Análise da modernização de sistemas monolíticos legados para micro-serviços à luz da dívida técnica: um estudo de caso corporativo

Caio Henrique Bos Loureiro

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciência da computação
Orientador: Profa. Dra. Ana Cristina Vieira de Melo

São Paulo, Agosto de 2018

Análise da modernização de sistemas monolíticos legados para micro-serviços à luz da dívida técnica: um estudo de caso corporativo

Esta é a versão original da dissertação elaborada pelo candidato Caio Henrique Bos Loureiro, tal como submetida à Comissão Julgadora.

Resumo

Loureiro, C. H. B. **Análise da modernização de sistemas monolíticos legados para micro-serviços à luz da dívida técnica: um estudo de caso corporativo**. 2018. 120 f. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2018.

Devido ao crescente apelo por agilidade e flexibilidade no desenvolvimento de software, sistemas legados tornaram-se grandes gargalos nas corporações globais devido a baixa manutenibilidade e evolutibilidade que apresentam, sendo resistentes às mudanças cada vez mais constantes no dia-a-dia. Para evitar a queda da produtividade das equipes, a estratégia de *Strangler Application*, uma técnica faseada de modernização de software, vem sendo amplamente adotada devido ao baixo risco e custo associado. Essa estratégia favorece a adoção de arquiteturas de micro-serviços em detrimento aos sistemas monolíticos, já que possibilita a criação de aplicações menores, coesas e autônomas a cada etapa do processo de modernização. Embora 88% das empresas planejem modernizar seus sistemas nos próximos anos, a literatura sobre *Strangler Application* é escassa assim como os estudos sobre a arquitetura de micro-serviços, já que tais tendências surgiram recentemente na indústria e ainda não puderam ser totalmente exploradas pela comunidade científica. Nesse contexto, um estudo de caso exploratório foi desenvolvido num contexto real de uma grande empresa brasileira baseado na modernização de um sistema monolítico legado em uma arquitetura de micro-serviços através da estratégia de *Strangler Application*. Métricas de dívida técnica foram coletadas das aplicações para que pudesse ser inferida a qualidade do sistema resultante a cada etapa do processo de modernização, uma da formas de avaliar o sucesso do mesmo. Além disso, a presença do pesquisador no dia-a-dia da empresa durante três anos garantiu o levantamento de uma ampla gama de dados qualitativos. Como resultado, percebeu-se que a dívida técnica pode variar conforme as características do sistema legado e principalmente de acordo com a priorização dos módulos migrados, o que também influência no acoplamento e adequação da nova arquitetura descentralizada construída aos padrões existentes na literatura de micro-serviços. Como contribuição, além da análise de alguns padrões de comunicação utilizados na arquitetura de micro-serviços, um guia de boas práticas de *Strangler Application* foi desenvolvido buscando consolidar a literatura existente e proporcionar uma transição mais suave entre o mundo de software legado e moderno.

Palavras-chave: modernização de software, arquitetura de software, dívida técnica, sistemas legados, *Strangler Application*, micro-serviços, sistemas monolíticos.

Abstract

Loureiro, C. H. B. **Análise da modernização de sistemas monolíticos legados para micro-serviços à luz da dívida técnica: um estudo de caso corporativo**.2018. 120 f. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2018.

Due to the growing appeal for agility and flexibility in software development, legacy systems have become major bottlenecks in global corporations because of the low maintainability and evolutionability they present, being resistant to ever-changing day-to-day changes. To avoid a fall in team productivity, the Strangler Application strategy, a phased technique of software modernization, has been widely adopted because of the low risk and associated cost. This strategy favors the adoption of micro-services architectures in detriment to monolithic systems, since it allows the creation of smaller, cohesive and autonomous applications at each stage of the modernization process. Although 88% of companies plan to modernize their systems in the coming years, the literature on Strangler Application is scarce as are studies on the architecture of micro-services since such trends have recently emerged in the software industry and have yet to be fully explored by the scientific community. In this context, an exploratory case study was developed in a real context of a large Brazilian company based on the modernization of a legacy monolithic system in a micro-services architecture through the Strangler Application strategy. Technical debt metrics were collected from the systems so that the quality of the resulting system could be inferred at each stage of the modernization process, one of the ways to evaluate its success. In addition, the presence of the researcher in the day-to-day business of the company for three years ensured the collection of a wide range of qualitative data. As a result, it was noticed that the technical debt can vary according to the characteristics of the legacy system and mainly according to the prioritization of the migrated modules, which also influence the coupling and adequacy of the new decentralized architecture built to the standards existing in the micro- services. As a contribution, in addition to analyzing some communication patterns used in the micro-services architecture, a Strangler Application good practice guide was developed to consolidate existing literature and provide a smoother transition between the legacy and modern software.

Keywords: software modernization, software architecture, technical debt, legacy systems, Strangler Application, microservices, monolithic systems.

Sumário

Lista de Abreviaturas vii

Lista de Figuras ix

Lista de Tabelas xi

1 Introdução 1

1.1 Motivação 2

1.2 Objetivos 3

1.3 Organização do trabalho 3

2 Fundamentos 5

2.1 Dívida técnica 5

2.1.1 Classificação da dívida técnica 6

2.1.2 Tipos de dívida técnica 7

2.1.3 Métricas de dívida técnica 9

2.2 Arquitetura de micro-serviços 10

2.2.1 Caracteríticas dos micro-serviços 11

2.2.2 Integração entre micro-serviços 15

2.2.3 Outras estratégias de modularização 19

2.2.4 Comparação com monolíticos 20

2.2.5 Anti-padrões da arquitetura de micro-serviços 22

2.3 Modernização de sistemas 24

2.3.1 Estratégias de modernização 25

2.3.2 Desafios na modernização 30

3 Estudo de caso e metodologia aplicada 33

3.1 Estudo de caso: um sistema legado 33

3.1.1 Histórico 34

3.1.2 Tecnologia 34

3.1.3 Domínio 35

3.1.4 Arquitetura 36

3.2 Metodologia aplicada . . . 40
3.2.1 Estratégia de pesquisa . . . 40
3.2.2 Processo de coleta e análise incremental . . . 41

4 Modernização do sistema legado 47
4.1 Padrões de apresentação . . . 47
4.2 Marco zero . . . 48
4.2.1 Contextualização . . . 48
4.2.2 Avaliação . . . 49
4.3 Marco um . . . 57
4.3.1 Contextualização . . . 57
4.3.2 Implementações realizadas . . . 59
4.3.3 Avaliação . . . 66
4.4 Marco dois . . . 77
4.4.1 Contextualização . . . 77
4.4.2 Implementações realizadas . . . 78
4.4.3 Avaliação . . . 80
4.5 Marco três . . . 85
4.5.1 Contextualização . . . 85
4.5.2 Implementações realizadas . . . 87
4.5.3 Avaliação . . . 89
4.6 Marco quatro . . . 93
4.6.1 Contextualização . . . 94
4.6.2 Implementações realizadas . . . 94
4.6.3 Avaliação . . . 95
4.7 Marco final . . . 100
4.7.1 Contextualização . . . 100
4.7.2 Avaliação . . . 100

5 Análise dos resultados e conclusões 103
5.1 Resultados . . . 103
5.1.1 Variação da dívida técnica durante o processo de modernização . . . 103
5.1.2 Padrões de comunicação em arquitetura de micro-serviços . . . 105
5.1.3 Guia de boas práticas de *Strangler Application* . . . 108
5.2 Ameaças à validade . . . 112
5.3 Conclusões . . . 112
5.3.1 Trabalhos futuros . . . 113

Referências Bibliográficas 115

Lista de Abreviaturas

API	Interface de Programação de Aplicações (*Application programming interface*)
BPM	Gestão de Processos de Negócio (*Business Process Management*)
DSM	Matriz de dependência estrutural (*Dependency Structure Matrix*)
EDI	*Electronic Data Interchange*
EJB	*Enterprise JavaBeans*
ERP	Planejamento de recurso corporativo (*Enterprise resource planning*)
ESB	Barramento de serviços (*Enterprise Service Bus*)
HTTP	Hypertext Transfer Protocol
ITIL	*Information Technology Infrastructure Library*
JVM	Serviço de Mensagem Java(*Java Message Service*)
JEE	*Java Platform, Enterprise Edition*
JVM	Máquina Virtual Java (*Java Virtual Machine*)
NoSQL	*No Structured Query Language*
REST	Transferência de Estado Representacional (*Representational State Transfer*)
RMI	Método de Invocação Remota (*Remote Method Invocation*)
SGBD	Sistema Gerenciador de Banco de Dados
SOA	Arquitetura Orientada a Serviços (*Service Oriented Architecture*)
SQALE	Software Quality Assessment based on Lifecycle Expectations
TD	Dívida Técnica (*Technical Debt*)
XML	Linguagem de Marcação Extensiva (*eXtensible Markup Language*)

Lista de Figuras

2.1 Classificação Martin Fowler: Quadrante do débito técnico 7
2.2 Heterogeneidade de tecnologias 11
2.3 Comparação da escalabilidade em arquiteturas monolíticas e de micro-serviços 13
2.4 Pipeline de integração contínua 13
2.5 Integração por orquestração 17
2.6 Integração por coreografia 18
2.7 Estimativa da produtividade em razão da complexidade em monolíticos e micro-serviços 21
2.8 Processo faseado de Strangler Application 26
2.9 Passos de uma etapa de estrangulamento 28
2.10 Modelagem de domínio através de contextos limitados 30

3.1 Arquitetura do account-collector 38
3.2 Metodologia de coleta e análise faseada 42

4.1 Marcos do processo de modernização do sistema legado 47
4.2 Exemplo de diagrama de componentes na representação de arquiteturas 48
4.3 Sistema account-collector 49
4.4 Histório de execução dos testes de regressão do account-collector 51
4.5 Fluxo de recorrência antes do marco um 58
4.6 Fluxo de recorrência após marco um 59
4.7 Representação gráfica de um processo BPMN 60
4.8 Elementros gráficos da notação BPMN 61
4.9 Processo BPMN de cobranças de recorrência 62
4.10 Fluxo de criação de cobranças no transaction-manager 64
4.11 Fluxo de recorrência com foco na integração AC e TM 65
4.12 Migração do sistema de mensageria 66
4.13 Comportamento da dívida técnica somada de todos os sistemas no marco um 70
4.14 Fluxo de validação de cartão antes do marco dois 78
4.15 Fluxo de validação de cartão após marco dois 79
4.16 Arquitetura do creditcard-authorizer 80
4.17 Comportamento da dívida técnica somada de todos os sistemas no marco dois 82

4.18 Fluxo do renova-fácil antes do marco três 86
4.19 Fluxo do renova-fácil depois do marco três 87
4.20 Arquitetura do pagseguro-gateway 88
4.21 Fluxo de renova-fácil dentro do processo de recorrência 88
4.22 Comportamento da dívida técnica somada de todos os sistemas no marco três 91
4.23 Comportamento da dívida técnica somada de todos os sistemas no marco quatro . . 99
4.24 Comportamento da dívida técnica somada de todos os sistemas no marco final 102

5.1 Variação da dívida técnica durante o processo de modernização 104
5.2 Dígrafo dos módulos do monolítico 110

Lista de Tabelas

4.1 Marco Zero: dívida técnica do account-collector 50
4.2 Resumo das premissas de integração atendidas pelas tecnologias utilizadas 55
4.3 Marco um: dívida técnica do account-collector 67
4.4 Marco um: dívida técnica do transaction-manager 67
4.5 Marco um: dívida técnica do billing-business-rule 68
4.6 Marco um: dívida técnica do BBPM 69
4.7 Marco um: dívida técnica do processo como um todo 69
4.8 Marco dois: dívida técnica do account-collector 81
4.9 Marco dois: dívida técnica do creditcard-authorizer 82
4.10 Marco dois: dívida técnica do processo como um todo 82
4.11 Marco três: dívida técnica do account-collector 90
4.12 Marco três: dívida técnica do pagseguro-gateway 90
4.13 Marco três: dívida técnica do BBPM 91
4.14 Marco três: dívida técnica do processo como um todo 91
4.15 Marco quatro: dívida técnica do account-collector 97
4.16 Marco quatro: dívida técnica do transaction-manager 97
4.17 Marco quatro: dívida técnica do billing-business-rule 98
4.18 Marco quatro: dívida técnica do processo como um todo 98
4.19 Marco final: dívida técnica do account-collector 101
4.20 Marco final: dívida técnica do processo como um todo 101

5.1 Detalhes da comunicação dos sistemas criados 106

Capítulo 1

Introdução

O cenário corporativo atual impõe cada vez mais dificuldades às empresas de tecnologia que não são capazes de entregar inovação em tempo hábil. Tais inovações, como já advertido por Lehman na década de 80, fazem parte da necessidade contínua de evolução dos requisitos dos sistemas para que continuem a atender seus propósitos e não caiam em desuso. Entretanto, apesar de imprescídiveis, essas próprias alterações de negócio ao longo do tempo são responsáveis por aumentar a complexidade e deteriorar a qualidade dos sistemas [Leh80].

Conceitos similares de envelhecimento de software também foram explorados por Parnas, que afirmava que assim como humanos, softwares inevitavelmente envelhecem e com o tempo deixam de desempenhar suas funções da melhor forma [Par94]. Embora medicinas preventivas possam ser adotadas, a degração do sistema é inerente ao tempo e pode somente ser postergada, mas não evitada. Esses sistemas muito antigos e obsoletos são conhecidos na literatura e na indústria como legados, *softwares* frágeis, lentos e de díficil extensão, resistentes à modificações e evolução, mas que ainda exercem um papel crucial nas operações das organizações [BLWG99][Ben95].

O envelhecimento de software pode inclusive ser acelerado caso novas implementações não sejam realizadas da melhor maneira possível [Par94]. Em 1992, Ward Cunningham cunhou o termo dívida técnica justamente como forma de expressar as consequências dessas escolhas subótimas durante o desenvolvimento de software [Cun92], que embora proporcionem agilidade e capacidade de entrega de novas funcionalidades no curto prazo, são responsáveis pela geração de dívidas técnicas que depreciam ainda mais a qualidade do sistema e a produtividade das equipes no longo prazo.

Uma das maneiras para previnir a precoçe formação desses legados é realizar a manutenção da qualidade interna do sistema, efetuando por exemplo o pagamento gradual das dívidas técnicas geradas ao longo do tempo. Essa prática, conhecida na literatura como gerenciamento da dívida técnica [SG11], requer que as dívidas do sistema sejam descobertas e monitoradas. Para isso, atualmente utiliza-se ferramentas capazes de extrair diversas métricas de código-fonte por meio de análise estática e classificá-las quanto a padrões de conformidade. Além disso, outras métricas relacionadas a documentação, infraestrutura, testes, tecnologia, design e arquitetura de software também podem ser monitoradas [TAV13], mas não de forma automatizada.

Entretanto, em sistemas cujo gerenciamento da dívida técnica foi negligenciado por longos períodos, técnicas básicas de manutenção e refatoração de código não são mais suficientes para garantir manutenibilidade e evolutibilidade ao sistema. Nestes casos, opta-se pela modernização do sistema, que representa sua completa transição de um estado obsoleto para um estado moderno

[BSJH14][Yam17].

A técnica de *Strangler Application* é um das estratégias de modernização de software existentes [Fow04]. Ela consiste na extração gradual de módulos de um complexo sistema em aplicações mais simples, menores e autônomas, os micro-serviços [Ric16]. Em oposição aos sistemas monolíticos, essa nova arquitetura composta por diversos serviços permite a separação das responsabilidades de negócio do sistema em diferentes aplicações, capazes de escalar e evoluir independentemente, contribuindo para um baixo acoplamento tanto do código como da estrutura organizacional da empresa [New16]. Em contrapartida, esta arquitetura apresenta maior complexidade de implementação e requer maior conhecimento técnico.

1.1 Motivação

A atual velocidade da evolução tecnológica e das mudanças de requisitos de negócio faz dos sistemas legados os grandes vilões quando o assunto é agilidade. Embora estime-se que hoje ainda 200 bilhões de linhas de códigos legados sejam utilizados em todo mundo [BSJH14], uma forte tendência de modernização de software vêm se propagando com o advento de novas tecnologias e arquiteturas. Segundo [BBB], mais de 88% das corporações mundiais pretendem modernizar ao menos 25% de seus sistemas nos próximos 18 meses, sendo apenas 2% o número de empresas que não realizarão nenhum tipo de modernização.

Dentre as estratrégias de modernização, as abordagens modulares e iterativas como *Strangler Application* vem ganhando cada vez mais espaço em relação ao redesenvolvimento completo do sistema de forma única, popularmente conhecido por *Big Bang*. Esta antiga abordagem pretende ser utilizada em apenas 9% dos casos, enquanto estratrégias faseadas em aproximadamente 77% dos casos de modernização [BBB]. Apesar disso, as referências sobre *Strangler Application* ainda são escassas e compostas somente por publicações *online* sem muito aprofundamento.

Modernizações faseadsa apresentam menor custo e risco, além de favorecer a adoção natural da arquitetura de micro-serviços, já que cada etapa do processo pode dar origem a uma nova aplicação, criada como resultado da extração de uma funcionalidade do monolítico. Esse tipo de arquitetura descentralizada, tendência na indústria [AAE16], ainda apresenta um número pequeno de publicações acadêmicas, que, embora em ascensão nos últimos anos, ainda não responderam ou confrontram todas suposições em relação as características desse padrão arquitetural.

Dessa forma, devido a relevância dos assuntos até aqui abordados no dia-a-dia de desenvolvimento de software e da relação intrínseca entre os mesmos, um estudo de caso exploratório será desenvolvido com a observação do fenômeno da modernização de um sistema monolítico legado em micro-serviços num contexto real, não simulado, dentro de uma grande companhia de software, o UOL, maior empresa brasileira de conteúdo, tecnologia, serviços e meios de pagamentos digitais.

A presença do pesquisador deste projeto como funcionário da companhia trouxe uma oportunidade para o desenvolvimento desta pesquisa, permitindo a análise sob ponto de vista acadêmico e formal de práticas adotadas pela indústria e também colaborando para a disseminação do conhecimento. Assim, espera-se identificar boas práticas em relação aos padrões arquiteturais e ao processo de modernização adotado, utilizando a dívida técnica como referência de qualidade e sucesso do processo como um todo. Além disso, almeja-se esclarecer pontos ainda pouco explorados

pela literatura sobre ambos os temas e coloborar com uma ampla de gama de desenvolvedores que enfrentram esses mesmos desafios nas grandes comporações globais.

1.2 Objetivos

O desenvolvimento do estudo de caso num cenário corporativo traz uma grande quantidade de dados e informações a respeito dos temas centrais deste projeto: dívida técnica, modernização de software e arquiteturas de micro-serviços, que serão explorados em relação a literatura e a percepção dos desenvolvedores observados durante o processo. Dessa forma, espera-se contribuir para o esclarecimento de diversos conceitos relacionados a esses temas e da própria relação entre os mesmos. Além disso, almeja-se aprofundar em tópicos específicos através dos seguintes objetivos:

- Analisar a variação da dívida técnica durante o processo de modernização do monolítico legado.
- Identificar os padrões de comunicação utilizados na arquitetura de micro-serviços e a motivação para cada uma deles, comparando com os demais existentes na literatura.
- Desenvolver um guia de boas práticas da estratégia de *Strangler Application*, com os principais aspectos e premissas a serem seguidas, consolidando a literatura existente.

1.3 Organização do trabalho

No capítulo 2 são apresentados os principais fundamentos abordados por esse trabalho: dívida técnica, arquitetura de sistemas e modernização de software. No capítulo 3 é detalhado o estudo de caso, o sistema de software sob análise e a metodologia de pesquisa aplicada na coleta e análise de dados durante todo o processo de modernização. No capítulo 4, por sua vez, é apresentado o desenvolvimento e a avaliação do processo de modernização através de diversos marcos que represetam diferentes etapas da migração do software. Os resultados finais do estudo são apresentados no capítulo 5, juntamente com as ameaças à validade, conclusões finais e os caminhos ainda abertos para estudo.

Capítulo 2

Fundamentos

Neste capítulo serão definidos os principais fudamentos relacionados aos temas dívida técnica, arquitetura de micro-serviços e modernização de software, alicerces para o desenvolvimento deste projeto de pesquisa.

2.1 Dívida técnica

A expressão dívida técnica refere-se às consequências da realização de atividades de forma subótima, inferiores a maneira ideal, durante o processo de desenvolvimento de software [KNOF13]. O termo foi introduzido por Ward Cunningham em 1992 no seu relatório da OOPSLA [Cun92] como forma de descrever uma situação na qual fosse preciso agilidade no curto prazo em detrimento da qualidade do código no longo prazo, gerando naquele momento uma dívida técnica para o produto. Segue abaixo a transcrição de um dos trechos:

“ ...this process leads to the most appropriate product in the **shortest possible time**. Although immature code may work fine and be completely acceptable to the customer,**excess quantities will make a program unmasterable**, leading to extreme specialization of programmers and finally an **inflexible product. Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite**.... The danger occurs when the debt is not repaid. **Every minute spent on not-quite-right code counts as interest on that debt**. ”

Além da criação da metáfora da dívida técnica, Cunningham também explicitou os riscos do acúmulo inadvertido de dívidas na manutenibilidade e evolutibilidade dos sistemas, além de definir os elementos da dívida: juros e principal. De forma análoga ao comportamento de débitos financeiros, o acúmulo de dívida técnica gera juros na forma de esforço adicional para a realização de atividades futuras no sistema em decorrência da má qualidade das implementações anteriores. Já o principal corresponde exatamente ao montante do débito existente, representado no software pela diferença entre a implementação desejada e a de menor qualidade que foi implementada [KNOF13][EBO+15].

Embora a metáfora tenha sido cunhada em 1992, somente nos últimos passou a ser aplicada e tornou-se popular na indústria de software [GRI+14]. O mesmo aconteceu na área acadêmica, especialmente a partir de 2010, quando começaram a surgir publicações sobre o tema buscando entender, definir e especialmente contextualizar o conceito com as necessidades atuais [KNOF13]. Entretanto, a dívida técnica não se trata de um novo fenômeno no processo de desenvolvimento

[SG11] e sim apresenta uma íntima correlação com ideias anteriores ou do mesmo período tais como envelhecimento de software [Par94][Leh96], gerenciamento de risco [Boe91] e controle de qualidade de software, conceito tão antigo quanto a própria engenharia de software [GRI+14].

Dessa forma, a compreensão da dívida técnica evoluiu e hoje agrega diversos aspectos relacionados ao ciclo de vida de desenvolvimento de software, desde requisitos até documentação [EBO+15]. Contudo, a qualidade interna de software ainda é a principal perspectiva explorada pela indústria [KNOF13][EBO+15] que provê ferramentas tais como o SonarQube [CP13], uma plataforma de análise estática de código capaz de obter diversas métricas de qualidade e representá-las de maneira fácil para qualquer desenvolvedor.

