UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

FABRIZIO SILVA MARMITT

REENGENHARIA VISANDO ARQUITETURA DE MICROSSERVIÇOS: UM ESTUDO DE CASO

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do grau de Especialista em Engenharia de Software

Prof. Dr. Marcelo Soares Pimenta
Orientador

Porto Alegre, maio de 2017

AGRADECIMENTOS

Agradeço aos professores do Instituto de Informática da UFRGS pelos conhecimentos compartilhados que agregaram muito em minha vida profissional nestes últimos 2 anos.

Agradeço especialmente ao professor Marcelo Soares Pimenta que aceitou me orientar durante a trajetória do trabalho de conclusão.

Agradeço a minha esposa pela paciência nos momentos em que tive me dedicar no curso e pelo apoio incondicional.

Agradeço aos meus amigos e colegas Junior, Robson e Tiago que compartilharam a jornada do curso de especialização em engenharia de software.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
AS	ASsinturas
HTML	HyperText Markup Language
HTTP	Hyperterxt Transfer Protocol
IP	Internet Protocol
IVC	Instituto Verificador de Comunicação
JDBC	Java Database Connectivity
LAN	Local Area Network
LoC	Lines of Code
ONG	Organizações não governamentais
REST	Representational State Transfer
ROI	Return on Investment
SO	Sistema Operacional
SQL	Structured Query Language
TCP/IP	Transmission Control Protocol
URI	Uniform Resource Identificator
XP	eXtreme Programing

LISTA DE FIGURAS

LISTA DE QUADROS

RESUMO

Este trabalho apresenta um projeto em que foi realizada a reengenharia de sistema legado monolítico para uma arquitetura de microsserviços, bem como descreve a plataforma tecnológica necessária para isso.

O trabalho mostra como é possível realizar a reengenharia de forma incremental em que o sistema legado e o novo convivam durante o processo de migração.

O trabalho foi aplicado em um sistema de gerenciamento de assinaturas de uma importante empresa jornalística no sul do país, e que é considerado um sistema legado.

O projeto envolveu uma engenharia reversa para obtenção do conhecimento necessário para sua reengenharia e, posteriormente, um novo *design* e uma nova implementação em que uma arquitetura de microsserviços foi proposta. Este trabalho também discute como é possível fazer a migração de todo o sistema tomando como base a implementação do primeiro módulo, o de cancelamento e suspensão, e traz uma discussão sobre os resultados obtidos nos capítulos anteriores.

Por fim, a conclusão retrata os resultados obtidos e as limitações deste trabalho.

Palavras-Chave: reengenharia incremental, engenharia reversa, sistemas legados, microsserviços.

1 INTRODUÇÃO

A mudança no software é algo quase inevitável. Em relação aos motivos em que estas mudanças ocorrem, destaca-se de forma principal, mas não exclusiva, a necessidade de adaptabilidade deste software com as novas demandas de negócio em que está inserido. A mudança contínua de um software, como descrita por Lehman (1980) é uma necessidade pois, caso contrário, um software se torna progressivamente menos útil a seus usuários.

Além da necessidade de adaptabilidade, os softwares mudam devido a erros, falhas e necessidade de novas funcionalidades. Estas mudanças podem ser classificadas em mudanças corretivas, mudanças para adaptação ou manutenção evolutiva. Independentemente do tipo de mudança, Rajlich e Bennett (2000) verificaram que pelo menos 50% do custo total do software está em mudanças ou manutenções, isto é, manter um software é geralmente mais caro do que desenvolvê-lo.

Um problema recorrente neste cenário é o aumento da complexidade de grandes softwares devido a frequentes mudanças que deterioram sua estrutura. A deterioração intrínseca de um software ao longo do tempo denomina-se entropia do software e este processo resulta em um software legado.

Neste trabalho, será analisado a situação de um importante sistema de gerenciamento de assinaturas que possui mais de vinte anos de existência de uma empresa do setor jornalístico situada no sul do Brasil. Este sistema vem sofrendo muitas mudanças desde seu primeiro *release* e hoje está no estágio de *servicing*, de acordo com a classificação de Rajlich e Bennett (2000). Nos últimos anos, com a popularização das novas mídias digitais para consumo de conteúdo jornalístico na internet, este sistema sofreu alterações substanciais em sua estrutura, pois a forma de comercializar noticiais no meio digital é diferente do meio impresso. Como consequência, o sistema apresenta seguidos *bugs* decorrentes de alterações adaptativas e evolutivas e, além disso, toda nova alteração no sistema possui um custo e tempo de desenvolvimento elevado.

Entre os motivos que justificam o elevado número de *bugs*, custo e tempo de desenvolvimento destacam-se o *design* ou arquitetura do sistema que não foi pensado para ser modificado; o déficit técnico dos diversos programadores que alteraram o sistema

no decorrer dos anos, ocasionando uma degradação do código-fonte; falta de documentação; falta de preocupação em refatoração e implementação de padrões de *design*, entre outros. Contudo, apesar dos problemas, evoluir o software continua sendo uma necessidade e para isso é preciso de um projeto ou plano de ação para fazer com que este software deixe de se degradar e atenda de forma rápida, eficiente e sem *bugs* o negócio ao qual serve.

Diversos padrões de *design* existem para solucionar problemas recorrentes e conhecidos no desenvolvimento de software. Arquitetura de microsserviços é um tópico que se popularizou recentemente, apesar de seus conceitos serem conhecidos desde os anos 70. Em resumo, o termo arquitetura de microsserviços descreve uma forma de projetar softwares como um conjunto de serviços (ou micro softwares) com implantação independente, cada um com seu próprio processo e comunicação com mecanismos simples e leves como HTTP.

Nos últimos anos, a adoção da arquitetura de microsserviços tem demonstrado bons resultados em empresas e negócios que mudam muito frequentemente, como o caso do Netflix, Amazon e Google. Analisando os objetivos de negócios apresentados pela empresa jornalística que consiste em 1) reduzir o *time-to-market* de projetos através do desenvolvimento rápido de sistemas, 2) alta escalabilidade para contemplar picos de demanda, 3) baixo custo, 4) efemeridade e 5) alta disponibilidade; verifica-se a grande aderência de uma arquitetura de microsserviços para a empresa.

Entretanto, adotar uma nova arquitetura implica em muitas mudanças no desenvolvimento dos softwares existentes, bem como suas integrações. Colocar os softwares atuais no lixo e começar do zero não é uma opção. Assim, o presente trabalho visa propor um projeto para implantação de uma arquitetura de microsserviços, a partir dos sistemas legados existentes através da proposição de plataformas, ferramentas, práticas e metodologias necessárias para isso.

1.1 Objetivo principal

Propor uma plataforma tecnológica que viabilize a reengenharia de sistemas monolíticos para uma arquitetura de microsserviços.

1.2 Objetivos secundários

- Estudar modelos para reengenharia de sistemas legados e definir metodologia e modelo de adoção de microsserviços.
- Estudar diferentes plataformas para microsserviços disponíveis no mercado.
- Definir infraestrutura e aplicações necessárias para implantação de arquitetura de microsserviços.
- Planejar a reengenharia de um componente de um sistema para microsserviços
- Documentar essa experiência para dar continuidade à migração dos sistemas legados da empresa.

1.3 Estrutura do trabalho

Este trabalho está dividido em 8 capítulos: introdução, revisão da literatura, metodologia, engenharia reversa, reengenharia, implementação do sistema, discussão e conclusão. Além destes capítulos, há a bibliografia ao final.

No Capítulo 1, o contexto em que o trabalho está inserido, bem como a sua justificativa são apresentados. Outro elemento importante deste capítulo, são os objetivos principal e secundários que se busca atingir.

O segundo Capítulo destaca conceitos relevantes para o entendimento e andamento deste trabalho, e resgata os conceitos sobre o ciclo de vida do desenvolvimento de software, os benefícios de uma arquitetura evolutiva e reengenharia de software. Após, dois outros conceitos centrais deste trabalho são apresentados que são arquitetura monolítica e arquitetura de microsserviços. Por fim, uma descrição geral da arquitetura de microsserviços é apresentada.

Na sequência, no Capítulo 3, a metodologia destaca como o trabalho foi conduzido para chegar nos objetivos almejados.

O Capítulo 4 expõe toda e engenharia reversa realizada na implementação do sistema de assinaturas a descreve a sua arquitetura, problemas através da análise de *code smells*, além dos requisitos e modelo conceitual.

No Capítulo 5, inicia-se a reengenharia a partir dos dados obtidos no capítulo anterior, e toda a proposta para o novo sistema é apresentado.

Já no Capítulo 6, são expostos detalhes sobre a nova implementação utilizando microsserviços e integração com o sistema legado.

O Capítulo 7 traz uma discussão sobre os resultados obtidos nos capítulos anteriores a faz uma extrapolação para estimar a migração de todo o sistema.

Por fim, a conclusão retrata os resultados obtidos e limitações deste trabalho.

2 REVISÃO DA LITERATURA

Neste capítulo estão descritos os principais conceitos que fundamentam este trabalho.

2.1 Ciclo de vida do desenvolvimento de software

Evolução é um processo de mudança progressiva no tempo nas propriedades, atributos, características ou comportamento. O fato é que evolução, em geral, torna as coisas mais complexas.

Um software é basicamente composto por 5 artefatos, a saber: requisitos, *design*, implementação, verificação e manutenção. Cada um destes artefatos evolui de forma diferente. A inconsistência de mudanças entre cada artefato consiste em um problema denominado problema da coevolução.

Um software entra em decadência quando as habilidades humanas e a integridade arquitetural são perdidas. O software se torna, então um software legado. Durante esta fase, é difícil e caro fazer alterações, ao mesmo tempo em que o software desempenha uma função crucial para a organização em que está inserido.

Sintomas típicos da decadência do software são:

- Complexidade excessiva no código-fonte. Código é mais complexo do que precisa ser;
- Excesso de código vestigial que não é mais utilizado ou não é mais necessário;
- Necessidade de mudanças corretivas frequentes;
- Presença de muitos bugs;
- Mudanças deslocalizadas são frequentes, ou seja, mudanças que afetam partes do código não previstas ou mapeadas;
- Excesso de *kludges* ou *gambiarras*, isto é, mudanças feitas de forma deselegante ou de maneira ineficiente, porém rápida (ex.: duplicação de trechos de código).

Excesso de dependências no código. A medida em que o número de dependências aumenta, efeitos secundários de mudanças tornam-se mais frequentes e possivelmente introduzem erros inesperados ao programa.

É importante, portanto, que todos os artefatos de software coevoluam, como por exemplo, código-fonte e objetos de banco de dados, *design* de código e documentação, código e testes, código e linguagem de programação, código é softwares terceiros, entre outros. A coevolução garante consistência no sistema.

Sistemas evolucionários, são aqueles que são constantemente modificados e os estágios do ciclo de vida destes sistemas são: desenvolvimento inicial, evolução, manutenção e morte.

Sistemas evolucionários são mais presentes em ambientes que estão em um processo contínuo de mudança. São exemplos empresas digitais, *start-ups* e setores que sofreram uma mudança de paradigma. Sistemas que sofrem alterações constantes geram os chamados "sistemas legados", que são definidos por Bennet (2002) como "sistemas essenciais para a organização, mas que não se sabe o que fazer com ele".

Algumas conclusões foram tomadas sobre o que fazer com sistemas legados, apesar de não existir bala de prata para solução desta situação. Bennet (2002) conclui que são parte importante do processo de evolução de sistemas legados:

1. Experiência humana e eliminação de débito técnico;
2. Amplo conhecimento sobre o contexto em que está inserido;
3. Desenhar uma arquitetura evolutiva

2.2 Arquitetura evolutiva

A mudança de software precisa ser planejada. Arquitetura evolutiva é um termo utilizado para refletir a necessidade de construir um software que certamente será alterado no futuro e que isto não deve ser encarado como um problema.

Contudo, desenhar uma arquitetura capaz de evoluir constantemente com a menor degradação possível é uma tarefa árdua. Muitos sistemas que existem há muitos anos, foram desenvolvidos como um conjunto de componentes que foram sendo alterados no decorrer do tempo e, apesar disso, não comprometeram a arquitetura inicial. Criar uma arquitetura evolucionária depende de muito conhecimento da equipe e liderança.

A degradação do software ocorre por vários motivos, mas entre os mais listados pela literatura destacam-se as pressões financeiras para tomar atalhos no desenvolvimento e entregar mudanças mais rápido, ignorando os potenciais conflitos arquiteturas que esta decisão pode acarretar. Por esse motivo, é importante pessoas com alta habilidade para entregar rápido e mesmo assim manter os padrões de software e arquitetura.

Para Bennett (2002), não existe uma resposta simples para fazer a arquitetura de um sistema evoluível ou perfectível. Analisando de forma pragmática projetos de software que estão a muito tempo no mercado, como UNIX, tipicamente mostra que o *design* ou arquitetura inicial foi realizado por um, ou alguns, indivíduos muito talentosos.

