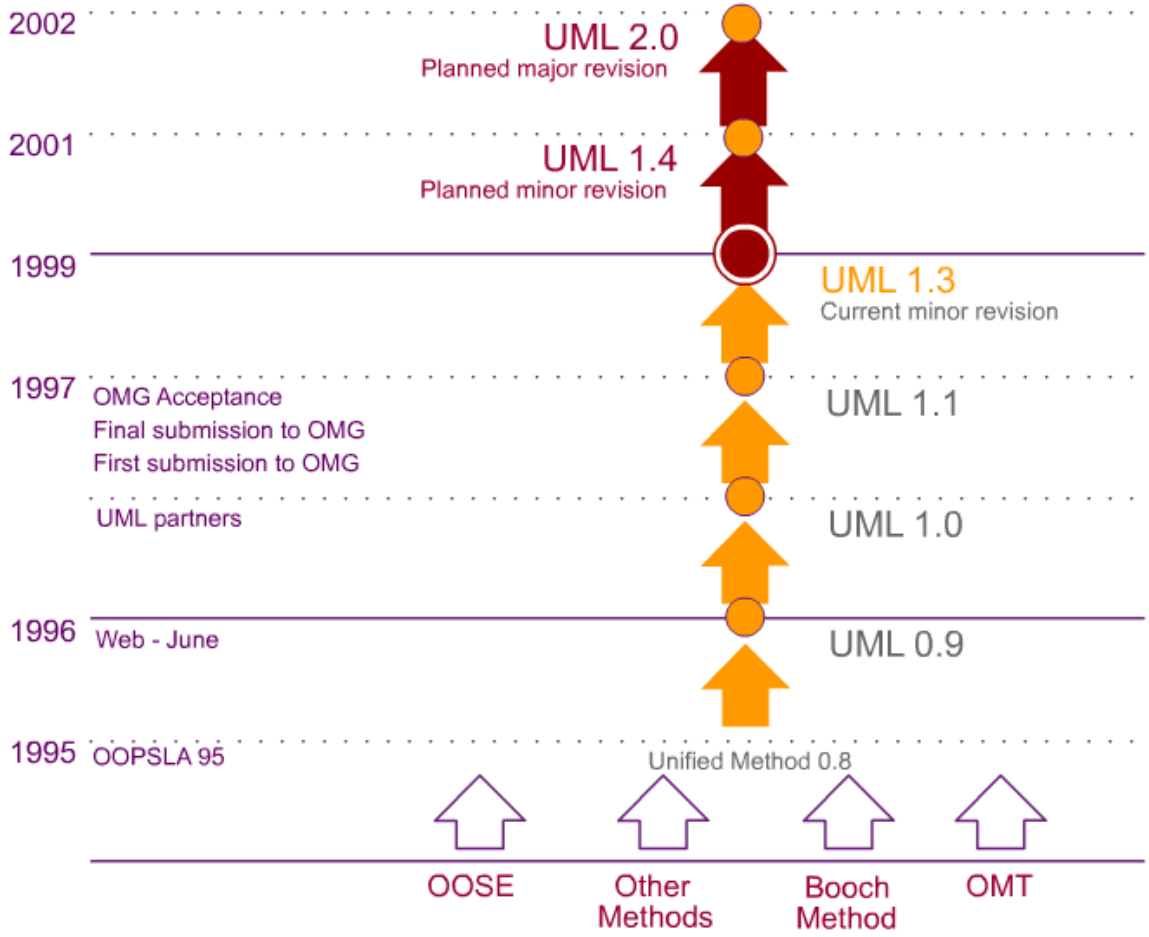
estabelecidos por eles, todos os elementos que compõem o modelo, suas características e relacionamentos, facilitando a efetiva comunicação entre os profissionais que a utilizam.

4.1.2.1 Evolução da UML

A UML foi desenvolvida na metade da década de 1990 por Grady Booch, James Rumbaugh e Ivar Jacobson, nessa época existiam três métodos que mais cresciam no mercado, eram eles: Booch'93 de Grady Booch, OMT-2 de James Rumbaugh e OOSE de Ivar Jacobson. Segundo Sampaio (2007) o OOSE possuía foco em casos de uso (use cases), OMT-2 se destaca na fase de análise de sistemas de informação e Booch'93 era mais forte na fase de projeto. O sucesso desses métodos foi, principalmente, devido ao fato de não terem tentado estender os métodos já existentes. Seus métodos já convergiam de maneira independente, então seria mais produtivo continuar de forma conjunta.

Fowler (2003) complementa escrevendo em seu livro que em outubro de 1994, começaram os esforços para unificação dos métodos. Já em outubro de 1995, Booch e Rumbaugh lançaram um rascunho do "Método Unificado" unificando o Booch'93 e o OMT-2. Após isso, Jacobson se juntou a equipe do projeto e o "Método Unificado" passou a incorporar o OOSE. Em junho de 1996, os três amigos, como já eram conhecidos, lançaram a primeira versão com os três métodos - a versão 0.9 que foi batizada como UML. Após a UML ser batizada, foram lançadas várias novas versões na qual podemos acompanhar através da figura 16.

Figura 16 – Linha do tempo da UML.



Fonte: http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/uml/historia_uml/historia_uml.htm

Segundo Vargas (2007) a OMG3 lançou um RFP (Request for Proposals) para que outras empresas pudessem contribuir com a evolução da então UML, chegando assim à versão 1.1. Após alcançar esta versão, a OMG3 passou a adotá-la como padrão e a se responsabilizar (através da RTF – Revision Task Force) pelas revisões. Essas revisões são, de certa forma, "controladas" a não provocar uma grande mudança no escopo original. Se observarmos as diferenças entre as versões atualmente, veremos que de uma para a outra não houve grande impacto, o que facilitou sua disseminação pelo mundo.

4.1.2.2 Uso da UML

Segundo Carlos Videira e Alberto Silva (2001) e Guedes (2009) a UML é usada no desenvolvimento de qualquer tipo de sistema. Ela sempre abrange alguma característica de sistema em seus diagramas. Também pode ser aplicada em diferentes etapas do desenvolvimento de um *software*, desde a especificação da análise de requisitos até a finalização, na etapa de testes.

A linguagem tem como intuito representar de alguma forma qualquer tipo de sistema, em forma de diagramas orientado a objetos. O seu uso mais comum é para gerar artefatos para o desenvolvimento de um *software*, porém, também pode ser usada para descrever sistemas mecânicos sem nenhum *software*.

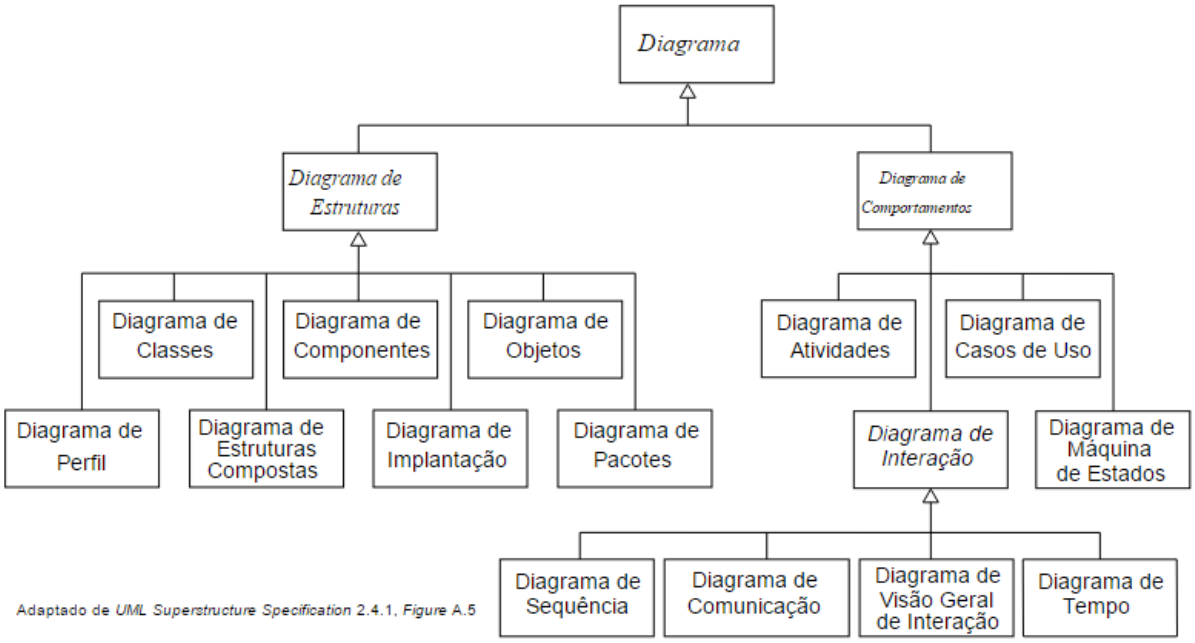
4.1.2.3 Diagramas da UML

A UML fornece diversos diagramas que são usados em conjunto, com a finalidade de obter diversas visões dos elementos do sistema. Segundo Guedes (2009) os diagramas da UML 2 encontram-se divididos em diagramas estruturais e diagramas comportamentais.

Os diagramas da UML possibilitam a visualização de um sistema sob diferentes perspectivas, apresentando os elementos que compõe o sistema em diversos diagramas, sendo que o mesmo elemento pode estar em mais de um diagrama, representando assim um objeto.

A divisão entre os diagramas estruturais e diagramas comportamentais podem ser vistas na figura 17.

Figura 17 – Diagramas da UML.



Fonte: http://www.ateomomento.com.br/o-que-e-caso-de-uso/

Estes são todos os diagramas da UML: Classe, Componentes, Objetos, Estrutura Composta, Instalação, Pacote, Atividade, Interação, Casos de uso, Estados, Sequência, Visão de Interação, Comunicação e Tempos.

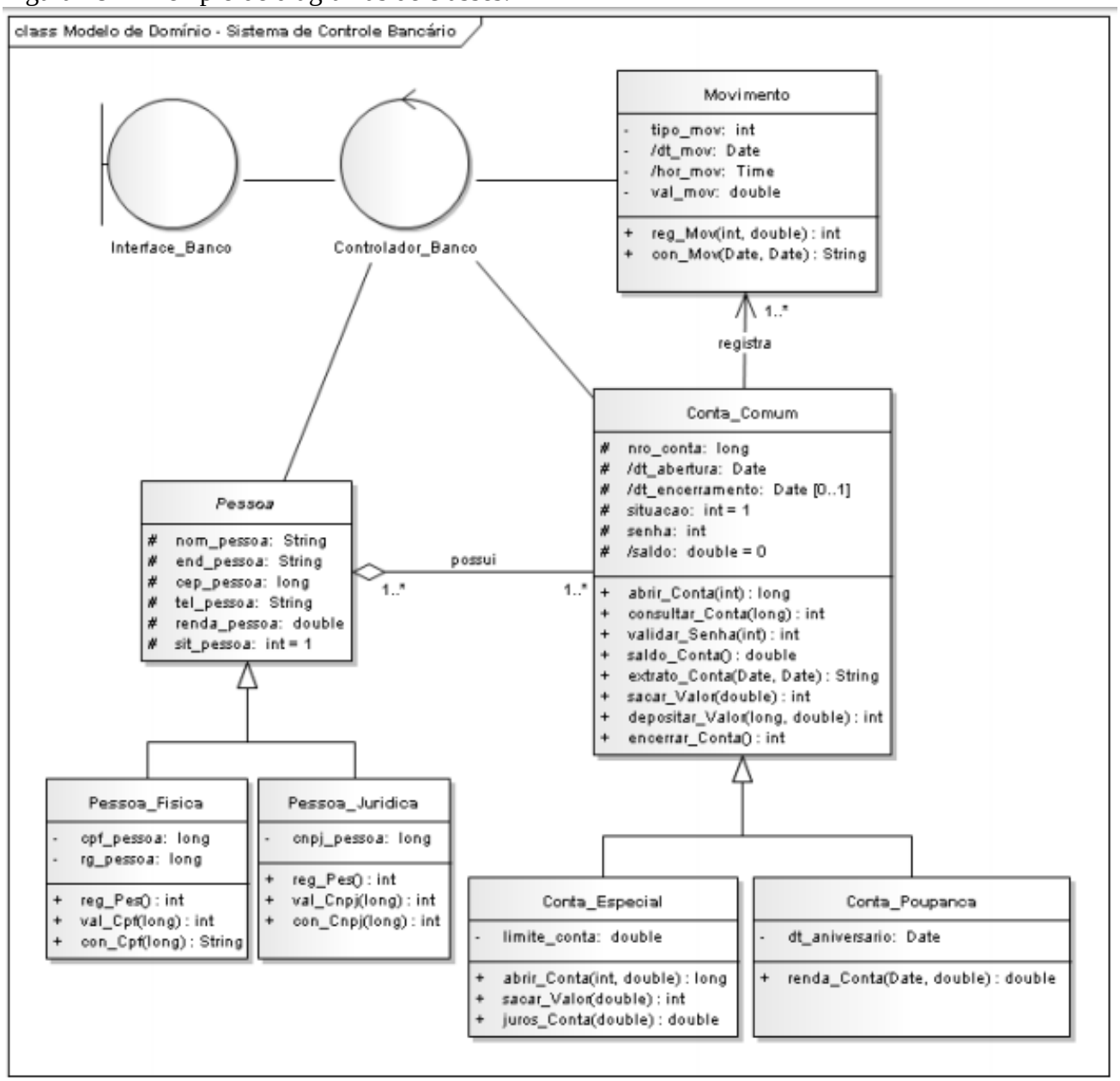
É sabido que os arquitetos de *software* e implementadores se baseiam em diagramas para entender como os sistemas se encaixam. Os diagramas por mais que possuam vários aparatos como caixas e linhas, devem possuir ar vago de certeza, ou seja, que ofereçam uma visualização bidimensional que represente um mundo ideal. Como este trabalho diz a respeito de uma arquitetura de microserviços, opta-se por dar um enfoque maior nos diagramas de Classe, Componentes e Implantação (também conhecido como diagrama de instalação).