Essa relação entre dívida técnica e qualidade muitas vezes não é clara na indústria [GRI+14]. A dívida técnica vai muito além de métricas de qualidade é do código-fonte dos sistemas. Ela pode ser aplicada como uma ferramenta estratégica durante o processo de desenvolvimento para garantir uma rápida entrega ao cliente final e gerar ganhos para o negócio [KNOF13]. Assim, além do ponto de vista técnico, a metáfora é também aceita por outras áreas da empresa que passam a entender a importância da agilidade de forma conjunta com a necessidade de manutenção da qualidade do sistema [EBO+15].

Nesse aspecto, para que a agilidade obtida seja sustentável ao longo da evolução do produto é necessário que haja um gerenciamento da dívida técnica, buscando atender as necessidades da companhia no curto e longo prazo. Para isto, diversas técnicas de gerenciamento podem ser aplicadas para evitar o acúmulo excessivo da dívida e o pagamento recorrente de seu juros, o que poderia comprometer a produtividade das equipes de desenvolvimento [BCG+10][CAAA15]. Entretanto, é comum na indústria, se não o padrão, casos onde a dívida técnica não é gerenciada e a qualidade dos sistemas comprometida, criando ao longos dos anos os legados apresentados no capítulo 2.3, cuja alternativa mais provável é a modernização.

2.1.1 Classificação da dívida técnica

A classificação da dívida técnica é dada de acordo com a origem ou razão pelo qual ela foi gerada. Dessa maneira, diferentes visões são encontradas na literatura, sendo a classificação como intencional e não intencional a mais adotada [Ste10][BCG+10]. A dívida não intencional é aquela gerada sem propósito, por descuido, falta de experiência ou desconhecimento da equipe de desenvolvimento. Já a dívida intencional tem propósito, gerada com o intuito de acelerar o desenvolvimento no curto prazo em prejuízo à manutenção e evolução no longo prazo.

Pelo fato de ser uma metáfora, não uma teoria ou conceito científico, a definição e classificação alimenta opiniões divergentes. Uncle Bob defende que bagunça é somente um código mal escrito e não uma dívida técnica [Bob09]. Já Martin Fowler propôs uma classificação mais abrangente capaz de abordar todas as diferentes opiniões [Fow09], conforme quadrante representado pela figura 2.1, condensando as dívidas técnicas de acordo com sua natureza em quatro tipos:

- **Irresponsável e intencional**: Não há tempo para o design então utiliza-se uma solução rápida sem preocupação com a qualidade e controle das dívidas técnica.
- **Prudente e intencional**: Não há tempo para o design então utiliza-se uma solução rápida e de menor qualidade, postergando o pagamento destas dívidas para um momento futuro.

- **Irresponsável e não intencional**: O time não tem experiência nem consciência dos princípios básicos de design e padrões de software, gerando de maneira não intencional um código cheio de dívidas.
- **Prudente e não intencional**: Mesmo em times experientes, somente quando a solução final é entregue que todos os requisitos ficam realmente claros e percebe-se que a solução desenvolvida poderia ser melhor.

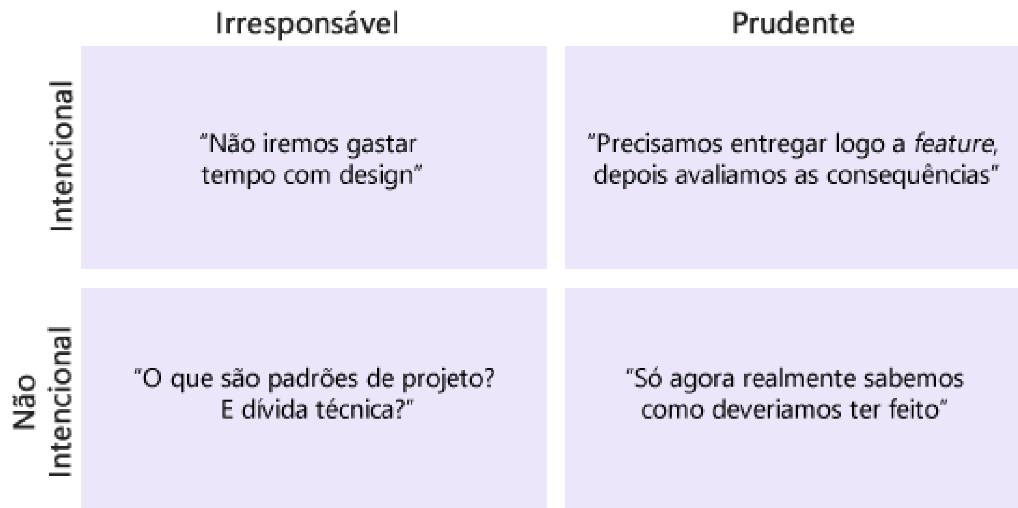

Figura 2.1: *Classificação Martin Fowler: Quadrante do débito técnico*

A classificação da dívida técnica evidência seu poder estratégico na tomada de decisões balanceando aspectos da qualidade e de negócio. Também mostra que a mesma pode surgir de forma não esperada devido simplesmente ao desconhecimento ou falta de preparo das equipes de desenvolvimento.

2.1.2 Tipos de dívida técnica

Embora o termo dívida técnica tenha sido cunhado em referência às práticas de codificação, hoje a metáfora é aplicada para uma ampla gama de atividades relacionadas ao desenvolvimento de software, desde modelagem arquitetural até a presença de documentação. A categorização das dívidas técnicas em tipos também apresenta divergências na literatura, as quais serão abordadas abaixo durante a descrição dos principais tipos existentes [TAV13].

Código

As dívidas técnicas de código são as mais comuns e umas das mais simples de serem corrigidas. O escopo desse tipo de dívida é pequeno, restrito a uma classe, método ou bloco de código. Geralmente resultam de código mal escrito, falta de padrão de codificação ou lógica mal organizada. São exemplos dessas dívidas a duplicação de código, a alta complexidade ciclomática e a existência de código sem utilização.

Documentação e distribuição do conhecimento

Esses dois tipos de dívida técnica podem ser agregados pois causam o mesmo efeito: queda de produtividade na manutenção e evolução do sistema em decorrência da falta de conhecimento sobre o mesmo. Enquanto a dívida de documentação caracteriza-se pela inexistência, desatualização ou dados errôneos na documentação, a dívida de distribuição do conhecimento é subjetivo e evidencia a falta de disseminação do conhecimento do sistema, restrito a uma pequena parcela dos desenvolvedores.

Ambiente de desenvolvimento

As dívidas relacionados ao ambiente de desenvolvimento contemplam as atividades de construção e suporte de software e infraestrutura. Essas são geralmente atividades operacionais recorrentes com alto potencial de automatização. A realização manual dessas tarefas caracteriza uma dívida pois implica num alto custo no longo prazo em função de maior rapidez no curto prazo. O pagamento da dívida é a simples automação da atividade.

Testes

A ausência de testes automatizados ou a baixa cobertura dos testes existentes caracteriza a dívida técnica de testabilidade. A baixa qualidade dos testes automáticos gera a necessidade da execução de testes manuais para cada entrega, em cada ambiente de desenvolvimento. Além disso, expõe a empresa a diversos riscos como perda de receita, *bugs* em produção e insatisfação dos clientes.

Defeitos

A categorização de defeitos como uma dívida técnica é divergente na literatura. Enquanto alguns não citam, outros argumentam que apenas elementos sem impacto ao usuário final podem ser considerados dívida técnica. Os estudos que consideram defeitos como dívida restringem-os apenas àqueles de menor importância, que não inviabilizam a utilização do sistema pelo usuário final.

Tecnologia

A dívida técnica de tecnologia também não é unanimidade nos estudos existentes. Relacionado diretamente ao envelhecimento do software, esse tipo de dívida é reflexo da evolução de novas ferramentas e tecnologias capazes de tornar o desenvolvimento de software mais eficaz. Quando não atualizadas, as tecnologias legadas não são capazes de entregar a mesma eficiência em relação às mais recentes, gerando um juros constante ao desenvolvimento.

Design

A dívida técnica de design aplica-se essencialmente para linguagens orientadas a objetos e corresponde à implementação de classes que não seguem os padrões de design estabelecidos na literatura. Tais feitos impactam diretamente no alto acoplamento e baixa coesão das classes do sistema. Diferentemente da dívida de código, o escopo da dívida de design é o conjunto de classes do sistema e como elas se relacionam.

Arquitetura

Muitas vezes, a dívida de arquitetura e design são apresentadas conjuntamente, dado a maior granularidade desses tipos de dívida. Entretanto, o escopo das dívidas arquiteturais é ainda maior, pois focam nas relações de dependência entre bibliotecas, pacotes e até mesmo outros sistemas. As violações de propriedades arquiteturais como modularidade, portabilidade e escalabilidade são exemplos de dívidas de arquitetura.

2.1.3 Métricas de dívida técnica

Na prática, a medição da dívida técnica de um software baseia-se em métricas de qualidade extraídas do código-fonte estático da aplicação e do resultado de sua compilação. Umas das ferramentas mais conhecidas e utilizadas na indústria para a medição dessas métricas é o SonarQube [CP13], ferramenta gratuita que vem sendo evoluída constantemente para facilitar a monitoração e gerenciamento de dívida técnica pelas equipes de desenvolvimento de software.

O SonarQube, também conhecido por Sonar, tem como principais características a flexibilidade e extensibilidade, obtidas através de mais de 50 plugins disponíveis e cobertura para mais de 20 linguagens diferentes. Para a linguagem java, por exemplo, o Sonar fornece plugins com a implementação de diversas regras de outras ferramentas amplamente utilizadas para garantir qualidade de código e padrões de codificação, tais como FindBugs, CheckStyle e PMD, provendo uma vasta detecção de anti-padrões. Entretanto, a plataforma Sonar deixou de fornecer nos últimas anos algumas métricas reconhecidas como LCOM e DSM, limitando sua análise em relação as dívidas de design e arquitetura. A seguir, as principais métricas de análise estática utilizadas para contextualizar dívida técnica serão apresentadas:

- ***Technical Debt***: Também conhecida como *SQALE quality index*, esta métrica representa de maneira consolidada o custo de remediação da dívida técnica total do sistema. Ela é calculada a partir do método SQALE, que utiliza um modelo de estimação baseado na análise estática de código, relacionando um custo de correção, em tempo, para cada não conformidade detectada [LI12]. Dessa forma, a métrica *Technical Debt*, ou simplesmente TD, é expressa em unidades de trabalho, unidades monetárias ou em homens/hora [SQA]. O SonarQube utiliza a unidade "dias de trabalho", padronizando o esforço de 1 homem.

- ***Technical Debt Ratio***: Ou simplesmente TD Ratio, é a métrica que associa a métrica *Technical Debt* ao tamanho do sistema, dada pela divisão do custo de remediação pelo custo de desenvolvimento total do software. Para isto utiliza-se algumas padronizações em relação ao custo de desenvolvimento de uma linha de software. Na prática, o *TD Ratio* fornece um dado muito mais preciso e útil para comparações entre sistemas de diferentes tamanhos que o TD.

- **Complexidade Ciclomática**: Métrica criada em 1976 por McCabe [McC76], capaz de mensurar a complexidade de um determinado módulo, classe ou método a partir da contagem do número de caminhos independentes que podem ser executados. Para cada ação condicional no código, o valor é incrementado. Um método com complexidade 5, por exemplo, precisa de 5 testes unitários para que sua cobertura seja total. McCabe aconselha complexidade máxima de 10 para métodos, sendo 21 já um número de elevado risco.

- **Complexidade Cognitiva**: Métrica desenvolvida pela SonarSource, empresa responsável pela ferramenta SonarQube, como forma complementar a complexidade ciclomática. Enquanto a complexidade ciclomática foca mais testabilidade, a complexidade cognitiva foca na manutenibilidade baseado no grau de dificuldade de entendimento de um método, classe ou módulo. O cálculo é similar, mas leva em consideração outros critérios como aninhamento de chamadas e a simplificação de métodos getter e setters [COG].

- ***Issues***: São possíveis problemas detectados no código, classificados em três diferentes níveis: *bugs*, vulnerabilidades e *bad smells* e. Os *bugs* são problemas que possivelmente podem causar erro ao usuário da aplicação e por isso apresentam maior risco. As vulnerabilidades são erros que podem comprometer a segurança do sistema. Os bad smells, por sua vez, são anti-padrões de codificação implementados no código que aumentam sua complexidade e podem comprometer a produtividade de desenvolvimento. Issues cobrem desde tratamento de exceções, detecção de deadlock, código-morto até estilos de escrita de codificação.

- **Duplicações**: Métrica que representa, em porcentagem, a quantidade de código duplicado existente, calculado a partir da detecção de blocos de linhas duplicados várias vezes no mesmo sistema.

- **Tamanho**: Mais uma categoria que uma simples métrica, o tamanho do sistema pode ser identificado por diversos dados obtidos através do código-fonte. Dentre os dados mais comuns estão o número de linhas de código(LOC), métodos e classes do sistema. Estas métricas além do tamanho, auxiliam no entendimento da complexidade como um todo do sistema.

2.2 Arquitetura de micro-serviços

O termo micro-serviço surgiu nos últimos anos para designar uma forma particular de arquitetura de sistemas, na qual um conjunto de pequenas aplicações autônomas e independentes são capazes de interagir entre si e atuar como se fossem apenas um único complexo sistema. Tais pequenas aplicações são conhecidas como serviços, comunicam-se geralmente através de protocolos leves e não intrusivos como HTTP e possuem uma única responsabilidade de negócio [FL14][New15].

Este padrão arquitetural começou a ser aplicado nas indústrias de software no começo da década e consolidou-se de vez em 2014, quando várias autoridades e praticantes, referências no assunto, passaram a escrever sobre o tema buscando defini-lo de uma forma mais clara [FL14][PZA+17]. Artigos acadêmicos sobre o assunto surgiram somente recentemente e ainda de forma escassa, sendo ainda um grande campo de estudo em aberto [AAE16].

Embora não haja uma definição formal da arquitetura de micro-serviços [FL14], existem diversas características que serão mostradas ao longo desta seção que distinguem esse padrão arquitetural dos demais, como o padrão monolítico. As arquiteturas monolíticas ainda são comuns mas vem perdendo cada vez mais espaço para os micro-serviços dado as implicações de se manter todas as funcionalidades em uma única e complexa aplicação, difícil de escalar, manter e evoluir, afetando a produtividade das equipes de desenvolvimento [AAE16][DGL+17].

Para proporcionar uma alta coesão a cada micro-serviço deve-se garantir que sua responsabilidade seja única e específica, conceito popularmente conhecido por *'Single Responsibility Principle'*

[New15]. Esse princípio enfatiza que todas as partes do código que mudem juntas por uma mesma razão precisam ser agrupadas, separando-as das demais partes que mudam apenas por razões diferentes. Dessa forma, os limites de cada serviço precisam ser determinados de acordo com específicas funcionalidades de negócio, evitando o risco de crescerem inadvertidamente. Micro-serviços forçam tal segregação de responsabilidades de maneira natural pois todo o código-fonte é mantido e entregue de forma separada, em diferentes aplicações.

Entretanto, não há uma regra específica que determine a granularidade exata de cada micro-serviço. Na prática, um serviço deve ser simples o suficiente para que possa ser reescrito num curto período de tempo e que qualquer desenvolvedor possa dar manutenção de maneira rápida. Além disso, caso toda entrega em um serviço seja acompanhada de uma entrega em outro serviço, sinal de que os mesmos estão muito acoplados e talvez pudessem ser agrupados. Em contrapartida, caso apenas uma pequena parte de um serviço seja alterada com frequência, ignorando-se as demais, possivelmente aquela funcionalidade deva ser extraída em um novo serviço [FL14][DGL$^+$17]. Mais detalhes de granularidade, responsabilidade e composição de micro-serviços serão aprofundados na seção 2.3.1.1, durante a apresentação das técnicas de *Strangler Application*.

2.2.1 Caracteríticas dos micro-serviços

Nesta seção serão conceituadas as principais características da arquitetura de micro-serviços e quais os benefícios e desafios ao adotar-se cada uma dessas qualidades.

Heterogeneidade

A modularização do sistema em diversos serviços segregados, cada qual com seu próprio código-fonte, permite a adoção de diferentes tecnologias para cada um. A escolha da linguagem, banco de dados ou framework para cada micro-serviço é livre e pode ser feita diante das funcionalidades necessárias para cada aplicação. Além disso, outras razões como o nível de performance desejado, conhecimento do time responsável e agilidade na entrega também afetam os critérios de escolha. A figura 2.2 exemplifica um sistema com tal heterogeneidade, composta por três aplicações, cada uma implementada em diferentes linguagens e com tecnologias de bancos de dados variadas.

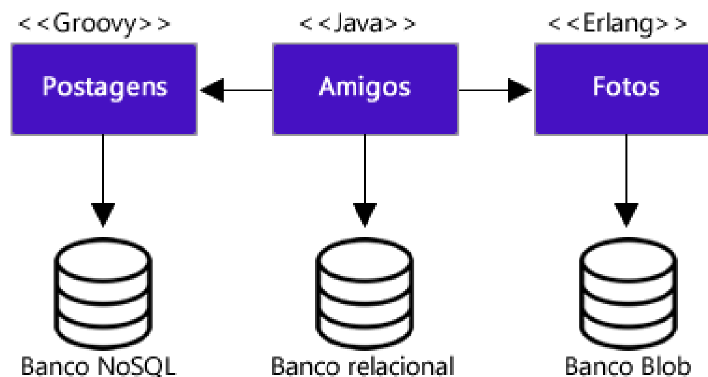

Figura 2.2: *Heterogeneidade de tecnologias*

Embora a adoção de múltiplas tecnologias e linguagens possa ser vantajosa, um alto custo implícito de manutenção pode ser gerado. Por isso, diversas empresas restringem o uso da tecnologia

a algumas linguagens específicas ou até mesmo plataformas. Netflix e Twitter são exemplos de companhias que utilizam apenas linguagens da plataforma JVM(*Java Virtual Machine*), tais como Java, Scale e Groovy [New15]. Tal restrição também facilita o compartilhamento de bibliotecas entre diferentes times da corporação, evitando retrabalho.

Resiliência

Resiliência é uma qualidade de quão recuperável, elástico e adaptável um corpo deve ser. Para micro-serviços, ser resiliente representa ter capacidade de recuperação dado que falhas aconteçam em alguma parte do sistema. Para isso, todos serviços precisam ser construídos tolerantes a falhas, uma vez que qualquer chamada a um serviço remoto pode não ser respondida a tempo dada indisponibilidade da aplicação ou latência de rede.

Um conceito chave em resiliência é a presença de um anteparo, algo capaz de conter a propagação de um erro. Micro-serviços devem ser construídos exatamente como anteparos, capazes de isolar erros e evitar que se propaguem por todo o sistema, gerando indisponibilidade apenas no serviço afetado enquanto os demais podem continuar funcionando e provendo suas funcionalidades, algo impensável numa arquitetura monolítica.

Serviços tolerantes a falhas precisam ter capacidade de recuperação dada qualquer indisponibilidade. Para isso mecanismos de retentativas são necessários, tais como filas assíncronas ou controles através de banco de dados. Além disso, os serviços precisam ser idempotentes, propriedade essencial que garante que qualquer ação não será aplicada duas vezes, ou que o resultado será o mesmo, independentemente de quantas vezes for executado. Esse controle é necessário em micro-serviços pois além de servir de controle de concorrência entre as diversas instâncias de uma mesma aplicação, evita que retentativas sejam processadas duas vezes.

Dessa forma, apesar de toda essa complexidade adicional de construir-se serviços tolerantes a falhas, uma maior confiabilidade e disponibilidade é garantida ao sistema como um todo.

Escalabilidade

Mais um benefício que uma característica, a propriedade de escalabilidade garante que apenas os serviços necessários sejam escalados em múltiplas máquinas ao invés de todo o sistema, como no caso dos monolíticos. Além da economia de custos de infraestrutura e da compatibilidade com os serviços de cloud existentes, arquiteturas de micro-serviços já são construídas para serem distribuídas e aptas a adição de novas instâncias concorrentes para uma mesmo serviço, diferentemente da realidade de grande parte dos sistemas monolíticos, não preparados para rodarem em múltiplas máquinas devido a falta de controle de concorrência e de serviços idempotentes. A figura 2.3 apresenta ambas realidades: à esquerda uma aplicação monolítica replicada em quatro servidores, como todas suas funcionalidades replicadas de forma igual, à direita um arquitetura de micro-serviços, onde cada serviço é responsável por uma única responsabilidade e pode ser replicado de acordo com a demanda independente dos demais.

Facilidade nas entregas em produção

A entrega em produção de alterações nos sistemas monolíticos quase sempre é realizada sob acompanhamento e necessita de testes e validações manuais. Mesmo uma pequena alteração apre-

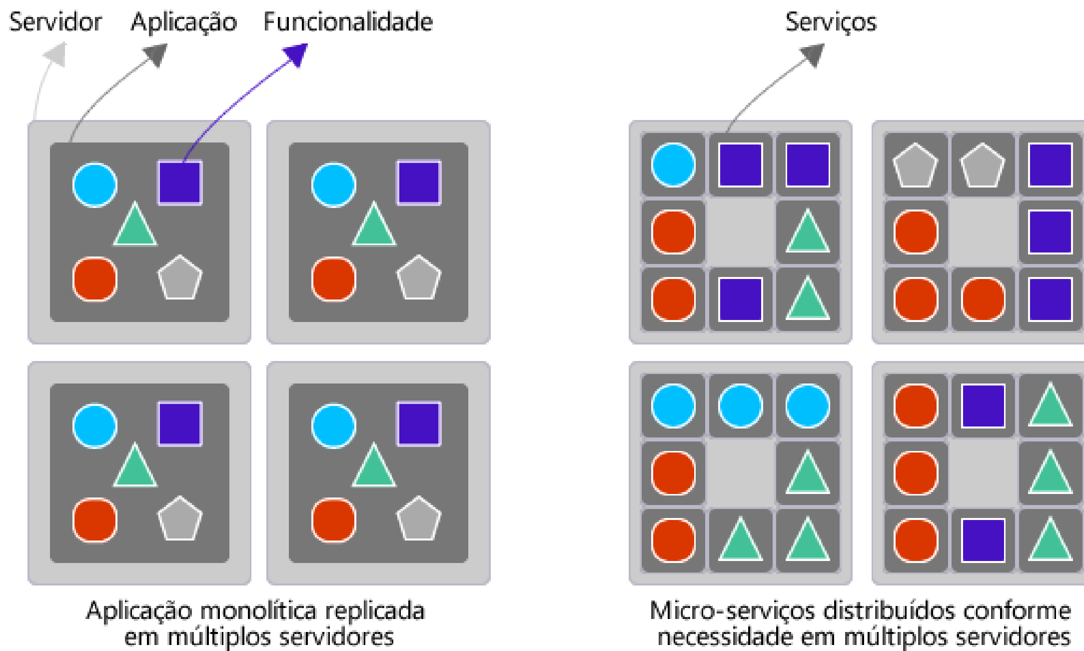

Figura 2.3: *Comparação da escalabilidade em arquiteturas monolíticas e de micro-serviços*

senta um alto risco pois pode impactar diversas das funcionalidades do sistema, dado que toda a aplicação é entregue simultaneamente. Na prática, evita-se entregas frequentes nos sistemas monolíticos devido ao alto risco e custo associado a cada entrega, além do medo e stress gerado entre a equipe que atuou no desenvolvimento.