Diversas abordagens para desenvolvimento evolucionário surgiram, como os métodos ágeis. Um exemplo é o *eXtreme Programming* (XP) ou Programação Extrema. O XP basicamente elimina a fase inicial de desenvolvimento, programadores trabalham próximo à área de negócios e escrevem histórias de forma evolutiva e interativa. O software é lançado em etapas denominadas *releases* e após cada *release* uma nova é planejada. Casos de testes são definidos antes de iniciar a programação. Neste método, o sistema inicia e termina evoluindo.

O termo desenvolvimento de software evolucionário consiste em um conjunto de práticas em engenharia de software visando a minimização do processo inicial de desenvolvimento, fazendo apenas uma versão inicial ou esquelética do programa. O restante do sistema é implementado no processo de evolução do sistema. Assim como o código, os requisitos também são coletados de forma fracionada.

2.3 Reengenharia de Software

Reengenharia de software é o termo cunhado para representar a renovação ou reconstrução de um software existente, visando corrigi-lo ou melhorá-lo (Bennet, 2002). Segundo Sommerville (2011), empresas tomam decisão relacionadas aos seus softwares baseados no retorno financeira desta decisão. Assim, iniciar um novo sistema do zero dificilmente é uma opção; é preciso acessar o problema do software legado de uma maneira mais realística.

A reengenharia torna-se, então, uma estratégia utilizada para sistemas legados. Geralmente é utilizada quando a qualidade do sistema foi perdida ou degradada devido à frequentes mudanças e pela falta de planejamento em realiza-las.

Entre as vantagens da reengenharia, destacam-se os custos reduzidos, bem como a redução dos riscos. Para estas vantagens serem reais, uma forma de aplicar a reengenharia, é através da modularização e implantação incremental. Uma vez que a reengenharia pode ser aplicada a partes do sistema, os riscos de má especificação e custos reduzem drasticamente.

2.4 Arquitetura Monolítica

A arquitetura monolítica é a arquitetura de software tradicional no mercado de TI e, também, a mais simples. Geralmente é composta por um único componente que é executado, como Jar, Zip, Exe, etc.

Segundo Gupta (2015), entre as vantagens de se utilizar aplicações monolíticas destacam-se:

- Amplamente utilizado no mercado: é o padrão arquitetural mais utilizado e por isso possui mais profissionais que o conhece além de possuir maior número de ferramentas disponíveis.
- IDE-*friendly*: a grande maioria dos IDEs disponíveis no mercado suportam muito bem esta arquitetura.
- Fácil compartilhamento: um único arquivo (e.g: zip, war, ear) é facilmente compartilhado entre times e através distintas fases do processo de desenvolvimento.
- Teste simplificado: uma vez que uma aplicação foi publicada, todos os serviços e funcionalidades estão disponíveis. Isto significa que para testar a aplicação não existem dependências adicionais, basta iniciar os testes.
- Fácil publicação: geralmente a publicação de uma aplicação monolítica é simplesmente copiar um arquivo de um diretório para o outro, o que é realmente muito simples.

Para o mesmo autor, as desvantagens de se utilizar uma arquitetura monolítica são:

- Agilidade limitada: cada pequena mudança na aplicação significa republicação de toda a aplicação. Isso diminui a agilidade do time de desenvolvimento e a frequência que novas funcionalidades podem ser entregues.

- Obstáculo para integração contínua: quando uma aplicação é pequena e simples, compilar e publicar são processos rápidos e fáceis. Contudo, aplicações reais são muito grandes e possuem diversos serviços e a compilação e publicação podem demorar horas para serem concluídas.
- Prisão tecnológica: a escolha das tecnologias a serem utilizadas no desenvolvimento de uma aplicação ocorre antes do início do seu desenvolvimento. Contudo, nessa fase, muitas coisas ainda estão incertas e pode-se descobrir no futuro que as tecnologias utilizadas *a priori* não foram as melhores escolhas para aquele contexto. Em uma aplicação monolítica, trocar de tecnologias para determinados serviços não é uma opção. Ou troca-se todo o sistema ou nada é trocado.

2.5 Arquitetura de Microsserviços

Arquitetura de Microsserviços é um termo que se popularizou em engenharia de software e representa uma forma de projetar aplicações favorecendo características do software como alta coesão, baixo acoplamento, autonomia e independência.

Este tipo de arquitetura tornou-se popular sendo uma espécie de antagonista da tradicional arquitetura monolítica, em que todas as partes de um sistema estão inseridas em um mesmo pacote ou executável. Assim, uma aplicação que utiliza uma arquitetura de microsserviço basicamente consiste em um conjunto de micro aplicações monolíticas especializadas em uma ou poucas funções, que pode ser evoluída independentemente do restante do sistema e se comunica com as demais partes do sistema através de APIs baseadas no protocolo HTTP.

Apesar de se popularizar recentemente, a ideia de minimizar software, separando-o em componentes, através da modularização é uma filosofia amplamente conhecida no desenvolvimento do Unix. Em suma, a filosofia Unix enfatiza a construção de softwares como um conjunto de componentes simples, limpos e extensíveis que podem ser facilmente mantidos e reutilizados.

Segundo Gupta (2015), microsserviços é um estilo arquitetural que exige decomposição funcional de uma aplicação. Para o autor, uma típica aplicação monolítica é dividida em múltiplos microsserviços, cada um publicado com seu próprio pacote executável. Após, estes serviços são compostos e uma única aplicação usando padrões

leves de comunicação, como REST através de HTTP. O termo “micro”, para o autor, não está relacionado com o número de linhas de um serviço (LoC), mas, sim, pelo número reduzido de funcionalidades.

As características de um microsserviços, para Gupta (2015), são:

- *Design* Orientado a Domínio: a decomposição funcional pode ser atingida pelos princípios do *Design* Orientado a Domínio de Evans (2003).
- Princípio da Responsabilidade Única: cada serviço deve ter apenas uma funcionalidade e deve ser especializada nela.
- Interfaces Explicitamente Publicadas: cada serviço deve publicar explicitamente, isto é, operações que executa, bem como a descrição de como utilizá-la, e deve honrá-las sempre. As interfaces devem ser versionadas e as versões antigas devem permanecer existentes para manter compatibilidade com todos os clientes.
- Publicação, *Upgrade*, Escalabilidade e Substituição independentes: cada serviço deve ser possível de ser publicado, escalado, substituído ou atualizado sem impactar o sistema como um todo. Cada serviço deve escalar independentemente no eixo X ou Z de acordo com Abbot e Fisher (2015).
- Potencial de heterogeneidade ou poliglótico: cada microsserviço poderá ser desenvolvido na tecnologia que melhor atender seus requisitos e isto não pode ser um problema para o sistema como um todo.
- Comunicação leve: a comunicação entre os serviços deve ocorrer utilizando protocolos leves como REST através de HTTP. Como HTTP é um protocolo síncrono, a utilização de padrões como *publish-subscribe* é amplamente utilizado para suporte a mensagens assíncronas.

Ainda para Gupta (2015), as vantagens da adoção de microsserviços são:

- Facilidade de publicação, manutenção e entendimento: uma das tarefas que mais levam tempo de um programador é compreender um código desconhecido. Por isso, um sistema restrito a uma função é mais fácil de compreender o que diminui o tempo de manutenção.
- Início rápido: um microsserviço deve executar mais rápido que um sistema monolítico, o que facilita teste e verificação.

- Mudanças locais são facilmente publicadas: criar uma nova instância do serviço para desenvolvimento em computador local deve ser uma tarefa simples e independente de outros serviços externos.
- Aumento do isolamento de falhas: falhas como perda de conexão com o banco de dados e *memory leak* afetarão apenas o serviço com falha e não toda a aplicação como em um sistema monolítico.

Em geral, uma arquitetura de microsserviços possui os artefatos:

- Clientes: aplicações *web*, *mobile* e integrações;
- Autenticação: para os clientes acessaram aos serviços;
- API Gateway: como elemento único de acesso e descobrimento de APIs pelos clientes;
- Microsserviços: que são pequenas unidades de software que implementam uma lógica de negócio específica;
- Mensageria: que implementa o padrão *publish-subscribe* na arquitetura como alternativa para evitar o uso microsserviços específicos para executar *workflows*;

3 METODOLOGIA

Este capítulo apresenta as etapas necessários para realizar a reengenharia do sistema de assinaturas e transformar sua atual arquitetura monolítica em uma arquitetura de microsserviços.

O *framework* base de reengenharia adotado é baseado em Nguyen (2011) e divide-se em duas grandes etapas, (i) engenharia reversa e (ii) reengenharia. Cada uma destas etapas é dividida em sub etapas que serão descritas em detalhes nas seções 3.1 e 3.2.

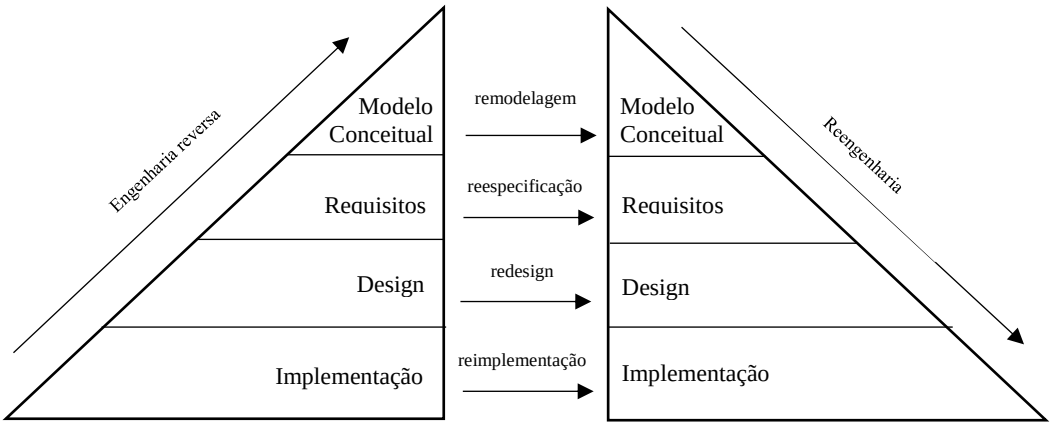

Figura 1 – Modelo geral da reengenharia de software
Fonte: elaborada pelo autor baseado em Nguyen (2011)

A Figura 1 apresenta o modelo geral de reengenharia de software em que o processo inicia pela engenharia reversa em que serão analisados a implementação, *design*, requisitos e modelo conceitual do sistema atual. Durante a engenharia reserva, o *design*, requisitos e modelo conceitual são elaborados a partir da análise da implementação.

Ainda na mesma figura, após a fase de engenharia reversa, a reengenharia inicia pela remodelagem do sistema em que é verificada a necessidade de alteração. Caso não seja necessário, avança para a reespecificação dos requisitos, re*design* e, por fim, a reimplementação.

3.1 Engenharia Reversa

Nesta fase são analisados todos os componentes que envolvem a implementação atual do sistema de assinaturas, como: (i) tecnologias utilizadas, (ii) arquitetura, (iii)

dependências, (iv) softwares de *middleware*, (v) banco de dados e (vi) *code smells*. Após, é elaborado uma descrição do *design* e arquitetura do sistema, bem como diagramas que ajudem a compreender o funcionamento. Na sequência, os requisitos são extraídos do código-fonte implementado e, por fim, é produzido o modelo conceitual.

3.2 Reengenharia

Posteriormente, inicia-se a fase de reengenharia em que a partir dos dados levantados na etapa de engenharia reversa, melhorias são identificadas e implementadas em todas as fases.

A primeira fase da reengenharia é a remodelagem do conceitual, ou seja, a partir do modelo conceitual elaborado, identificar e aplicar melhorias, se for o caso. A segunda fase é a reespecificação, em que do mesmo modo que a remodelagem, melhorias são identificadas e implementadas a partir das especificações existentes.

Com o novo modelo conceitual e requisitos em mãos, resta redesenhar a aplicação do ponto de vista arquiteturas, linguagem de programação e definir tudo que for necessário para a nova implementação. Por fim, a reimplementação é a fase em que o novo software é construído.

3.3 Migração Incremental

Por tratar-se em um grande sistema e por ser crítico em sua organização, o processo de reengenharia ocorrerá de forma incremental, isto é, o sistema será subdividido em módulos e submódulos e, então, cada um gradualmente migrado de acordo com a necessidade de negócio associada.

Por esse motivo, a engenharia reversa iniciará no sistema como um todo, para se ter uma noção do tamanho do sistema, bem como capturar seus elementos como arquitetura, tecnologia, *code smells*, etc. Porém para iniciar a reengenharia, apenas um módulo será escolhido para ser migrado. Este módulo também servirá para poder mensurar o trabalho necessário para migrá-lo e, então, poder-se elaborar uma estimativa mais assertiva sobre o tempo e custo de migrar todo o sistema. Ressalta-se que enquanto todo sistema antigo não for migrado, os dois sistemas coexistirão. Assim, um desafio adicional é manter os dois sistemas o mais integrado possível. Ademais, é possível que os dois sistemas coexistam para sempre, pois será uma decisão de negócio, baseado em custo e retorno em

que a decisão de migrar todo o sistema será feita. Portanto, criar um *design* prevendo isso faz-se muito importante.