Esta escolha foi feita com base na melhor representação gráfica para o microserviço que foi criado. Tendo em vista demonstrar as classes do microserviço, com quem interage e como deve ser feito seu *deploy*.

4.1.2.3.1 Diagrama de Classe

Segundo Guedes (2009) este diagrama provavelmente é o mais utilizado e é um dos mais importantes da UML. Pois, serve de apoio para a maioria dos demais diagramas. Este diagrama define a estrutura das classes utilizadas pelo sistema, definindo seus atributos e métodos, além de demonstrar como as classes se relacionam e trocam informações entre si. A figura 18 apresenta um exemplo desse diagrama.

Figura 18 – Exemplo de diagramas de classes.



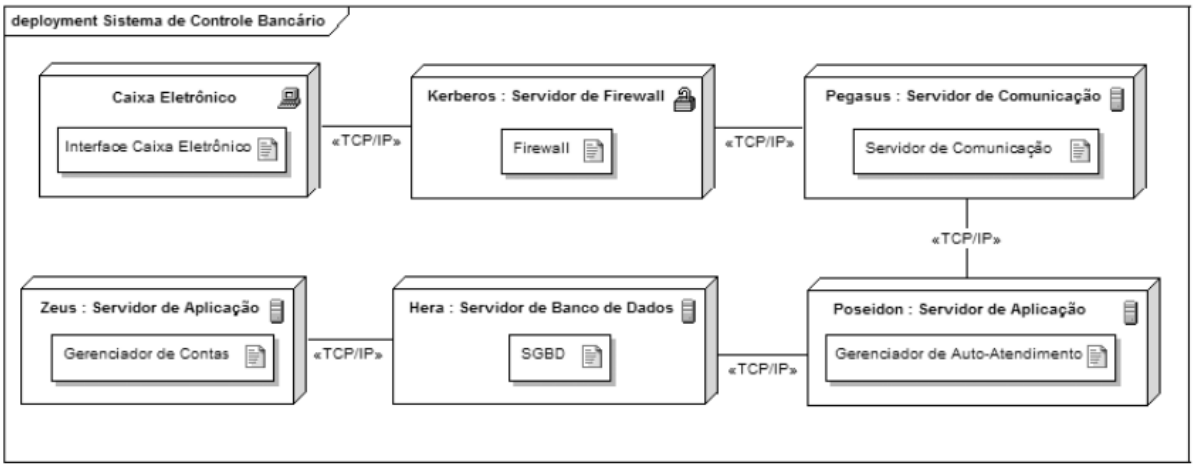
Fonte: Guedes (2009, p 32)

O diagrama de classes define as classes contidas no sistema – atributos e métodos de cada uma das classes – facilitando a visualização do contexto de cada um dos componentes.

4.1.2.3.2 Diagrama de Implantação

Segundo Guedes (2009) este diagrama determina as necessidades de *hardware* do sistema, as características físicas como servidores, estações, topologias e protocolos de comunicação, ou seja, todo o contexto físico sobre o qual o *software* deverá ser executado. A figura 19 apresenta um exemplo desse diagrama.

Figura 19 – Exemplo de diagrama de implantação.



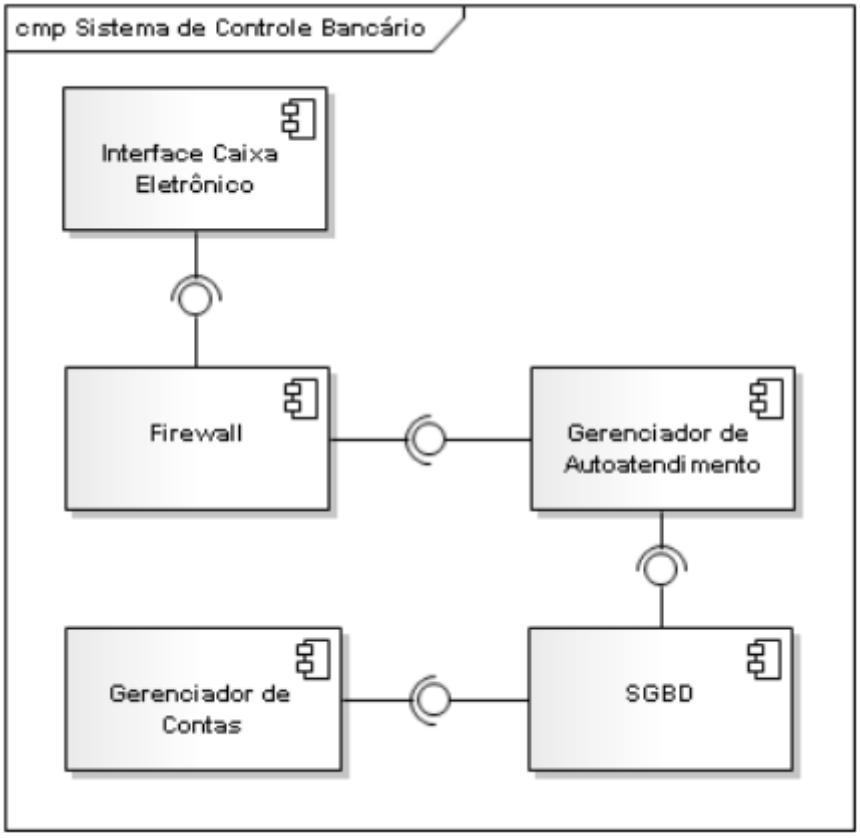
Fonte: Guedes (2009, p 39)

Como vemos no exemplo, neste diagrama podemos visualizar os componentes e suas formas de comunicação, o que facilita o entendimento dos relacionamentos entre os componentes. Estes componentes que podem ser melhor visualizados no digrama de componentes.

4.1.2.3.3 Diagrama de Componentes

Segundo Guedes (2009) esse diagrama representa os componentes do sistema, quando o mesmo deve ser implementado em termos de módulos (poliglotas ou não), bibliotecas, formulários, arquivos de ajuda etc. e indica como estes estarão estruturados e irão interagir para que o sistema funcione de maneira correta. A figura 20 apresenta um exemplo desse diagrama.

Figura 20 – Exemplo de diagrama de componentes.



Fonte: Guedes (2009, p 39)

Com base nos diagramas apresentados acima, podemos concluir através de suas explicações que estes são os mais indicados para modelar um microserviço, pois temos a visualização de todos os componentes através do diagrama de componentes, o contexto de cada componente através do diagrama de classes e a comunicação entre os componentes através do diagrama de implantação. Coloca-se em prática este aprendizado na seção 4.2, a qual contemplará o projeto de solução com estes 3 diagramas devidamente modelados.

4.1.2.4 Ferramentas da UML

Para Weinrich (1999) ferramentas CASE (do inglês *Computer-Aided Software Engineering*) é:

> Uma classificação que abrange todas ferramentas baseada em computadores que auxiliam atividades de engenharia de *software*, desde análise de requisitos e modelagem até programação e testes. Podem ser consideradas como ferramentas automatizadas que tem como objetivo auxiliar o desenvolvedor de sistemas em uma ou várias etapas do ciclo de desenvolvimento de *software*.

Segundo Guedes (2009) as ferramentas de UML se encaixam nesse seguimento de ferramentas e algumas serão abordadas incluindo as suas características.

IBM Rational Requisite Pro: Programa reconhecido no mercado de fácil utilização para gerenciamento de requisitos e de referência de utilização que promove melhor comunicação, aprimora o trabalho em equipe e reduz o risco do projeto.

Umbrello UML: Programa de modelagem para a plataforma Linux. Permite criar todos diagramas da UML no formato padrão. É uma ferramenta *open-source*.

Poseidon para UML: O Poseidon é uma evolução da ferramenta de código-aberto ArgoUML estando entre as ferramentas de modelagem mais conhecidas. Seu principal foco está na facilidade de uso que a torna simples de aprender e usar.

Além dessas, existem outras várias as ferramentas que podem ser utilizadas na modelagem com a UML (Rational Rose, MagicDraw, Omondo, Jude, TogetherSoft, ArgoUML, Fujaba, Visual Paradigm, Objecteering, Enterprise Architect, Dia, etc.).

4.1.3 Etapas da modelagem

O projeto inicial consiste de uma API que é consumida por diversos sistemas. Esta API trata diferentes domínios de negócio, efetuando o controle em nível de sistema de cidades, estados, pessoas e os cadastros que podem envolver esta entidade – funcionários,

fornecedores, usuários do sistema -, até o controle das entidades e regras de negócio relacionadas ao setor financeiro.

A ideia do projeto é segmentar os elementos dessa API respeitando os conceitos anteriormente apresentados nesse trabalho, avaliando os domínios de negócios existentes e construindo microserviços que utilizam um barramento de comunicação baseado em REST.

Este projeto visa desacoplar todo domínio de negócio inerente ao financeiro desta API inicial. Para isto foi necessário avaliar as dependências das entidades, as regras de negócio inseridas no sistema, a melhor arquitetura para o desenvolvimento do microserviço – a empresa que utilizará o microserviço possui uma stack que define as tecnologias e ferramentas à serem usadas para o desenvolvimento de novos sistemas. Após estas avaliações foi definido um projeto de solução, que será apresentado na próxima sessão, assim como os diagramas que exemplificam a arquitetura proposta.

4.2 PROJETO DE SOLUÇÃO

A ideia central deste projeto é propor uma arquitetura baseada em microserviços para a migração de sistemas baseados em uma arquitetura monolítica. Seguindo os conceitos de microserviços e respeitando as regras de negócio inseridas no sistema legado, foi criado um diagrama de classe, um diagrama de componente e um diagrama de implantação, que englobam o contexto ao qual o microserviço desenvolvido está inserido.