A arquitetura de micro-serviços possibilita uma abordagem totalmente distinta, focada na entrega contínua e constante das aplicações. Dado seu caráter modular, os micro-serviços provém isolamento e independência ao realizar entregas de novas features em produção, evitando a propagação de erros a outras funcionalidades. Além disso, micro-serviços são tolerantes a falhas e suportam retentativa, possibilitando uma rápida atuação e correção caso o sistema fique indisponível.

Outra vantagem ao adotarmos micro-serviços é a realização de testes ágeis automatizados [HRJ+16]. Os testes ágeis são construídos sob perspectiva dos clientes da aplicação e focam exclusivamente nas funcionalidades do sistema, garantindo o comportamento adequado perante os usuários. Em micro-serviços a criação desses testes é facilitada pois as funcionalidades são enxutas e bem definidas. Além disso, podem ser executados em um curto período de tempo se comparados aos longos testes integrados dos sistemas monolíticos.

Figura 2.4: *Pipeline de integração contínua*

Escopo reduzido, testes ágeis e o surgimento de novas tecnologias de automação possibilitaram a realização de entregas contínuas em produção. O conceito de entrega contínua [Fow06] foca na constante entrega de pequenas alterações de forma automatizada através de diversas etapas sequen-

ciais de compilação e testes até sua chegada ao ambiente de produção, conforme exemplificado na figura 2.4. Essa abordagem, característica em arquiteturas de micro-serviços, encoraja as equipes a realizar entregas cada vez mais rápidas e sem receios, proporcionando baixo risco e velocidade nas entregas.

Alinhamento Organizacional

Os conceitos de granularidade aplicados a arquitetura de software também podem ser empregados na granularidade das equipes de desenvolvimento. A maioria das desvantagens em se manter grandes sistemas também são evidentes na manutenção de grandes equipes de desenvolvimento. Já em 1968, Melvin Conway observou que qualquer empresa que projeta um sistema inevitavelmente produz um projeto cuja estrutura é uma cópia da própria estrutura existente na organização [Con68]. Tal observação ficou conhecida como Lei de Conway e se aplica a todas áreas de negócio, inclusive no desenvolvimento de sistemas.

A Lei de Conway prevê que grandes equipes trabalhando em grandes sistemas tendem a promover alto acoplamento, enquanto pequenos times independentes focados em projetos de menor escopo produzem sistemas com boa interface e baixo acoplamento. Empresas como Netflix e Amazon já adotam esse tipo de organização [New16] através de pequenas equipes independentes e com alta autonomia, responsáveis por todo ciclo de vida dos serviços que deram origem, desde seu desenvolvimento a sua manutenção em produção [Gra06].

Equipes pequenas e autônomas trabalhando em micro-serviços isolados e com funcionalidades de negócio específicas inevitavelmente precisam ter conhecimento em áreas distintas como negócio, interfaces de usuário, serviços de backend, banco de dados, etc. Dá-se o nome de multi-funcionais a times com essas características, capazes de executar todas as atividades relativas ao desenvolvimento de software por conta própria [FL14]. Embora não seja regra, na maioria dos casos cada membro é polivalente e pode atuar em diferentes tipos de tarefas.

Dessa forma, para que a arquitetura de micro-serviços seja aplicada com sucesso toda uma cultura de desenvolvimento precisa ser revista e ajustada para otimizar os resultados. Muitas das empresas podem falhar ao adotar esse padrão arquitetural caso não adequem também seu padrão comportamental e organizacional, principalmente caso ignorem os conceitos de granularidade e polivalência das equipes, correndo o risco de produzirem sistemas altamente acoplados.

Design evolucionário

Em grandes corporações é comum ainda a presença de grandes legados, escritos em linguagens ultrapassadas e não mais suportadas, cuja manutenção é custosa e requer coragem por parte dos desenvolvedores. Embora pareça a opção correta, a substituição desses sistemas pode ser sempre postergada devido a sua alta complexidade, custo e risco. Caso esse grande sistema fosse composto de pequenos micro-serviços independentes, a substituição ou mesmo descarte de cada um deles seria uma tarefa simples, de baixo-risco e realizável num curto período de tempo [New15]. Esta habilidade de fácil substituição e evolução inerente a micro-serviços garantirá maior adaptabilidade e rapidez na tomada de decisões às empresas que adotarem tal arquitetura.

2.2.2 Integração entre micro-serviços

A adoção da arquitetura de micro-serviços traz consigo uma vasta gama de novos desafios em relação a como esses serviços devem se comunicar. As chamadas que antes eram de métodos locais agora são realizadas por mecanismos de comunicação entre processos, ou *inter-process communication(IPC)* [Ric14], através de chamadas a serviços remotos através do protocolo HTTP ou filas de mensageria [FL14]. A separação física dos serviços em distintas aplicações trouxe diversos benefícios mas também uma maior complexidade à integração de todo sistema.

Os aspectos da integração entre os serviços são de extrema importância para o sucesso de toda arquitetura de micro-serviços. Tais aspectos são determinantes para garantir o baixo acoplamento e a autonomia entre os serviços, permitindo entregas de forma isolada e sem impacto aos clientes. Também são fundamentais para assegurar uma alta coesão e definição clara dos objetivos de cada serviço. Entretanto, para que a integração entre os micro-serviços seja bem sucedida, algumas premissas precisam ser consideradas na hora da escolha das tecnologias e estilos existentes [New15]. As premissas mais relevantes são:

- **Novas entregas sem impacto aos clientes**: A capacidade de entregar alterações em produção de forma rápida e com baixo risco dado o escopo reduzido dos serviços é uma das grandes vantagens da arquitetura de micro-serviços. Entretanto, caso seja comum alterações que impactem e demandem mudanças em outros serviços clientes, todo esse benefício está perdido. Dessa forma, busca-se por alternativas que não afetem e indisponibilizem o serviço àqueles que já o consomem, mesmo em casos onde a interface de comunicação é alterada.

- **API's independentes de tecnologia**: A área de desenvolvimento de software muda muito rapidamente. A cada dia novas tecnologias surgem e muitas outras se tornam obsoletas, levando consigo as aplicações que as utilizam. Embora esse aspecto seja muito bem tratado na arquitetura de micro-serviços, dado seu design evolucionário, as formas de integração também precisam garantir o total desacoplamento entre os serviços, permitindo a substituição dos mesmos sem que um impacte o outro. Para isso, utilizam-se mecanismos de comunicação independentes de tecnologia e construídos sob protocolos consagrados. Um exemplo atual é o REST [WPR10], padrão de comunicação baseado no protocolo HTTP que trafega texto em formato XML ou JSON.

- **Interfaces sem detalhes da implementação**: Consumidores não devem conhecer detalhes internos da implementação de outros serviços. Caso conhecessem, suas implementações poderiam fazer uso dessas informações e acoplar de forma indevida os sistemas. Tal acoplamento além de dificultar a clareza e responsabilidade dos serviços poderia gerar grandes impactos não esperados em caso de alterações no serviço provedor.

Apesar das premissas limitarem as escolhas, diversas maneiras de comunicação ainda são possíveis de serem aplicadas em micro-serviços, dependendo das necessidade de cada projeto ou negócio. Os principais modos, paradigmas e estilos de comunicação serão confrontados durante essa seção, clarificando todas as possibilidades e razões pela qual cada um deve ser adotado.

2.2.2.1 Base de dados compartilhada

Embora sistemas gerenciadores de bancos de dados(SGBD's) tenham sido criados com a finalidade de persistência, até hoje os bancos de dados também são utilizados de forma alternativa para a comunicação entre sistemas. Devido a sua simplicidade, esse estilo de integração é o mais adotado na indústria de software [New15] pois permite de forma rápida a escrita e o acesso a dados compartilhados.

Utilizado desde o surgimento dos SGBD's, a integração de sistemas através das bases de dados ainda é comum em sistemas monolíticos mas está em desuso na arquitetura de micro-serviços pois não garante a coesão nem baixo acoplamento entre os serviços. Esses problemas podem ser identificados facilmente através das premissas estabelecidas, uma vez que nenhuma delas é atendida por esta maneira de integração.

A primeira premissa comprometida é a da entrega sem impactos. Com o compartilhamento de dados através de tabelas é inevitável que mudanças estruturais nessas tabelas não impactem todos os serviços consumidores. Dessa forma, ou todas as consultas aos dados são alteradas em todos os serviços, ou simplesmente evita-se que tais mudanças sejam feitas, prejudicando a evolutibilidade dos serviços.

O acoplamento demasiado gerado por essa integração também afeta a independência tecnológica uma vez que não se pode mais alterar o meio onde os dados estão sendo persistidos sem que haja consequências aos consumidores. Por exemplo, se a aplicação provedora desejar migrar seus dados para outra tecnologia, como um banco NoSQL [SF12], todos outros serviços precisam migrar de tecnologia conjuntamente.

A terceira e última premissa, interfaces sem detalhes de implementação, também não é respeitada, já que todos os dados internos estão disponíveis de forma crua aos consumidores externos. Ao infringir as três premissas básicas, essa forma de integração pode ser considerada um anti-padrão para a arquitetura de micro-serviços, devendo ser evitada ao máximo nesse contexto.

2.2.2.2 Comunicação síncrona ou assíncrona

O modo de sincronia na comunicação entre serviços é uma das principais decisões arquiteturais a serem tomadas pois determina o estilo de colaboração a ser seguido entre os serviços, ou seja, como efetivamente vão se comunicar. A comunicação síncrona baseia-se no envio de uma requisição por um serviço cliente a um serviço servidor, que recebe, executa e responde ao cliente no mesmo canal de comunicação, de forma blocante. Já no caso do modo assíncrono, o cliente envia uma requisição ou simplesmente uma mensagem a outros envolvidos, que podem ou não retornar uma resposta para aquela operação em outro canal, de forma não blocante.

Mais comum e mais simples de ser implementada, a comunicação síncrona permite fácil monitoração do resultado das operações pois os eventos devem acontecer de forma sequencial e num curto espaço de tempo. Por outro lado, apesar da comunicação assíncrona apresentar maior complexidade de implementação e de controle de estado, este modo provê maior performance às aplicações, capazes de executar uma quantidade superior de operações devido ao fato de não manterem as conexões abertas com os clientes. Além disso, é apropriado para operações de longa vida cuja duração

inviabiliza o uso de conexões síncronas.

Estes dois modos de comunicação apresentados acima servem como base para dois paradigmas diferentes de colaboração entre serviços: o padrão requisição-resposta e o baseado em eventos [New15]. O padrão requisição-resposta necessita obrigatoriamente de dois serviços, sendo o primeiro o iniciador da requisição e segundo o servidor, responsável por executar a operação e retornar uma resposta. Embora seja muito semelhante ao modo síncrono de comunicação, esse paradigma também pode ser aplicado de forma assíncrona através do registro de chamadas de retorno, fazendo o servidor estabelecer uma nova conexão quando a operação estiver concluída.

O paradigma de colaboração baseado em eventos requer apenas um serviço iniciador, que envia uma mensagem em razão a algum evento ocorrido, alertando os demais serviços interessados para que tomem providências caso necessário. Neste cenário, centenas de serviços podem ser notificados sem mesmo que o serviço iniciador saiba de sua existência, garantindo um desacoplamento total entre os mesmos. Essa funcionalidade de *multicast* é geralmente provida por sistemas de mensageria como o RabbitMQ [VW12], capazes de registar tópicos de eventos e consumidores associados. Além disso, essa forma de colaboração é totalmente assíncrona, não blocante e não requer respostas por parte do serviço inicial.

2.2.2.3 Orquestração ou coreografia

As classificações dos modos de comunicação e dos paradigmas de colaboração entre os serviços, citados anteriormente, elucidaram as tecnologias de integração existentes mas não explicitaram como as regras de negócio devem ser dispostas numa arquitetura de micro-serviços. Dessa maneira, uma nova categorização complementar aos conceitos anteriores é utilizada para discriminar dois estilos de integração possíveis: a orquestração e a coreografia.

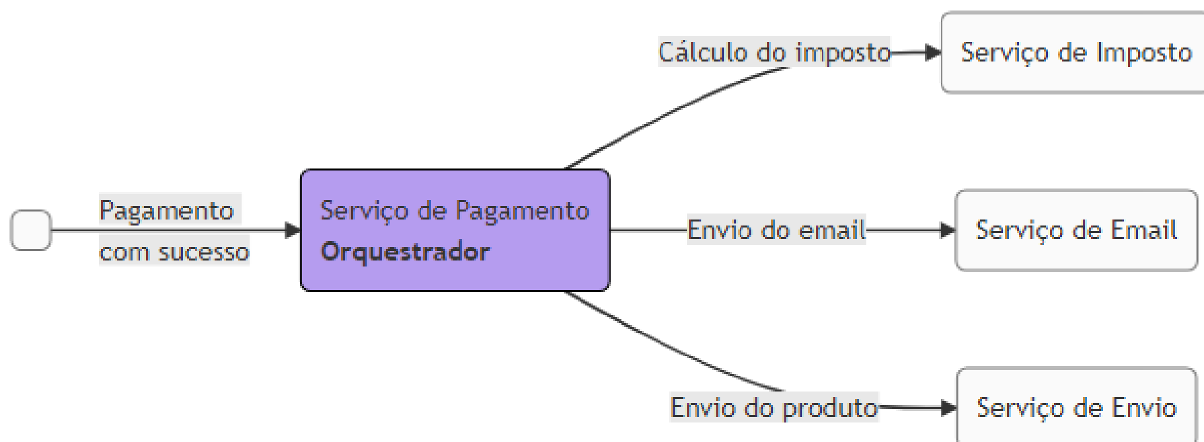

Figura 2.5: *Integração por orquestração*

A orquestração baseia-se na composição de serviços para resolução de um processo de negócio através de um controlador central, similar a um maestro de uma orquestra, capaz de executar uma séria de requisições-respostas a serviços clientes de forma sequêncial ou paralela, conforme exemplificado na figura 2.5, na qual o orquestrador 'Serviço de Pagamento' executa requisições a três diferentes serviços de acordo com a regra de negócio, para cálculo do imposto, envio de

email e do produto ao cliente. Na orquestração, essas requisições podem ser tanto síncronas como assíncronas, já que segue os mesmos princípios do paradigm requisição-resposta.

Já a coreografia, de forma oposta, não possui um coordenador central, mas sim uma lógica distribuída onde cada serviço sabe seu papel e como interagir diante de um evento específico, tomando uma ação e gerando outros eventos, assim como numa dança de balé, seguindo os mesmos princípios da colaboração baseada em eventos. Dessa forma, a comunicação é somente assíncrona através de sistemas mensageria que fazem a distribuição das mensagens ou eventos, sinônimos neste caso. A figura 2.6 representa uma implementação via coreografia das mesmas regras de negócio do exemplo anterior. Neste caso, o 'Serviço de Pagamento' não orquestra chamadas dado um pagamento, apenas envia um evento para uma fila do sistema de mensageria, na qual os outros três serviços estão inscritos e escutando cada mensagem, comunicação representada no desenho através de linhas tracejadas.

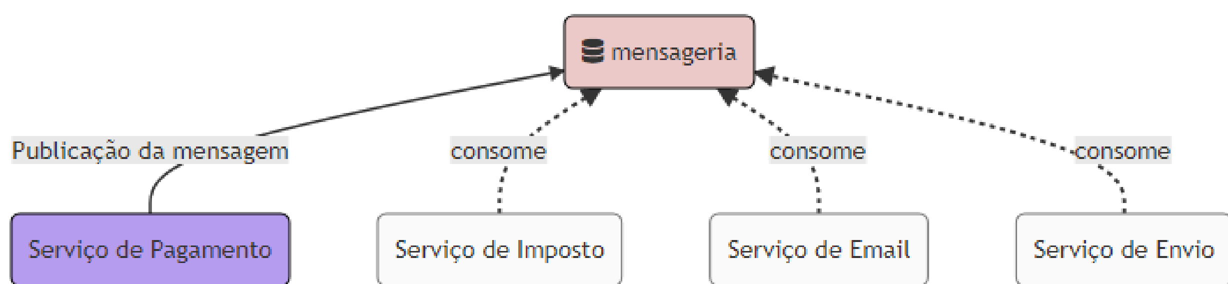

Figura 2.6: *Integração por coreografia*

Cada um desses estilos tem suas peculiaridades, vantagens e desvantagens. A orquestração oferece maior controle e centralização, o que pode resultar em serviços anêmicos integrados por um orquestrador sobrecarregado, iminente ponto único de falha. Para evitar tal resultado, a orquestração pode ser dividida entre vários orquestradores, cada um responsável por um fluxo de negócio de menor granularidade. Além disso, toda inteligência deve ser mantida nos próprios serviços, com exceção das regras de negócio de alto nível de abstração. A adoção do estilo de orquestração também garante maior visibilidade aos fluxos dos processos, facilitando o entendimento à todas áreas envolvidas. Essa vantagem fica ainda mais evidenciada com a utilização de ferramentas visuais para modelagem das regras de negócio, que auxiliam na manutenção e criação de novos processos.

O estilo de integração por coreográfia tem benefícios opostos a orquestração. A ausência de um eixo central elimina o ponto único de falha em razão da distribuição das responsabilidades, mas também dificulta na monitorização do estado das operações e na identificação dos fluxos de negócio, restringindo a visibilidade do sistema. Totalmente descentralizado, esse estilo provê maior escalabilidade e total desacoplamento, permitindo que novos serviços sejam criados e integrados ao sistema sem que haja qualquer alteração nos serviços existentes.

Assim, somente diante das características e dos requisitos funcionais e não funcionais de cada projeto é possível definir qual dos estilos de integração deve ser usado, baseado nas vantagens e limitações de cada um. No entanto, nada impede que ambos estilos sejam utilizados em conjunto, proporcionando a cada serviço a forma mais adequada de comunicação.

2.2.3 Outras estratégias de modularização

A ideia de modularização de software não é algo recente. Os primeiros artigos datam da década de 60 e já expunham preocupações com acoplamento e coesão de sistemas através de princípios como *'separation of concerns'*, *'information hiding'* e *'design for change'* [BC68][Par72], utilizados até hoje em dia.

Em 1994, David Parnas, um dos pioneiros no assunto, desenvolveu um estudo retratando o envelhecimento inevitável do software [Par94], contextualizando refatoração, modularização e substituição de sistemas com o que hoje chamamos de dívida técnica. Embora escrito mais de 20 anos atrás, o estudo ainda retrata problemas da atualidade e expõe de maneira genérica como solucioná-los.

Todo o empenho aplicado na área de engenharia de software e arquitetura de sistemas nas últimas décadas não foi suficiente para que todos os problemas fossem resolvidos. A evolução tecnológica trouxe consigo também uma série de novas necessidades e a busca incessante de melhores soluções. Dessa forma, a arquitetura de micro-serviços apresenta-se como um passo adiante nessa melhoria contínua em busca de software de alta qualidade.

A seguir algumas das estratégias de modularização que antecederam a arquitetura de micro-serviços serão apresentadas nesta seção.

2.2.3.1 Arquitetura orientada a serviços

A comparação entre micro-serviços e o padrão arquitetural orientado a serviços é muito comum. Essa arquitetura, conhecida pela sigla SOA, *Service Oriented Architecture*, foi amplamente disseminada na década anterior no combate aos desafios de comunicação entre sistemas monolíticos. Para isso, a arquitetura SOA prega reusabilidade por meio da disponibilização das funcionalidades das aplicações através de serviços definidos por interfaces e conectados por um barramento de serviços, o tal ESB, *Enterprise Service Bus* [New15].

Apesar de ter por objetivo uma maior reusabilidade e definição clara do escopo de cada aplicação, na prática, a falta de consenso em torno de definições específicas e a ambiguidade sobre os conceitos aplicados a SOA culminaram no declínio desse padrão arquitetural. A falta de valor de negócio gerado para as empresas e o acoplamento central em torno dos ESB's foram outros fatores fundamentais que levaram grandes corporações ao insucesso [FL14].

A maior parte das diferenças entre SOA e micro-serviços estão justamente nos conceitos que SOA deixou em aberto. Apesar de ambas arquiteturas utilizarem o termo serviço, seu significado é diferente em cada uma. Em micro-serviços, um serviço tem a dimensão de uma aplicação, responsável por uma única responsabilidade de negócio. Já em SOA, serviço define uma funcionalidade exposta através de uma interface, não especificando onde realmente a implementação está. É justamente essa pequena diferença que afeta a granularidade das aplicações e consequentemente todo os princípios relacionados a essas arquiteturas.

Embora a implementação de ambas seja bem divergente, a arquitetura de micro-serviços estende os aspectos fundamentais de SOA, podendo ser considerada seu sucessor natural já que a maioria das técnicas utilizadas tiveram origem na década passada durante a integração dos monolíticos e

que, principalmente, muito das novas características decorrem das aprendizagens e do erros cometidos nesse período.

2.2.3.2 Outras técnicas

A decomposição de parte das aplicações em múltiplas bibliotecas compartilhadas é a técnica mais comum de modularização e software, disponível em praticamente todas as linguagens de programação [New15]. As bibliotecas de software consistem de código, scripts e dados pré-determinados e estáticos responsáveis por funcionalidades específicas dentro de um contexto. São geralmente construídas para suprir necessidades comuns a muitos sistemas, tais como mecanismos de integração com outras aplicações e de acesso a banco de dados.

Durante o desenvolvimento de uma aplicação é comum que bibliotecas de auxílio sejam criadas, podendo ser compartilhadas entre outras aplicações desse sistema ou empresa. Entretanto, a maioria das bibliotecas utilizadas hoje pelas aplicações são externas, disponíveis através de repositórios para toda a comunidade, evitando o retrabalho à milhares de desenvolvedores.

Apesar de facilitar o reuso e a padronização de certas atividades de software, as bibliotecas têm poder limitado e não são favorecem a heterogeneidade de tecnologias, não são capazes de auxiliar na escalabilidade das aplicações e nem mesmo isolar erros e garantir resiliência ao sistema. Além disso, bibliotecas que atuam na integração entre sistemas podem favorecem o acoplamento, amarrando-a a uma tecnologia específica [New15].

Outro método de modularização de código advém da criação de módulos dentro de uma mesma aplicação. Entretanto, essa técnica ainda é restrita a algumas linguagens de programação e ao final ainda apresenta as mesmas limitações das bibliotecas compartilhadas.

Tanto essas técnicas apresentadas como a arquitetura de micro-serviços podem ser aplicadas concorrentemente e independentemente, cada uma auxiliando dentro de seu escopo específico: as bibliotecas e módulos facilitando a decomponibilidade do código-fonte das aplicações enquanto os serviços são responsáveis pela composição de todas as funcionalidades do sistema.