4 ENGENHARIA REVERSA

Neste capítulo, será exposto a análise do Sistema de Assinaturas ou AS, como é conhecido. Esta análise consiste no resultado da engenharia reversa executada no sistema em que, a partir da implementação (i.e, código-fonte e arquitetura), foram extraídas estatísticas sobre o sistema, como uma forma de dimensionar seu tamanho. Ademais, foi produzida uma descrição da arquitetura atual do sistema, bem como foram descritos os requisitos e o modelo conceitual.

4.1 Análise descritiva

O Sistemas de Assinaturas (AS) é uma importante aplicação da empresa e é responsável pelas funcionalidades, a saber:

1. **Operação de vendas ativas e receptivas de assinaturas** de conteúdo, como venda de assinaturas digitais ou impressas, bem como produtos promocionais oferecidos pela empresa para alavancagem de vendas;
2. **Operação de serviços ou suporte aos clientes**, como troca de endereço de entrega, reclamação por não recebimento, entre outros serviços;
3. **Roteirização de entregas** de jornais, revistas e outros produtos tangíveis comercializados ou promocionais;
4. **Cadastro de produtos e tabela de preços**, como a precificação de assinaturas de acordo com o conteúdo e plataforma que o cliente deseja ter acesso;
5. **Remuneração de vendedores** ou controle de comissão de vendas;
6. **Gerenciamento de assinaturas** para liberação de conteúdo de acordo com o tipo de assinatura do cliente, bem como verificação para interrupção da prestação de serviços em caso de não pagamento ou suspensão da assinatura;
7. **Cobrança recorrente**, como controle da data de vencimento de cada assinatura para cobrar periodicamente, de acordo com parâmetros negociados em cada assinatura;

8. **Gerenciamento de clube de assinantes**, que é um produto que oferece benefício aos associados, como descontos em parceiros conveniados;
9. **Gerenciamento de convênios,** para garantir que os associados do clube tenham acesso aos descontos nos parceiros conveniados. Consiste em integração com sistemas de empresas parceiras;
10. **Conciliação bancária** para enviar as faturas a serem cobradas para os bancos de adquirentes de cartão de crédito e, após, receber o retorno das cobranças e atribuir a cada assinatura o status pago ou não pago;
11. **Contabilização** que consiste em gerar informações contábeis e fiscais sobre as cobranças realizadas, além de solicitar a emissão de notas fiscais e atribuir receitas nas rubricas contábeis corretas;
12. **Analítico**, como emissão de relatórios e agregação e transformação de dados para extração de informações analíticas gerenciais;
13. **Auditoria,** como a extração de dados relacionados ao índice de circulação para ser auditado pelo IVC (Instituto Verificador de Circulação), uma ONG que emite um selo sobre a quantidade de circulação de jornal impresso e digital, utilizado para precificar a publicidade;
14. **Gerenciamento de Usuários** para controle de acesso de usuários ao sistema.

Todas estas 14 atividades são as principais da operação de venda de assinaturas para entrega de conteúdo aos consumidores. Em uma empresa jornalística, comercialização e entrega do conteúdo é a segunda atividade mais importante depois da criação do conteúdo jornalístico.

O sistema é composto por quatorze tarefas de alta complexidade e de naturezas distintas (como venda e roteirização de entrega, por exemplo), o que explica a dificuldade encontrada em alterá-lo, visto que uma mudança no gerenciamento de usuários, por exemplo, pode afetar todas as funcionalidades do sistema.

Com isto posto, há um primeiro diagnóstico de baixa coesão e alto acoplamento no sistema, duas características combatidas pela arquitetura de microsserviços e um desafio a ser superado para a migração de plataforma.

4.2 Análise da implementação

Nesta seção é retratada a análise da implementação do sistema em que serão avaliadas as tecnologias utilizadas, componentes existentes, infraestrutura de hardware, código-fonte escrito e processo de desenvolvimento aplicado.

4.2.1 Tecnologias

O sistema está implementado, em sua maioria, utilizando as tecnologias Oracle®. A camada de apresentação utilizada pelo usuário final foi desenvolvida em Oracle Forms 6i. Já na camada de persistência de dados é utilizado o Oracle Database 11. Na camada em que estão implementadas as regras de negócio, foi utilizada a tecnologia PL/SQL, que consiste em uma linguagem de programação que é executada no próprio Oracle Database, sendo esta uma extensão do SQL padrão. Existe, ainda, uma camada de integrações desenvolvidas em Java, com o *framework* Spring, executadas por um Apache Tomcat 7.1, que expõem rotinas do PL/SQL como API's REST para garantir integrações com outros sistemas.

Os sistemas operacionais utilizados com estas tecnologias são Oracle Linux 7.1, nos servidores Oracle Database, CentOS 7 nos servidores de API's em que é executado o Tomcat. Já o Oracle Forms é executado em estações de trabalho com Windows 7 e se conectam ao Oracle Database através do Oracle Instant Client 9g de 32 bits. Toda a comunicação envolvida ocorre via TCP/IP em uma rede LAN de 10 GB.

Em resumo as tecnologias utilizadas (ou *technology stack*) são:

- Camada de Apresentação: Oracle Forms 6i executados em estações de trabalho com Windows 7;
- Camada de Integração: Java 7 utilizando Spring Framework 3.0.3 executado em um servidor web Java Tomcat 7 hospedados em um servidor Linux virtualizado CentOS 7;
- Camada de Negócio: PL/SQL executado no Oracle Database;
- Camada de Persistência de Dados: Oracle Database 11g executado em sistema operacional Oracle Linux em hardware Dell;
- Camada de Virtualização (Hypervisor): Microsoft Hyper-V

- Camada de Hardware: Servidores Dell

A Figura 2 apresenta o diagrama de implementação (*deployment diagram*) do sistema de assinaturas, descrevendo como estão organizadas as tecnologias do sistema, bem como seus ambientes de execução.

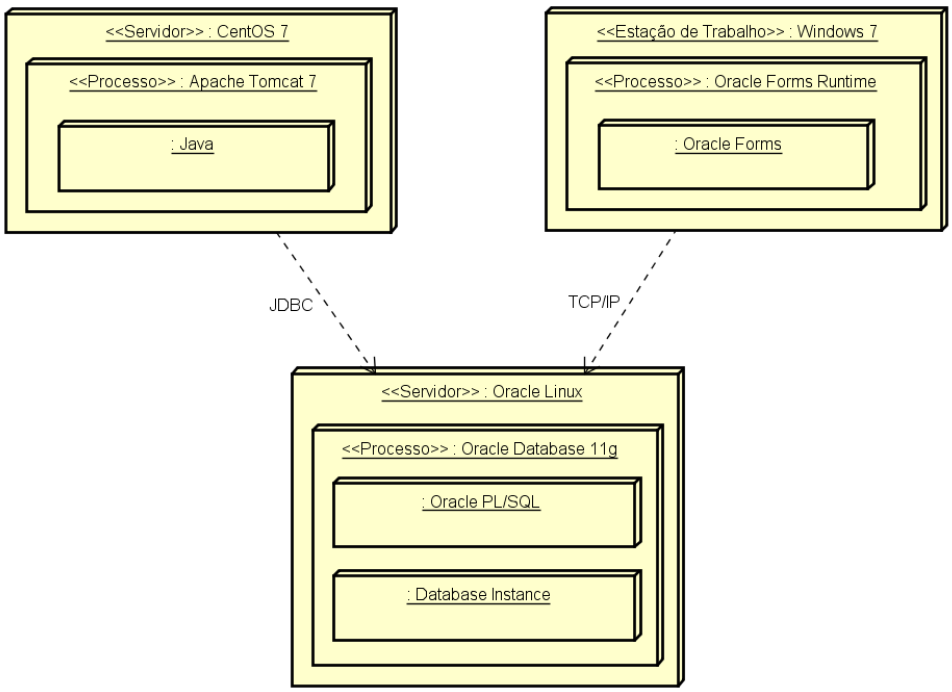

Figura 2 – Diagrama de implementação do Sistema de Assinaturas
Fonte: Elaborado pelo Autor

Como pode ser observado, são 3 ambientes de execução principais:

1. **Servidor** com sistema operacional **Oracle Linux** que executa os processos do **Oracle Database** que, por sua vez, executa as lógicas de negócio escritas em **PL/SQL** e **persiste os dados** em uma instância de banco de dados;
2. **Servidor** com sistema operacional **CentOS 7** que executa os processos do **Apache Tomcat 7** e este, executa o programa em **Java** que expõem as API's do sistema de assinaturas para integração com outros sistemas;
3. **Estação de trabalho**, que consistem em computadores de mesa, com sistema operacional **Windows 7** que executam o **Oracle Forms Runtime** responsável por executar os **Oracle Forms**, que, por fim, são utilizados pelos usuários do sistema. O Oracle Forms, portanto, é interface de interação entre o sistema o os seus usuários.

Em relação à escalabilidade, as estações de trabalho que rodam Oracle Forms podem crescer indefinidamente, enquanto o Oracle Database suportar conexões. Em cada estação de trabalho, basta instalar o Oracle Forms Runtime 6i, instalar os Oracle Forms do sistema de assinaturas, instalar o Oracle InstantClient 10g, responsável pela conexão entre a estação de trabalho o servidor o Oracle Database, e configurar no arquivo TNSNAMES do InstantClient os endereços IP do Oracle Database.

A escalabilidade dos servidores de API, basta criar mais um servidor e fazer a implantação do código Java no Tomcat e configurar o context.xml do servidor web com os dados JDBC corretos para conectar com o Oracle Database. Contudo, é necessário incluir na arquitetura um LoadBalancer que irá distribuir a carga entre os servidores.

Já em relação à escalabilidade do OracleDatabase, é preciso executar uma escalabilidade vertical (no eixo "y"), incluindo mais memória ou capacidade de processamento ao servidor. Ou, ainda, é preciso criar um cluster do servidor. Atualmente o servidor Oracle Database já é "clusterizado" com alta disponibilidade e redundância para evitar indisponibilidades.

Trata-se de um *technology stack* tradicional, com aplicações monolíticas, isto é, componentes da aplicação são implementados e executados em um único pacote ou programa executável.

4.2.2 Dados sobre a implementação

O sistema de assinaturas, como supracitado, é um importante sistema e afeta diversas atividades e áreas da empresa. Para dimensionar o tamanho do sistema, bem como o número de dependências, linhas de código (LoC, do inglês *Lines of Code*), tipos componentes, linhas de código por artefatos ou objetos, linguagem de programação, entre outras informações, foi feita uma mineração nos dados do sistema. Em resumo, os dados mais importantes obtidos sobre o sistema são:

1. **Usuários**: 835 distribuídos em 5 áreas;
2. **Telas**: 213 telas (Oracle Forms) que são utilizadas pelos usuários;
    a. 80% do uso está concentrado em 59 elas;
    b. Estas 59 telas correspondem a 27% do total de telas e possuem 140.223 LoC, isto é, 19,484% do total e LoC do sistema e 27,389% do total de LoC das telas do sistema.

3. **Linhas de Código** (LoC, *Lines of Code*): 719.672 LoC distribuídas em:
    a. **Telas Oracle Forms:** 213 telas e 511.988 LoC (71,142%) e média de 2151 LoC/Tela;
    b. **Java**: 234 arquivos e 18.787 LoC (2,610%) e média de 80 LoC/arquivo;
    c. **PL/SQL:** 1.967 objetos e 188.897 LoC (26.248%) e média de 2436 LoC/objeto;
        i. **Package Body**: 135.571 LoC ou 18,838% do total;
        ii. **Procedure**: 26.395 LoC ou 3,668%;
        iii. **Trigger**: 15.542 LoC ou 2,160%;
        iv. **Packages**: 5.867 LoC ou 0.815%;
        v. **Functions**: 5.522 LoC ou 0.7867%
4. **Tamanho do Banco de Dados**: 7,5 TB

Com os dados obtidos, é possível ter uma ideia de dimensão do sistema. Para tentar converter o LoC em horas de trabalho, podemos considerar que um programador escreva 20 LoC em 1 hora, logo, 719.672 LoC são escritas em 35.983 horas, ou 1500 dias, ou 214 semanas, ou 53 meses, ou 4 anos e meio.

Contudo, como citado na metodologia deste trabalho, a abordagem da migração será incremental, isto é, não serão migrados todos módulos e LoC do sistema de uma vez e sim, separado em módulos que serão gradativamente migrados. Como módulo para ser o primeiro a ser migrado, foi escolhido pelo gerente de sistemas o módulo de Cancelamento e Suspensão de Assinaturas por ser um módulo importante do sistema, sendo o sexto mais utilizado até janeiro de 2017, e por ser relativamente pequeno, representando menos de 3% do total de LoC do sistema como é detalhado na seção 4.3.2.

4.2.3 Code Smells

Como descrito na revisão bibliográfica deste trabalho, código limpo é sinônimo de produtividade e redução dos custos de mudança de software. Neste trabalho, a procura por *Code Smells* seguiu o guia do código limpo de Martin (2009).