4.2.1 Diagrama de Classe

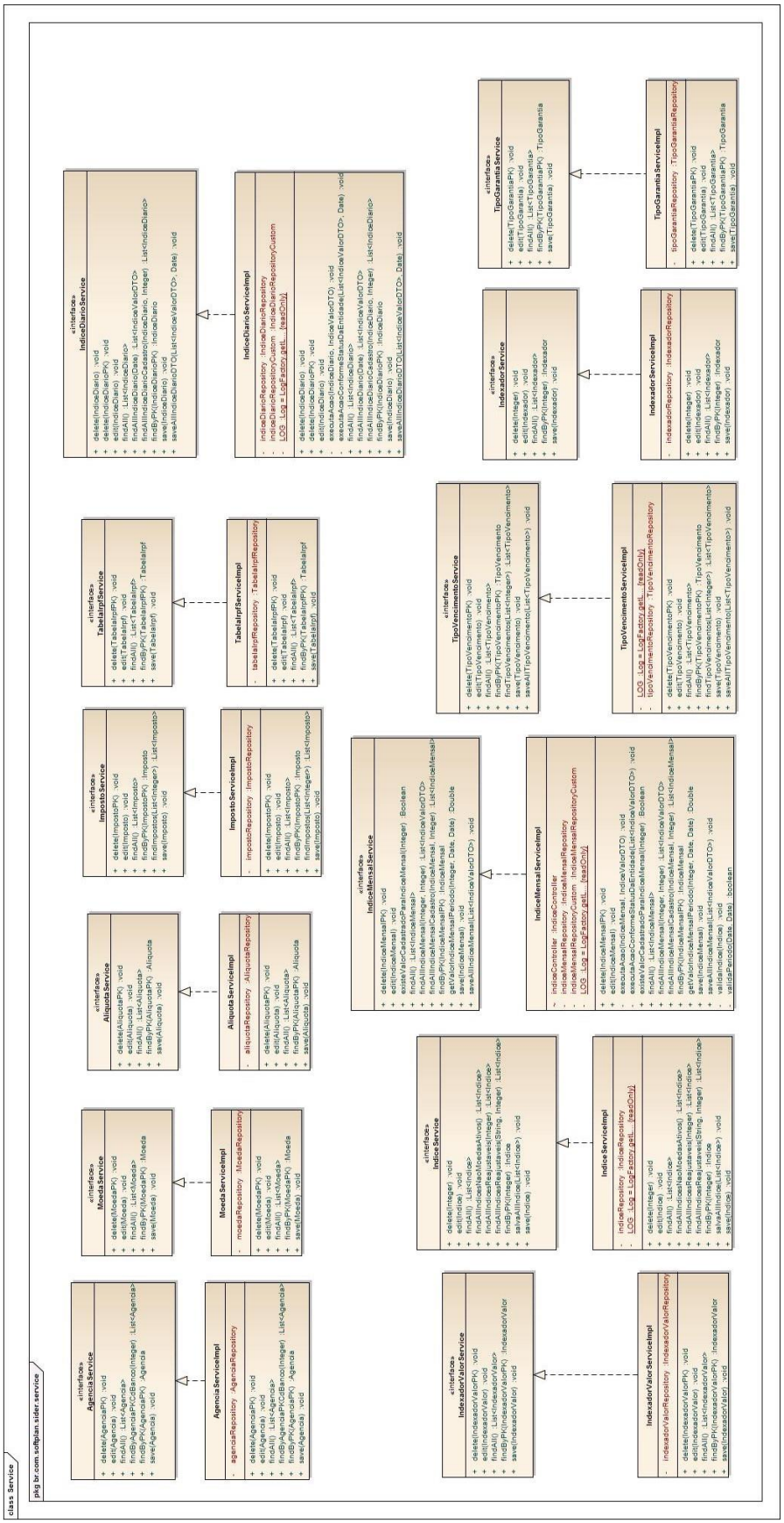
Os diagramas que serão apresentados nas figuras 21, 22, 23 e 24 representam os pacotes do sistemas, assim como as classes que estão envolvidas no contexto do microserviço criado. Uma visão completa dos pacotes e suas classes pode ser vista no primeiro apêndice deste trabalho.

Figura 21 – Modelagem diagrama do pacote *Controller*.

Fonte: Autoria própria

O diagrama apresentado na figura 21 mostra a camada referente aos controladores, que disponibilizam a interface que será consumida.

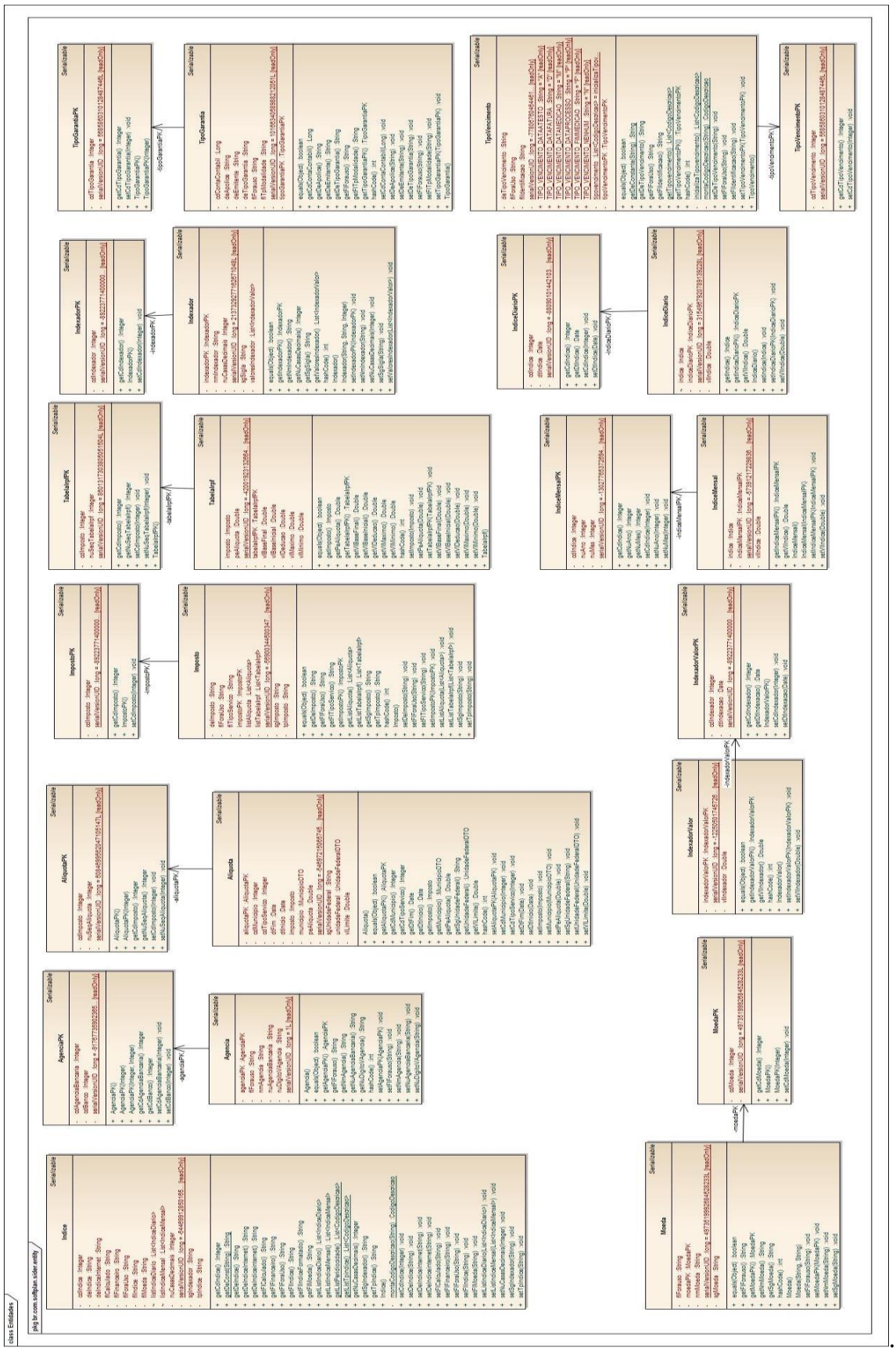
Figura 22 – Modelagem diagrama do pacote *Service*.



Fonte: Autoria própria

A figura 22 apresenta a segunda camada é referente aos *services* e suas implementações, essa camada é responsável pelas validações das regras de negócio.

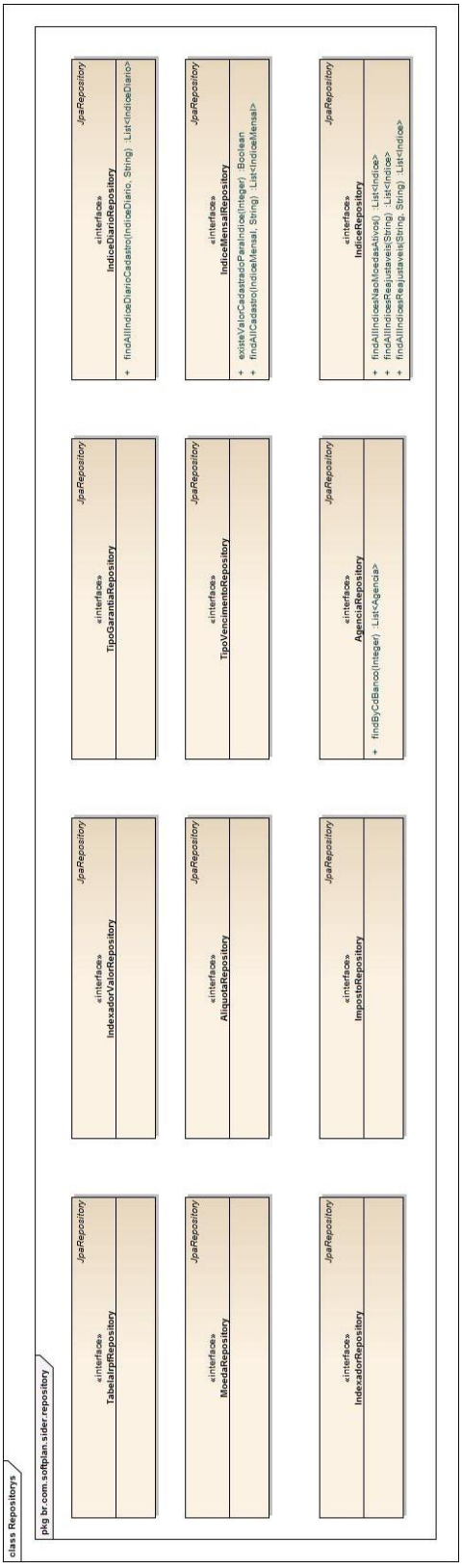
Figura 23 – Modelagem diagrama do pacote *Entity*



Fonte: Autoria própria

A figura 23 representa a terceira camada da aplicação desenvolvida que é das entidades e suas chaves primárias.