2.2.4 Comparação com monolíticos

Ao longo de toda seção 2.2 a expressão ‘monolíticos’ foi utilizada para definir a mais tradicional forma de se arquitetar um sistema de software: diversas funcionalidades em uma única aplicação. Nessas ocasiões, grande parte dos problemas desse tipo de arquitetura foram expostos em contrapartida às vantagens providas pelos micro-serviços. Esta seção apresenta uma comparação sob outra perspectiva, analisando as perdas e dificuldades na implementação dos micro-serviços em relação aos monolíticos.

Na grande maioria das situações, sistemas de micro-serviços tendem a ter melhor evolutibilidade, manutenibilidade e resiliência que monolíticos, uma vez que foram projetados para esse fim. Dificilmente alguma equipe de desenvolvimento, ao deparar-se com uma mesma solução implementada em ambas arquiteturas, optaria pelo padrão monolítico. Entretanto, o caminho até a solução final é árdua. A implementação de um sistema em micro-serviços escalável, resiliente, desacoplado, coeso e com integração contínua exige maior complexidade e maior custo de desenvolvimento [FL14].

Nos micro-serviços, a substituição das chamadas de métodos locais por chamadas à aplicações remotas traz à tona toda a complexidade de se gerir um sistema distribuído. Além do risco de falha relacionado à rede de comunicação, questões transacionais e de inconsistência entre os serviços também devem ser tratadas em cada aplicação através de mecanismos de retentativa e lógica para prover idempotência. Por fim, gerir diversas instâncias de múltiplos serviços requer uma maior inteligência operacional, preferencialmente automatizada.

A arquitetura de micro-serviços também pode não ser a mais adequada para certas necessidades. Alguns exemplos:

- Ambientes limitados e escassos não são adequados a esse tipo de arquitetura, devido ao maior processamento computacional e tráfego de rede gerado na comunicação entre os serviços.
- Aplicações que exigem respostas em tempo muito baixo pode ser impactadas pela latência maior na rede.

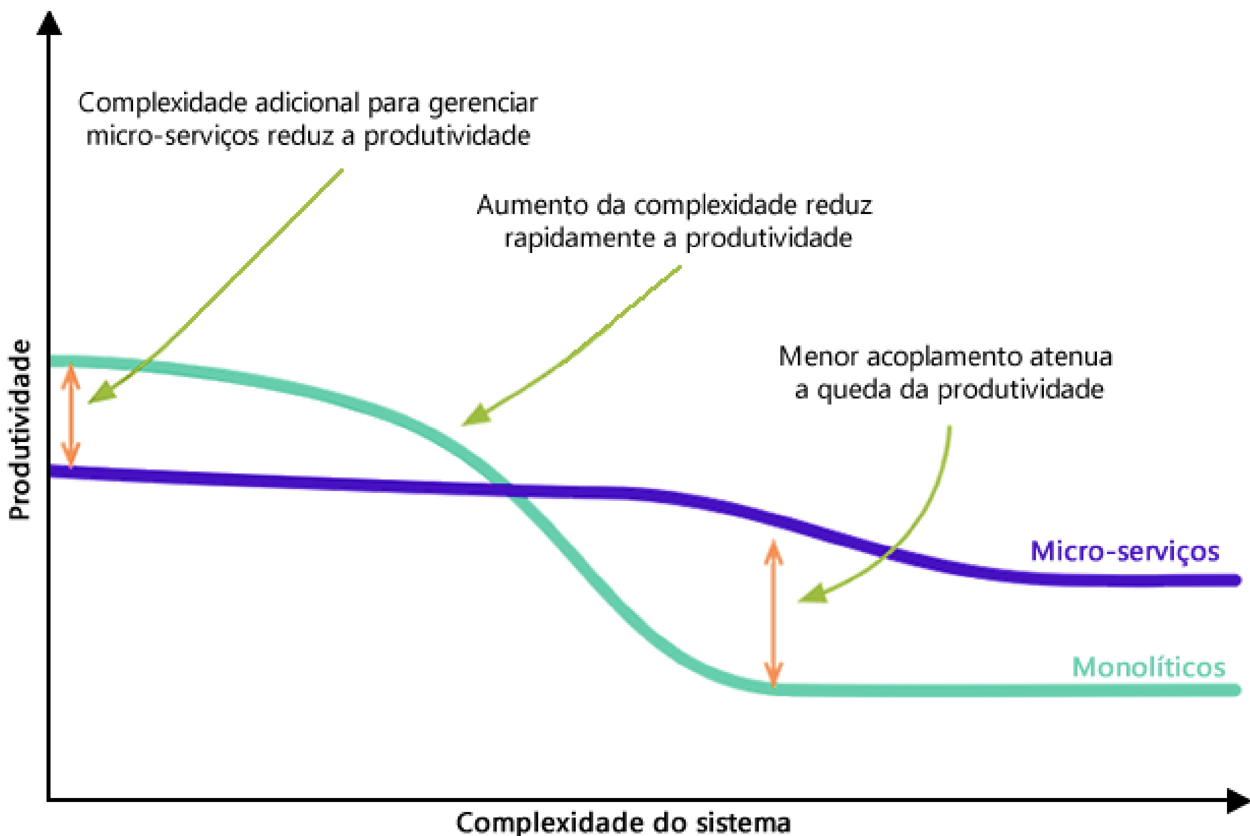

Figura 2.7: *Estimativa da produtividade em razão da complexidade em monolíticos e micro-serviços*

Muitas vezes o termo monolítico é usado de forma pejorativa, criando-se a impressão de que sistemas monolíticos são sempre legados, o que é uma falácia. Esse tipo de padrão requer menos esforço de implementação inicial, o que oferece uma vantagem competitiva no curto-prazo. Além disso, é adequado para sistemas de baixa complexidade, onde o esforço adicional para a manutenção de micro-serviços traz mais perdas que ganhos [FL14]. Uma estimativa da produtividade das equipes de desenvolvimento de acordo com a complexidade do sistema em cada uma dessas arquiteturas

é apresentada na 2.7, corroborando que arquiteturas de micro-serviços tendem a apresentar maior produtividade conforme maior complexidade do sistema, em relação aos monolíticos.

Uma das discussões mais recentes sobre esses dois estilos é de qual se adotar no início de um novo projeto. A análise acima pressupõe que sistemas simples devem adotar o padrão monolítico enquanto sistemas mais complexos devem aderir aos micro-serviços. Entretanto, não é essa a opinião de um dos mais pesquisadores dessa área, Martin Fowler. Em seu estudo [Fow15], Martin Fowler defende que mesmo para sistemas que pareçam demasiadamente complexos, uma implementação inicial num monolítico é ideal por ser mais simples e rápida, ajudar a validar o sistema e auxiliar na definição das interfaces de cada funcionalidade. Assim, gradualmente as funcionalidades podem ser migradas para micro-serviços conforme necessidade, através da estratégia de *Strangler Application*, detalhada a seguir na seção 2.3.1.1.

2.2.5 Anti-padrões da arquitetura de micro-serviços

A implementação da arquitetura de micro-serviços em grandes empresas por todo o mundo trouxe diversas experiências de implementações de sucesso e também de erros cometidos durante o caminho. Muitas dessas experiências negativas foram compartilhadas através de publicações online [PER][HIJ][ISM][MIC][ALA] ou acadêmicas [TL18], consolidando uma base inicial de anti-padrões que devem ser evitados em arquiteturas de micro-serviços.

Para facilitar o entendimento desses dez *bad smells* apresentados, eles serão agrupados em quatro diferentes grupos: comunicação, compartilhamento de dados, decomposição e outros.

Anti-Padrões de Comunicação

A expressão *"dependency hell"* abrange vários possíveis problemas em relação a dependência entre os micro-serviços, desde o suporte a retrocompatibilidade de versões de API's [PER][ISM] até a organização das chamadas entre os mesmos, ou seja, o grafo de dependência que determina o grau de normalização da arquitetura [HIJ]. Essas questões são fundamentais para que a independência entre os micro-serviços esteja garantida ao invés de dificultar tanto o desenvolvimento como a entrega de novas versões entre duas aplicações adjacentes no grafo. Por serem assuntos amplos e complexos, optou-se por classificar os anti-padrões numa menor granularidade em relação a expressão *"dependency hell"*, listados abaixo:

1. **Não versionamento de API's**: O não versionamento ou a falta de compatibilidade de uma API entre duas versões do software, caso haja alguma alteração no formato dos dados, pode ocasionar erros às aplicações clientes ou necessitar que ambas sejam entregues ao mesmo tempo. Aconselha-se que exista um versionamento e que as entregas possam prover retrocompatibilidade [TL18][PER][ISM][ALA].

2. **Dependências cíclicas**: Não devem existir uma cadeia de chamadas entre micro-serviços que seja cíclica. Dependências cíclicas podem ser difíceis de manter e resultar em ciclos infinitos de chamadas [TL18][ISM]. Indicam que talvez a modelagem do domínio não esteja correta.

3. **Dependências transitivas**: Essas dependências são um sub-domínio das dependências cíclicas e também devem ser evitadas [HIJ], principalmente se forem chamadas síncronas em longas cadeias de transitividade, comprometendo o tempo de resposta dos serviços [MIC].

4. **Ausência de *API Gateway***: *API Gateways* são sistemas capazes de fazer a composição de micro-serviços, orquestrando as chamadas entre os micro-serviços para um propósito de negócio em comum, conforme apresentado na seção 2.2.2. Dessa forma, toda chamada a um recurso do sistema deve ser encapsulada por um *API Gateway* que pode consultar um ou mais serviços para prover os dados necessários. Assim, evita-se que os micro-serviços integrem-se diretamente e evoluam para uma arquitetura spaghetti descentralizada [PER], sem nenhuma governança e de difícil manutenção [TL18][HIJ][ALA].

Anti-Padrões de Compartilhamento de dados

Essa classificação refere-se aos anti-padrões relacionados ao compartilhamento de dados entre micro-serviços, outro grande desafio quando o domínio está dividido em diversas aplicações. São dois os *bad smells* catalogados:

5. **Persistência compartilhada**: Micro-serviços não devem compartilhar o mesmo banco de dados, muito menos acessarem os dados das mesmas tabelas. Como já apresentado em 2.2.2, integração por banco de dados é intrusiva e comprometem as premissas de integração. Aconselha-se utilizar banco de dados diferentes para cada micro-serviço ou no mínimo um esquema separado para cada um, quando sob o mesmo banco de dados, nunca acessando as mesmas tabelas [TL18].

6. **Intimidade inapropriada**: Quando um micro-serviço necessita de dados privados de outros micro-serviços e os acessa de forma não adequada, conectando diretamente ao banco de dados ou por API's temporárias. Talvez os dados tenham sido modelados de forma errada e agrupar os dois serviços seja uma solução [TL18].

Anti-Padrões de Decomposição

A decomposição de um sistema em micro-serviços corresponde a como os micro-serviços estão divididos, quais os limites e responsabilidades de cada um, tema que será aprofundado no capítulo 2.3.1.1. Segundo Chris Richardson, uma decomposição de forma errada do sistema é grave pois se dá no nível de arquitetura ao invés de código-fonte, sendo o pior cenário um monolítico distruibuído [RIC]. A literatura apresenta alguns anti-padrões relacionados:

7. **Corte errado**: Também classificado como "Arquitetura de serviços em camadas", ocorre quando um sistema é decomposto em micro-serviços baseado nas camadas da aplicação, tais como apresentação, negócio e de dados, ao invés da decomposição por funcionalidades de negócio ou contextos limitados [TL18][ALA]. Um exemplo comum é a criação de serviços que servem apenas de camada para expor o banco de dados através de serviços.

8. **Micro-serviço ganancioso**: Dada a tendência de criação de micro-serviços, times de desenvolvimento podem exagerar e criar serviços para funcionalidades muito pequenas, sem que haja necessidade [TL18]. Uma explosão no número de micro-serviços pode apenas afetar a manutenibilidade do sistema.

Outros

Alguns outros *bad smells* são mais genéricos e não se encaixam em nenhuma das três categorias acima. São eles:

9. **Excesso de tecnologias**: Embora esse seja um dos benefícios da arquitetura de microserviços, abusar do uso de diferentes linguagens, protocolos e frameworks pode comprometer o desenvolvimento futuro da empresa com a mudança de profissionais ou mesmo prejudicar a produtividade dado a falta de padrão. Não deve-se adotar de forma indiscriminada sempre o último *hype* de tecnologia [TL18].

10. **Bibliotecas compartilhadas**: O compartilhamento de bibliotecas privadas entre os microserviços pode reduzir a dependência entre eles e também entre os times de desenvolvimento. A extração da biblioteca para um serviço separado é aconselhada [TL18].

2.3 Modernização de sistemas

Modernização de software é um termo genérico utilizado para representar a transição de sistemas obsoletos para novas aplicações modernas e aptas a atender as necessidades atuais. Outros termos comumente utilizados para designar modernização são substituição, migração, evolução [Yam17] ou até mesmo reengenharia e refatoração arquitetural [Zim15]. O processo de modernização geralmente é aplicado quando as técnicas básicas de manutenção e refatoração de código não são mais suficientes para garantir manutenibilidade e evolutibilidade ao sistema [BSJH14][Zim15]. Os sinais abaixo são alguns dos indícios da necessidade de modernização do sistema [Zay17]:

- Longos ciclos de desenvolvimento.
- Não há separação dos módulos e há um alto acoplamento no código.
- Tecnologias utilizadas ultrapassadas.
- Baixa escalabilidade.
- Dificuldade de entendimento do sistema por parte de novos desenvolvedores.

Na maioria das referências encontradas o termo modernização acompanha o adjetivo legado, outro termo comumente utilizado no mundo de software [BLWG99][BSJH14][Yam17]. Sistemas legados são geralmente grandes e antigos sistemas de software que, embora obsoletos, até hoje desenvolvem um papel crucial nas operações e negócios das organizações [Ben95][BSJH14]. Geralmente são frágeis, lentos e de díficil extensão, resistentes a modificações e evolução [BLWG99][BSJH14][VAC]. Se um sistema é pequeno demais ou não apresenta grandes custos de manutenção, ele não é considerado um legado [BLWG99].

A atual velocidade da evolução tecnológica e das mudanças de requisitos de negócio faz dos sistemas legados os grandes vilões quando o assunto é agilidade. Estes sistemas comprometem o *time-to-market* das empresas e impactam na produtividade das equipes de desenvolvimento [BSJH14]. Embora estime-se que hoje ainda 200 bilhões de linhas de códigos legados sejam utilizados em todo mundo [BSJH14], uma forte tendência de modernização de software vêm se propagando com o advento de novas tecnologias e arquiteturas. Segundo [BBB], 88% das corporações mundiais pretendem modernizar ao menos 25% de seus sistemas nos próximos 18 meses, sendo apenas 2% o número de empresas que não realizarão nenhum tipo de modernização, o que confirma tal tendência.

2.3.1 Estratégias de modernização

Software legado e sua modernização é um problema antigo que vem sendo estudado há algumas décadas. Já em 1980, Lehman [Leh80] citava a constante necessidade de mudanças nos sistemas para que continuassem a servir seu propósito. Além disso, argumentava que tais alterações ao longo do tempo inevitavelmente aumentariam a complexidade e degradariam a qualidade do sistema até que sua substituição fosse necessária.

Umas das estratégias mais básicas de modernização é a citada por Lehman, a substituição completa do sistema. Esta estratégia também conhecida por *Big-Bang*, ou *Big-Bang rewrite*, consiste na reconstrução completa de um novo sistema capaz de substituir de uma só vez todas as funcionalidades do sistema legado, utilizando arquitetura, ferramentas e hardware modernos [BLWG99]. Embora tenha sido muito adotada no passado, está técnica já é considerada ingênua e de alto risco desde a década de 90 [BLW+97], uma vez que o novo sistema pode não atender da mesma forma todos os requisitos do sistema antigo e apresentar novas dificuldades de manutenção [ACD10][Eva13][VAC][HAM]. Além disso, o alto custo e prazo associados a um redesenvolvimento dessa proporção podem ser ainda mais impactados em caso de mudanças de requisitos, exigindo a alteração em ambos os softwares. Todo este contexto gera pressão na entrega do novo sistema, que acaba sendo construído de maneira inferior à desejada. Novos defeitos em produção ou mesmo necessidade de refatoração são outros problemas comuns [Fow04].

Entretanto, outras estratégias além da substituição completa foram sendo abordadas ao longo do tempo. Já em 1994, Keith Bennett propunha uma estratégia menos radical, em etapas, conforme sugere o trecho extraído de [Ben95]:

“ Another promising approach is to “freeze” and encapsulate the legacy system as a component in a new implementation. The functions provided by the legacy system can then progressively be taken over by the new software until the legacy software becomes redundant. ”

Ainda na década de 90, novos trabalhos surgiram para contextualizar as estratégias de modernização na época. Em [BLWG99], além da estratégia de *Big-Bang*, apresentada como “redesenvolvimento de software”, novas duas outras abordagens foram citadas: “empacotamento de software” e “migração de software”. O empacotamento é uma solução de curto prazo e baixo risco que consiste no encapsulamento da complexidade do sistema legado em novas aplicações, provendo interfaces mais amigáveis para os sistemas clientes sem que haja necessidade de alterações no legado. Já a migração, dentre as três abordagens, é a estratégia intermediária em relação a riscos e custos, que prevê a coexistência entre o novo sistema criado e o legado, migrado em diversas etapas.

As referências mais atuais seguem basicamente as mesmas classificações definidas no passado. Em [ACD10], diversas técnicas são citadas para evolução do legado em relação a três tipos de abordagens: empacotamento, migração e substituição, cujo significado é o mesmo de redesenvolvimento ou *Big-Bang*. Já em [Zim15], a noção de modernização é trazida para o dia dos desenvolvedores através do termo refatoração arquitetural, composta por técnicas de separação de componentes, migração de responsabilidades entre componentes ou até mesmo camadas da aplicação.

Embora a literatura seja difusa e apresente diversos estudos com propostas similares de modernização mas com diferentes classificações ou nomes, um termo que vêm ganhando cada vez mais notoriedade dentro da comunidade de desenvolvimento de software é o *Strangler Application*, ou estrangulamento da aplicação, que será contextualizado e detalhado a seguir.

2.3.1.1 *Strangler Application*

O expressão *Strangler Application* foi cunhada por Martin Fowler em 2004, antes mesmo do surgimento dos micro-serviços, como uma forma gradual de substituição de sistemas complexos [Fow04]. Essa metáfora faz analogia a alguns tipos de plantas tropicais e até mesmo de uvas que, em meio a matas densas, germinam no topo das árvores e crescem suas raízes em direção ao solo, ao redor da árvore hospedeira, estrangulando a mesma.

A estratégia de *Strangler Application* consiste na extração faseada de módulos de uma aplicação monolítica, geralmente legada, construindo novas aplicações ao seu redor. A Figura 2.8 apresenta de forma simplificada este processo iterativo de modernização, fase por fase, até eventualmente o fim do sistema legado. Do ponto de vista de modernização, esta estratégia encaixa-se na categoria de migração de software, abordagem incremental de baixo risco e menor prazo quando comparado a técnicas de substituição de software como o *Big-Bang* [BLWG99] e que favorece naturalmente a adoção dos micro-serviços [Ric16].

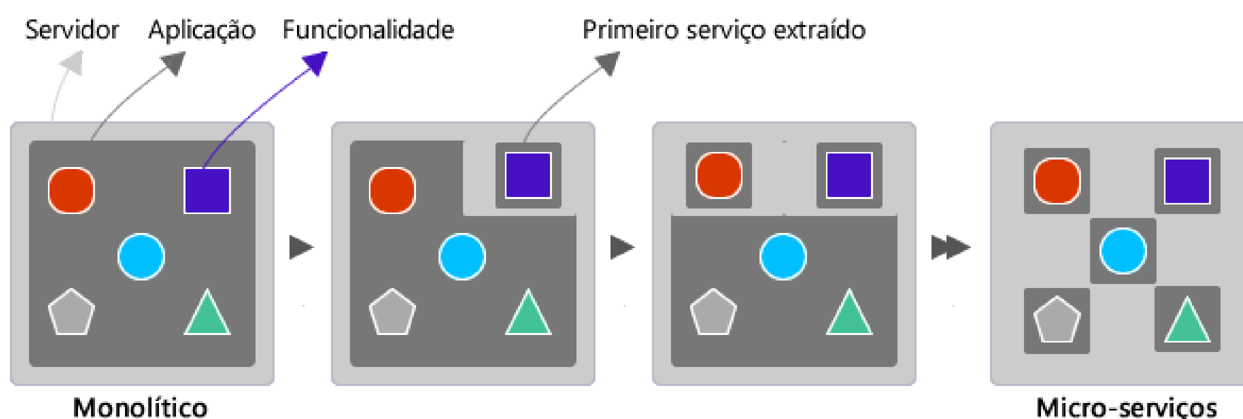

Figura 2.8: *Processo faseado de Strangler Application*

O conceito que se tem hoje de *Strangler Application* ainda não está totalmente definido. Embora esta estratégia de migração modular e faseada venha sendo citada desde antes de Martin Fowler, o termo cunhado por ele ganhou popularidade e se fundiu com outras técnicas similares como *bubble context* [Eva13], criada por Eric Evans. Nos últimos anos, novos estudos vêm sendo publicados online geralmente contextualizando os padrões e abordagens adotados pela indústria [Ric16][Bro][VAC] sem que haja referências bibliográficas acadêmicas de artigos do tema. Dessa forma, não há uma definição exata do processo como um todo e mesmo a escassa literatura existente pode apresentar opiniões divergentes.

Técnicas de implementação

Uma vez identificado que o sistema legado necessite modernização, o primeiro passo é evitar que implementações de novas funcionalidades sejam realizadas no seu código-fonte, estancando o crescimento de sua complexidade. Em seguida, inicia-se o processo faseado de extração de módulos, partes de código que possuem um significado ou valor de negócio em comum, para novos sistemas autônomos e independentes [Ric16]. Entretanto, a escolha de quais módulos devem ser prioritariamente migrados é uma decisão que depende de cada sistema e cada equipe envolvida no processo. A literatura apresenta algumas boas práticas para auxiliar nesta priorização:

- Aconselha-se que as iterações iniciais sejam simples, escolhendo-se módulos pequenos de fácil extração [Ric16][Bro] para que o time ganhe experiência com a estratégia e aprofunde seu conhecimento do domínio legado.

- Para maior produtividade, devem ser priorizados módulos que estão em constante mudança. Tais módulos, uma vez migrados, podem ser entregues e atualizados de forma muito mais rápida quando desacoplados do monolítico [Ric16].

- Para maior desempenho, os módulos que exigem recursos diferentes do resto do sistema ganham prioridade, uma vez que o novo micro-serviço pode ser construído numa linguagem mais adequada e através de técnicas mais propícias para aquela funcionalidade [Ric16].