O Quadro 1 foi elaborado em conjunto com a equipe que mantém o sistema composta por 4 programadores, 1 arquiteto de *software*, 1 gerentes de sistema e 2 usuários-chave. A avaliação foi obtida através de uma nota atribuída de 0 a 10 em que 0 a pessoa discorda totalmente com a afirmação da coluna “Análise” e 10 concorda totalmente. Por fim, foi feita uma média das respostas e transcrito no Quadro 1.

CARACTERÍSTICA	ANÁLISE	AVALIAÇÃO
RIGIDEZ	O sistema apresenta dificuldade de mudança e cada pequena mudança desencadeia outras pequenas mudanças subsequentes.	10
FRAGILIDADE	O software apresenta erros em diversos lugares devido a uma única alteração.	8

IMOBILIDADE	Não é possível reutilizar partes do sistema em outras partes devido a riscos que podem envolver comportamento inesperado.	9
VISCOSIDADE DO *DESIGN*	Atalhar e introduzir um débito técnico exige menos esforço do que fazer o que é correto.	10
VISCOSIDADE DO AMBIENTE	Compilação, implantação, testes, Jobs e outras tarefas levam muito tempo.	9
COMPLEXIDADE DESNECESSÁRIA	A arquitetura e o código apresentam elementos que não são mais úteis, porém o risco de removê-los e erros inesperados ocorrerem são altos.	8
REPETIÇÃO DESNECESSÁRIA	Código apresenta muita repetição.	10
OPACIDADE	O código é difícil de compreender. Ao custo de uma nova alteração é adicionado o tempo de compreensão do que o código faz.	10
SMELLS SCORE		**9,25**

Quadro 1 – Smells Score do sistema de assinaturas
Fonte: elaborado pelo autor

O Quadro 1 apresenta o indicador *Smells Score* que corresponde a uma média aritmética das notas atribuídas a cada característica avaliada no código. Sendo 0 a isenção de *Code Smells* e 10 sendo a total presença, o sistema pontuou 9,25, o que é uma nota elevada e demonstra que o sistema apresenta muito débito técnico na implementação e no *design*.

4.3 *Design* do sistema

A segunda etapa do padrão geral da reengenharia de software consiste em reproduzir os padrões de *design* e arquitetura adotados a partir da implementação do código existente. Neste capítulo serão apresentados diagramas sobre a arquitetura utilizada, bem como os padrões de *design* encontrados.

4.3.1 Arquitetura

Como já pode ser observado na Figura 2, o sistema adota uma arquitetura de 3 camadas, sendo elas: camada de apresentação, camada de regras de negócio ou de negócio e camada de persistência. Apesar do sistema possuir as três camadas facilmente identificadas, duas camadas rodam no mesmo ambiente de execução, como é o caso da camada de negócios e camada de persistência, o que é combatido pela arquitetura de microsserviços, que presa pela autonomia e independência dos serviços.

A Figura 3 representa as camadas da arquitetura que estão em verde. A mesma figura mostra, ainda, onde cada camada está sendo executada, como, por exemplo, o Oracle Forms é executado pelo Oracle Forms Runtime que, por sua vez, está rodando em um Windows 7.

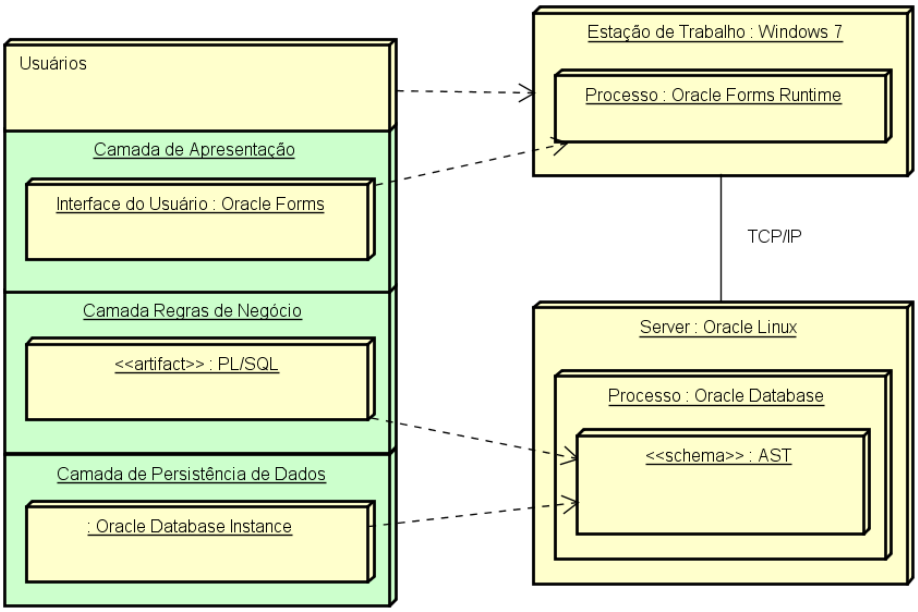

Figura 3 – Representação das camadas da arquitetura
Fonte: elaborado pelo autor

4.3.2 Dependências do sistema

Nesta seção é apresentado como os elementos do sistema, apresentados na seção 4.2.2, dependem uns dos outros, exceto a camada de integração em Java. Devido ao elevado número de objetos de banco de dados e telas o mapa de dependências fica praticamente inelegível, como demonstra a Figura 4.

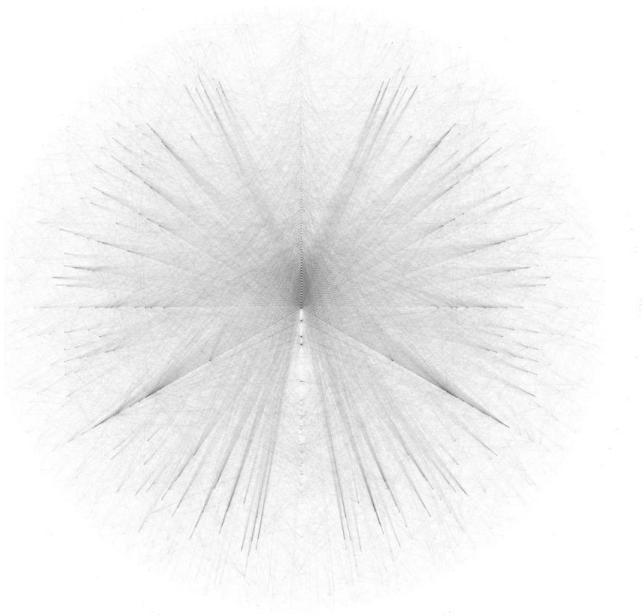

Figura 4 – Gráfico de dependência do sistema de assinaturas
Fonte: elaborado pelo autor

A Figura 4 representa, através de um diagrama de grafos concêntrico, como os elementos se relacionam. Estes elementos são objetos de banco de dados Oracle, como *tables, packages, procedures, functions, triggers, index* e *views* e telas em Oracle Forms. O centro do gráfico está o nodo que mais utilizado no sistema e é a tabela de banco de dados "assinatura", o que é bem previsível tratando do sistema de assinaturas. Nas pontas mais externos do círculo estão os nodos que representam os elementos menos utilizados no sistema.

Analisando o módulo de Cancelamento e Suspensão de Assinaturas, que corresponde ao módulo que será migrado para a nova arquitetura neste trabalho, verifica-se que este é composto pela tela ASCA0500 e pelos os objetos de banco de dados Oracle que estão descritos no Quadro 2.

Rótulos de Linha	**Contagem de Objetos**	**Soma de LoC**
TABLE	105	N/A
FUNCTION	21	2054
PACKAGE	12	1568
PROCEDURE	12	801
PACKAGE BODY	5	7966
SEQUENCE	3	N/A
VIEW	3	N/A
INDEX	2	N/A
Totais	**163**	**12389**

Quadro 2 – Resumo das dependências da ASCA0500
Fonte: elaborado pelo autor

Todos os objetos de banco de dados Oracle descritos no Quadro 2 são dependências diretas ou indiretas da tela ASCA0500 e juntos compõem o módulo de Cancelamento e Suspensão de Assinaturas. Verifica-se, portanto, o que o total de dependências diretas e indiretas desta tela é 163 e totaliza 12.389 LoC. Destas 163 dependências, 105 são *tables*, 12 *packages*, 5 *packages body*, 21 *functions*, 12 *procedures*, 2 *index*, 3 *sequence* e 3 *views*. Somando as 12.389 LoC das dependências de objetos de banco de dados com as 2.989

LoC da tela ASCA0500, totaliza-se 15.378 LoC no módulo de Cancelamento e Suspensão de Assinaturas, cerca de 3% do sistema.

4.4 Requisitos do sistema

O levantamento de requisitos do sistema é a etapa da engenharia de software que consiste na compreensão do sistema, seu domínio, prioridades, entre outros. Na etapa de engenharia reversa, a etapa de requisitos ocorre de forma um pouco diversa da reengenharia. Neste caso, se extrai a partir da implementação os requisitos implantados, para mapear a situação atual do sistema. A partir deste mapeamento, se verifica com os *stakeholders* se estes requisitos ainda são válidos e, após, são alterados de acordo com os novos requisitos.

Nesta seção, serão levantados os requisitos implementados na tela ASCA0500 e suas dependências. Após, será apresentada a revalidação e alterações para a reimplementação.

4.4.1 Descrição da ASCA0500

A ASCA0500 é o objeto que realiza a interface de interação entre o usuário do sistema de assinaturas e o sistema para cancelamento de uma assinatura e é a sexta tela mais utilizada do sistema.

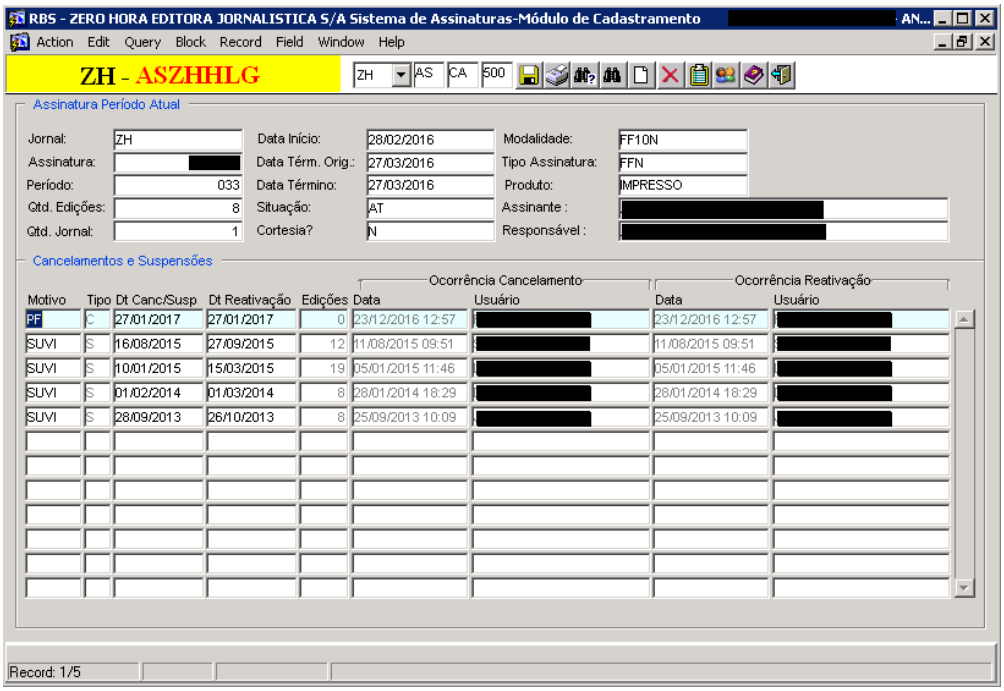

Figura 5 – Visão geral da ASCA0500
Fonte: elaborado pelo autor

A Figura 5 exibe como os usuários visualizam a ASCA0500 e como interagem com ela. Na figura, as tarjas pretas ocultam informações sensíveis para empresa. O uso geral desta tala é o cancelamento ou suspensão de uma assinatura. Portanto, o funcionamento básico consiste em selecionar o assinante, selecionar a assinatura que deseja cancelar ou suspender, inserir a data de início de cancelamento ou suspensão e salvar.

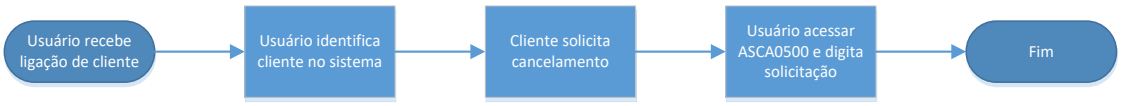

Figura 6 – Processo simplificado do cancelamento de uma assinatura
Fonte: elaborado pelo autor

Apesar de muito simples o funcionamento desta tela, 15.378 LoC é bastante código. Isto se deve a diversas regras de negócio e requisitos existentes nesta tela que serão apresentados nas próximas subseções.

4.4.2 Objetivo Principal

A ASCA0500 é uma parte importante do sistema que visa suprir a necessidade do cliente em cancelar ou suspender sua assinatura sempre que julgar conveniente. Para empresa, claro, prestar este serviço é uma necessidade.