Figura 24 – Modelagem diagrama do pacote *Repository*



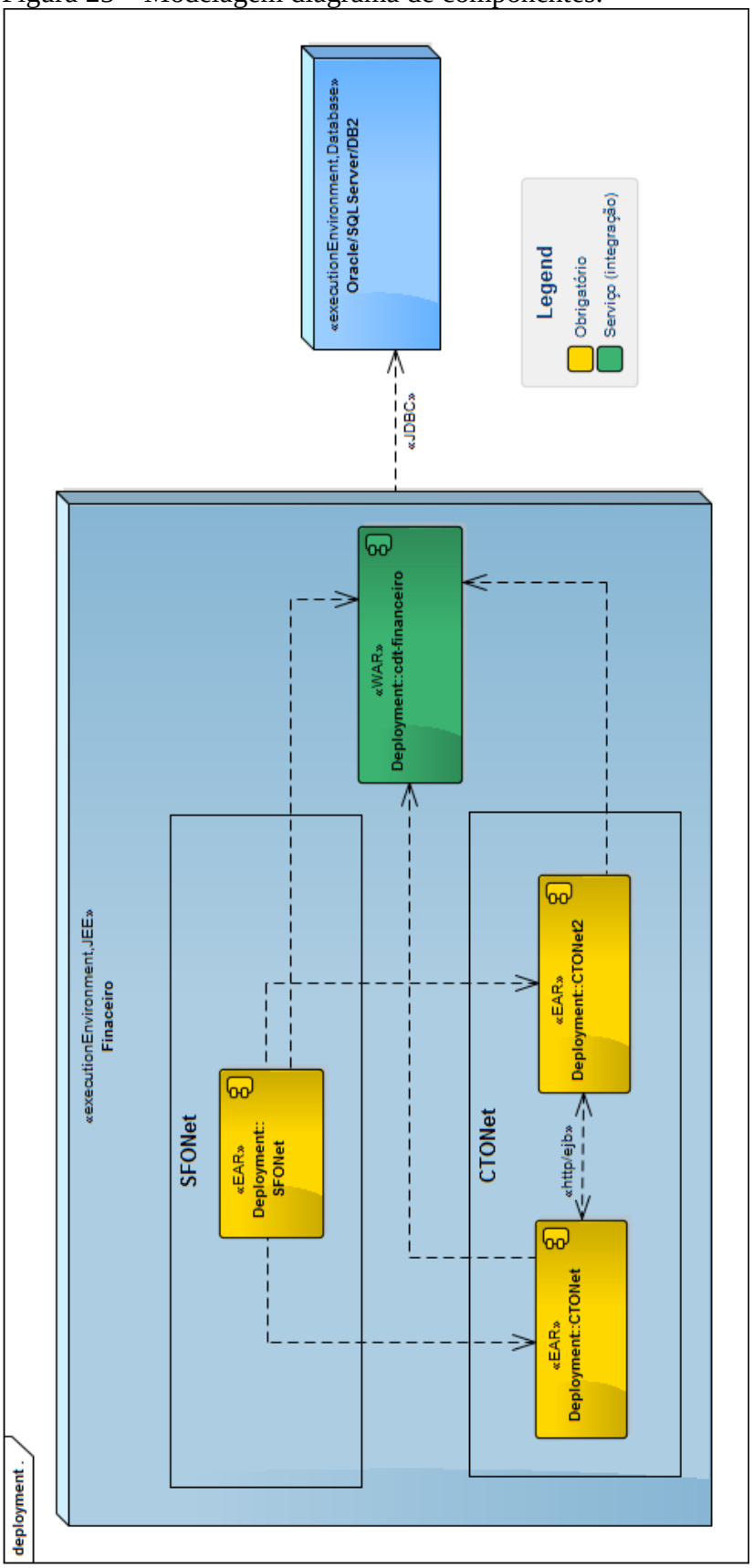
Fonte: Autoria própria

A figura 24 apresenta a quarta camada representa os repositórios e seus comportamentos.

4.2.2 Diagrama de Componente

Neste diagrama serão apresentados os componentes – sistemas e domínios de negócio - que estão envolvidos no contexto do microserviço criado.

Figura 25 – Modelagem diagrama de componentes.



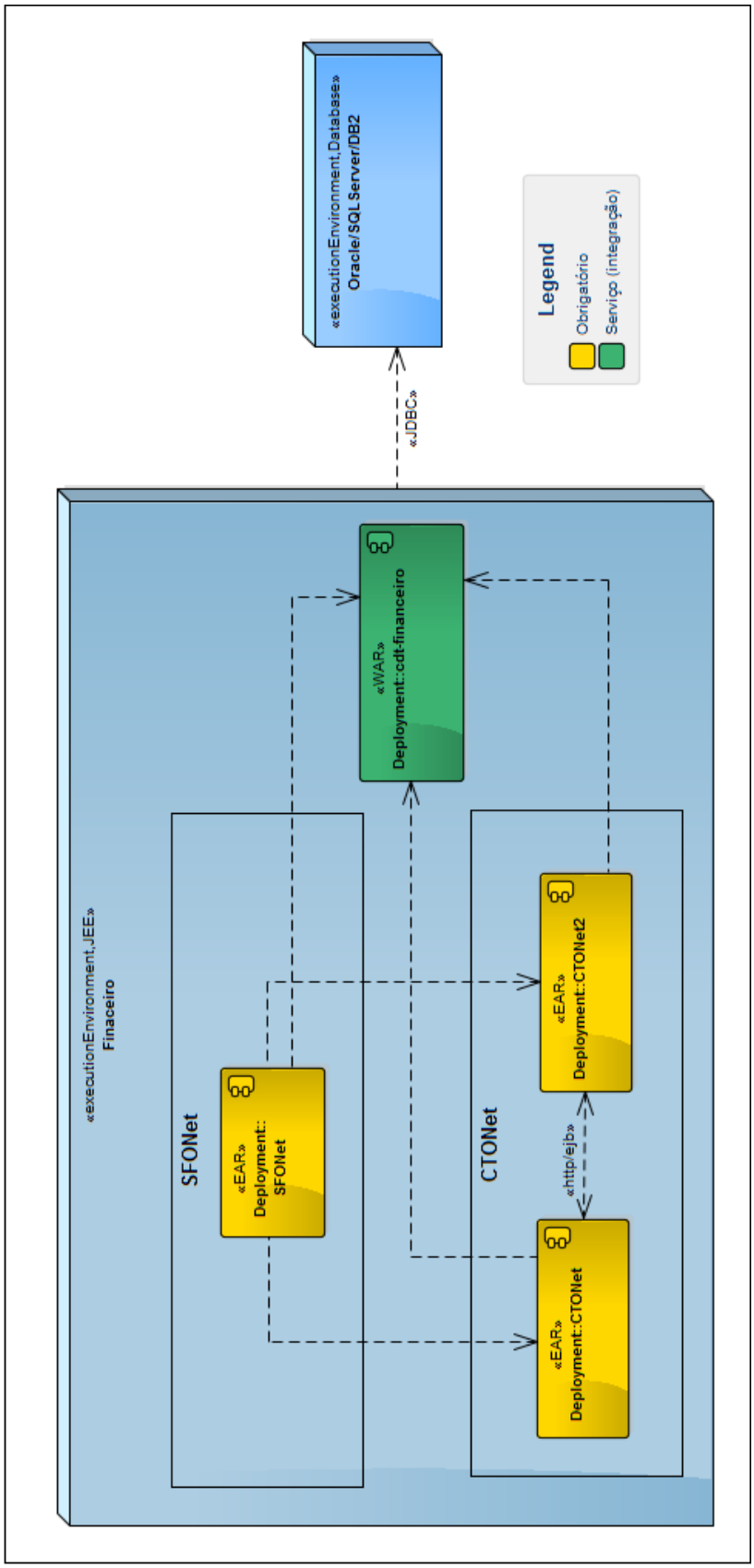
Fonte: Autoria própria

O diagrama apresentado na figura 25 mostra o diagrama de componentes do contexto em que o microserviço desenvolvido está inserido. Como pode-se ver na figura 25, existem dois sistemas - SFONet e CTONet - que consomem as informações disponibilizadas pelo microserviço do cdt-financeiro.

4.2.3 Diagrama de Implantação

Para este diagrama visualiza-se o contexto todo da aplicação. O microserviço migrado neste trabalho foi o cdt-financeiro como pode-se ver na figura 26.

Figura 26 – Modelagem diagrama de implantação.



Fonte: Autoria própria.

O diagrama de *deployment* apresentado na figura 26, apresenta todos os componentes que fazem parte do sistema financeiro, assim como as suas dependências e as formas de comunicação entre os componentes. O microserviço desenvolvido se encontra junto de outros serviços já existente e que são usados pelo sistema financeiro.

5 DESENVOLVIMENTO E VALIDAÇÃO DA PROPOSTA

Neste capítulo, são apresentadas as ferramentas escolhidas e utilizadas para o desenvolvimento da proposta, bem como o processo de desenvolvimento do microserviço, demonstrando as principais vantagens da arquitetura de microserviços, e a avaliação da proposta.

5.1 FERRAMENTAS E TECNOLOGIA UTILIZADAS

Nesta seção, são apresentadas as ferramentas utilizadas para assegurar a funcionalidade da arquitetura da solução proposta.

5.1.1 Git

Git é um sistema livre e de código aberto distribuído de controle de versão projetado para lidar com tudo, desde pequenas a grandes projetos com rapidez e eficiência. Git é fácil de aprender e tem um bom desempenho. Ele supera ferramentas de SCM como Subversion, CVS, Perforce, e ClearCase com recursos de ramificação local, áreas de preparação conveniente, e vários fluxos de trabalho.

Figura 27 – Logo Git.

Fonte: https://git-scm.com/

Essa ferramenta foi utilizada para versionar os arquivos do projeto do microserviço, ela foi escolhida pela empresa para este trabalho pelo fato de já ser usada por outros projetos e fazer parte do ambiente de integração continua.

Informações tiradas do site do fabricante: https://git-scm.com/

5.1.2 Gerrit

Gerrit é uma ferramenta *web* gratuita de revisão de código construída em cima do sistema de controle de versão git. Escrito em Java (Java EE Java Servlet) e desenvolvido pelo Google por Shawn Pearce para o desenvolvimento do projeto Android.

Figura 28 – Logo Gerrit.

Fonte: https://www.gerritcodereview.com/

Essa ferramenta foi utilizada para verificação e controle de qualidade do código produzido durante o desenvolvimento. As revisões foram feitas pelo arquiteto de softwate e tutor do projeto na empresa, ela foi escolhida pela empresa para este trabalho pelo fato de ser gratuita, já ser usada por outros projetos e fazer parte do ambiente de integração continua.

Informações tiradas do site do fabricante: https://www.gerritcodereview.com/

5.1.3 GitBlit

Gitblit é uma ferramenta de código aberto, escrita com Java puro, serve para visualizar, gerar e administrar repositórios Git. Foi projetado principalmente como uma ferramenta para pequenos grupos de trabalho que querem hospedar repositórios centralizados.

Figura 29 – Logo Gitblit.

Fonte: http://gitblit.com/

Essa ferramenta foi utilizada para gerar e administrar o repositório criado para o projeto deste trabalho. Ela foi escolhida pela empresa para este trabalho pelo fato de ser gratuita, já ser usada por outros projetos e fazer parte do ambiente de integração continua.

Informações tiradas do site do fabricante: http://gitblit.com/

5.1.4 Eclipse

Eclipse é uma plataforma que foi projetada desde o início para desenvolvimento de aplicações *web*. A plataforma não fornece uma grande quantidade de funcionalidades para o usuário final por si só. O valor da plataforma é: desenvolvimento rápido de recursos integrados com base em um modelo de plug-in. Cada plug-in pode se concentrar em fazer bem um pequeno número de tarefas. Que tipo de tarefas? Definição, análise, animação, edição, compilação, depuração, diagramação ... o único limite é a sua imaginação.

Eclipse também fornece um modelo comum de *interface* de usuário (UI) para trabalhar com ferramentas. Ele é projetado para funcionar em vários sistemas operacionais, proporcionando a integração robusta com cada sistema operacional subjacente.

Figura 30 – Logo Eclipse.

Fonte: http://www.eclipse.org/

Essa ferramenta foi utilizada para codificação do projeto deste trabalho. Ela foi escolhida pela empresa para este trabalho pelo fato de ser gratuita e já ser usada pelos integrantes deste trabalho.

Informações tiradas do site do fabricante: http://www.eclipse.org/

5.1.5 Java

O Java é uma tecnologia usada para desenvolver aplicações que tornam a *web* mais divertida e útil. O Java não é a mesma coisa que o JavaScript, que é uma tecnologia simples usada para criar páginas *web* e só é executado no seu *browser*.

O Java permite executar jogos, fazer *upload* de fotos, bater papo *on-line*, fazer *tours* virtuais e usar serviços, como treinamento *on-line*, transações bancárias *on-line* e mapas interativos. Se você não tiver o Java, muitas aplicações e *websites* simplesmente não funcionarão.

Figura 31 – Logo Java.

Fonte: https://www.java.com

Essa linguagem foi utilizada para codificação do projeto deste trabalho. Ela foi escolhida pela empresa para este trabalho pelo fato de ser gratuita e já ser usada por outros projetos.

Informações tiradas do site do fabricante: https://www.java.com

5.1.6 SGBD - MySQL

MySQL é o mais popular sistema de gerenciamento de banco de dados SQL Open Source, é desenvolvido, distribuído e apoiado pela Oracle Corporation.

Figura 32 – Logo MySQL.