Cada etapa do processo de modernização consiste numa sequência lógica de passos que resultará na migração de um módulo do legado para um novo serviço. Em [Bro], os passos são divididos em transformar, coexistir e eliminar, que consistem na criação do novo serviço, migração da funcionalidade e remoção do código legado, respectivamente. Já em [Ric16], Chris Richardson faz um maior aprofundamento nos detalhes de extração das funcionalidades do legado, definindo a sequência de passos apresentadas na figura 2.9 e descritas abaixo:

1. Definir o módulo que será extraído. No caso, módulo C.

2. Definir uma API, um contrato de comunicação entre o módulo a ser extraído e o restante do monolítico.

3. Extrair o módulo para um serviço independente, que pode ser entregável de forma totalmente desacoplada, removendo o código legado do monolítico.

A Figura 2.9, além de apresentar o passo-a-passo de uma etapa de extração, adiciona novos elementos na arquitetura. Um destes elementos é a camada anti-corrupção, mais conhecida na literatura por *anti-corruption layer(ACL)* ou *glue-code*, que exerce o papel de um fachada para leitura e escrita de dados do monolítico, capaz de abstrair a complexidade do domínio legado e realizar a tradução entre esses dois mundos. Desta forma, o novo serviço pode ser criado de maneira totalmente desacoplada, sem os ruídos proveniente do domínio do sistema legado. A camada anti-corrupção pode ser construída dentro do novo serviço, do monolítico ou em ambos. Um código apartado e descartável, que pode ser removido assim que a comunicação com o antigo sistema não for mais necessária. Novas integrações com o novo sistema não devem ter conhecimento sobre a camada anti-corrupção.

Embora o conceito de camada anti-corrupção tenha sido criado por Eric Evans em 2003 no seu livro *Domain-Driven Design(DDD)* [Eva03], somente em 2013 passou a ser utilizado no contexto de modernização de sistemas [Eva13]. Nesta função, a camada de anti-corrupção tem sido utilizada de maneiras diferentes [Ric16][Eva13]:

- *ACL-backed repository*: nesta implementação, a camada de abstração lê e escreve os dados através de API's do sistema legado ou até mesmo diretamente do banco de dados legado, encapsulando toda a complexidade para dentro da ACL e simplificando o novo micro-serviço.

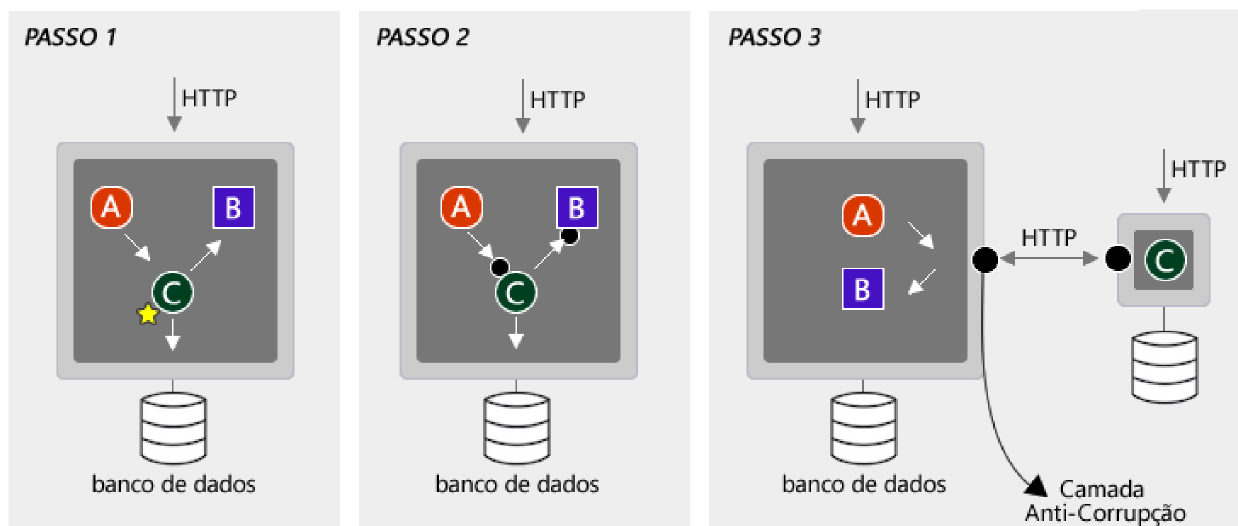

Figura 2.9: *Passos de uma etapa de estrangulamento*

- *ACL synchronized*: o novo micro-serviço consulta diretamente a sua base de dados que é periodicamente sincronizada com a base de dados do legado. *Batches* periódicos, ou mesmo sistemas de mensageria, podem ser utilizados dependendo da necessidade de sincronismo.

Além desses pontos principais, a literatura fornece também outras boas práticas identificadas ao longo dos anos pela comunidade de software em relação ao processo como um todo:

- Priorize módulos que não dependerão de dados do monolítico, minimizando as alterações necessárias no legado [Deh18].

- O sistema legado não deve conhecer o novo micro-serviço, ou seja, o roteamento da comunicação para o novo micro-serviço deve ser externo ao legado [VAC]. Entretanto, outras referências tratam essa abordagem como normal e inclusive indicam que a dependência inversa é um problema maior [Deh18].

- Além da migração, adicionar novos valores de negócio às entregas pode melhorar a percepção do processo de estrangulamento em todas as áreas da companhia [HAM].

- A utilização de testes funcionais e de integração é vital para garantir a segurança do que está sendo desenvolvido e a assertividade quanto as regras de negócio migradas [HAM].

- Tecnologias ágeis facilitam a adoção da estratégia faseada de *Strangler Application*, que não seria possível através de processos longos como o *waterfall* [BBB].

- Utilizar a técnica de *feature-toggle* pode ser muito importante para evitar problemas de integração e facilitar o *rollback*, isto é, o retorno da versão anterior às alterações, caso seja encontrado algum problema durante o processo de migração [RQRA16]. Na prática a técnica consiste basicamente numa validação condicional, um *if/else* para determinar em tempo de execução o fluxo que será seguido, o novo ou o anterior.

Técnicas de decomposição

A teoria apresentada até o momento foi simplista pois abstraiu as dificuldades relacionadas à identificação dos módulos, funcionalidades e fluxos existentes nos sistemas legados. Na prática, a alta complexidade, alto acoplamento, falta de documentação e de conhecimento dos sistema legados dificultam a decomposição do código do monolítico em pedaços menores e coesos durante cada etapa do estrangulamento.

Embora a literatura sobre o tema *Strangler Application* seja escassa, as pesquisas acadêmicas relacionadas à identificação e divisão de fluxos e regras de negócio em sistemas complexos é ampla. Dos estudos analisados, boa parte propõem técnicas baseadas na construção de grafos de dependência a partir de análise estática de código ou de documentações padronizadas, capazes de revelar os fluxos entre as chamadas de métodos e também os pontos de menor acoplamento da aplicação [LTV15][MH08]. Outros propõem técnicas de análise lexicográfica a fim de identificar os módulos através da repetição de palavras [MCL17], ou ainda abordagens com heurísticas para identificação de regras de negócio [PdP16]. Entretanto, os estudos ainda são incipientes e as ferramentas desenvolvidas ainda muito restritivas, ou com limitações que dificultam a utilização em aplicações da indústria.

Na prática, as literaturas relacionadas a modernização recomendam as mesma técnicas utilizadas na construção de novas arquiteturas de micro-serviço, isto é, enfatizam primeiramente a modelagem do domínio da aplicação dividindo o modelo de negócio em sub-domínios e suas respectivas entidades que mapeiam o mundo real [Ric16][New15][Zay17][Bro]. A essa divisão coesa do domínio através de funcionalidades e regras de negócio é dado o nome de *bounded-context*, ou contexto-limitado, termo notório também cunhado por Eric Evans em seu livro *Domain-Driven Design(DDD)* [Eva03]. Um contexto limitado representa um sub-domínio claro do negócio que tem sua própria linguagem ubíqua [Eva03] e algumas entidades do sistema. Estas entidades podem coexistir em vários contextos limitados, tendo um comportamento único para atender à necessidade específica de cada contexto. A Figura 2.10 exemplifica esse conceito através da modelagem de um domínio em dois contextos limitados: vendas e suporte. No exemplo, tanto a entidade “cliente” como “produto” existem em ambos contextos, com implementações diferentes em cada uma, especializadas de acordo com a função que exercem em cada contexto.

Além da decomposição segundo os princípios de DDD e contextos limitados, outras abordagens também são citadas na literatura:

- ***Business capabilities***: As *business capabilities*, ou capacidades de negócio, definem quais são as funcionalidades de negócio mais importantes com foco na entrega de valor, dentro de um domínio particular [Deh18]. Este conceito não é tão simples de ser definido nos sistemas pois as funcionalidades de negócio podem ser interpretadas de diferentes maneiras em diferentes negócios e granularidades de especificação.
- **Verbos/Casos de uso**: De acordo os casos de uso da aplicação que represetam os verbos, as ações realizadas [RIC].
- **Nomes/Recursos**: De acordo com a modelagem dos recursos existentes no sistema, uma entidade ou grupos de entidades do domínio [RIC].

Como todas essas abordagens necessitam de um amplo conhecimento da organização e do negócio em si, não existem ferramentas para automatizar tais decisão. Além disso, muitas vezes o sistema

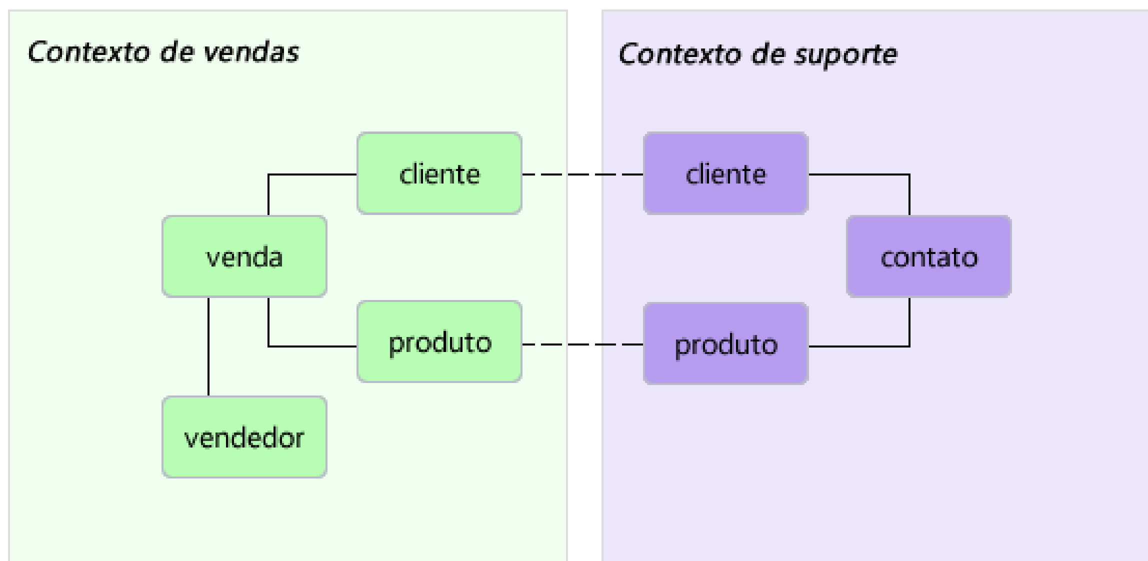

Figura 2.10: *Modelagem de domínio através de contextos limitados*

legado possui um domínio ultrapassado que ao ser migrado é também atualizado para atender a novas necessidades. Ao fim da implementação, espera-se que o novo serviço extraído atenda aos princípios de micro-serviços apresentados na seção 2.2.1.

2.3.2 Desafios na modernização

De forma similar aos seres humanos, software envelhece, torna-se obsoleto e passa a comprometer a produtividade, confiabilidade e competitividade das organizações [Par94]. Lidar com estes sistemas legados não é uma atividade simples e requer um alto custo de manutenção [Ben95]. Entretanto, a outra opção existente, sua modernização, também apresenta diversos desafios, riscos e um alto custo [BSJH14][Yam17]. Todas essas variáveis devem ser analisadas de acordo as necessidades futuras do sistema e objetivos da organização para definir se, em termos econômicos, a modernização é a melhor escolha [BLWG99].

Das diversas referências bibliográficas que citam os desafios na área de modernização de sistemas, duas pesquisas se destacam pelo seu caráter empírico. Em [BSJH14], uma grande pesquisa qualitativa composta por entrevistas com 26 profissionais da indústria traz um catálogo dos maiores desafios vivenciados na prática, listados abaixo:

- Restrições de tempo e custo: um projeto de modernização tende a durar muito tempo e os recursos durante este período são sempre escassos, desde profissionais que têm conhecimento sobre o domínio até documentação atualizada. Longos ciclos de desenvolvimento.

- Migração dos dados: os dados dos antigos e novos sistemas nunca são compatíveis ou devem coexistir ao longo do tempo, o que torna complexa a implementação.

- Complexa arquitetura do sistema: a arquitetura do sistema legado nunca é simples, geralmente monolítica e com alto acoplamento. A integração com outros sistemas também ocorre por meios obsoletos e intrusivos, como banco de dados por exemplo. Ao fim, testar todas as funcionalidades do legado no novo sistema pode requerer muito esforço.

- Falta de conhecimento: o desconhecimento e a falta de documentação do comportamento do sistema legado só aumenta os riscos na migração para um novo sistema. Além disso, a modernização em si exige conhecimento de pessoas capazes de criar um plano de migração desde a parte de infraestrutura e hardware até as funcionalidade de negócio.

- Dificuldade de extração e priorização das regras de negócio: resultado da falta de conhecimento, a identificação das regras de negócio é outro grande desafio, especialmente em monolíticos de grande acoplamento e com diversos domínios. Há um grande risco em realizar a migração de funcionalidades se não se sabe ao certo todos os detalhes de implementação no legado.

- Resistência da organização: existe uma resistência natural por parte do desconhecido e isto se aplica na adoção de novas tecnologias. Além disso, muitos profissionais temem que após a modernização seu conhecimento não seja mais necessário.

- Fatores não técnicos: esta categoria engloba assuntos não técnicos relacionados à questão financeira da modernização. Não é trivial prever o retorno financeiro sobre o investimento nem comunicar as consequências e as necessidades da modernização, o que pode dificultar que o projeto de modernização seja financiado e saia do papel.

Já no estudo [Yam17], uma grande análise foi feita a fim de identificar padrões recorrentes no processo de modernização tendo como base a experiência vivenciada do autor nos diversos projetos que atuou como consultor em mais de dez anos. Os maiores desafios foram divididos em quatro principais áreas:

- Retrabalho da solução: falta de comunicação ou falta de conhecimento e visão do todo fazem com que seja constante a necessidade de refatoração e correção da solução recém implementada, impactando no andamento do projeto.

- Guiado por defeitos: é comum que diversos defeitos e bugs surjam em razão da modernização dadas as faltas de conhecimento e de uma profunda análise dos impactos das mudanças que podem gerar alguns efeitos colaterais.

- Tarefas consumidoras de tempo: diversas tarefas, inclusive manuais ou processuais, tomam muito tempo do desenvolvimento durante a modernização de software. Por exemplo, a coleta de dados e informações em diferentes áreas da empresa que podem ser clientes ou dependências do sistema em questão.

- Planejamento: um planejamento prévio com um maior levantamento de informações a respeito do sistema legado pode ser extremamente importante para uma melhor tomada de decisão sobre a alocação de recursos em relação à modernização. Talvez até mesmo uma nova etapa de planejamento para coleta de informações e discussão sobre os próximos passos seja necessária dentro de cada iteração da evolução.

Embora categorizados de maneiras diferentes, a essência dos desafios de modernização mostram-se bem similares nas referências analisadas e compreendem desde questões técnicas como arquitetura até questões culturais ou financeiras que podem afetar todo o processo [BBB].

Capítulo 3

Estudo de caso e metodologia aplicada

Neste capítulo será apresentado o estudo de caso e a metodologia de pesquisa aplicada durante o desenvolvimento deste projeto, detalhando desde o sistema de software sob estudo até o processo sistemático de coleta e análise de dados.

3.1 Estudo de caso: um sistema legado

O estudo de caso tem como objeto de investigação a modernização de um sistema monolítico legado, o "account-collector", de vital importância para a frente de cobrança de pagamentos do UOL, o Universo Online, maior empresa brasileira de conteúdo, tecnologia, serviços e meios de pagamentos digitais. Como característico em sistemas legados, apesar de exercer um papel crucial para a companhia, o account-collector apresenta alta complexidade e alto custo de manutenção, além de ser inapto à mudanças e dificultar a implementação e evolução de requisitos funcionais e não funcionais.

Problemas em produção, incidentes e outras dificuldades diárias relacionadas à manutenção do account-collector traziam sempre a tona a inevitabilidade de substituição desse legado. No entanto, a necessidade de entregas de negócio sempre tiveram priorização e grande parte das vezes as novas implementações eram realizadas no próprio account-collector, que continuava a expandir-se. Este padrão persistiu até meados de 2015, quando um esforço conjunto entre desenvolvedores e gerência estabeleceu algumas normas e ações para que o antigo anseio de eliminar a aplicação legada fosse atingido. Além de passar a evitar que novas implementações de negócio fossem realizadas no monolítico, parte das funcionalidades do account-collector começaram a ser reescritas e migradas para outros sistemas menores, coesos e autônomos, os micro-serviços. Não houve um planejamento específico de como seria essa evolução, que não tinha um prazo determinado nem o foco de uma equipe específica para atender a migração. O processo passou a ocorrer juntamente com as demandas de negócio, buscando-se sempre construir novos micro-serviços capazes de atender as novas necessidades e também extrair parte das funcionalidades implementadas no legado, de acordo com o domínio alterado.

Com duração de aproximadamente três anos, desde julho de 2015 até agosto de 2018, o estudo de caso acompanhou além das mudanças relacionadas ao processo de modernização do legado, toda a transformação ocorrida na estrutura e cultura de desenvolvimento da área na qual o sistema está inserido, conhecida como plataforma corporativa. Esta area é responsável por suportar todo o ciclo

de vida dos clientes UOL, provendo informações, ferramentas e serviços a todas as outras áreas da companhia.

Das transformações ocorridas na plataforma corporativa, muitas das quais serão aprofundadas durante a avaliação do estudo de caso no capítulo 4, várias delas tiveram efeitos sobre a frente de cobrança, subárea responsável pelas equipes que atuam na manutenção e evolução do account-collector e de outros sistemas de pagamentos. No inicio da pesquisa, por exemplo, em torno de cinco equipes de desenvolvimento com cerca de quatro profissionais cada atuavam de maneira indiscriminada no código-fonte da aplicação legada, de acordo com as necessidades de negócio. Entretanto, a partir de meados de 2017, buscando-se uma maior organização e governança das aplicações, a frente de cobrança foi repartida em domínios específicos e os sistemas de software divididos entre eles. Deste momento em diante, embora outros times também efetuassem alterações pontuais no código legado devido ao seu extenso domínio, toda governança do sistema concentrou-se em apenas uma equipe.

As seções a seguir apresentaram detalhadamente todas as características do account-collector, sistema de software base do estudo de caso.

3.1.1 Histórico

Analisando os comentários presentes no código do account-collector, a informação de data mais antiga encontrada refere-se ao ano 2000, ou seja, o sistema tem ao menos 18 anos de idade. Criado para ser o responsável pelas cobrança de pagamentos online, a aplicação foi ao longo dos anos acumulando diversas novas regras e responsabilidades de negócio, crescendo descontroladamente diante das constantes mudanças de requisitos e satisfazendo cada vez menos ao 'princípio de responsabilidade única' [New15]. O processo de desenvolvimento de novas funcionalidades no sistema legado continuou até meados de 2015, data na qual novas entregas foram congeladas devido ao alto custo de desenvolvimento e risco de inserção de novos bugs.

Datada antes mesmo do manifesto ágil [BBB+01], a aplicação passou por uma série de culturas de processo de desenvolvimento, paradigmas e frameworks diferentes. A mesma diversidade ocorreu com os times de desenvolvimento responsáveis pela manutenção e evolução da aplicação, sendo estimada a atuação de mais de uma centena de desenvolvedores ao longo dos anos, grande maioria não mais presente na companhia. Como resultado, o legado apresenta um código-fonte sem padrão de codificação e uma arquitetura confusa e sem governança, longe dos padrões atuais de qualidade. Diante de todos esses sintomas, realizar a modernização desse sistema passou a ser a alternativa mais indicada para a contínua evolução da área de cobrança da companhia.

3.1.2 Tecnologia

O account-collector foi desenvolvido na linguagem Java e atualmente é executado na versão 5. O servidor de aplicação utilizado é o JBoss 4.2 [JBO04], um container robusto porém demasiadamente pesado, que amarra a aplicação a uma série de *frameworks* e tecnologias legadas. Dentre essas amarras estão os serviços implementados em EJB2, além de alguns já migrados para EJB3. A falta de padrão também atinge a configuração destes serviços, sendo parte via arquivos XML e a outra via anotações no código-fonte.

A adoção de tecnologias mais recentes é essencial para retardar o envelhecimento de software

[Par94][Leh96] e também para evitar dívidas técnicas de tecnologia, descrita na seção 2.1.2. Anos atrás, essa percepção natural fez com que mais um *framework* fosse adicionado a aplicação, o Spring [SPR], a fim de realizar parte do papel do já ultrapassado JBoss 4.2 e a plataforma JEE utilizada. Leve e não intrusivo, o Spring propõem técnicas de inversão de controle e injeção de dependência e não se limita às API's do JEE. Embora tenha sido uma atitude inovadora, a utilização deste *framework* ficou restrita às novas linhas de código desenvolvidas, sem que houvesse um *refactoring* para modificar o código já existente. Dessa forma, ambas as tecnologias passaram a conviver no dia-a-dia da aplicação, o que aumentou ainda mais a complexidade do sistema. Além disso, diversos problemas transacionais surgiram em decorrência do gerenciamento dos beans do JBoss e do Spring, que não possuem nenhuma integração.

A não padronização destas questões tecnológicas foram determinantes para que o sistema se tornasse obsoleto. A presença de diversos *frameworks* e API's com a mesma funcionalidade dentro desta única aplicação apenas aumentou a curva de aprendizagem e a dificuldade na manutenção e evolução do sistema.

3.1.3 Domínio

Por definição básica, para uma aplicação ser considera um micro-serviço seu escopo deve ser pequeno e sua responsabilidade única, conforme o princípio de responsabilidade única [New15]. Embora tamanho e responsabilidade seja subjetivo, quando múltiplos domínios se cruzam e para cada um há uma ramificação de funcionalidades implementadas através de diferentes serviços, fica evidente que não se trata de um micro-serviço, mas sim um monolítico.