Cancelar e Suspender são duas operações do sistema que produzem o mesmo efeito com a diferença que a primeira é definitiva enquanto a segunda é temporária. O cancelamento da assinatura consiste na decisão do cliente, ou da empresa, por qualquer motivo, encerrar sua assinatura.

O efeito imediato desta operação é a interrupção do recebimento do conteúdo jornalístico nos canais contratados, como os canais físicos (jornal, encartes e revistas) ou como os canais digitais (aplicativos e web site), interrupção dos benefícios, como descontos do clube de assinantes, e interrupção do pagamento do cliente a partir do fim do período atual. No caso da suspensão, os efeitos são os mesmos, com a diferença que há um período em que eles são válidos, isto é, existe uma data de início da suspensão e outra de fim e a partir da data de fim, o conteúdo, benefícios e pagamento retomam seu funcionamento normal.

4.4.3 Fluxo Básico

O fluxo básico de funcionamento das operações de cancelamento e suspensão é:

1. Cliente contata o *contact center* através de um dos canais disponíveis;
2. Operador inicia atendimento solicitando informações para identificação do cliente e de segurança;
3. Cliente informa que deseja cancelar sua assinatura;
4. Operador informa a existência da opção de suspensão como alternativa ao cancelamento;
5. Cliente informa sua decisão:
6. Operador inicia operação;
7. Se operação for suspensão, então operador insere as datas de suspensão;
8. Se operação for cancelamento, operador consulta data de fim do período e informa ao cliente;
9. Operador informa que a operação foi concluída e encerra o atendimento.

4.4.4 Pré-Requisitos e Pressupostos

Sendo as operações de venda parte de um sistema maior é preciso que alguns pré-requisitos e pressupostos necessários que estão expostos nesta subseção.

4.4.4.1 Integração com sistema de contact center

Já existe um sistema que provê ao operador acesso aos canais como telefone ou chat pela internet. Portanto, o sistema de assinaturas está integrado com este sistema com o sistema de *contact center*.

4.4.4.2 Operador de atendimento é um perfil de usuário do sistema devidamente identificado

O operador que está fazendo o atendimento de cancelamento ou suspensão deve estar identificado pelo sistema com sua credencial única e deve ter permissões específicas para executar a operação.

4.4.4.3 Tela de identificação do cliente

A tela de identificação do sistema está disponível e a operação de cancelamento/suspensão é um redirecionamento ou link a partir dela.

4.4.5 Requisitos Funcionais

4.4.5.1 Período atual

O período atual corresponde ao período de tempo vigente para prestação do serviço contratado e pago pelo cliente. Por exemplo, se o contrato do cliente é para pagamento mensal, a cada mês, uma cobrança nova será gerada. O período atual corresponderá ao mês de prestação de serviço ao qual o cliente pagou por último e ainda não encerrou.

4.4.5.2 Cancelamento da Assinatura

- O sistema deve remover do roteiro de entrega dos canais físicos a partir da data de fim do período atual;
- O sistema deve interromper o acesso aos canais digitais a partir da data de fim do período atual;

4.4.5.3 Suspensão da Assinatura

- O sistema deve remover do roteiro de entrega dos canais físicos a partir da data de início da suspensão;
- O sistema deve retomar do roteiro de entrega dos canais físicos a partir da data de fim da suspensão;
- O sistema deve revogar o acesso aos canais digitais a partir da data de início da suspensão;
- O sistema deve conceder o acesso aos canais digitais a partir da data de fim da suspensão;

4.5 Modelo Conceitual

A partir da análise da implementação, do *design* e dos requisitos do sistema de assinaturas, foi elaborado o modelo conceitual, que expõem como se relacionam as entidades do módulo de cancelamento e suspensão de assinatura. A Figura 7 apresenta o modelo conceitual.

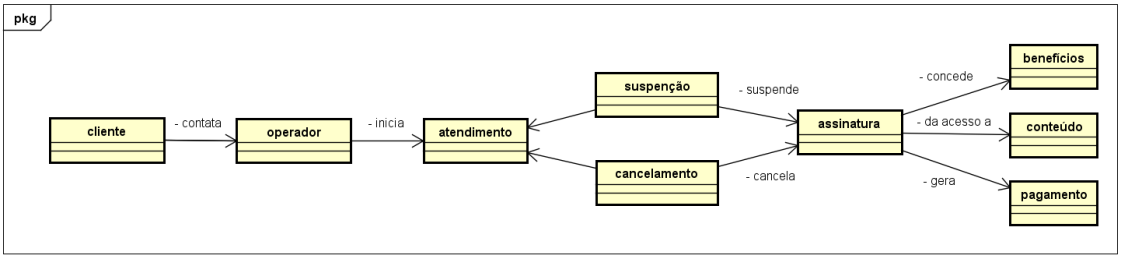

Figura 7 – Modelo conceitual
Fonte: elaborado pelo autor

5 REEGENHARIA

Neste capítulo será apresentada como foi realizada a reengenharia do sistema de assinaturas com base nos dados obtidos durante a etapa de engenharia reversa. No capítulo anterior, foi citado que o módulo escolhido para iniciar a migração foi o de cancelamento e suspensão de assinaturas.

5.1 Remodelagem

O módulo de suspensão e cancelamento não foi alterado em nada do ponto de vista funcional, permanecendo válido o modelo conceitual da Figura 7. Além do modelo conceitual, o objetivo principal, fluxo básico e requisitos funcionais permanecem os mesmos do seção 4.4.

Entretanto, as tecnologias utilizadas, plataforma e requisitos não-funcionais foram alterados e serão exibidos nas próximas seções deste capítulo.

5.2 Reespecificação

Nesta seção será apresentada a reespecificação do sistema de assinaturas baseado na especificação elaborada na etapa de engenharia reversa.

5.2.1 Descrição Geral da Nova ASCA0500

A Figura 8 exibe a nova aparência da ASCA0500 na nova arquitetura. Ressalta-se que o *design* e experiência do usuário no cliente não está no escopo deste trabalho. Contudo, o *mockup* da nova tela demonstra aspectos importantes sobre a nova arquitetura, como o fato de estar em um ambiente *web*.

5.2.2 Pré-Requisitos e Pressupostos

Nesta seção será apresentado os pré-requisitos e pressupostos para o desenvolvimento da nova ASCA0500.

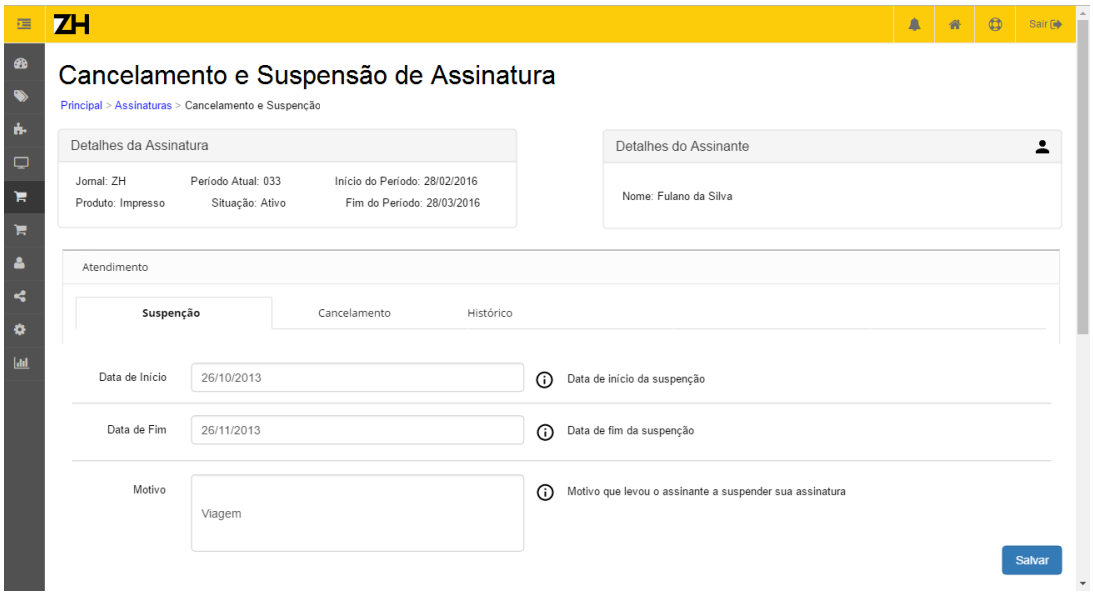

Figura 8 – Visão geral da nova ASCA0500
Fonte: elaborado pelo autor

5.2.2.1 Integração com sistema de contact center

Do mesmo modo em que a ASCA0500 atual tem integração com o sistema de *contact center*, que provê ao operador acesso aos canais como telefone ou chat pela internet. A integração em um primeiro momento será manual, isto é, o operador utilizará os dois sistemas ao mesmo tempo, alternando entre as janelas de um e de outro.

Contudo, deve haver um *roadmap* de integração em que o estado da arte é o usuário visualizando apenas uma janela (um sistema) que esteja visualmente e funcionalmente integrado, sendo possível, por exemplo, identificação automática do cliente que está entrando em contato.

5.2.2.2 Integração com sistema atual

São requisitos da ASCA0500, que o usuário esteja *logado* no sistema, tenha encontrado o cadastro do assinante e sua assinatura em uma tela específica para isso e que tenha permissões para executar a operação de cancelamento.

Pela migração ser executada de forma incremental, existirá um tempo em que o sistema antigo e o novo coexistirão. Assim, como o novo sistema não terá a tela de login nem a de busca de assinante e assinatura, no sistema atual, quando o usuário pressionar o botão que redireciona para a ASCA0500, deverá redirecionar para a nova tela.

5.2.2.3 Requisitos Não-Funcionais

- O sistema deverá ser desenvolvido para plataforma web;
- O sistema deverá ser responsivo;
- O sistema deverá executar em todos os navegadores;
- Todas as funcionalidades devem ser serviços acessíveis via REST;
- O sistema deverá seguir as normas de acessibilidade;

5.3 Re*design*

Uma parte essencial para a migração de um sistema é a definição tecnológica do sistema destino. Uma das grandes vantagens da adoção da arquitetura de microsserviços é sua característica agnóstica à tecnologia, ou seja, é possível integrar diversas tecnologias e um mesmo sistema. Por esse motivo, a investigação tecnológica será por cada elemento da arquitetura.

5.3.1 Plataforma

Plataforma refere-se a qualquer hardware ou software utilizado para hospedar uma aplicação ou serviço. No sistema de assinaturas atual, as plataformas são servidores físico com SO Linux e estações de trabalho físicas com SO Windows.

No mundo dos microsserviços, a virtualização sob demanda ou conteinerização se tornou extremamente popular. Assim, fazer o *deploy* de um microsserviço em um contêiner que está em um alto nível de virtualização é uma estratégia muito simples e uma das estruturas que viabiliza a arquitetura de microsserviços.

É possível adotar a arquitetura de microsserviços sem utilização de contêineres. Utilizar servidores físicas como ambiente de execução ainda é muito eficiente do ponto de vista de performance, mas não muito eficiente pela ótica de facilidade de escalabilidade se você precisar de centenas deles. Outra opção é a tradicional virtualização, contudo, a utilização desta tecnologia não é simples e ainda é muito cara, além do *overhead* de memória e processamento para rodar pequenos serviços.

Diante destas considerações, os contêineres se tornaram uma alternativa barata e de fácil utilização, totalmente alinhada com os princípios da arquitetura de microsserviços e por esse motivo será adotado neste trabalho.

Existem basicamente três escolhas tecnológicas para adoção de contêineres: escolher o software de conteinerização, escolher o software de gerenciamento de contêineres e escolher a plataforma para estes dois softwares. Em geral, é mais fácil começar escolhendo a plataforma, pois é comum que suporte apenas um software de conteinerização e um software de gerenciamento de contêineres.

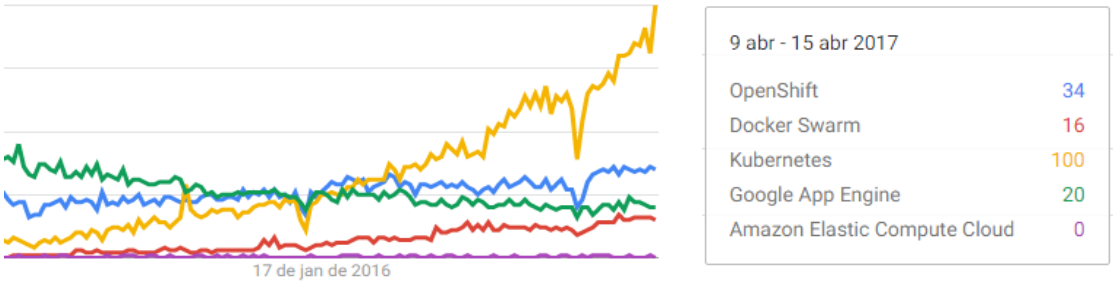

Figura 9 – Popularidade de buscas das plataformas de contêinerização
Fonte: Google Trends (Abril de 2017)

Como pode ser observado na Figura 8, Kubernetes é a plataforma mais popular nas pesquisas do Google, segundo o Google Trends, na data especificada na legenda. OpenShift está em segundo lugar e Docker Swarm vem ganhando relevância.