Fonte: http://www.mysql.com/

Este banco de dados foi utilizado para armazenamento dos dados projeto deste trabalho. Ela foi escolhida pela empresa para este trabalho pelo fato de ser gratuita, e ser exigida por alguns clientes.

Informações tiradas do site do fabricante: http://www.mysql.com/

5.1.7 Enterprise Architect

Enterprise Architect é uma ferramenta excepcional com recursos de ponta e um rico conjunto de recursos para ajudar a gerenciar informações e inovar no ambiente complexo e exigente de hoje. A um preço significativamente mais baixo do que as ferramentas concorrentes, Enterprise Architect oferece a você, sua equipe e sua empresa a oportunidade de criar modelagens e construir recursos de ponta a um preço amigável.

Enterprise Architect tem uma longa e comprovada reputação em uma ampla gama de indústrias em mais de 160 países. Por 15 anos, tem sido continuamente desenvolvido, melhorado e refinado para satisfazer as necessidades emergentes de programadores, analistas de negócios, arquitetos corporativos, testadores, gerentes de projeto, *design*ers e outros.

Baseado em padrões abertos, Enterprise Architect pode confortavelmente escalar de pequenos modelos individuais a grandes repositórios.

Figura 33 – Logo Enterprise Architect.

Fonte: http://www.sparxsystems.com.au/

Esta ferramenta de modelagem foi utilizada para gerar os artefatos de modelagem do projeto deste trabalho. Ela foi escolhida pela empresa para este trabalho pelo fato de ser a mais popular e de fácil uso.

Informações tiradas do site do fabricante: http://www.sparxsystems.com.au/

5.1.8 Spring boot

Spring Boot é um *framework* que facilita a criação de aplicações *stand-alone* voltadas para o negócio que utilizam Spring, onde você pode apenas "rodá-las". Foi utilizada uma visão a partir de algumas opiniões de usuários da plataforma Spring e bibliotecas de terceiros na criação do *framework*, sendo assim, é possível criar um projeto do zero com o mínimo de esforço. A maioria dos projetos que utilizam spring boot, necessitam de apenas pequenas configurações do Spring.

Figura 34 – Logo Spring Boot.

Fonte: http://projects.spring.io/spring-boot/

Este *framework* foi utilizado para facilitar o desenvolvimento do projeto deste trabalho. Ele foi escolhido pelo integrante desse projeto pelo fato de ser voltado para o desenvolvimento de microserviços, ser popular com grande base de conhecimento na *web* e de fácil uso.

Informação tiradas do site do fabricante: http://projects.spring.io/spring-boot/

5.1.9 Postman

Postman é um poderoso cliente HTTP para testar serviços *web*. Criado por Abhinav Asthana, um programador e *designer* nascido em Bangalore, Índia.

Postman torna mais fácil para testar, desenvolver e documentar APIs, permitindo que os usuários coloquem com facilidade juntos os dois pedidos de HTTP, os simples e os complexos. Está disponível como um aplicativo do Google Chrome.

Figura 35 – Logo Postman.

Fonte: http://www.getpostman.com/

Esta ferramenta de modelagem foi utilizada para efetuar requisições HTTP para testar os *endpoints*[18] do projeto deste trabalho. Ela foi escolhida pelos integrantes desse projeto pela facilidade de uso e pela *interface* amigável.

Informações tiradas do site do fabricante: http://www.getpostman.com/

5.1.10 Jenkins

Jenkins é um mecanismo para automação com um ecossistema de plug-ins paralelo que oferece suporte a todas as ferramentas para entrega, independente se o objetivo seja integração continua, testes automatizados ou entrega continua.

[18] Meio de comunicação disponibilizado para consumo do microserviço

Figura 36 – Logo Jenkins.

Fonte: https://jenkins.io/doc/

Esta ferramenta foi utilizada para facilitar a geração de versões do projeto deste trabalho. Ela foi escolhida pela empresa para este trabalho pelo fato de ser gratuita, já ser usada por outros projetos e fazer parte do ambiente de integração continua.

Informações tiradas do site do fabricante: https://jenkins.io/doc/

5.1.11 SonarQube

Sonarqube é uma plataforma aberta para gerenciamento de qualidade de código. Para tal a ferramenta utiliza os 7 pilares da qualidade de código: arquitetura e projeto, comentários, regras de codificação, potenciais falhas, complexidade, testes unitários e duplicidade de código.

Figura 37 – Logo SonarQube.

Fonte: http://www.sonarqube.org/

A ferramenta possui uma maneira muito eficiente de navegação, um balanço entre visões de alto nível, *dashboards*, *TimeMachine* e ferramentas para detecção de *bugs*. Isso permite mostrar de maneira eficiente projetos que não possuem cobertura de testes ou componentes que possuem débito técnico, isso permite que estratégias referentes ao código sejam traçadas.

Esta ferramenta foi utilizada para manter a qualidade do código, pois a cada *commit* era feita uma validação das regras configuradas para o projeto. Ela foi escolhida pela

empresa para este trabalho pelo fato de ser gratuita, já ser usada por outros projetos e fazer parte do ambiente de integração continua.

Informações tiradas do site do fabricante: http://www.sonarqube.org/

5.1.12 Apache

O projeto servidor HTTP Apache é um esforço para desenvolver e manter um servidor HTTP de código aberto para sistemas operacionais modernos, incluindo UNIX e Windows. O objetivo do projeto é fornecer um servidor seguro, eficiente e extensível que fornece serviços HTTP em sincronia com os padrões HTTP atuais.

O servidor HTTP Apache ("http") foi lançado em 1995 e tem sido o servidor *web* mais popular da Internet desde abril de 1996. Em fevereiro de 2015 o projeto celebrou seu aniversário de 20 anos.

Figura 38 – Logo Enterprise Architect.

Fonte: https://httpd.apache.org

Esta ferramenta foi utilizada para disponibilizar o microserviço para seus clientes. Ele foi escolhido para o projeto devido a expertise dos integrantes do grupo com a ferramenta, além de ser o servidor *web* com maior comunidade, o que facilita contornar eventuais dificuldades.

Informações tiradas do site do fabricante: https://httpd.apache.org

5.2 PROCESSO DE DESENOLVIMENTO (HISTÓRICO)

Durante o processo de estabelecimento de novos paradigmas da empresa onde os integrantes desse projeto trabalham, houve a necessidade de uma experiência com a estrutura de microserviços, esta experiência foi nos proporcionada. O desafio era migrar uma funcionalidade da aplicação monolítica para um microserviço, o que originou o desenvolvimento do projeto deste trabalho.

Para que a migração fosse feita, foi necessário avaliar a estrutura da funcionalidade anterior – código legado, dependências internas e externas, regras de negócio – para que a melhor estratégia fosse traçada.

Após o entendimento da estrutura desta funcionalidade e separação conceitual do domínio de negócio que seria inerente ao microserviço, se iniciou o processo de modelagem da arquitetura proposta. Devido a empresa já possuir licenças da ferramenta Enterprise Arquitect, este foi o *software* utilizado para modelagem do novo microserviço.

Com a modelagem do microserviço já estabelecida era necessário entender os conceitos desejados para o desenvolvimento do microserviço. Para isso, os integrantes desse projeto efetuaram pesquisas nas áreas de microserviços, TDD, DDD e *clean code*. Os motivos que levaram a um melhor entendimento sobre microserviços são devidos à arquitetura que o novo componente terá. A pesquisa sobre TDD, surgiu devido a necessidade de prevenção de erros futuros no projeto, garantindo mais qualidade para os componentes que irão consumir o microserviço. As pesquisas relacionadas à DDD foram para entender se aplicação que estava em desenvolvimento de fato respeita os conceitos de domínio de negócio, já que este é um dos pontos mais críticos no desenvolvimento de um microserviço.

Por fim, a pesquisa voltada à *clean code* se deu à busca dos integrantes do projeto por desenvolver uma ferramenta de fácil manutenção e de código leve, para que os conceitos de microserviço fiquem evidenciados não só na teoria, mas também na prática da aplicação.

Durante o desenvolvimento do microserviço os integrantes do projeto se depararam com algumas dificuldades com o *framework* spring boot – devido à falta de experiência no desenvolvimento utilizando esta ferramenta.

Estas dificuldades foram solucionadas a partir de conversas e *pair programming* com o orientador disponibilizado pela empresa, além de pesquisas feitas em fóruns na *internet*.

Como o desenvolvimento do projeto se deu fora do horário de trabalho, houve situações de demora e/ou dificuldade de comunicação com membros mais experientes da equipe, para solucionar esta dificuldade foram enviados e-mails e marcadas reuniões após o surgimento de dúvidas mais críticas.

Durante o desenvolvimento do microserviço, foram feitos alguns ajustes no código fonte, devido a todos os *commits* passarem pelo Gerrit e serem aprovados pelo arquiteto de *softwares* da equipe – orientador disponibilizado pela empresa –, além de serem submetidos à verificação de qualidade de código da ferramenta SonarQube.

Após a conclusão do projeto, devido a utilização de ferramentas que facilitaram o desenvolvimento e a utilização de uma ferramenta de integração contínua – Jenkins – não houveram problemas na realização de *deploys* da aplicação desenvolvida no servidor de *web*.

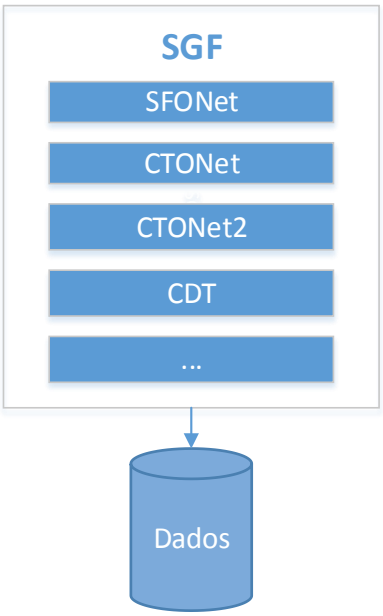
5.3 APRESENTAÇÃO DO MICROSERVIÇO

Nesta seção, são demonstradas as principais vantagens da arquitetura de microserviços para a empresa.

5.3.1 Arquitetura inicial

Conforme a figura 39 representa a arquitetura inicial do projeto é contemplada por um monólito, onde a funcionalidade escolhida pelos integrantes do projeto, fazia parte do corpo do projeto como um todo, sendo assim, integrando a solução.

Figura 39 – Arquitetura inicial do projeto.



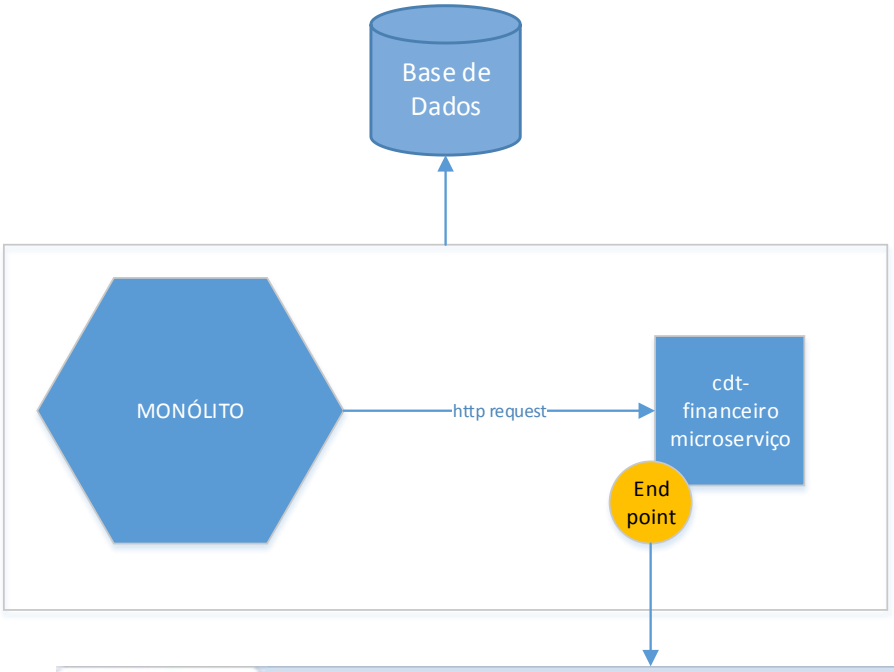
Fonte: Autoria própria.