O account-collector, AC, cujo nome não soa esclarecedor, tem como principal domínio transações de pagamento, isto é, é o serviço de *backend* responsável pela integração com agentes financeiros para realização das cobranças de produtos de diversas áreas do UOL. Essas transações podem ser tanto de compras online, onde há um cliente presencial, como de compras de assinaturas, onde periodicamente é realizada uma cobrança.

Há diversas peculiaridades no complexo domínio de cobranças. Além da transação em si, diversas outras funcionalidades relacionadas precisam existir para atender com qualidade aos clientes UOL e também as regras do Banco Central do Brasil, o BACEN. É possível, por exemplo, que o cliente conteste o pagamento e requeira a devolução do valor juntamente a operadora de cartão, operação conhecida por *chargeback* e que precisa ser refletida no sistema. Também existem serviços responsáveis por iniciar e processar cancelamentos e reembolsos em qualquer data futura para uma data transação. Por fim, uma outra funcionalidade sempre muito utilizada é a de quitação, uma transação específica que tem por objetivo sanar a dívida de um cliente que por algum motivo não conseguiu ter sua assinatura cobrada com sucesso.

Entretanto, nem todas as transações são processadas diretamente pela aplicação em estudo. Parte das transações de recorrência de assinaturas são efetuadas por outros sistemas, capazes de processar cobranças para meios de pagamento boleto físico e débito bancário. Esses sistemas têm uma característica em comum de não serem online, uma vez que o retorno do pagamento vem apenas através de arquivos EDI, *Electronic Data Interchange* [IBD95]. No entanto, uma vez que todo o domínio de pagamentos está sendo gerenciado no account-collector, todas essas informações de transações realizadas por outros sistemas são sincronizadas nele através de um complexo batch

de importação. Este e outros batches, serviços escalonados que são executados periodicamente, também fazem parte do código do sistema monolítico.

As transações processadas pelo account-collector restringem-se a cobranças de cartão de crédito e boleto online, fluxos que sofreram importantes alterações ao longo do tempo. Inicialmente, as cobranças de cartão de crédito eram realizadas através da integração com um sistema externo chamado Sitef, que usa tecnologia TEF: Transferência Eletrônica de Fundos, comumente utilizada em supermercados e lojas em geral por comunicar-se com um grande número de adquirentes [ADQ] e dessa maneira atender uma vasta gama de bandeiras de cartões.

Contudo, em meados de 2013, com o desenvolvimento de um *gateway* de pagamento dentro do próprio grupo UOL, o PagSeguro, identificou-se a possibilidade de migração da plataforma Sitef para esse novo sistema desenvolvido, gerando diversos ganhos para a companhia e, principalmente, enquadrando a plataforma UOL nos requisitos para tornar-se *PCI compliance* [RW06], um padrão de segurança de dados reconhecido mundialmente. O processo de migração do *gateway* de pagamento ocorreu em diversas etapas, geralmente produto a produto da companhia, até que todas as transações via Sitef passaram a ocorrer via gateway do PagSeguro. Embora essa mudança tenha sido conduzida com sucesso do ponto de vista de negócio, a urgência do projeto fez com que os melhores padrões de desenvolvimento não fossem seguidos, gerando dívida técnica, complexidade e ainda mais funcionalidades ao sistema legado.

Além de gerenciar todo domínio de transações, realizar toda a complexa integração com diversos gateways de pagamento, prover serviços de *chargeback*, cancelamento, quitação, validação de cartão de crédito, entre outros, o account-collector é também um orquestrador. É ele o responsável pela comunicação de qualquer alteração de status das transações para diversos tipos de clientes através de diferentes formas de integração, questão a ser explorada na seção de arquitetura.

3.1.4 Arquitetura

O estado da arte da arquitetura do sistema legado pode ser explorada sob duas óticas de diferentes granularidades: design de código e integração entre sistemas. Tais óticas serão detalhadas durante esta subseção, respectivamente.

Desenvolvido em Java, uma linguagem orientada a objetos, o sistema faz uso restrito de funcionalidades como encapsulamento, polimorfismo e interface. Embora herança seja frequentemente utilizada, é comum encontrarmos classes e métodos muito grandes semelhantes ao paradigma procedural, orquestrado por uma sequência de loops e estruturas condicionais. Tal fato fica evidenciado na classe TransactionDao.java, a qual tem 2,6 mil linhas, complexidade ciclomática de 182 e complexidade cognitiva de 262.

Além de não se aprofundar na utilização dos conceitos de orientação a objetos, o código do sistema legado também apresenta poucas implementações de padrões de projeto, termo mais comumente conhecido pelo nome em inglês: *'design patterns'* [GHVJ94]. Tais padrões, criacionais, estruturais ou comportamentais são altamente difundidos na comunidade de software como casos de sucesso dado os benefícios que apresentam em conceitos como acoplamento e coesão, fatores que afetam diretamente a manutenibilidade e evolutibilidade dos sistemas.

A falta de padrão no desenvolvimento de software é um problema comum em sistemas monolíticos legados em constante evolução, cujas boas práticas aplicadas sofreram mudanças ou foram

ignoradas ao longo dos muitos anos de desenvolvimento. No account-collector, tais questões podem ser observadas através de itens como:

- **Estratificação em camadas**: Nenhum tipo de padrão arquitetural de estratificação é seguido, tais como *model-view-controller* ou *layered-pattern* [Som10]. A falta de regras para criação de classes com papéis específicos como *Services*, *Controllers* e *Repositories* afeta o encapsulamento dos dados e a complexidade como um todo.

- **Padronização de nomes**: Também conhecido por *'Naming convention'* [NAM], este conceito foi ignorado durante o crescimento da aplicação. A falta de uniformidade é notável na nomenclatura de classes que mesmo com propósitos iguais, apresentam diferentes prefixos, sufixos ou estrutura. O mesmo pode ser observado na hierarquia dos pacotes Java, o que dificulta a leitura, navegação e entendimento dos fluxos do sistema, além de atestar a falta de estratificação em camadas.

A arquitetura do monolítico também pode ser analisada sob um contexto mais amplo: seu papel na orquestração de cobranças e como integra-se aos demais sistemas de toda companhia. Estes aspectos que vão além de código-fonte têm maior granularidade em relação ao design de código do sistema e acarretam resultados e impactos muito mais significativos [SSS16][HLH14], simultaneamente atrelados a uma maior complexidade e custo de desenvolvimento e evolução.

Como contextualizado na subseção de domínios, o account-collector é peça central da área de cobranças do UOL, sendo servidor e cliente de diversos outros sistemas. A figura 3.1 representa de forma simplificada essa arquitetura: clientes, dependências e as formas de comunicação utilizadas.

Em relação às formas de comunicação, nota-se uma grande heterogeneidade com o uso de diferentes tecnologias de integração para cada cenário. Ao total, coexistem quatro tipos de tecnologia distintas: RMI, HTTP, JMS e banco de dados. Além dos impactos relacionados a cada um desses tipos, que serão discutidos a seguir, o excesso de formas de comunicação por si só foi um fator determinante na alta complexidade do sistema. Infelizmente, não há ferramentas hoje capazes de prover métricas consistentes de problemas arquiteturais assim como há para as métricas de análise estática de código-fonte [LLA13][SCG+12].

O tecnologia RMI[Wal98], invocação remota de métodos, foi durante muito tempo a forma de comunicação padrão utilizada para expor serviços a outras aplicações. Somente nos últimos anos, com a ascensão do padrão de comunicação REST [WPR10], novos serviços passaram a ser expostos através dessa abstração do protocolo HTTP em detrimento da tecnologia RMI. Os ganhos de desacoplamento com essa mudança de tecnologia são grandes já que é totalmente *stateless* e ignora os detalhes da implementação de cada componente. O REST também torna possível a entrega contínua de software [Fow06], sem que haja indisponibilidade ao usuário, através da utilização de um software de balanceamento de carga, SLB [CMAM11], e um serviço de *health-check* na aplicação. Entretanto, como os serviços legados do account-collector nunca foram migrados para REST, a aplicação ainda gera indisponibilidade aos clientes RMI toda vez que uma nova entrega é realizada.

Dos clientes que ainda utilizam RMI, o checkout-online merece relevância pois é responsável pela venda online dos produtos comercializados pelo UOL. Este sistema oferece integração web que captura dos dados de meio de pagamento do usuário para em seguida invocar o serviço de processamento de cobrança do account-collector. Além deste, o AC dispõe de uma gama de serviços

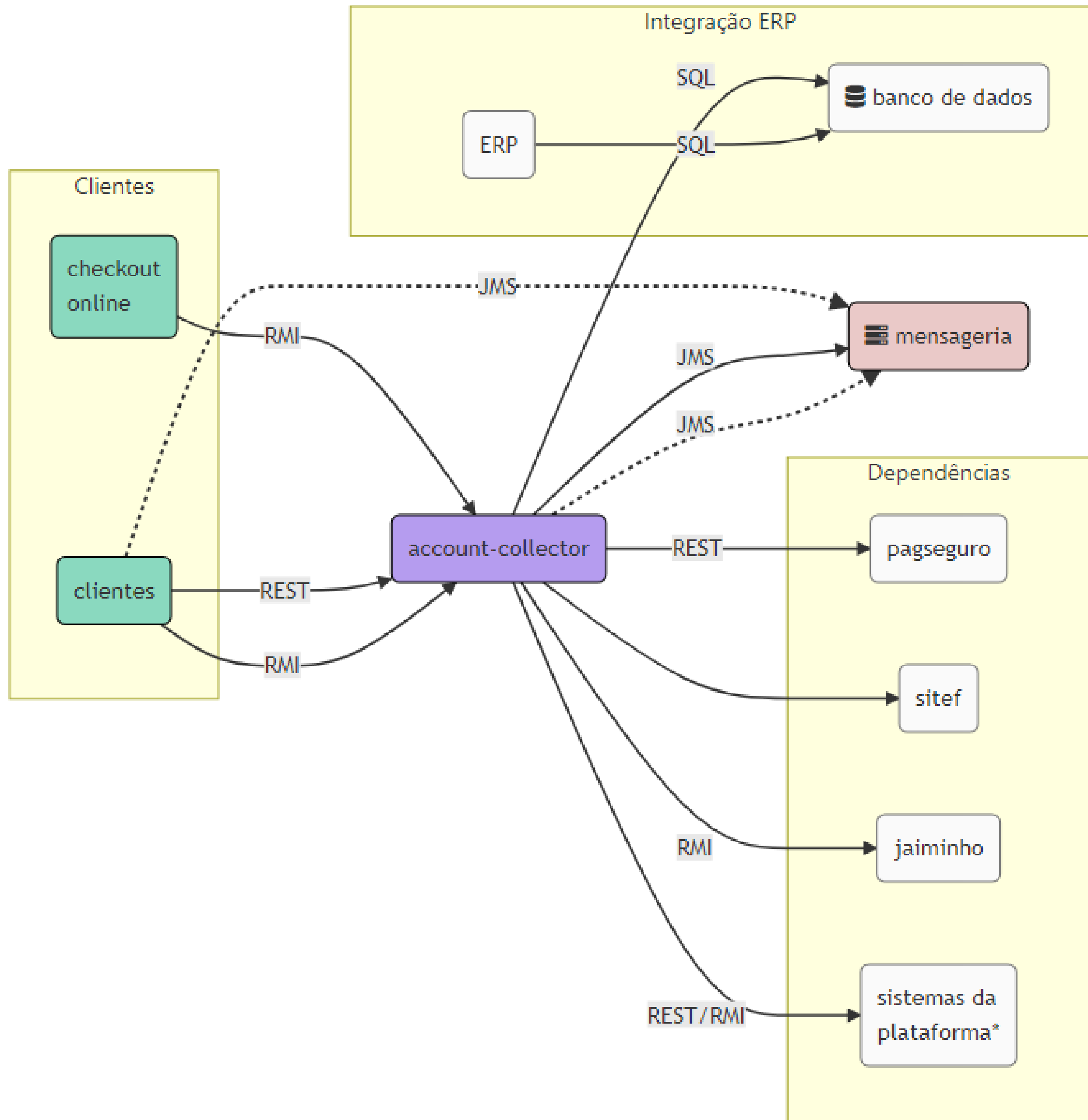

Figura 3.1: *Arquitetura do account-collector*

relacionados às cobranças expostos em RMI ou REST, consumidos por diversos sistemas que fazem interface com usuário, URAs, etc.

A tecnologia JMS[Dea12], *Java Message Service*, consiste em uma API Java de mensageria que permite integração de forma assíncrona, padrão arquitetural de coreografia descrito na seção 2.2.2. Através do servidor JBossMQ, o account-collector provê filas duráveis para as quais envia mensagens que serão lidas por um consumidor em específico. Também provê 'tópicos' que podem ser acessados por diversos assinantes, permitindo que uma mensagem seja lida por muitos consumidores.

Descrita na seção 2.2.2, integração de sistemas via banco de dados também está presente no monolítico. A necessidade surgiu assim que o sistema foi criado e precisava integrar-se ao sistema de ERP [UHU03], que na época não possuía outro tipo de comunicação. Através de batches que rodam

periodicamente, o account-collector lê tabelas do banco de dados onde o sistema de ERP grava as informações das cobranças de assinaturas. Assim, todo processamento das cobranças de assinatura do UOL tem início a partir da leitura destas tabelas. Após o processamento é a vez do AC comunicar ao ERP o resultado das transações, utilizando novamente as tabelas como forma de integração. Este mesmo padrão existe para diversos outros cenários, tais como quitação e cancelamento de transações.

Os parágrafos anteriores contextualizaram os blocos de 'Clientes' e 'Integração ERP' presentes na figura 3.1. Já o bloco 'Dependências' representa a integração com os sistemas no qual o account-collector é o cliente, as dependências externas. Duas destas dependências já foram evidenciadas na subseção de domínios, o Sitef e PagSeguro, sistemas utilizados para a comunicação com as adquirentes de cartão de crédito para processamento dos pagamentos. Embora todos os fluxos de cobrança já tenham sido migradas do Sitef para o PagSeguro, o fluxo de validação de cartão de crédito ainda utiliza a plataforma Sitef, que, dessa maneira, não pode ser removida do código-fonte do account-collector.

A integração com o Sitef é implementada por meio de bibliotecas externas que provêm uma API de acesso ao sitef-client, aplicação instalada nos servidores UOL mas desenvolvida pelo próprio Sitef, que realiza a comunicação final com seus servidores. Além de oferecer uma gama de serviços limitada, essa integração é totalmente obscura e dependente do Sitef, o que dificulta o *troubleshooting* de possíveis erros sistêmicos ou de erros em algumas cobranças em si.

De forma antagônica, a integração com o PagSeguro ocorre de modo transparente através do protocolo HTTP com a troca de mensagens em XML, uma comunicação semelhante mas que não segue exatamente todos os padrões REST. Desenhada na figura 3.1 de forma simplificada, essa comunicação ocorre em várias etapas, de forma assíncrona, na seguinte ordem:

1. **Checkout**: O account-collector faz uma chamada HTTPS ao PagSeguro, que retorna sincronamente o id da transação que foi aberta.

2. **Callback/Bell**: O PagSeguro, numa url de *callback* configurada, faz uma chamada HTTPS para o account-collector sem nenhuma informação de dados da transação, apenas com um código da notificação. Essa notificação ocorre somente quando há alguma mudança no estado da transação do lado do PagSeguro.

3. **Probe**: O account-collector faz outra chamada HTTPS ao PagSeguro passando como parâmetro o devido código de notificação previamente recebido. É somente nesta conexão que os dados da transação são trafegados, tais como estado atual, valor cobrado, presença de cancelamentos, etc.

Este mesmo fluxo é também utilizado para processamentos de cancelamentos e devoluções, alterando-se apenas o serviço do PagSeguro chamado na etapa 1. Embora garanta total segurança, essa integração via HTTP é suscetível a falhas de rede e problemas de concorrência, o que aumenta a complexidade de implementação. De forma similar, essas características aproximam-se aos obstáculos encontrados na adoção de micro-serviços que utilizam comunicação HTTP, conforme apresentado na seção 2.2.2.

Além das dependências relacionadas às cobranças em sí, o monolítico também integra-se a diversos outros sistemas. Tais como:

- **Jaiminho**: Sistema interno do UOL responsável pela efetivação do envio de emails e SMS's. Contém os templates com a formatação adequada para cada tipo de email cadastrado. No entanto, o responsável pela orquestração dos envio é o account-collector, que determina qual o tipo de email será enviado em cada situação relacionada ao ciclo de uma transação de pagamento do cliente, comunicando-se via RMI com o Jaiminho.

- **Sistemas da plataforma**: Correspondem a diversos sistemas da plataforma corporativa aos quais o account-collector se integra via REST e RMI para consulta e cadastro de dados. Estes sistemas são responsáveis pelos domínios do cliente, desde dados de endereço até meios de pagamento. Muitos destes dados são utilizados para a integração com os outros sistemas.

Ainda há outras integrações sistêmicas existentes que serão abstraídas nesse primeiro momento mas que serão mencionadas no capítulo 4, conforme o detalhamento do assunto.

3.2 Metodologia aplicada

3.2.1 Estratégia de pesquisa

O estudo de caso é a estratégia de pesquisa qualitativa mais comumente aplicada na área de informação de sistemas [Mye97], capaz de atender tanto propósitos exploratórios como explanatórios através da profunda observação de um fenômeno contemporâneo dentro de seu contexto na vida real [Yin08][Mye97]. Este fenômeno é a unidade de estudo investigada empiricamente pelo pesquisador durante um instante ou um período de tempo, fornecendo materiais, evidências e experiências vivas e cheia de interpretações que não são possíveis de capturar através de teorias [Yin08]. Dessa forma, este tipo de estudo se adequa muito bem à área de engenharia de software cujo objetivo de análise é amplo e se expande além das fronteiras técnicas para questões organizacionais, comportamentais e humanas [Mye97]. Também é apropriado para questões nas quais os estudos são poucos e recentes, onde não há ainda limites bem definidos na literatura entre o fenômeno e o contexto [Mye97], exatamente como o panorama atual de modernização e micro-serviços.

Uma vez identificada a adequação da aplicação do estudo de caso no contexto dos fenômenos de software apresentados durante todo capítulo 2, a presença do pesquisador como funcionário desde 2012 no dia-a-dia de uma grande organização de desenvolvimento de software surgiu como oportunidade para a condução deste projeto de pesquisa e também como grande fator motivador para a análise sob ponto de vista acadêmico e formal de práticas adotadas pela indústria. A conjuntura mostrou-se apropriada para uma investigação a fundo dessas questões contemporâneas em meio a um contexto real, apresentando grande relevância para a literatura uma vez que ainda são poucos os estudos capazes de realizar a integração entre o dia-a-dia da indústria de software com assuntos notórios porém escassos na literatura acadêmica.

Neste contexto, embora o pesquisador tenha pouco ou nenhum controle dos eventos relacionados ao fenômeno, sua proximidade constante e diária com o objeto sob investigação durante um grande período apresenta características intrínsecas aos estudos etnográficos, muito comuns em pesquisas nas áreas sociais. A pesquisa etnográfica tem características semelhantes ao estudo de caso mas exige uma profunda imersão do pesquisador na cultura do objeto em estudo durante um longo período de tempo [Mye99]. Além disso, o estudo etnográfico tem como principal forma de coleta

de informações a observação do objeto e dos participantes durante sua rotina diária [Mye97]. Por conseguinte, a metodologia aqui apresentada incorpora tanto traços da estratégia de estudo de caso como da etnografia, dado a imersão diária por mais de três anos do pesquisador no desenvolvimento deste projeto de pesquisa que teve como principal fonte de coleta de dados qualitativos as anotações e experiências observadas diretamente e através dos participantes durante todo esse período.

As estratégias de pesquisas utilizadas podem basear-se em diversos tipos de dados, coletados através de diferentes métodos. No caso deste projeto, optou-se por utilizar tanto dados qualitativos como quantitativos, abordagem conhecida na literatura como métodos mistos ou triangulação [Cre08]. Os dados qualitativos não podem ser medidos e serão inferidos a partir do método de observação. Já os dados quantitativos serão extraídos a partir do código-fonte do software atrelado ao estudo de caso. Dessa forma, além da coleta, a interpretação conjunta desses dois tipos de dados fornecerá resultados mais precisos e reduzirá as limitações e o viés de cada um dos métodos, quando utilizados separadamente, convergindo para um resultado mais robusto e confiável.

3.2.2 Processo de coleta e análise incremental

Dado o caráter faseado do processo de modernização adotado, *Strangler Application*, a avaliação do estudo de caso será também dividida em marcos, não regulares, de acordo com as importantes entregas de software realizadas relacionadas ao account-collector. Dessa forma, cada marco pode ter duração e tamanho diferentes, definidos para esta pesquisa para facilitar a apresentação das informações mas sem nenhuma influência no dia-a-dia da empresa. Os gatilhos abaixo definirão o início de uma novo marco:

- Criação de um micro-serviço em substituição a uma responsabilidade da aplicação monolítica.
- Migração de alguma responsabilidade do monolito para algum micro-serviço já existente.
- Reestruturação dos micro-serviços recém-criados.
- Remoção de código morto do legado, cuja funcionalidade já não é mais utilizada.

Como essas implementações podem ocorrer de maneira paralela e em diversos sistemas ao mesmo tempo, uma data próxima no tempo será escolhida de forma a garantir uma atomicidade entre todas as alterações com um mesmo propósito. Implementações de pequenas funcionalidades ou correções de *bugs* no legado devido a alguma urgência da área de negócio serão tratadas junto ao contexto dos marcos mas não serão gatilhos. A figura 3.2 apresenta de forma simplicada a metodologia até aqui descrita, com a segmentação da coleta e análise dos dados em cada marco estabelecido, do marco inicial M0 até o último, Mn, permitindo um investigação incremental e histórica de todo o processo realizado.

Para que o estudo de caso possa ser avaliado sob diversos ângulos e abrangentes contextos, os dados serão coletados e analisados mediante um processo sistemático em relação às três áreas de conhecimento desta pesquisa: dívida técnica, arquitetura de micro-serviços e modernização de software. Além disso, de forma a tentar reduzir o viés da pesquisa, tanto dados quantitativos como qualitativos serão coletados. Do ponto de vista quantitativo, métricas de código-fonte do account-collector e das novas aplicações criadas serão utilizadas para agregar valor a análise. Do ponto de

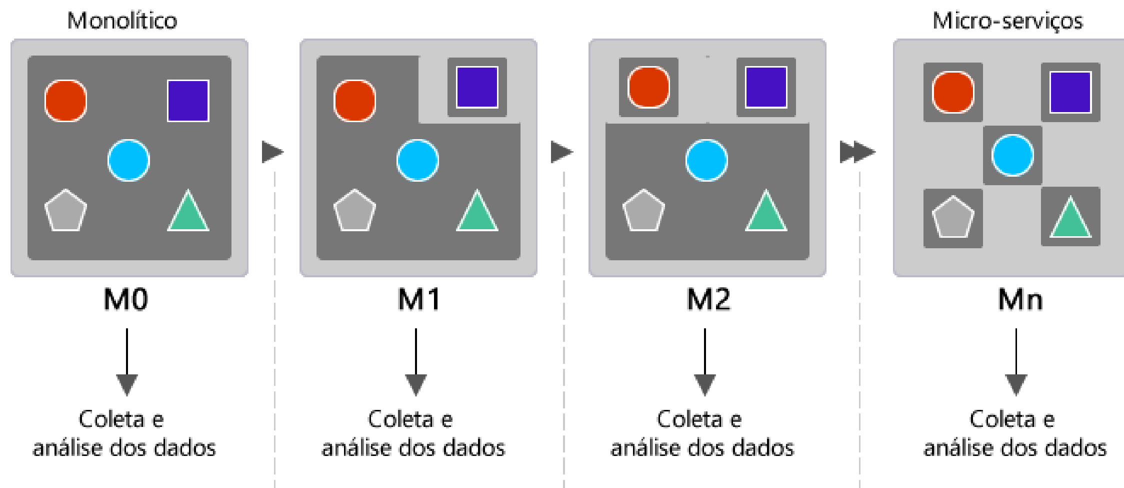

Figura 3.2: *Metodologia de coleta e análise faseada*

vista qualitativo, a observação terá como enfoque diversas questões pré-estabelecidas, que serão avaliadas em cada marco dependendo do contexto das alterações.