OpenShift é uma plataforma Open-Source da Red Hat que utiliza Kubernetes em sua estrutura, de forma que oferece as funcionalidades do Kubernetes e outras mais, por esse motivo foi selecionado o OpenShift como a plataforma para o sistema destino.

Uma vez escolhido o OpenShift, como este software funciona apenas com Kubernetes como software de gerenciamento de contêineres e Docker como software para conteinerização, estes já estão automaticamente escolhidos.

5.3.2 Linguagens e Frameworks para Microsserviços

Uma das grandes vantagens da arquitetura de microsserviços é que a tecnologia escolhida importa pouco. Em outras palavras, o programador ou equipe fica livre para escolher a tecnologia que melhor desempenha determinada tarefa.

Como a empresa objeto de estudo deste trabalho possui muitos programadores Java, C# e JavaScript, as prioridades será para frameworks para estas linguagens. Neste contexto, o framework Spring Cloud oferece um aparato de componentes prontos para arquitetura de microsserviços e, por isso, será utilizado neste trabalho.

5.3.3 API Gateway

API Gateway é um padrão adotado na arquitetura de microsserviços que tem como objetivo centralizar as API´s de todos os serviços em um único ponto de acesso pelos clientes, de forma garantir segurança, documentação e governança das API´s.

Existem diversas opções para implementação de um API Gateway, que vai desde a compra do software de um grande fabricante, ou utilizar frameworks que facilitam a sua implementação. Entre os grandes desafios de implementar o API Gateway, é fazê-lo de forma que não se torne um gargalo para uso das API´s, ao mesmo tempo provenha segurança, controle de uso e logs.

Neste cenário, uma ferramenta desenvolvida pelo Netflix chamada Zuul ganhou grande popularidade. Atualmente existe implementação em diversas linguagens e frameworks, entre eles o Spring Cloud, que é um framework Java que visa prover os componentes necessários para implementar a arquitetura de microsserviços.

Devido a sua simplicidade de implementação e compatibilidade com Spring para Java, que é uma linguagem amplamente utilizada pela empresa, neste trabalho o API Gateway será desenvolvido com Zuul.

5.3.4 Mensageria

Um padrão muito utilizado para filas de processamento e implementação do padrão *publish-subscribe* é a utilização de um *broker*. Neste trabalho será utilizado o RabbitMQ devido ao seu amplo uso com Spring, além de ser nativamente suportado para utilizar Docker como ambiente de execução.

5.3.5 Modelagem e Documentação

Para modelar e documentar, os padrões mais adotados são RAML e Swagger. Para este trabalho será adotado Swagger por ser entendido como a solução mais completa, desde a geração, vinculada ao código fonte, e a exibição com Swagger-UI.

5.3.6 Arquitetura

A Figura 10 apresenta um esquema do diagrama de implementação e as camadas envolvidas em cada processo. Apresenta, também, como cada processo interage com

outro. De forma resumida, os usuários poderão acessar o sistema de assinatura por diversos clientes, como o portal online ou pelo contact center. Cada cliente acessará as funcionalidades do sistema de assinaturas através dos serviços expostos pelo API Gatway, que verifica a autenticação e redireciona a requisição ao serviço responsável por ela.

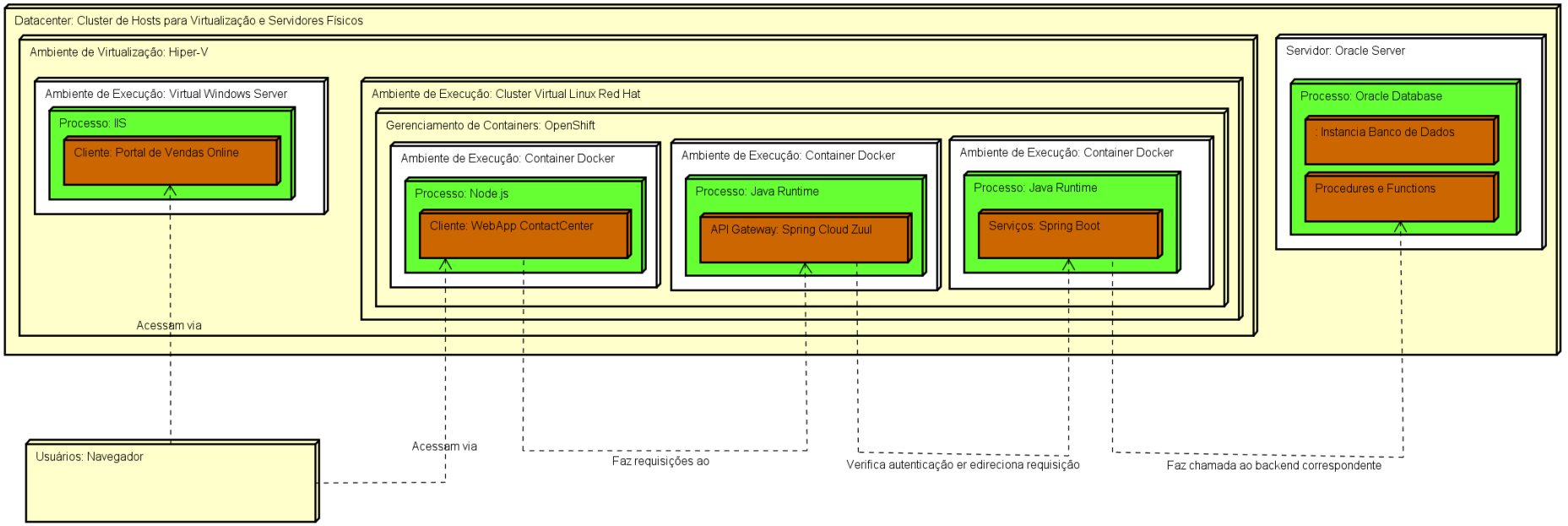

Figura 10 – Esquema do diagrama de implantação e interação entre componentes
Fonte: elaborada pelo autor

Em resumo, a arquitetura geral do sistema terá 7 componentes principais:

1. Cliente: aplicação que faz interface com o usuário do sistema (site, aplicação, etc.);
2. API Gateway: catálogo de API's do sistema de assinatura que funciona como autorizador e autenticador das requisições;
3. Microsserviço: serviço que implementa a lógica de cada operação;
4. Mensageria: serviço que implementa o padrão *publish-subscribe* para encadeamento de ações;
5. Gerenciamento de configuração: componente responsável por replicar mudanças de configurações em instâncias distribuídas de serviços. Este componente não é obrigatório para a primeira versão do sistema, contudo, é muito importante em cenários que existem muitas instâncias distribuídas de muitos serviços;
6. Descoberta de instâncias de serviços: assim com o gerenciamento de configuração, o serviço de descoberta de instâncias é importante em cenários em que existem muitas instâncias de muitos serviços em execução ao mesmo tempo. Com este serviço, é possível identificar quais instâncias estão disponíveis para uma determinada requisição, além de funcionar como um distribuidor de carga;
7. Integração com legado: basicamente são os componentes necessários para manter a compatibilidade com o sistema atual;

O fluxo de informações na arquitetura atual seguirá o diagrama de sequência para o módulo cancelamento e suspensão representado na Figura 11. Toda requisição iniciará em um cliente que invocará uma API de um serviço. No exemplo da Figura 11, o cliente é o AS WebApp que faz uma solicitação de cancelamento ou suspensão, informando a assinatura e o período (em caso de suspensão) para o API Gateway. Este, por sua vez, verificará se o cliente está autenticado e se possui permissões para fazer a operação requisitada, se tiver permissão, redireciona a requisição ao serviço responsável. O Serviço Canc/Suspens irá processar a requisição, armazenar os dados em seu repositório de persistência (que não mais será compartilhado, mas cada serviço terá o seu, seja um banco de dados, índice, arquivo, etc.), e notifica, de forma assíncrono, o serviço de mensageria sobre o evento que acabou de ocorrer (Cancelamento ou Suspensão).

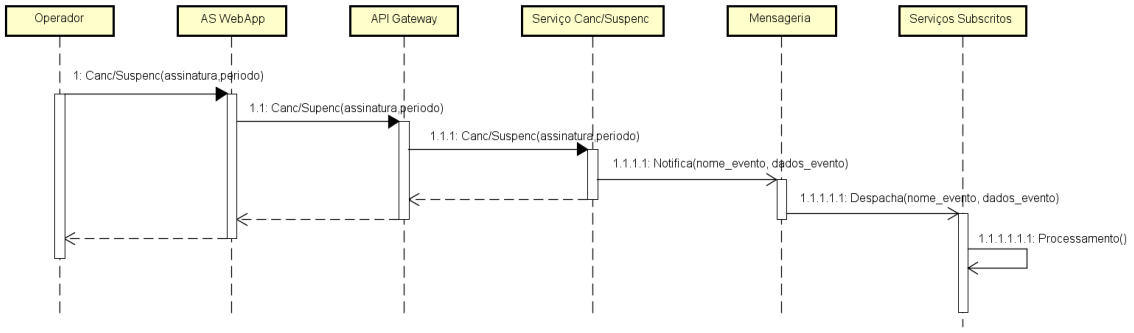

Figura 11 – Esquema geral do fluxo de informações na nova arquitetura
Fonte: elaborada pelo autor

Logo, o serviço de mensageria publicará aos serviços subscritos o evento e cada serviço executará a lógica implementada para cada situação. Como exemplo de serviços subscritos, para o caso do cancelamento e suspensão, pode-se citar o serviço que irá excluir os pagamentos futuros (evitando que a empresa cobre o assinante de um produto que acabou de cancelar), remova a assinatura do roteiro de entrega, remove a assinatura dos veículos digitais (cancelando a entrega de conteúdo digital) e excluído a assinatura do cadastro de benefícios. São, portanto, quatro serviços subscritos aos eventos de cancelamento e suspensão.

Além destes, outro serviço é necessário, que é o serviço de replicação de alteração de dados. Na arquitetura de microsserviços, cada serviço terá sua própria solução de persistência. O sistema de assinaturas possui um banco de dados compartilhado com todo o sistema, além da parte analítica. Este serviço de sincronização irá manter a compatibilidade com o sistema legado, replicando as alterações de dados no banco de dados atual.

6 IMPLEMENTAÇÃO DO SISTEMA

Para dar início a implementação do novo sistema, alguns componentes básicos para dar suporte a arquitetura de microsserviços são necessários. Neste capítulo, será apresentado como foi implementado cada um destes componentes e o código fonte está disponível em https://github.com/fabriziomarmitt/tcc-eng-software sob licença MIT.

Atualmente, a lógica para executar o Cancelamento e Suspensão está em uma PACKAGE no banco de dados Oracle. Esta PACKAGE possui muitas outras responsabilidades. Além de processar a operação, ela também executa as consequências do cancelamento, como remoção dos pagamentos futuros, remoção da assinatura do roteiro de entrega, remoção da liberação do conteúdo digital e cancelamento dos benefícios. Cada uma destas consequências deveriam ser rotinas separadas que fossem iniciadas pelo evento disparado quando um cancelamento ou suspensão fosse processado.

Assim, o trabalho necessário a ser feito para a migração deste módulo é:

- Isolar a lógica para cancelamento e suspensão da PACKAGE PCK_CANC_SUSPENSAO e implementar um microsserviço com duas ações, cancelamento e suspensão e remover a lógica da PACKAGE.
- Isolar a lógica de cada consequência e implementar um microsserviços para cada um deles.
- Implementar a PCK_CANC_SUSPENSAO chamando os novos serviços para promover compatibilidade com o restante do sistema.
- Implementar os eventos ASSIATURA_SUSPENSAO e ASSINATURA_CANCELAMENTO no serviço de mensageria e implementar a subscrição destes eventos nos serviços de consequências do cancelamento.
- Alterar camada de apresentação do sistema de assinaturas para invocar o novo cliente *WebApp*.

6.1 API Gateway com Spring Cloud Zuul

O API Gateway corresponde ao elemento da arquitetura de microsserviços que agrega os *endpoints* dos serviços de *backend* em um único ponto de acesso. Nesta implementação optou-se pela tecnologia Spring Cloud Zuul pelos motivos já mencionados.

A implementação completa do API Gateway com Spring Cloud Zuul pode ser acessada em https://github.com/fabriziomarmitt/tcc-eng-software/tree/master/api-gateway. Aqui está descrito o processo básico para seu uso.

O primeiro passo para sua implementação é criar a estrutura de diretórios como apresentado pela Figura 12.

Na Figura 12 é possível observar 6 arquivos principais que resumem a aplicação. O arquivo principal ApiGatewayApplication.java contém as definições necessárias para executar a aplicação, enquanto PreFilter.java executas funções que interceptam a requisição antes e depois de encaminhar ao *endpoint*.