Pode-se ver na figura 39 que a funcionalidade que será migrada para microserviço está no mesmo contexto dos demais módulos. Na arquitetura inicial, a separação de domínios de negócio acaba se perdendo, não respeitando o conceito de DDD, tanto em nível de código, porque o módulo SFONet é dependente dos módulos CTONet e CTONet2, sendo a reciproca verdadeira. Também há acoplamento do banco de dados da aplicação.

5.3.2 Arquitetura atual

A figura 40 mostra a arquitetura após a construção do microserviço escolhido pelos integrantes do projeto, onde as funcionalidades do sistema que consomem as informações referentes ao módulo migrado foram alteradas para consumir a partir do novo microserviço criado.

Figura 40 – Arquitetura atual do projeto.



IndiceController.java

```java
package br.com.softplan.sider.controller;

import java.util.List;

@RestController
@RequestMapping(value = "/indice")
public class IndiceController {

    @Autowired
    private IndiceService indiceService;

    @RequestMapping(method = RequestMethod.POST)
    public Indice save(@PathVariable Indice indice) {
        this.indiceService.save(indice);
        return indice;
    }

    @RequestMapping(method = RequestMethod.PUT)
    public void edit(@PathVariable Indice indice) {
        indiceService.edit(indice);
    }

    @RequestMapping(value = "/{cdIndice}", method = RequestMethod.DELETE)
    public void delete(@PathVariable Integer cdIndice) {
        indiceService.delete(cdIndice);
    }

    @RequestMapping(method = RequestMethod.GET)
    public List<Indice> findAll() {
        return indiceService.findAll();
    }

    @RequestMapping(value = "/{cdIndice}", method = RequestMethod.GET)
    public Indice findByPK(@PathVariable Integer cdIndice) {
        return indiceService.findByPK(cdIndice);
    }

    @RequestMapping(value = "/findAllIndicesReajustaveis/{cdSistema}", method = RequestMethod.GET)
    public List<Indice> findAllIndicesReajustaveis(@PathVariable Integer cdSistema) {
        return this.indiceService.findAllIndicesReajustaveis(cdSistema);
    }

    @RequestMapping(value = "/findAllIndicesNaoMoedasAtivos", method = RequestMethod.GET)
    public List<Indice> findAllIndicesNaoMoedasAtivos() {
        return this.indiceService.findAllIndicesNaoMoedasAtivos();
    }
}
```

Fonte: Autoria própria.

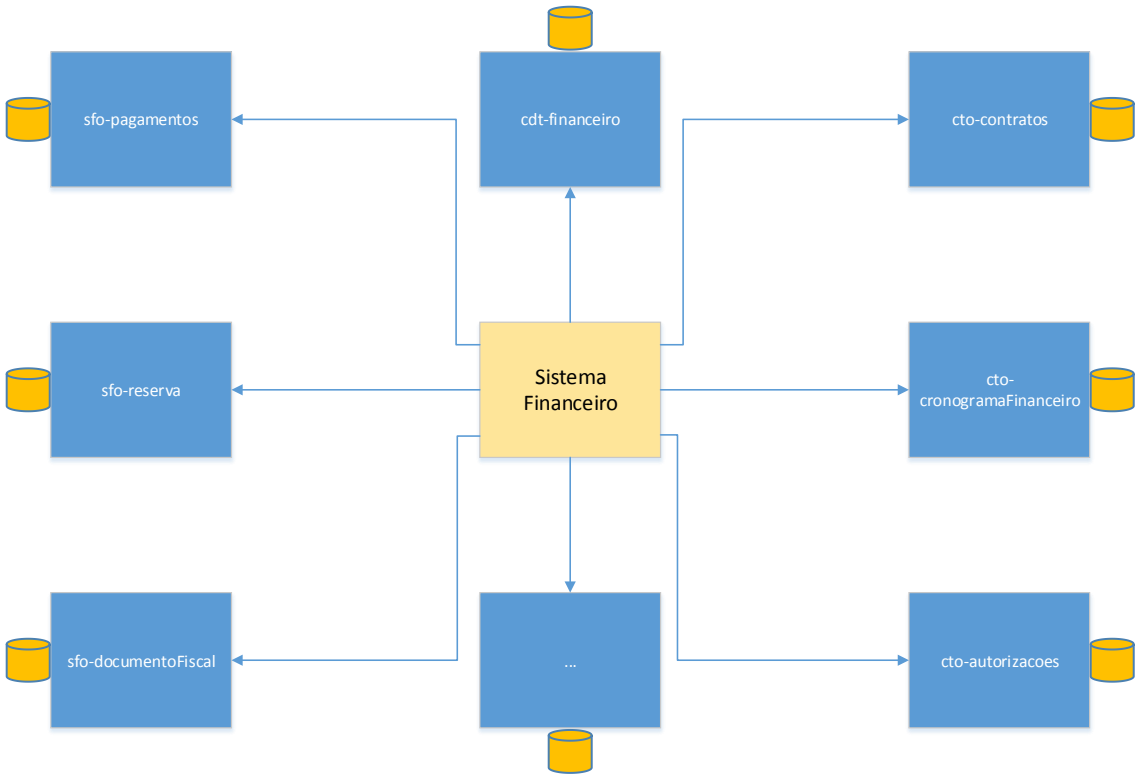
Como apresentado na figura 40, agora o sistema consome as informações relacionados ao financeiro, utilizando conexões HTTP, desta maneira o microserviço é um projeto pequeno e com o escopo reduzido, tornando o domínio de negócio bem definido.

O código fonte apresentado na figura 36 é relacionado ao controlador da entidade índice, que representa a camada de comunicação deste entidade. Todas as outras entidades que disponibilizam serviços para serem consumidos, possuem um *endpoint*.

5.3.3 Arquitetura proposta

A figura 41 apresenta a arquitetura de microserviços desejada, onde os domínios são bem definidos, os escopos são reduzidos e os projetos pequenos, facilitando a comunicação entre eles e tornando a manutenção destes projetos mais simples.

Figura 41 – Arquitetura proposta.



Fonte: Autoria própria.

Pode-se ver na figura 41 que a arquitetura baseada em microserviços apresenta sistemas independentes, o que minimiza pontos únicos de falha. Com a arquitetura proposta também se pode observar a presença de conceitos como, DDD, facilidade na entrega contínua e na integração contínua, além de todas as características relacionadas a microserviços já apresentadas neste trabalho.

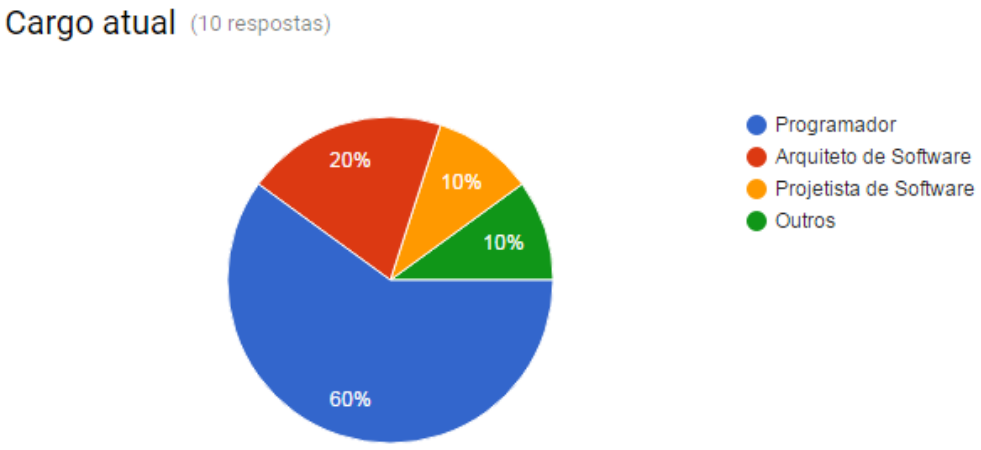
5.4 AVALIAÇÃO DO MICROSERVIÇO

Para avaliação do microserviço foi estruturado um questionário pela ferramenta Google Forms com questões voltadas a arquitetura utilizada no projeto desenvolvido neste trabalho. As perguntas eram relacionadas a conceitos de microserviços, domínio de negócio, usabilidade do projeto desenvolvido e aplicabilidade da nova arquitetura de software. O questionário que contém 6 perguntas fechadas e uma aberta foi submetido a 10 colaboradores da empresa que utilizaram o microserviço desenvolvido. O resultado obtido foi o seguinte.

Os entrevistados possuem uma média de idade de 28,5 anos e trabalham na área de tecnologia em média a 7,4 anos.

A primeira pergunta era relacionada ao cargo atual do entrevistado.

Figura 42 – Respostas da pergunta número 1 em forma de gráfico.



Fonte: Autoria própria por Google Forms.

Conforme a figura 42 mostra dos 10 entrevistados 6 eram programadores, 2 arquitetos de *software*, 1 projetista de *software* e 1 possui outro cargo não técnico.

A segunda questão era relacionada a aceitação da utilização da arquitetura de microserviços

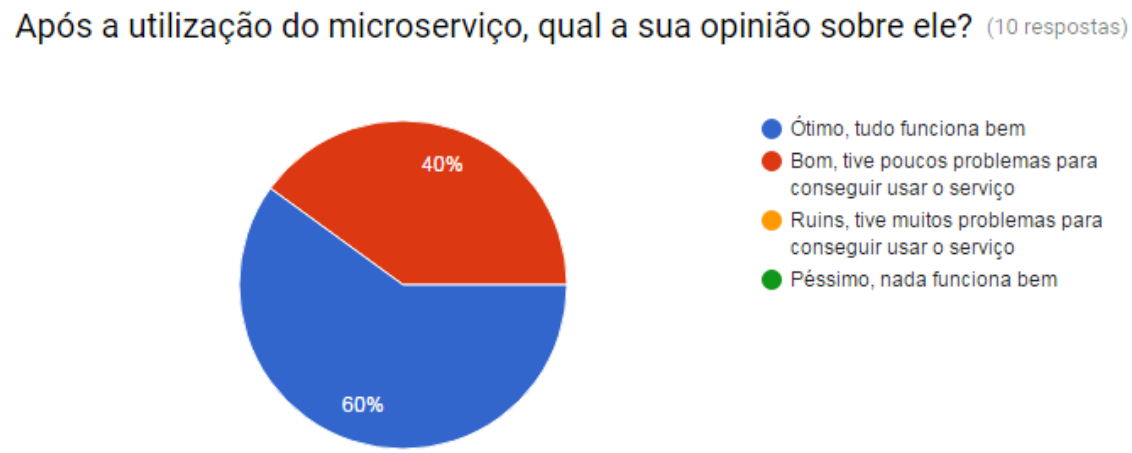
Figura 43 – Respostas da pergunta número 2 em forma de gráfico.

Fonte: Autoria própria por Google Forms.

Como podemos ver na figura 43 10% dos entrevistados são a favor da utilização da arquitetura de microserviços e os outros 90% são a favor, porém dependendo do tipo de projeto.

A terceira questão era relacionada ao microserviço desenvolvido. Após a utilização do microserviço o entrevistado deu a sua opinião a respeito do seu funcionamento.

Figura 44 – Respostas da pergunta número 3 em forma de gráfico.