3.2.2.1 Dívida Técnica

A dívida técnica, além do aspecto gerencial e estratégico, fornece uma ampla gama de métricas de software que serão utilizadas para prover informações quantitativas a respeito dos sistemas envolvidos a cada marco, tanto do legado como dos novos micro-serviços. A ferramenta SonarQube será utilizada para realizar a extração dessas métricas, conceituadas na seção 2.1.3:

- ***Technical Debt***: Fornecerá fácil visibilidade do impacto das alterações pois sintetiza numa unidade específica as outras métricas relacionadas a manutenibilidade. Será utilizada a unidade de homem/dia.
- ***Technical Debt Ratio***: Fornecerá evidências de comparação da dívida técnica entre os sistemas.
- ***Issues***: Somatória do número de bugs, vulnerabilidades e bad smells do sistema.
- **Complexidade**: Essa variável será dada pela soma das duas complexidades extraídas a partir do SonarQube, ciclomática e cognitiva. Por utilizarem unidades de mesma grandeza, o valor agregado destas duas métricas fornecerá um dado sumarizado em relação a complexidade do sistema como um todo, proveniente do resultado de cada método e classe do sistema.
- **Duplicações**: Porcentagem do código duplicado do sistema, dado em relação ao número de blocos duplicados pelo número total de linhas do sistema.
- **LOC**: ou *"lines of code"*, representa a quantidade de linhas de código-fonte do sistema, o melhor indicador do tamanho do sistema.
- **Métodos**: Outra métrica de tamanho, representa o número de métodos do sistema, fornecendo informações sobre coesão e acoplamento do mesmo.

- **Classes**: Outra métrica de tamanho, representa o número de classes do sistema, fornecendo informações sobre coesão e acoplamento do mesmo.

Além das métricas primárias obtidas, métricas secundárias serão geradas para fornecer maior embasamento quanto as características e qualidade dos sistemas analisados:

- **Média do tamanho das classes**: LOC/Classes. Média do número de linhas de código por classe do sistema.
- **Média do tamanho dos métodos**: LOC/Métodos. Média do número de linhas de código por método do sistema.
- **Média de métodos por classe**: Métodos/Classes. Média do número de métodos por classe do sistema.
- **Frequência de issues**: LOC/Issues. Significa a existência de 1 Issue a cada certo número de linhas.
- **Frequência de complexidade**: LOC/complexidade. Significa a existência de 1 unidade de complexidade a cada certo número de linhas.

A dívida técnica, além da análise individualizada de cada componente, será utilizada também como forma de avaliação marco a marco do processo de modernização como um todo, identificando se houve aumento ou queda da dívida técnica total. Para isto, será considerado como sistema resultante do processo o agregado de todos os sistemas envolvidos, desde o legado a cada nova aplicação criada. Assim, para cada métrica de dívida técnica, seu valor para o sistema resultante será dado em função da somatória abaixo:

$$R_i = debt(L) + \sum_{n=1}^{x} debt(M_n)$$

Sendo:
R_i: dívida total no marco i para tal métrica.
$debt(L)$: dívida do monolito legado para tal métrica.
$debt(M_n)$: dívida da nova aplicação n para tal métrica.
x: quantidade de novas aplicações criadas.

As métricas acima serão comparadas e análisadas caso a caso, dependendo do contexto da alterações realizadas no marco, provendo informações significativas do andamento do processo de modernização e da características dos sistemas. Além disso, permitirá uma análise histórica do comportamento das métricas e qual a relação entre elas. No entanto, conforme apresentado na seção 2.1.2, são vários os tipos de dívida técnica que vão além do código-fonte e que não estão contidos na análise acima. Dessa forma, do ponto de vista qualitativo, algumas outras questões serão analisadas para complementar os resultados obtidos. São elas:

- **Dívida técnica além do código-fonte**
  Como conceituado na seção 2.1.2, existem outros diversos tipos de dívida que vão além do código-fonte da aplicação. Qual o comportamento dessas dívidas durante o andamento do estudo-de-caso?

- **Origem da dívida técnica**
  Conforme quadrante da dívida técnica contextualizado na seção 2.1.1, qual a origem da dívida técnica nos sistemas? Intencional? Prudente?

- **Reflexo da dívida técnica no dia-a-dia**
  A dívida técnica teve reflexo no dia-a-dia de desenvolvimento de software?

3.2.2.2 Arquitetura

A avaliação dos aspectos arquiteturais de micro-serviços através de dados quantitativos não é algo familiar na literatura. Entretanto, diversos estudos acadêmicos fornecem embasamento suficiente para que as aplicações possam ser avaliadas qualitativamente em relação as seguintes questões:

- **As aplicações seguem o padrão de uma arquitetura de micro-serviços?**
  Isto é, avaliar se cada aplicação possui um domínio específico, responsabilidade única e outras características inerentes ao padrão arquitetural de micro-serviços descritas no capítulo 2.2.

- **Quais paradigmas de comunicação foram adotados?**
  Requisição-resposta ou baseado em eventos? Comunicação síncrona ou assíncrona? Por quê a escolha em cada cenário? A seção 2.2.2.2 será utilizada como referência.

- **A comunicação entre as aplicações atende as premissas de integração?**
  Para cada integração entre os serviços, *IPC*, as premissas descritas na seção 2.2.2 foram respeitadas?

- ***Bad Smells* de micro-serviços**
  Algum *bad smell* do catálogo de anti-padrões conceituados na seção 2.2.5 pode ser identificado?

3.2.2.3 Modernização de software

A modernização, conceito antigo da literatura de engenharia de software, também não fornece métricas quantitativas para sua validação. Entretanto, a literatura provê diversas boas práticas que serão úteis na análise das questões abaixo, de forma qualitativa, observadas marco a marco do processo.

Em relação ao processo de modernização:

- **Indícios para a modernização do legado**
  Conforme contextualizado na seção 2.3, havia indícios suficientes para que fosse necessário um processo de modernização?

- **Estratégia de modernização adotada**
  Dentre as estratégias de modernização presentes na literatura conceituadas na seção 2.3.1, houve algum indício da adoção de alguma estratégia de modernização específica?

- **Critério para a priorização de migração dos módulos**
  Qual o critério utilizado para se definir a priorização do módulo a ser migrado? As boas práticas presentes na literatura descritas na seção 2.3.1.1 foram adotadas?

- **Adição de novas funcionalidades no legado**
  Houve adição de novas funcionalidades no sistema legado? Poderiam ter sido construídas já em um novo sistema?

- **Entrega de valor para o negócio**
  Além das questões técnicas relacionadas a modernização, houve alguma entrega de valor para o negócio?

Em relação a implementação técnica:

- **Técnicas de decomposição de micro-serviços**
  Com base nos conceitos apresentados na seção 2.3.1.1, qual técnica de decomposição foi utilizada para definir o escopo da extração no marco?

- **Técnicas de isolamento de dados do domínio**
  O novo micro-serviço teve seu domínio preservado em relação aos dados do legado? Alguma técnica como *anti-corruption layer* foi utilizada? Qual o relacionamento de dados entre o sistema legado e os novos micro-serviços?

- **Técnicas de migração e validação das alterações**
  Como foram testados os módulos migrados? Foi utilizado *feature-toggle*?

Capítulo 4

Modernização do sistema legado

Neste capítulo, toda experiência observada e vivenciada durante o desenvolvimento do estudo de caso será apresentada detalhadamente e avaliada de acordo com os procedimentos estabelecidos na seção 3.2.2.

O estudo de caso será apresentado em 6 seções, cada uma representando uma etapa do processo de modernização do sistema legado, em ordem cronológica, do marco zero ao marco 5, conforme apresentado na figura 4.1. Ao todo foram 5 iterações de desenvolvimento, com início em julho de 2015 e finalização em julho de 2018, duração exata de 3 anos.

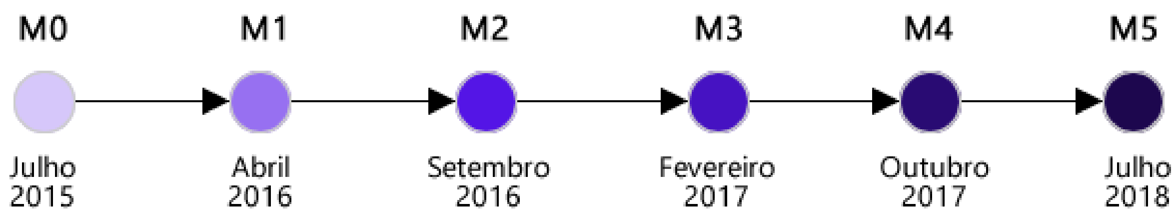

Figura 4.1: *Marcos do processo de modernização do sistema legado*

4.1 Padrões de apresentação

Durante todo esse capítulo, diversas figuras exibindo a arquitetura dos sistemas e a forma de integração entre os mesmos serão utilizadas para facilitar a visualização dos complexos fluxos e das mudanças realizadas durante a modernização. Essas figuras seguirão o formato de diagramas de componentes, padrão mais comum utilizado para representação dessas arquiteturas [AAE16], com algumas simplificações e extensões exemplificadas na figura 4.2 e detalhadas abaixo:

- Para facilitar a visualização do fluxo de mensagens entre os sistemas sem que seja necessário também um diagrama de sequência, nas próprias setas que representam as conexões entre os componentes estará especificado o protocolo de comunicação utilizado e também a ordem de execução do mesmo.
- Setas lisas representam a conexão entre os componentes de forma síncrona e blocante.

- Setas tracejadas representam retornos assíncronos de aplicações servidoras como resposta à chamadas passadas realizadas por aquele cliente, ou seja, o cliente faz uma chamada sícrona ao servidor, recebe um retorno imediato com a promessa de que a requisição será processada. Em algum momento futuro, quando o processamento tiver sido concluído, o servidor abrirá uma nova chamada com o cliente enviando-lhe os dados de resposta daquela requisição.

- Setas tracejadas também representam comunicação assíncrona com filas, quando o cliente é um *subscriber* do sistema de mensageria e realiza o consumo de suas mensagens.

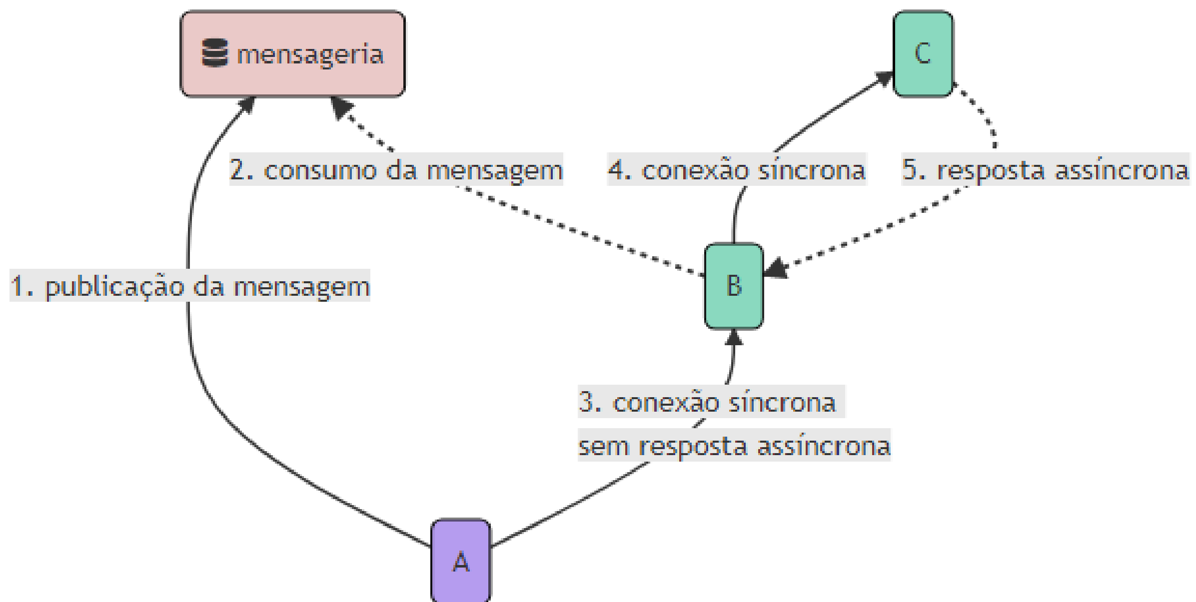

Figura 4.2: *Exemplo de diagrama de componentes na representação de arquiteturas*

4.2 Marco zero

O marco zero corresponde a iteração inicial, ao ponto de partida da pesquisa, iniciada em julho de 2015. Seu escopo compreende apenas uma aplicação, o monolítico legado conhecido por account-collector, ou simplesmente AC, um dos principais sistemas do complexo ambiente de cobranças do UOL.

4.2.1 Contextualização

Todos as características do account-collector foram apresentados na seção do estudo de caso 3.1, na qual foram detalhadas as evidências do ponto de vista histórico, tecnológico e arquitetural da grande complexidade do sistema legado. A figura 4.3 apresentada novamente abaixo representa uma síntese de boa parte dessas características, que serão citadas como referência ao longo de todo este capítulo de modernização.

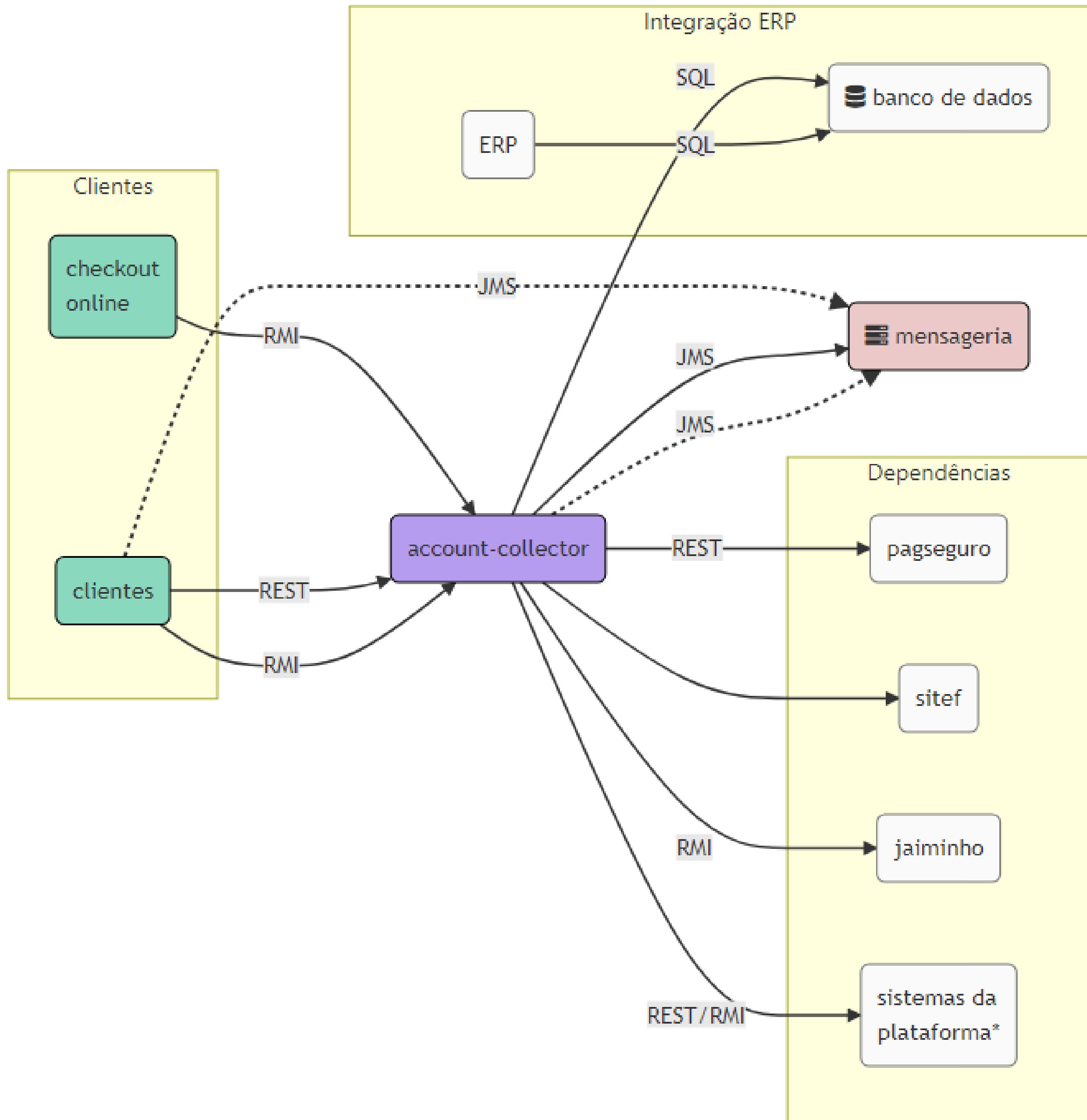

Figura 4.3: *Sistema account-collector*

4.2.2 Avaliação

Por ser o marco inicial do estudo de caso desta pesquisa, a avaliação deste marco tem um caratér excepcional já que tem como objeto de observação somente a aplicação legada sem que haja ainda um processo de modernização. Entretanto, muitas das questões levantadas na seção 3.2.2 também podem ser aplicadas neste contexto.

4.2.2.1 Dívida Técnica

Dívida técnica e o legado account-collector possuem uma relação intrínseca. Os detalhes de implementação e os diversos problemas até aqui explorados revelaram vários indícios de dívidas técnicas, especialmente arquiteturais. Além disso, a falta de governança da aplicação contribuiu

para o acúmulo inadvertido da dívida técnica ao longo dos anos, um dos aspectos determinantes para que o processo de modernização fosse necessário.

Essa relação de causa-efeito da dívida técnica pode ser percebida através das dívidas de código-fonte, capazes de serem mensuradas através da análise estática de código em dados quantitativos, isolados de interpretação. Dessa forma, assim como será feito nas demais iterações e para todas as aplicações envolvidas no processo de modernização, as métricas descritas na seção 3.2.2 foram extraídas e disponibilizadas na tabela 4.1, corroborando as evidências descritas ao longo deste capítulo.

Métricas	**Marco Anterior**	**Marco zero**	**Variação**
Technical Debt	-	1.472 dias	-
Technical Debt Ratio	-	11,8%	-
Issues	-	85.111	-
Complexidade	-	29.155	-
Duplicações	-	23,9%	-
LOC	-	199.761	-
Métodos	-	13.867	-
Classes	-	2.114	-
Média do tamanho das classes	-	94,49	-
Média do tamanho dos métodos	-	14,41	-
Média de métodos por classe	-	6,56	-
Frequência de issues	-	2,35	-
Frequência de complexidade	-	6,85	-

Tabela 4.1: *Marco Zero: dívida técnica do account-collector*

Os números obtidos das métricas de dívida técnica são significativos mas não fornecem muito valor sozinhos, já que podem variar demais de acordo com o tipo de projeto, domínio e até a ferramenta de extração utilizada. A maior relevância pode ser notada quando estas métricas são comparadas a elas mesmas, através de diversas iterações ao longo do tempo. Estes valores obtidos no marco zero servirão como base para a avaliação ao longo do processo de modernização nos demais marcos do estudo.

A métrica *Techinical Debt*, que expressa o custo de manutenibilidade do sistema, cálcula em 1.472 dias o tempo para todas as dívidas possam ser pagas ao custo de 1 desenvolvedor. Entretanto, dado as limitações ainda existentes hoje, esse número refere-se somente as dívidas de código-fonte extraídas, não levando em consideração as dívidas arquiteturais, de documentação, ambiente, testes, etc. Na prática, seria necessário um esforço ainda maior para que esse sistema pudesse atingir um nível alto de qualidade.

Em relação às outras métricas que se destacam, percebe-se que há um grande número de duplicações de código, quase $\frac{1}{4}$ do sistema, um total aproximado de 50.000 linhas duplicadas. A quantidade de *issues* do sistema é outro valor signficativamente alto, um montante de 85.111 *bugs*, vulnerabilidades e *bad smells*, cuja frequência indica a presença de algum problema a cada 2,35 linhas de código.

Dívida técnica além do código-fonte

Como conceituado no capítulo 2.1.2, os tipos de dívida técnica vão além do código-fonte da aplicação e das métricas acima obtidas. Destes outros tipos, grande maioria pode ser encontrada no account-collector, tais como tecnologia, testes, ambiente de desenvolvimento e documentação.

A dívida técnica de tecnologia é uma das mais comuns no legado dado o tempo de existência da aplicação. Como não houve uma atualizam das ferramentas, linguagens e frameworks com o passar dos anos, o account-collector apresenta restrições inerentes à linguagem Java 5, ao servidor de aplicação JBoss 4.2 e à comunicação via RMI, além de outros aspectos.

Outra questão constante no dia-a-dia do legado é a dívida técnica de testes. Embora existam testes automatizados, por volta de 900, boa parte são testes mal escritos, intermitentes e que demoram em torno de 24 horas para rodar. Dos quase mil testes, uma média de 150 falham a cada regressão, conforme demonstrado na figura 4.4, obtida da ferramenta Jenkins responsável pela execução da suíte de testes. O eixo horizontal da figura corresponde as diversas execuções dos testes de regressão, enquanto o eixo vertical mostra a quantidade total de testes em azul, e os testes que falharam em vermelho. Como nunca houve priorização para a correção dos mesmos, as equipes de desenvolvimento passaram a apenas monitorar que não houvesse um grande aumento no número de falhas. Entretanto, com o passar do tempo, os testes que não garantiam quase nenhuma segurança passaram a ser ignorados e não mais atualizados pelos desenvolvedores, tornando-se crescente o número de falhas e o risco de novas entregas.