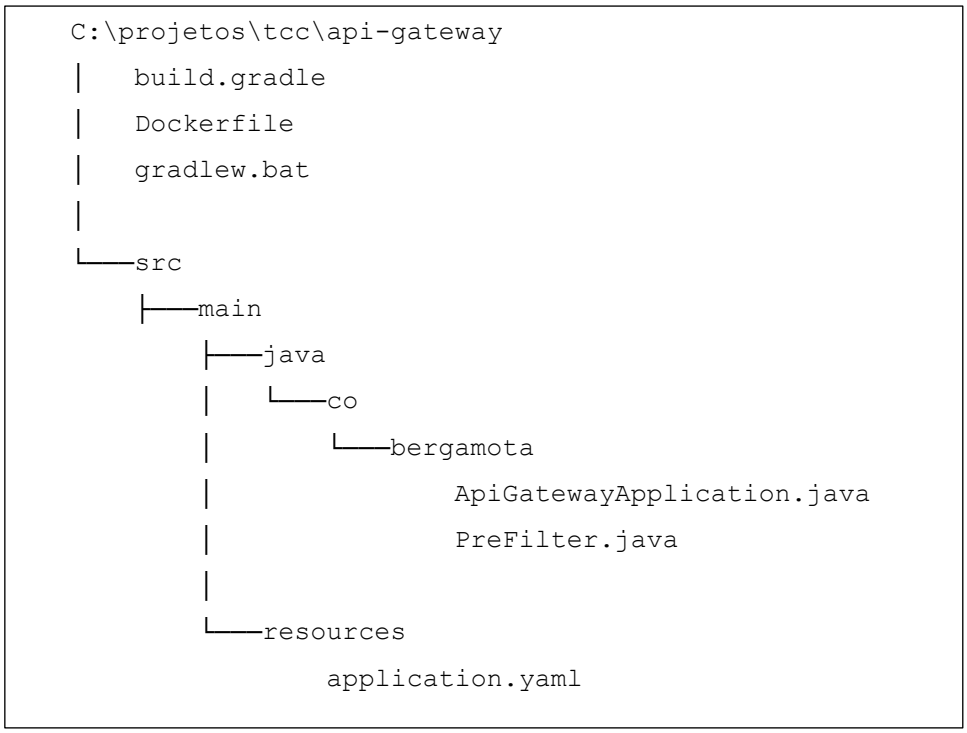

Figura 12 – Estrutura de diretórios API Gateway
Fonte: elaborada pelo autor

O arquivo *application.yml* é o arquivo necessário para o Spring saber em qual porta deve iniciar o serviço e é onde estão definidas as rotas para o Zuul.

Analisando a Figura 13, iniciando pela configuração de rotas do API Gateway, como pode ser observado, é muito simples adicionar novas rotas. Na figura, está registrada a rota *httpbin* que é acionada quando a aplicação for acessada na *url* http//localhost:8080/httpbin e, então, será direcionada para http://httpbin.org. Todo comportamento do API Gateway é definido, então, no arquivo de configuração.

```
zuul:
  routes:
    httpbin:
      path: /httpbin/**
      url: http://httpbin.org
ribbon:
  eureka:
    enabled: false
server:
  port: 8080
```

Figura 13 – Arquivo de configuração Spring Cloud Zuul
Fonte: elaborada pelo autor

Outros pontos importantes deste arquivo de configuração, é que nele você configura a porta para o serviço, bem como seu vínculo ao Discovery Server, que será visto nas próximas seções. Ressalta-se que com a integração do *Configuration Server*, este arquivo de configuração estará no repositório de configurações gerenciado por este componente. Em outras palavras, não será preciso editar o arquivo de configuração do API Gateway toda vez que uma rota for adicionada, basta alterar o *Configuration Server* e este se responsabilizará por distribuir a nova configuração entre as instâncias dos API Gateways.

Por fim, para executar a aplicação, pode-se utilizar o Docker ou o Gradle. Com Gradle, basta executar o comando conforme Figura 14.

```
.\gradlew.bat bootRun
```

Figura 14 – Executar API Gateway com Gradle
Fonte: elaborada pelo autor

Para executar com Docker, basta executar os comandos especificados na Figura 15.

```
docker build -t api-gateway .

docker run -p 8080:8080 --name api-gateway-instance-1 -d api-gateway
```

Figura 15 – Executar API Gateway com Docker
Fonte: elaborada pelo autor

Por fim, basta acessar uma das rotas configuradas no navegador, como por exemplo, http://localhost:8080/httpbin e então você deverá visualizar o mesmo conteúdo que acessar http://httpbin.org.

6.2 Message Broker com Spring Cloud Bus e RabbitMQ

O *RabbitMQ* consiste em um projeto que implementa um sistema de mensageria e é executado de maneira autônoma. No contexto desta aplicação, o RabbitMQ será executado através de um *contêiner* Docker e acessado via TCP/IP pelos microsserviços através da biblioteca Spring Cloud Bus que é responsável em fazer a integração com o sistema de mensageria.

Uma vez que o *RabbitMQ* está sendo executado, para implementar a integração com o microsserviço, basta adicionar a dependência *org.springframework.boot:spring-boot-starter-amqp* no arquivo *buid.gradle* e implementar a interface *RabbitListenerConfigurer* na classe publica principal do microsserviço.

Após realizadas as configurações, o microsserviço terá a capacidade de receber e enviar eventos ao *message broker*.

6.3 Microsserviços com Spring Boot

Os microsserviços contêm as lógicas de negócio extraídas do sistema legado e para este componente serão necessários 2 microsserviços que estão descritos nesta subseção.

6.3.1 Cancelamento e Suspensão de Assinaturas

Neste componente, será implementada a lógica contendo as regras de negócio de cancelamento e suspensão de uma assinatura, que atualmente estão na PACKAGE PCK_CANC_SUSPENSAO.

Os *endpoints* deste microsserviço são:

- `POST /v1/assinaturas/assinatura/{assinaturaId}/cancelar`: executa cancelamento da assinatura.
- `POST /v1/assinaturas/assinatura/{assinaturaId}/suspender`: executa suspensão da assinatura e o corpo da requisição ainda deve conter a data de início da requisição e a data de fim da requisição.

Os eventos gerados ao Message Broker por este microsserviço são:

- ASSINATURA_CANCELADA: evento informando aos serviços subscritos que a determinada assinatura foi cancelada. O *payload* deste evento será o ID da assinatura (assinaturaId) e o *timestamp* do cancelamento.
- ASSINATURA_SUSPENSA: evento informando aos serviços subscritos que a determinada assinatura foi suspensa. O *payload* deste evento será o ID da assinatura (assinaturaId) e as datas de início e fim da suspensão da assinatura.

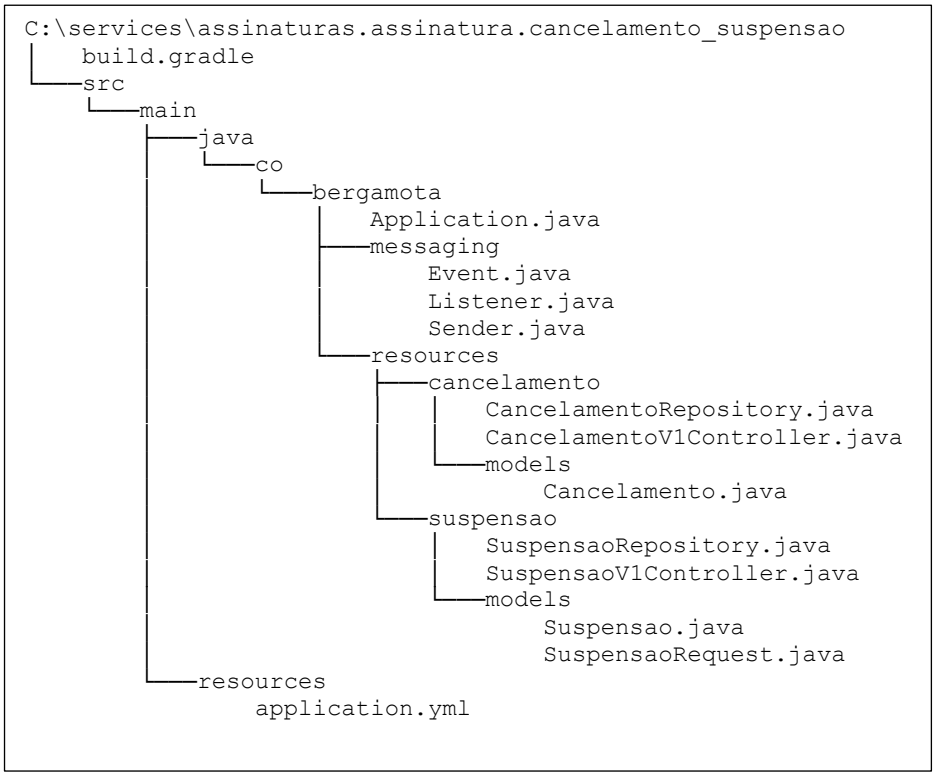

Figura 16 – Estrutura de arquivos do microsserviço Cancelamento e Suspensão
Fonte: elaborada pelo autor

A dependência direta deste microsserviços será do serviço de assinaturas, pois será preciso consultar o ID da assinatura para executar regras e validações de negócio. Devido

a esta dependência, pode-se argumentar que cancelamento e suspensão deveriam ser *endpoints* do microsserviço de assinaturas, descrito no próximo tópico, contudo, por entender que são grupos de domínio diferentes o suficiente para fazer com que evoluam em velocidades diferentes, optou-se por separar em dois serviços.

Sobre as lógicas de negócio, devido à confidencialidade, será apresentada um fluxo simplificado, em que os serviços apenas geraram um registro persistente de cancelamento e suspensão e disparará os eventos ao *Message Broker*.

A Figura 16 apresenta a estrutura de diretórios necessárias para um microsserviço. Esta estrutura mudará muito pouco de serviço para serviço, basicamente mudando os *models* e *constrollers*. Entre os arquivos mais importantes, destacam-se o *Application.java*, que é o arquivo principal que inicializa o serviço; *application.yml*, que contém as configurações de acesso ao banco de dados e ao *message broker*, os arquivos *repository*, que implementam a interface de acesso ao banco de dados; os arquivos *controller*, que contém a implementação dos *enpoints*, e os arquivos contidos em *messaging*, que implementam a comunicação com o *message broker*.

```
1. @RestController
2. @RequestMapping("/api/v1/assinaturas")
3. public class CancelamentoV1Controller {
4.     @Autowired
5.     CancelamentoRepository cancelamentoRepository;
6.     @Autowired
7.     Sender sender;
8.     @RequestMapping(value = "/assinatura/{assinaturaId}/cancelar",
         method = RequestMethod.POST)
9.     public ResponseEntity cancelar(@PathVariable String assinaturaId) {
10.         Cancelamento cancelamento = cancelamentoRepository
                  .save(new Cancelamento (assinaturaId, new Date()));
11.         sender.sendMessage(new Event(cancelamento, "ASSINATURAS_ASSINATURA_CANCELAMENTO"));
12.         return ResponseEntity.status(HttpStatus.CREATED).body(cancelamento);
13.     }
14.}
```

Figura 17 – Implementação do Enpoints de Cancelamento
Fonte: elaborada pelo autor

Como pode ser visto na Figura 17, implementar um *endpoint*, requer apenas algumas linhas de código. Nesta figura pode ser observado nas linhas 10, 11 e 12, a operação de salvamento no banco de dados, o envio do evento ao *message broker* e a envio da resposta ao cliente da solicitação. Estas 3 linhas representa as funções principais do microsserviço de cancelamento e suspensão.

6.3.2 Assinaturas

O microsserviço de assinaturas apresenta a mesma estrutura de diretórios da Figura 16, porém com seus c*ontrollers* e *repositories* específicos.

Uma particularidade deste microsserviço que é ressaltada, é o fato deste ser um *subscriber* do evento enviado na linha 11 da Figura 17. Uma vez que uma assinatura é cancelada no microsserviço de cancelamento, um evento é disparado ao *message broker* que, por sua vez, notifica aos *subscribers*.

A Figura 18 apresenta a implementação da classe e do método que recebe o evento de cancelamento de assinatura e muda o *status* da assinatura no banco de dados. A linha 6 desta figura apresenta a anotação com o evento que este método está esperando, isto é, quando o evento "ASSINATURAS_ASSINATURA_CANCELAMENTO" ocorrer, o método *handleEvent* é disparado. As linhas 9, 10 e 11 mostram a operação de busca da assinatura que foi cancelado, sua alteração de *status* e seu salvamento.

```
1. @Service
2. public class CancelamentoAssinaturaHandler {
3.   @Autowired
4.   AssinaturaRepository assinaturaRepository;
5.   private static final Logger log =
          LoggerFactory.getLogger(CancelamentoAssinaturaHandler.class);
6.   @EventHandler("ASSINATURAS_ASSINATURA_CANCELAMENTO")
7.   public void handleEvent(Event event){
8.       String assinaturaId = ((LinkedHashMap)
             event.getPayload()).get("assinaturaId").toString();
9.       Assinatura assinatura = assinaturaRepository.findOne(assinaturaId);
10.      assinatura.status = Assinatura.Status.CANCELADO;
11.      assinaturaRepository.save(assinatura);
12.      log.info("Assinatura {} cancelada com sucesso", assinatura.assinaturaId);
13.  }
14. }
```

Figura 18 – Implementação do *subscriber* do evento de cancelamento de assinatura
Fonte: elaborado pelo autor

6.4 *WebApp* e a nova interface da Aplicação

Como já foi apresentado na Figura 8, a nova interface do sistema para o *Contact Center* consiste em uma aplicação *web* que se comunica com as API's do sistema.