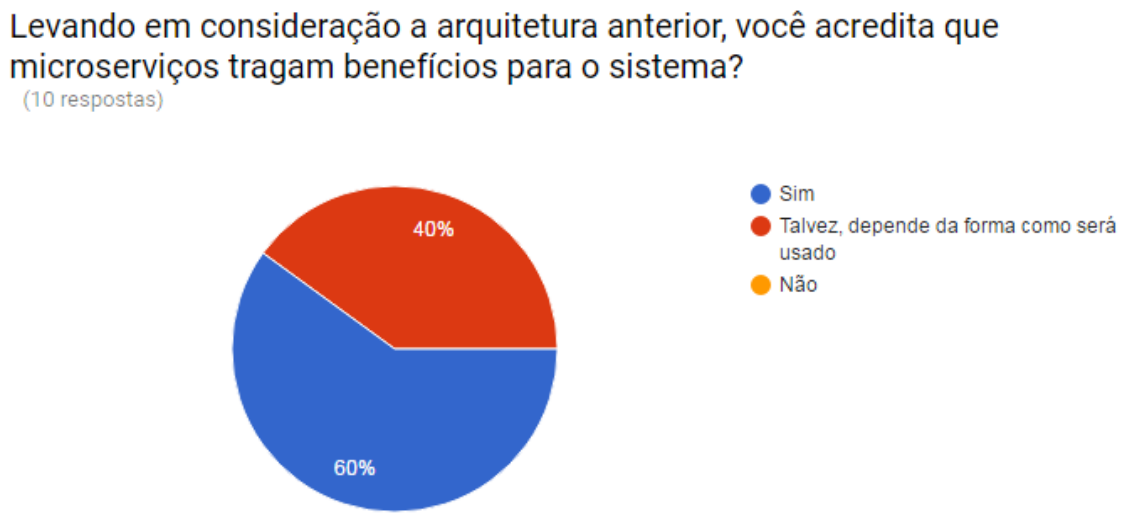


Fonte: Autoria própria por Google Forms.

Como mostra a figura 44 acima, 60% dos entrevistados não teve problema algum para utilizar o microserviço, já 40% conseguiu utilizar com poucos problemas.

A quarta pergunta era relacionada a arquitetura inicial do sistema em comparação a arquitetura de microserviços proposta neste trabalho.

Figura 45 – Respostas da pergunta número 4 em forma de gráfico.

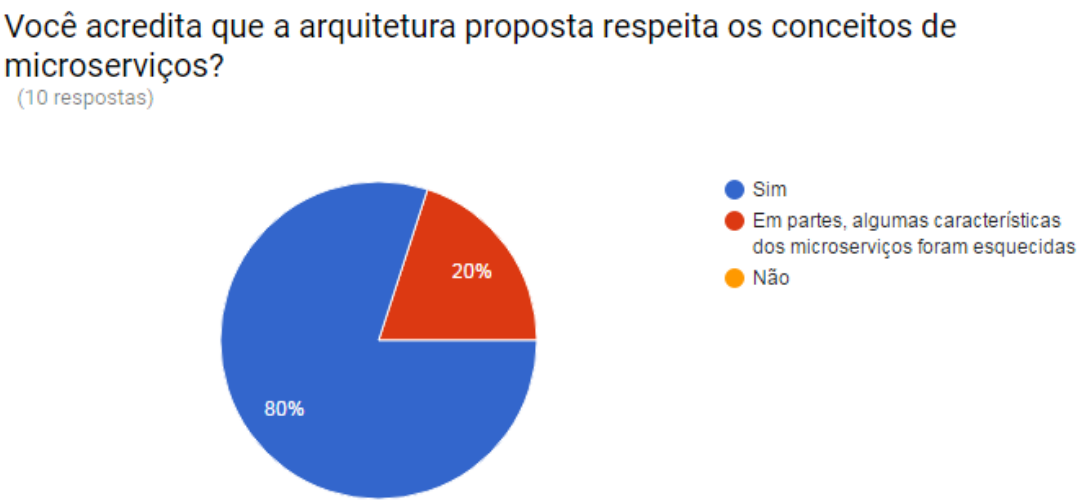


Fonte: Autoria própria por Google Forms.

Dos 10 entrevistados 6 acreditam que a arquitetura de microserviços traz beneficios ao sistema em comparação a arquitetura anterior, já 4 dos intrevistados acredita que os beneficios trazidos pela arquitetura de microserviços dependerá da forma como estes serão usados.

A quinta questão era relacionada aos conceitos de microserviços aplicados ao projeto e se estes estavam sendo respeitados.

Figura 46 – Respostas da pergunta número 5 em forma de gráfico.

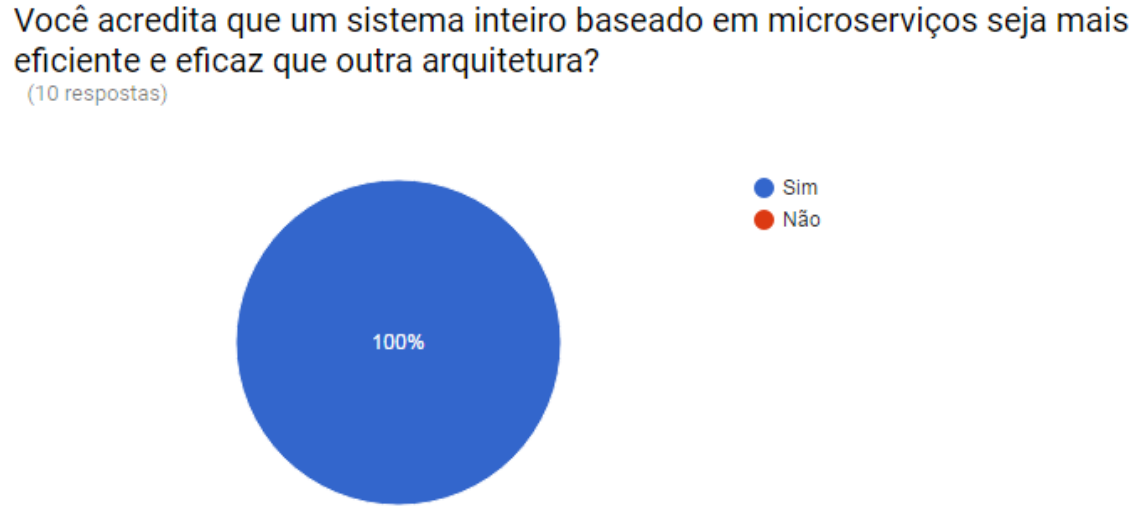


Fonte: Autoria própria por Google Forms.

Como se pode observar na figura 46 8 dos 10 entrevistados acreditam que os conceitos de microserviços foram respeitados no desenvolvimento do novo componente, já 2 dos 10 entrevistados acreditam que algumas caracteristicas foram esquecidas.

A sexta questão era relacionada a opiniao dos entrevistados em relacão a eficiencia e eficacia da arquitetura baseada em microserviços em comparação a outras arquiteturas.

Figura 47 – Respostas da pergunta número 6 em forma de gráfico.

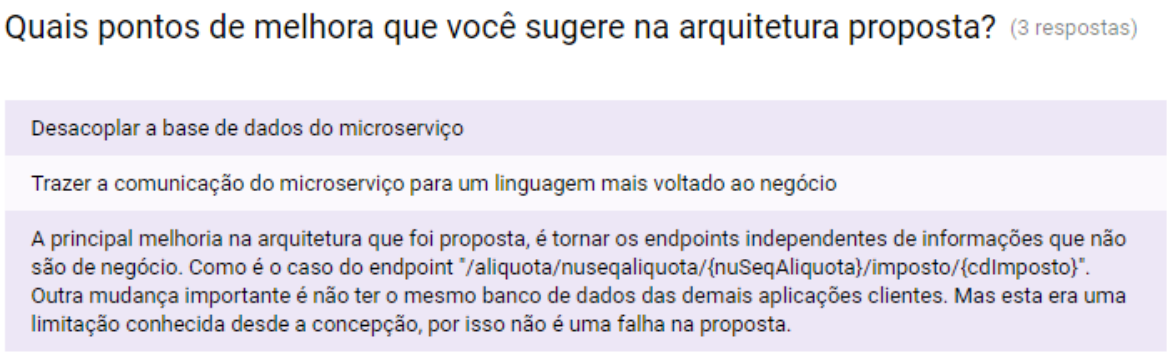


Fonte: Autoria própria por Google Forms.

Como a figura 47 mostra, 100% dos entrevistados concordam que uma arquitetura baseada em microserviços é mais eficiente e eficaz que outras arquiteturas.

A última questão do questionário era aberta e opcional, relacionada aos pontos de melhora que o entrevistado observou no microserviço desenvolvido.

Figura 48 – Respostas da pergunta número 7 de forma dissertativa.



Fonte: Autoria própria por Google Forms.

Como a figura 48 mostra, houve três entrevistados que encontraram pontos de melhora no microserviço desenvolvido, sendo que a última resposta apresentada convergiu as duas primeiras repostas.

6 CONCLUSÃO

Neste capítulo, é apresentado a conclusão do projeto deste trabalho, a partir da visão dos integrantes do projeto. Também será apresentado os trabalhos futuros que os integrantes do projeto pretendem executar utilizando os conceitos apresentados neste trabalho.

6.1 CONCLUSÕES

A realidade encontrada nas empresas que não acompanharam alguns aspectos do avanço tecnológico, principalmente as que possuem bastante código legado com poucos testes e com *softwares* robustos baseados em uma arquitetura monolítica vem ocasionando falta de produtividade, perda de segurança na qualidade das entregas de novas versões e até mesmo dificuldade do time de desenvolvedores para compreender o código legado, gerando atraso nas manutenções.

Com o surgimento de novos conceitos arquiteturais, algumas empresas acompanharam e aplicaram esses conceitos e algumas estão buscando meios de se adequar a essas novas características de arquitetura, que é o caso da empresa onde os integrantes deste projeto trabalham. Percebendo essa nova realidade os integrantes deste projeto se propuseram a idealizar uma prova de conceito sobre a arquitetura baseada em microserviços, respeitando as delimitações da empresa. A proposta foi aceita e então os integrantes começaram a desenvolve-la.

A empresa já utilizava algumas ferramentas de integração e entrega continua, como Jenkins, SonarQube, Git, Gerrit e GitBlit, o que facilitou em partes o desenvolvimento da proposta. Porém, tivemos que nos adequar tanto aos conceitos da arquitetura de microserviços como também em nível técnico para utilização do *framework* e ferramentas utilizadas para o desenvolvimento, exemplo: Postman e SpringBoot, além de definir qual barramento de comunicação utilizado pelo microserviço, que foi o barramento de comunicação já bem consolidado no mercado de desenvolvimento de software que é o baseado em REST.

Durante o desenvolvimento da solução foi notória a diferença na forma de pensar e codificar o microserviço, em comparação a arquitetura anterior. Pois, para cada linha de código escrita sempre era pensado em deixar o código simples e eficaz que são conceitos do *Clean Code*. Também se preocupando com as maneiras de comunicação com o cliente, sempre visando tornar esta simplista. Além desses pontos, desenvolvemos o projeto aplicando o conceito de TDD, ou seja, desenvolvendo testes para toda funcionalidade existente no microserviço.

A principal questão levantada durante a construção da problemática foi "É possível migrar uma solução monolítica para uma arquitetura de microserviços, respeitando o objetivo original da aplicação?". Pode-se constatar com o desenvolvimento desse trabalho que não só é possível migrar as soluções do monólito, como também a nova arquitetura favorece diversos fatores para o desenvolvimento e manutenção do código gerado. Durante o processo de desenvolvimento desse projeto, pode-se verificar também que nem todos os sistemas devem ser desenvolvidos utilizando a arquitetura de microserviço. O principal fator que define qual a melhor arquitetura a ser usada é a robustez e o tamanho do sistema que será feito. Sistemas pequenos e que não possuem uma grande quantidade de acessos, podem ser desenvolvidos utilizando uma arquitetura monolítica, já os sistemas mais robustos, com muitos domínios de negócios, tendem a ter suas características favorecidas pela arquitetura baseada em microserviços.

Após finalizado o projeto respeitando todos os conceitos relacionados a arquitetura de microserviços já citados neste trabalho, e, observando o resultado do questionário, a nova arquitetura proposta foi bem aceita pelo time de desenvolvimento da empresa, e também pelos gestores, devido ao ganho de produtividade.

Concluindo, levando em consideração as respostas observadas no questionário proposto, onde 100% dos entrevistados consideraram o microserviço desenvolvido ótimo ou bom e 60% dos entrevistados consideram a arquitetura de microserviços traz mais benefícios que a arquitetura anterior, podemos afirmar que a arquitetura proposta teve uma boa aceitação e contribuirá para futuras manutenções no código existente e também na criação de novas funcionalidades, seja a nova funcionalidade pertencente a um domínio de negócio de um microserviço já existente ou de um novo domínio de negócio que acarretará na criação de um novo microserviço.