A implementação inicial da nova interface pode ser acessada através https://github.com/fabriziomarmitt/tcc-eng-software/tree/master/client-web-app . Em resumo, consiste em uma aplicação escrita em Node.js e HTML, que utiliza o próprio *engine* do Node.js como servidor web. O funcionamento básico, o *token* do usuário é verificado e este estiver autorizado, é redirecionado para a página de Cancelamento e Suspensão. Nesta página, realiza a operação e assim que finalizada, o usuário retorna ao Oracle Forms.

6.5 Integrando Oracle Forms e WebApp

Um dos desafios deste projeto é a convivência necessária entre a nova interface web e a interface em Oracle Forms. Uma maneira que ganhou bastante destaque através do desenvolvimento mobile é o protocolo plugável (*Pluggable Protocols*).

O funcionamento básico dos protocolos plugáveis é a utilização do URI (Identificador de recursos uniforme) utilizado para identificar recursos na internet. Cada URI possui uma parte que consiste no esquema (*schema*) que informa qual o protocolo do recurso, como por exemplo, HTTP na URI http://www.google.com indica que o recurso localizado em .com.google.www é HTTP.

Utilizando esta lógica, pode ser aplicada na comunicação entre Oracle Forms e a nova interface web. O primeiro, ao fazer uma chamada ao segundo, abre a URI http://localhost/cancelamento?token=123, nesse momento, o sistema operacional identificará o usuário deseja abrir um recurso HTTP situado em *localhost* e então executa o software padrão de execução de recursos HTTP, que são navegadores como o Internet Explorer ou Google Chrome. O navegador, então, carrega o recurso informando a variável *token* que será utilizada para efetuar a autenticação e autorização do usuário.

Na integração inversa, quando a interface web irá comunicar-se com a interface em Oracle Forms, pode-se utilizar a mesma lógica alterando apenas o protocolo. Quando a interface web deseja invocar, a URI é invocada é asforms://asca0001?token=456. Neste momento, o sistema operacional irá procurar em seus registros o programa padrão para

executar o esquema *asforms*, que será o Oracle Forms. Quando este for invocado pelo sistema operacional, receberá os parâmetros *token* para realizar a autenticação e autorização e o recurso *asca001* indica o *Form* que o usuário deseja abrir.

7 DISCUSSÃO

A implementação do módulo de cancelamento e suspensão demonstrou como é possível migrar o sistema sem ser necessário criar um novo sistema do zero. A metodologia utilizada possibilita a migração parcial de módulos ou submódulos do sistema que podem coexistir e interagirem de forma integrada e transparente.

Foi possível migrar totalmente as 2.989 LoC da tela ASCA0500 para a nova interface web que possui 59 LoC de JavaScript, 78 LoC de HTML e 134 LoC de CSS. É possível notar uma grande diferença entre a simplicidade da nova implementação da interface com a antiga e ainda a pontos de melhoria a serem feitas na nova interface. O tempo de desenvolvimento da nova tela foi de 271 LoC em 20 horas. Por uma outra perspectiva, migrou-se 2.989 LoC legado em 20 horas, uma média de 150 LoC de interface por hora.

Já o *backend* (isto é, o código que não está na interface), totaliza 12.389 LoC, distribuídas em 12 *Packages*, 12 *Procedures* e 21 *Functions* de PL/SQL. Como nem todas estas dependências são utilizadas apenas pelo cancelamento e suspensão, nem todas foram 100% migradas. De objetos de banco de dados, foram migrados 1.262 LoC da *Package* PCK_CANC_SUSPENSAO. Além disso, o módulo de cancelamento e suspensão foi totalmente desacoplado do sistema, ou seja, dos 163 objetos que o módulo dependia, agora depende apenas do serviço de assinatura que irá receber o evento de cancelamento assinatura. Mesmo assim, o serviço de assinatura não precisa estar funcional para que o módulo de cancelamento e suspensão funcione, pois estão totalmente desacoplados.

A migração das 1.262 LoC de *backend* resultou em 607 LoC em Java com *framework* Spring Boot. Este código foi desenvolvido em 100 horas, resultando a migração de 13 LoC de *backend* por hora. Além do tempo do tempo de desenvolvimento da interface e *backend*, teve o tempo de 80 horas de implementação do API Gateway, Message Broker e deploy com Docker.

Em suma, considerando que cada 1 hora se migra 50 LoC da interface e 13 LoC do *backend*, a migração das 59 telas mais usadas do sistema levaria aproximadamente 2.800 horas para interface e a migração total do *backend* levaria aproximadamente 15 mil horas.

Estas estimativas são extrapolações da primeira implementação e precisa ser calibrada a cada nova interação.

Mais uma vez a abordagem incremental adotada se torna um aliado ao negócio que não precisa investir as 17 mil horas de projeto de uma vez, a migração pode ocorrer à medida que for sendo necessária.

Para realizar uma comparação de resultados entre a nova arquitetura e a antiga, retomemos os motivos que levaram a empresa a buscar alternativas como a adoção de microsserviços para reengenharia do sistema de assinaturas: 1) reduzir o *time-to-market* de projetos através do desenvolvimento rápido de sistemas, 2) alta escalabilidade para contemplar picos de demanda, 3) baixo custo, 4) efemeridade e 5) alta disponibilidade.

O desenvolvimento através de microsserviços acelerou-se o desenvolvimento por dois motivos principais: a independência de cada serviço (desacoplamento) permitiu a paralelização do desenvolvimento do software, de forma que mais membros puderam-se juntar ao time. Como ressalta o gerente de sistemas da empresa, Anderson de Oliveira: "com a arquitetura antiga, monolítica, um trabalho de manter o trabalho paralelizado compatível era muito grande. O quanto mais alterações tinha uma nova versão do sistema, mais a probabilidade de *bugs* surgirem. Com a nova arquitetura, cada funcionalidade do sistema ficou independente, de forma que *deploys* de uma funcionalidade nada implica na outra, o que deixou todos muito mais confiantes em adicionar mais membros na equipe, além de ter reduzido muito o trabalho de operação para realizar mudanças".

Em relação à escalabilidade, apesar da dependência do Oracle Forms não permitir a escalabilidade total possível com a arquitetura de microsserviços, utilizar o OpenShift e os *contêineres* Dockers aumentaram a capacidade de escalar o sistema, de forma que quando é necessário mais carga, basta adicionar novas instâncias de cada *contêiner* e devido ao OpenShift e Kubernetes, esta operação é automatizável, o que reduz muito os custos para escalar a aplicação.

A redução de custos se dá em diversos fatores. A própria maneira de escalar a aplicação tornou-se muito mais barata, pois requer apenas mais capacidade de máquina virtual para os *contêineres* Docker serem executados e isto é muito mais econômico do que comprar licenças para mais processamento em um banco de dados Oracle. Além disso, ciclos de desenvolvimentos menores, reduzem os custos para implementação de

novas funcionalidades. Por fim, o uso de ferramentas Open Source, reduzem muito os custos com licenciamento.

Sobre a efemeridade, a redução de custos e facilidade de desenvolvimento possibilitou a criação de funcionalidades testes com um custo muito reduzido. E a redundância obtida através da *conteinerização* possibilitou a alta disponibilidade do sistema que, além de tudo, pode ser executado de forma distribuída, ou seja, no datacenter da empresa e na Amazon AWS ao mesmo tempo, por exemplo. A independência entre os serviços e a forma que interagem entre si através de eventos controlados e via protocolo HTTP, possibilita isto.

8 CONCLUSÃO

Apresentar a proposta de migração de um sistema com 17 mil horas é praticamente inviável, pois migrar um sistema não possui um ROI direto para o negócio. Por outro lado, manter um sistema legado, além de um risco para a empresa, progressivamente os custos de manutenção e evolução vão aumentando.

Este trabalho propôs uma abordagem para reengenharia do sistema, visando retirá-lo do estado de sistema legado para um sistema evoluível, planejado para ser mudado, através da implementação da arquitetura de microsserviços e migração incremental. Essa última, por sua vez, tem grande importância na questão financeira, como mencionado.

O objetivo principal de propor uma plataforma tecnológica que viabilize a reengenharia de sistemas como microsserviços foi cumprido. Com esta mesma abordagem, não apenas o sistema de assinatura pode ser migrado, mas, também outros sistemas da empresa que estão na mesma situação.

Comparando os resultados deste trabalho com as especificações de microsserviços descritos na Seção 2.5, foi possível atingir a decomposição funcional através da modularização do sistema e da migração de um módulo único com sucesso. O princípio da responsabilidade única também foi atendido também pela modularização funcional, apesar do módulo migrado executar duas funções (Cancelamento e Suspensão), acredita-se que elas executam operações suficientemente similares para serem consideradas uma mesma função.

Como mostra a Seção 6.3, foi possível cumprir o requisito da explicitação das interfaces. Já o Docker e OpenShift viabilizaram a publicação, *upgrade*, escalabilidade e substituição independentes. Por fim, como apresentado no Capítulo 6 de forma geral, o potencial poliglótico e comunicação leve foram contemplados devido a forma em que se comunicam os serviços através de HTTP e REST.

Com isto, acredita-se ter obtido os benefícios já citados na Seção 2.5, que são a facilidade de publicação, manutenção e entendimento, rápida execução, facilidade em efetuar mudanças, isolamento de falhas, além do aumento da possibilidade de colocar mais funcionalidades em produção de forma mais rápida, segura e barata que são requisitos de negócio importantes para este projeto de migração de arquitetura.

Como limitações deste trabalho, pode-se citar a falta de foco nos mecanismos de segurança necessários para implementar esta arquitetura. O motivo principal para que não tenha sido abordado foi o fato de segurança em microsserviços ser um tema amplamente documentado, não se fazendo necessário repeti-lo aqui. Ressalta-se, porém, que se faz necessário implementar esses mecanismos de segurança. Outra limitação importante, é a forma que foi elaborada a estimativa total do sistema que foi baseada na implementação teste. É preciso fazer a implementação de alguns módulos, rodando algumas interações e medindo tempo e LoC migradas para se obter uma estimativa mais real. Acredita-se que números aqui apresentados estão superestimados.

Por fim, a necessidade de implementação de testes automatizados é latente, visto que a degradação da qualidade do software foi um problema no passado e é preciso implementar medidas que impeçam que isso repita-se no futuro. Entre os motivos que não foi abordado este tema, destaca-se o caráter experimental deste trabalho que visa provar o conceito e metodologias aqui propostos.

BIBLIOGRAFIA

ABBOTT, Martin L.; FISHER, Michael T. **The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise**. 2. ed. Crawfordsville: Addison-wesley Professional, 2015. 624 p.

BENNETT, K.h.; RAJLICH, V.t.; WILDE, N.. **Software Evolution and the Staged Model of the Software Lifecycle**. Advances In Computers, [s.l.], p.1-54, 2002. Elsevier BV. http://dx.doi.org/10.1016/s0065-2458(02)80003-1.

EVANS, Eric. **Domain-Driven Design: Tackling Complexity in the Heart of Software**. Portland: Addison-wesley Professional, 2003. 563 p.

GUPTA, Arun. **Microservices, Monoliths, and NoOps**. 2015. Disponível em: <http://blog.arungupta.me/microservices-monoliths-noops/>. Acesso em: 11 jun. 2017.

LEHMAN, M.m.. **Programs, life cycles, and laws of software evolution**. Proceedings Of The Ieee**,** [s.l.], v. 68, n. 9, p.1060-1076, 1980. Institute of Electrical and Electronics Engineers (IEEE).

LEWIS, James; FOWLER, Martin. **Microservices**. 2014. Disponível em: <http://www.martinfowler.com/articles/microservices.html>. Acesso em: 25 nov. 2016.

LIENTZ, Bennet P.. **Issues in Software Maintenance. Acm Computing Surveys**, [s.l.], v. 15, n. 3, p.271-278, 1 set. 1983. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/356914.356919.

Martin, Robert Cecil. **Clean Code: A Handbook of Agile Software Craftsmanship**. 2009, Prentice Hall. ISBN 9780132350884.

NGUYEN, Phuc V.. **The Study and Approach of Software Re-Engineering. 2011**., Ho Chi Minh, 2011. http://cds.cern.ch/record/1408290/linkbacks.

RAJLICH, V.t.; BENNETT, K.h.. **A staged model for the software life cycle**. Computer, [s.l.], v. 33, n. 7, p.66-71, jul. 2000. Institute of Electrical and Electronics Engineers (IEEE). http://dx.doi.org/10.1109/2.869374.

SOMMERVILLE, Ian. **Software Engineering**. 2011., Pearson. ISBN 978-0-13-703515-1.