Tal modularização que a arquitetura de microserviços defende permite a comercialização de módulos desagregados, não havendo a necessidade de o cliente receber

em seu código fonte funcionalidades desnecessárias para seu negócio. O fato do cliente receber apenas aquilo que lhe “convém” torna o produto final mais acessível.

6.2 TRABALHOS FUTUROS

Devido a boa aceitação da arquitetura proposta, há o interesse dos integrantes desse projeto em continuar a migração de outras funcionalidades do sistema monólito para microserviços.

Há também o interesse na busca por padrões de projetos para auxiliar no desenvolvimento de novos projetos que utilizem a ferramenta de microserviços ou na migração de funcionalidades de módulos monolíticos para microserviços.

Testar outro padrão de barramento. Assim como outros frameworks de desenvolvimento.

O principal desafio durante o desenvolvimento do projeto foi o de entender os limites de domínio de negócio. Pretende-se com o conteúdo adquirido nesse projeto realizar uma apresentação sobre DDD aos cargos técnicos da empresa, visando melhorar a modularização das futuras soluções.

REFERÊNCIAS

ANICHE, Maurício. **TDD, Test-*Driven* Development**. Disponível em: < http://tdd.caelum.com.br/> Acesso em: 15 jan. 2016

AVRAM, Abel. **The Benefits of Micro*services***. Disponível em: <http://www.infoq.com/news/2015/03/benefits-micro*services*> Acesso em: 1 nov. 2015

AVRAM, Abel. ***Domain-Driven Design* Quickly.** Raleigh, North Carolina, United States: Lulu.com, 2007. 104 p.

BAETJER JUNIOR, Howard. ***Software* as Capital**: An Economic Perspective on *Software* Engineering. Piscataway, Nj: Wiley-ieee Computer Society Press, 1997. 206p.

BEZERRA, Eduardo. **Princípios de análise e projeto de Sistemas com UML**. 2. ed. Rio de Janeiro: Elsevier Editora, 2007. 380 p.

BOAVENTURA, Edivaldo M. **Metodologia da Pesquisa:** Monografia, dissertação, tese. São Paulo. Editora Atlas 2004. 160 p.

CARDOSO, André. **TDD, por que usar?** Disponível em: < http://tableless.com.br/tdd-por-que-usar/> Acesso em: 3 fev. 2016.

CONTINO. **The Benefits Of Micro*Services***. Disponível em: <http://contino.co.uk/the-benefits-of-micro*services*/> Acesso em: 1 nov. 2015.

DEMO, P. **Pesquisa e construção de conhecimento:** metodologia científica no caminho de Habermas. 2. ed. Rio de Janeiro: Tempo Brasileiro, 1996. 125p.

DESLAURIERS, J. P. **Recherche qualitative:** guide pratique, Montreal, McGraw Hill, 150p.

EVANS, Eric; FOWLER, Martin. **Domain-driven Design:** Tackling Complexity in the Heart of Software. Boston: Addison-wesley Professional, 2003. 529 p.

FERRARI, A. T. **Metodologia da pesquisa científica.** São Paulo: McGraw-Hill do Brasil, 1982. 318 p.

FIELDING, Roy Thomas. **Architectural styles and the *design* of network-based software architectures**. 2000. Tese de Doutorado. University of California, Irvine.

FLORAC, W. A.; PARK, R. E.; CARLETON, A. D. (1997). **Practical *software* measurement: measuring for process management and improvement**. Pittsburgh, PA: *Software* Engineering Institute, 1997. 240 p.

FONSECA, J. J. S. **Metodologia da pesquisa científica.** Fortaleza: UEC, 2002. Apostila. 127 p. Disponível em: <http://www.ia.ufrrj.br/ppgea/conteudo/conteudo-2012-1/1SF/Sandra/apostilaMetodologia.pdf> Acesso em: 4 nov. 2015

FOWLER, Martin. **Micro*service* Trade-Offs**. Disponível em: <http://martinfowler.com/articles/micro*service*-trade-offs.html#summary> Acesso em: 1 nov. 2015. A

FOWLER, Martin. **MonolithFirst**. Disponível em: <http://martinfowler.com/bliki/MonolithFirst.html> Acesso em: 27 set. 2015. B

FOWLER, Martin. **UML Distilled: A Brief Guide to the Standard Object Modeling Language**. Boston, MA: Addison-Wesley professional, 2003. 208p.

FOWLER, Martin; LEWIS, James. **Micro*services*, a definition of this new architectural term**. Disponível em: < http://martinfowler.com/articles/micro*services*.html> Acesso em: 27 jul. 2015.

FOWLER, Marin, 2014. **Bounded Context**. Disponível em: < http://martinfowler.com/bliki/BoundedContext.html > Acesso em: 10 fev. 2016.

FUGGETTA, A., 2000, ***Software* Process: A Roadmap**, Disponível em: <http://alarcos.esi.uclm.es/per/fruiz/cur/pso/comple/fuggetta.pdf> Acesso em 17 de out. 2015.

GIL, Antônio Carlos**. Métodos e técnicas de pesquisa social.** 6 ed. São Paulo: Atlas, 2008. 216 p. Disponível em: <https://ayanrafael.files.wordpress.com/2011/08/gil-a-c-mc3a9todos-e-tc3a9cnicas-de-pesquisa-social.pdf> Acesso em: 4 nov. 2015.

GUEDES, Gilleanes T. A. **UML 2: Uma Abordagem Prática**. 1. ed. São Paulo: NovaTec Editora, 2009. 488 p.

GUPTA, Arun. **Getting Started With Micro*services*.** Disponível em: <https://dzone.com/refcardz/getting-started-with-micro*services*> Acesso em: 10 out. 2015.

HAFEMANN, Lodemar José. **Protótipo de Gerador de Código Fonte baseado em Diagramas de Sequencias.** 2000. 64 f. TCC (Graduação) - Curso de Ciência da Computação, Universidade Regional de Blumenau, Blumenau, 2000. Disponível em: < http://dsc.inf.furb.br/arquivos/tccs/monografias/2000-1lodemarjosehafemannvf.pdf > Acesso em: 2 fev. 2016.

HAGLER Dave. **On monoliths, *service*-oriented architectures and micro*services***. Disponível em: <http://odino.org/on-monoliths-*service*-oriented-architectures-and-micro*services*> Acesso em: 26 set. 2015.

HUMPHREY, W. S. **Managing the *Software* Process**, New York: Addison-Wesley, 1989. 512 p.

IBM (EUA). **Micro*services* from Theory to Practice:** Creating Applications in IBM Bluemix Using the Micro*services* Approach. Armonk: Redbooks, 2015. 148 p.

JALOTE, Pankaj. ***Software* Project Management in Practice**. Boston, MA: Addison-wesley Professional, 2002. 288 p.

JUNIOR, José. **MongoDB: sharding e mapReduce().** Disponível em: <http://*core*.eti.br/2010/11/14/mongodb-sharding-e-mapreduce> Acesso em: 11 out. 2015.

LAKATOS, E. M.; MARCONI, M. A. **Fundamentos da Metodologia Científica.** São Paulo: Atlas, 1986. 238p.

NAMIOT Dmitry; SNEPPE Manfred. **On Micro-*services* Architecture**. International Journal of Open Information Technologies, 2(9), 2014.

NEWMAN, Sam. **Building Micro*services***. Sebastopol: O'Reilly Media, 2015. p. 280.

PARMAR, Ketan. **Monolithic vs Micro*service* Architecture**. Disponível em: <https://www.linkedin.com/pulse/20141128054428-13516803-monolithic-vs-micro*service*-architecture> Acesso em: 27 set. 2015.

PEMBER, Steve. **Monolithic Architecture Doesn't Scale!** Disponível em: <http://cantina.co/monolithic-architecture-doesnt-scale> Acesso em: 27 set. 2015.

PRESSMAN, Roger S. **Engenharia de *software*.** 5. ed. São Paulo: Makron Books, 2002. 872p.

PRESSMAN, Roger S. **Engenharia de *Software*.** 6. ed. São Paulo: Mcgraw-hill, 2006. 720p.

PRESSMAN, Roger S. **Engenharia de *Software***: uma abordagem profissional. 7. ed. Porto Alegre: AMGH, 2011. 780p.

RAMOS, R. A. **Treinamento Prático em UML**. 1. ed. São Paulo: Digerati, 2006. v. 1. 144p

RIBEIRO, Leilane Ferreira. **Modelagem de *Software* Utilizando UML: Análise Comparativa Entre as Ferramentas Astah UML e Umbrello UML Modeller.** 2012. 57 f. Tese (Mestrado em Engenharia de Sistemas) - Curso de Pós-Graduação em Engenharia de Sistemas, Escola Superior Aberta do Brasil, Espírito Santo, Vila Velha. Disponível em: < http://www.esab.edu.br/wp-content/uploads/monografias/leilane-ferreira-ribeiro.pdf> Acesso em: 2 fev. 2016.

RICHARDSON, Chris. **Micro*services*: Decomposição de Aplicações para Implantação e Escalabilidade**. Disponível em: < http://www.infoq.com/br/articles/micro*services*-intro> Acesso em: 2 nov. 2015 A.

RICHARDSON, Chris. **Pattern: Monolithic Architecture**. Disponível em: <http://micro*services*.io/patterns/monolithic.html> Acesso em: 27 set. 2015 B.

RICHARDSON, R. J. **Pesquisa social:** métodos e técnicas. 3. ed. São Paulo: Atlas, 1999. 334p.

ROSA, Thiago Pereira. **UM MÉTODO PARA O DESENVOLVIMENTO DE *SOFTWARE* BASEADO EM MICROSERVIÇOS.**2015. 30 f. TCC (Graduação) - Curso de Engenharia de *Software*, Universidade Federal do Ceará, Quixadá, 2015.

SAMPAIO, M. C. **História de UML**. [s.n.], 2007. Disponível em: <http://www.dsc.ufcg.edu.br/sampaio> Acesso em: 3 fev. 2016.

SAUDATE, Alexandre. **REST**: Construa API's inteligentes de maneira simples. São Paulo: Casa do Código, 2014. 314 p.

SILVA, Edna Lúcia da. **Metodologia da pesquisa e elaboração de dissertação/Edna Lúcia da Silva, Estera Muszkat Menezes.** 4. ed. Florianópolis: UFSC, 2005. 138 p.

SOMMERVILLE, Ian. **Engenharia de *software***. 8ª ed. São Paulo: Pearson Addison-Wesley, 2007. 552 p.

SOMMERVILLE, Ian. **Engenharia de *Software***. 9. ed. São Paulo: Pearson Education, 2011. 568 p.

STENBERG, Jan. **Micro*services* and Evolutionary Architecture**. Disponível em: < http://www.infoq.com/news/2015/03/qcon-micro*services*-architecture> Acesso em: 5 nov. 2015.

VARGAS, Thânia C. de Souza. **A História da UML e seus Diagramas**. Disponível em: <https://projetos.inf.ufsc.br/arquivos_projetos/projeto_721/artigo.tcc.pdf> Acesso em: 10 fev. 2016.

VUORINEN, Carl, 2014. **What is Clean Code and why should you care?** Disponível em: < http://cvuorinen.net/2014/04/what-is-clean-code-and-why-should-you-care/ > Acesso em: 10 jul. 2015.

SILVA, Alberto; VIDEIRA, Carlos. **UML - Metodologias e Ferramentas CASE**. 1. ed. Lisboa: Centro Atlântico, 2001. 284 p.

WEINRICH, Jair; GRAHL Everaldo**. *Software* de apoio a avaliação e seleção de ferramentas case baseado na norma ISO/IEC 14102.** Artigo SEMINCO FURB - Universidade Regional de Blumenau, 1999.

MARTIN, Robert; FEATHERS, Michael; OTTINGER, Timothy. **Clean Code: A Handbook of Agile Software Craftsmanship.** 4. ed. Nova Jersey: Prentice Hall, 2009. 431 p.

APÊNDICES