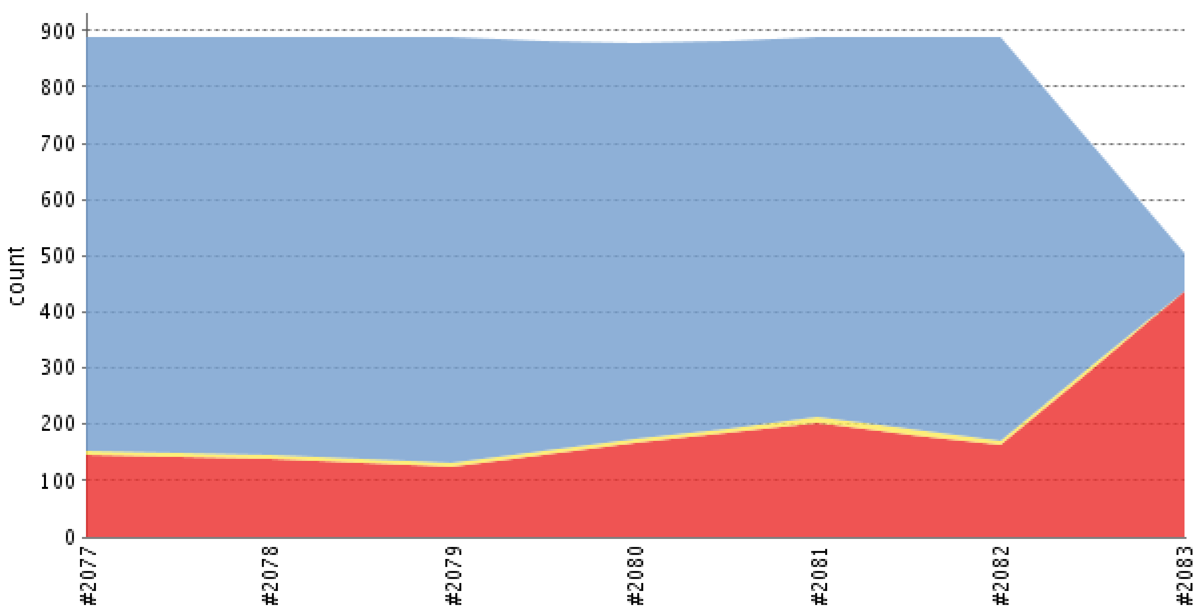

Figura 4.4: *Histório de execução dos testes de regressão do account-collector*

Nesse período de 2015, à entrega em produção do sistema legado ainda era uma tarefa manual e custosa, dívida técnica relacionada ao ambiente de desenvolvimento. O processo para geração do pacote era parcialmente automatizado com a entrega sendo realizada de maneira manual através de uma equipe de implantação responsável. Além disso, devido a limitações da clusterização do account-collector, as entregas eram realizadas somente em período de janela, nos horários de menor uso da aplicação, geralmente entre as 2h e 6h da manhã. Mesmo assim, indisponibilidade poderia ser gerada durante o período do *delivery*, afetando clientes que estivessem realizando compras neste

horário.

A documentação e distribuição do conhecimento desta aplicação também era escassa, sendo essa mais uma dívida não mensurável presente na rotina dos desenvolvedores. O código de difícil entendimento e falta de documentação dificultava a quebra das histórias de desenvolvimento, a estimativa dos prazos para novas funcionalidades e a percepção dos efeitos colaterais que poderiam ser gerados.

Origem da dívida técnica

Do ponto de vista classificatório, quanto à natureza ou origem da dívida técnica, conforme descrito na seção 2.1.1, a forma **irresponsável e não intencional** de desenvolvimento foi predominantemente aplicada ao longo dos anos passados na aplicação. Isto porque o conceito de dívida técnica é algo recente na indústria se comparado à idade do sistema e dessa forma não havia o conhecimento nem a preocupação com seu gerenciamento. Outro razão que suporta essa conclusão é a falta de governança do sistema, que foi sendo modificado por desenvolvedores de diversos níveis e culturas em diferentes equipes desde os anos 2000, sem que houvesse uma política de revisão de código e controle de qualidade.

Entretanto, em muitas situações no passado as dívidas de caráter **irresponsável e intencional** também foram utilizadas. A urgência para atender o *time-to-market* das áreas de negócio devido às características inerentes aos sistemas de pagamento foi durante muito tempo tida como prioridade em detrimento de um bom *design*. A falta de gerenciamento coloborou com essa irresponsabilidade.

Além disso, independentemente da área, qualquer sistema tende a se tornar obsoleto com o passar de 18 anos devido a dívida técnica de tecnologia, conforme sugerido por Parnas e seu estudo relacionado ao envelhecimento de software [Par94]. Este tipo de dívida técnica comum no account-collector pode ser classificada como **prudente e não intencional**, uma vez que não há responsabilidade e nem intenção, apenas a inevitável ação do tempo. Entretanto, vale ressaltar que medidas paliativas de manutenção podem ser tomadas para evitar tamanho acúmulo de dívida.

Entretanto, nos últimos anos a maior governança sobre o sistema fez com que a natureza da dívida técnica fosse alterada. Desta vez, a mais comumente aplicada é a dívida **prudente e intencional**, dado que a necessidade de entregas rápidas persiste mas há um comprometimento futuro para que a melhor solução seja desenvolvida. Desta forma é possível encontrar evidências das quatro classificações existentes de dívida técnica no legado, em diferentes períodos do seu ciclo de vida.

Reflexo da dívida técnica no dia-a-dia

A queda da produtividade das equipes de desenvolvimento é o impacto mais fácilmente perceptível em decorrência do acúmulo das dívida técnica até aqui apresentadas no legado. Segue a lista de evidências:

- A enorme quantidade de *issues*, muitos deles bugs, aliados a baixa resiliência do sistema, tornam frequentes pequenos incidentes em produção que precisam ser analisados e tratados pelas equipes de desenvolvimento.

- A falta de testes confiáveis torna maior o risco de novas entregas e mais fácil a criação de novos bugs.

- As tecnologias antigas apresentam restrições e não são capazes de garantir a mesma eficiência e rapidez durante o desenvolvimento, testes e entrega de software.
- As entregas manuais em produção exigem a presença de desenvolvedores durante o período noturno, tomando um tempo que poderia ser utilizado para o desenvolvimento.
- A falta de documentação leva a uma extensiva busca por entendimento e dificulta uma correta tomada de decisão por parte dos desenvolvedores e gerência.

Além disso, o reflexo dessas dívidas também é perceptível no ânimo dos desenvolvedores, que apresentam dificuldades e receios ao executarem atividades no código legado. Por vezes, optam por tomar decisões mais simples para evitar a propagação de erros e efeitos colaterais no sistema.

4.2.2.2 Arquitetura

Conforme exposto ao longo da seção de domínio 3.1.3 e arquitetura 3.1.4, vários anti-padrões podem ser encontrados no sistema legado. Uma breve análise será realizada nesta seção mediante as questões especificadas na seção 3.2.2.

As aplicações seguem o padrão de uma arquitetura de micro-serviços?

Como contextualizado durante toda pesquisa, o account-collector é uma aplicaçao monolítica e não um micro-serviço. As evidências abaixo confirmam tal distinção:

- Possuí múltiplas funções, muitas delas totalmente independentes, ferindo princípio de responsabilidade única.
- Um único sistema com diversas funcionalidades ao invés de pequenas aplicações autônomas e independentes.
- Além de possuir domínio próprio bem amplo, faz a orquestração de diversos fluxos independentes, podendo ser considerado uma aplicação orquestradora em diversos cenários.
- Não se comunica através de protocolos leves e não intrusivos. Integração via banco de dados, por exemplo, é uma das formas intrusivas utilizadas.
- Não apresenta resiliência e tolerância a falhas em diversas situações, necessitando de atuação manual em casos de erros de rede na comunicação com o PagSeguro, por exemplo.
- Complexo para escalar devido as características do cluster JBoss 4.2 e falta de automação. Além disso, somente é possível escalar todas as funcionalidades de uma vez.
- Entregas em produção são custosas e arriscadas.
- Não favorece um alinhamento organizacional desacoplado com uma boa definição de interfaces.
- Não apresenta design evolucionário, sendo de difícil substituição.

Quais paradigmas de comunicação foram adotados?

Embora não seja comum mesmo em grandes sistemas, o account-collector utiliza não apenas uma, mas quatro diferentes tecnologias de comunicação: REST, RMI, JMS e Banco de dados, apresentadas caso a caso na seção 3.1.4. Essas tecnologias compreendem três dos paradigmas de comunicação expostos em 2.2.2, utilizados de diferentes maneiras para diferentes funções.

No caso do paradigma **requisição-resposta**, tanto RMI quanto REST implementam este paradigma de comunicação caracterizado pelo estabelecimento de uma conexão direta entre as aplicações cliente e servidor. Entretanto, é comum que no padrão HTTP alguns cenários possam ser implementadas de modo assíncrono, como foi feito no caso da integração com o PagSeguro, por exemplo. Já no caso da tecnologia RMI apenas implementações síncronas e blocantes foram implementadas.

O paradigma de comunicação **baseado em eventos** também é empregado no account-collector através da tecnologia JMS, *Java Message Service*, uma API que permite a comunicação assíncrona entre os sistemas de forma totalmente desacoplada e com alta escalabilidade, pois garante que novas aplicações tenham acesso aos dados sem que haja qualquer alteração nos sistemas. Este paradigma pode ser utilizado de duas maneiras:

Uma outra tecnologia utilizada é o JMS, *Java Message Service*, uma API construída sob o paradigma **baseado em eventos**, que permite a comunicação assíncrona do account-collector com outros sistemas de forma totalmente desacoplada e com alta escalabilidade, pois garante que novas aplicações tenham acesso aos dados sem que haja qualquer alteração no account-collector. Este paradigma pode ser utilizado de duas maneiras:

- **Modelo ponto a ponto**: Composto por filas que tem mensagens publicadas por apenas uma aplicação produtora e consumidas também por apenas uma aplicação consumidora. O account-collector utiliza este tipo de comunicação para comunicar-se com outras aplicações da plataforma-corporativa em cenários específicos.

- **Modelo *publish/subscribe***: Neste caso, filas são substuídas por tópicos, elemento no qual diversas aplicações podem inscrever-se para receber as mensagens nele publicadas. O account-collector faz uso deste modelo através de um tópico de notificação de mudança no estado das transações, no qual publica mensagens com os dados das mesmas. Neste tópico, diversos clientes da plataforma corporativa e também de outras áreas estão inscritos e consumindo tais mensagens. Dessa forma, no caso do lançamento de novos produtos e serviços, novas aplicações podem se conectar e escutar as mensagens relacionadas ao produto em específico sem que haja desenvolvimento do lado do account-collector, uma grande vantagem deste tipo de paradigma.

A quarta e última forma de integração utilizada no account-collector é via **banco de dados**. Este tipo de integração não possuí um paradigma específico uma vez que os sistemas gerenciadores de banco de dados não foram criados com a finalidade de comunicação, e sim persistência de dados. Dada a característica obscura desta integração, ela sempre ocorre de forma assíncrona, sem que os sistemas tenham conhecimento um do outro. Apesar da semelhança com o paradigma baseado em eventos, confunde-se nesse tipo arquitetura os papéis de cliente e servidor uma vez que não há responsabilidade definida. Dessa forma, nunca se tem certeza da origem dos dados, que podem ter sido adicionados ou alterados manualmente ou por qualquer sistema que tenha acesso a tais tabelas.

A comunicação entre as aplicações atende as premissas de integração?

As três premissas de integração conceitudas na seção 2.2.2: "novas entregas sem impacto aos clientes", "API's independentes de tecnologia"e "interfaces sem detalhes da implementação", servirão como base para avaliação das quatro tecnologias de integração utilizadas no account-collector.

O REST, conjunto de princípios sobre o protocolo HTTP, provê alto desacoplamento entre cliente e servidor, permitindo que alterações sejam realizadas no servidor, como a adição de novos campos, sem que o cliente seja impactado. Além disso, é possível utilizar versionamento dos recursos para que mudanças mais drásticas não afetem as aplicações clientes. Em relação a tecnologia, por basear-se no padrão HTTP, permite a integração de qualquer linguagem de programação via texto. Além disso, os recursos podem ser expostos de uma forma que abstraia todos os detalhes de implementação.

Já a integração via RMI, um implementação de *remote procedure call* da plataforma Java, necessita da existência de uma JVM, *Java Virtual Machine*, limitando as escolhas de tecnologia. Além disso, não garante o desacoplamento REST uma vez que é *statefull* e trafega objetos entre as aplicações, podendo sofrer com problemas de serialização. Além disso, as chamadas são realizadas diretamente às implementações dos clientes, sistemas distribuídos cujas interfaces expõem até o tipo de variável dos parâmetros.

JMS, por sua vez, é uma API criada pela Sun Microsystems destinada a plataforma Java, conforme indicado no seu próprio nome. Embora algumas implementações JMS também utilizem outros protocolos e permitam a comunicação com outras linguagens, não é este o caso do account-collector, que utiliza como sistema de mensageria o JBossMQ, que não provê tal independência de tecnologia. No entanto, ambas outras premissas são atendidas já que o JMS garante o maior desacoplamento dentre as outras tecnologias presente no legado.

Quanto a integração por banco de dados, a avaliação já foi realizada no próprio capítulo de conceitos, seção 2.2.2.1, expondo as deficiências e as razões pelas quais esse tipo de integração não atende a nenhuma dos aspectos abordados. Dessa forma, das quatro tecnologias implementadas no account-collector, somente a comunicação via REST satisfaz as três premissas de integração com base nas boas práticas de arquiteturas de micro-serviços. A tabela 4.2 apresenta estes dados de forma resumida:

Tecnologias	**Novas entregas sem impacto aos clientes**	**API's independentes de tecnologia**	**Interfaces sem detalhes da implementação**
REST	Sim	Sim	Sim
RMI	Não	Não	Não
JMS - JBossMQ	Sim	Não	Sim
Banco de dados	Não	Não	Não

Tabela 4.2: *Resumo das premissas de integração atendidas pelas tecnologias utilizadas*

4.2.2.3 Modernização de software

Em relação a modernização, somente uma única questão pode ser avaliada neste primeiro momento.

Havia indícios suficientes para que fosse necessário um processo de modernização?

Os índicios presentes na litetura da necessidade de modernização de um sistema foram expostos na seção 2.3. Dentre eles, todos são perceptíveis no account-collector:

- **Longos ciclos de desenvolvimento**: Tendem a serem maiores do que em outras aplicações mais simples devido principalmente ao alto custo associado ao procedimento de novas entregas. Além disso, o prazo para desenvolvimento de funcionalidades e a realização de testes de regressão manuais tendem a consumir um longo período de tempo.
- **Não há separação dos módulos e há um alto acoplamento no código**: O alto acoplamento e a falta de padrões de design foram contextualiados na seção 3.1.4.
- **Tecnologias utilizadas ultrapassadas**: As tecnologias são antigas e não garantem mais a mesma produtividade das equipes de desenvolvimento, conforme contextualidado na seção 3.1.
- **Baixa escalabilidade**: Embora o legado seja de baixa escalabilidade, este nunca foi um problema que gerasse impacto para o time de desenvolvimento ou negócio, uma vez que performance nunca foi uma grande preocupação. Entretanto, em relação ao custo, soluções de *cloud* mais baratas deixam de ser utilizadas devido ao aspecto monolítico da aplicação.
- **Dificuldade de entendimento do sistema por parte de novos desenvolvedores**: Dentre todos os itens, este apresenta maior relevância para as equipes de desenvolvimento e consequentemente para os interesses da companhia. A curva de aprendizagem do sistema é muito alta, o que dificulta a atuação de desenvolvedores com menos experiência. Além disso, mesmo desenvolvedores já familiarizados com o sistema têm dificuldades e gastam tempo desnecessário na identificação de fluxos. Isto se dá devido a alta complexidade do sistema, falta de padrões de *design*, falta de documentação, excesso de funcionalidades e a utilização de tecnologias defasadas.

Além dos itens acima, outros índicios foram determinantes para que a modernização tivesse realmente início. Dentre eles, a necessidade de constante evolução inerente aos sistemas de pagamentos, que a todo momento precisam se adaptar à mudanças na lei, aproveitar oportunidades de taxas menores, atender uma crescente gama de meios de pagamento e de preocupações relacionadas a segurança da informação, dentre outros itens, sempre em um curto espaço de tempo. Se o account-collector fosse um legado que não sofresse evoluções, provavelmente o interesse pela sua modernização seria menor.

Outro item importante refere-se a manutenibilidade do legado, ou seja, o custo relacionado a sua susbsistência, continuidade em produção. Uma inferência desse custo pode ser obtida através da quantidade de incidentes e problemas que ocorrem em produção e que exigem análise manual, tomando tempo recorrentemente das equipes de desenvolvimento. Neste quisito, mais uma vez o account-collector acena para a necessidade de modernização, já que muitos desses incidentes são intrínsecos à arquitetura do sistema e às tecnologias legadas, uma dívida técnica cujo juros continuará a ser cobrado.

4.3 Marco um

O marco um representa o conjunto das primeiras entregas significativas do processo de modernização do account-collector, na data de abril de 2016, oito meses após o início do estudo de caso. Essa primeira iteração têm um escopo abrangente pois compreende a criação de três novos componentes que foram desenvolvidos concomitantemente: billing-business-process-model(BBPM), transaction-manager(TM) e billing-business-rule(BBR).

4.3.1 Contextualização

As dificuldades enfrentadas pelas equipes de desenvolvimento na manutenção e evolução do sistema legado, apresentadas no decorrer deste pesquisa, foram o grande fator motivador das implementações realizadas neste marco. Buscou-se então, através da criação dos novos componentes, atingir justamente as questões julgadas como mais críticas e relevantes referentes ao cerne do account-collector: o acúmulo de responsabilidades e a falta de visibilidade das regras de negócio.

Dentre todos os fluxos de cobrança existentes no account-collector, superficialmente citados seção de domínios 3.1.3, o fluxo de recorrência foi o escolhido para ser o primeiro a ser migrado. Este fluxo é responsável por efetuar as cobranças de cartão de crédito de todas as assinaturas de todos os produtos do grupo UOL, um valor financeiro muito significativo. A figura 4.5 representa de forma simplificada o fluxo deste processo, descrito sequencialmente abaixo:

1. O primeiro passo corresponde ao início do fluxo, que ocorre de maneira periódica, uma vez por dia, quando um *batch* do account-collector é inicializado. Este *batch* executa diversas consultas SQL em um banco de dados de integração, onde outros sistemas de ERP se conectam. São estes sistemas que geram as entradas nas tabelas e que fornecem os dados iniciais para o processamento da cobrança.

2. Após a consulta, para cada cobrança a ser processada, o account-collector faz uma chamada RMI para si próprio enviando os dados da cobrança.

3. Após o processamento interno dentro do account-collector, baseado em diversas regras de negócio implementadas, uma nova transação é criada no domínio do legado e enviada ao PagSeguro, cujos detalhes da comunicação não blocante foram expostos durante a apresentação da arquitetura do legado, seção 3.1.4.

4. Após o retorno assíncrono do PagSeguro, os dados da transação são processados e persistidos no domínio do account-collector.

5. Na sequência, o account-collector inicia a comunicação do processamento da transação para diversas aplicações clientes. Uma dessas formas é através de um sistema de mensageria assíncrono, um tópico JMS no qual clientes estão inscritos.

6. O último passo ocorre de forma assíncrona, quando o AC, através de mais um *batch* periódico, realiza a atualização dos status das cobranças nas tabelas de integração.

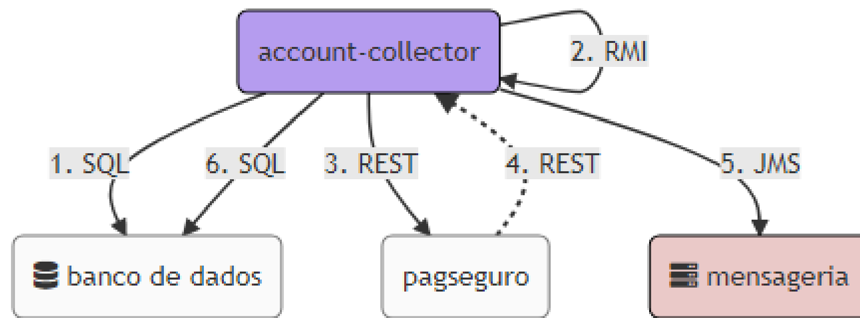

Figura 4.5: *Fluxo de recorrência antes do marco um*

Após o desenvolvimento realizado durante este marco, o processo de cobrança de recorrência sofreu diversas alterações, que podem ser observadas na figura 4.6:

1. O batch permanece no account-collector, mas de forma simplificada. Ele apenas consulta as tabelas de integração e envia as identificações das cobranças que devem ser processadas para outro sistema, o recém criado BBPM. Este sistema, que será detalhado na próxima seção, passa a gerenciar os status temporários das tabelas de integração e prover o controle da concorrência em detrimento do código do *batch*.

2. O passo seguinte mostra justamente a integração com o BBPM através de uma interface REST, onde um processo de cobrança é criado. Neste ponto, o retorno é síncrono.

3. Este passo corresponde a integração com o sistema BBR, outro nova aplicação criada neste marco responsável por encapsular regras específicas de meios de pagamento de toda área. Também via REST de forma síncrona.

4. O passo seguinte consiste na chamada REST do BBPM ao TM, terceiro e último sistema criado neste marco. É ele o novo responsável pelo domínio de transações, anteriormente restrito ao account-collector. No entanto, embora a comunicação no passo 4 seja síncrona, seu resultado é somente uma promessa de execução. A resposta definitiva será dada de forma assíncrona num passo futuro.

5. O TM, por motivos que serão descritos no decorrer desta seção, comunica-se então ao account-collector através de um novo serviço REST criado. Este serviço criado no account-collector executa os mesmos passos que eram efetuados no passo 2 da figura 4.5, só que via REST ao invés de RMI.

6. A integração com o PagSeguro ocorre da mesma maneira assíncrona.

7. Após o retorno assíncrono do PagSeguro, os dados da transação são processados e persistidos no domínio do account-collector.

8. Na sequência, o account-collector inicia a comunicação do processamento da transação para diversas aplicações clientes. Uma dessas formas é através de um sistema de mensageria assíncrono, um tópico JMS no qual clientes estão inscritos.

9. O account-collector é um dos clientes do seu próprio tópico JMS. Ele consome as mensagens publicadas anteriormente e filtra aquelas pertinentes a serem notificadas ao TM.

10. O account-collector, identificando tais transações iniciadas pelo TM, efetua a chamada REST com os dados da transação, um *callback* assíncrono.

11. O TM, por sua vez, processa e consolida o status da transação no seu domínio, decidindo se a mesma se encontra paga ou não, além de outros estados intermediários. Após isso, notifica ao BBPM via REST o resultado do processamento daquela transação.

12. No último passo o BBPM atualiza o status nas tabelas de integração com o ERP, sem que seja necessária a atuação do *batch* de retorno anterior utilizado anteriormente.

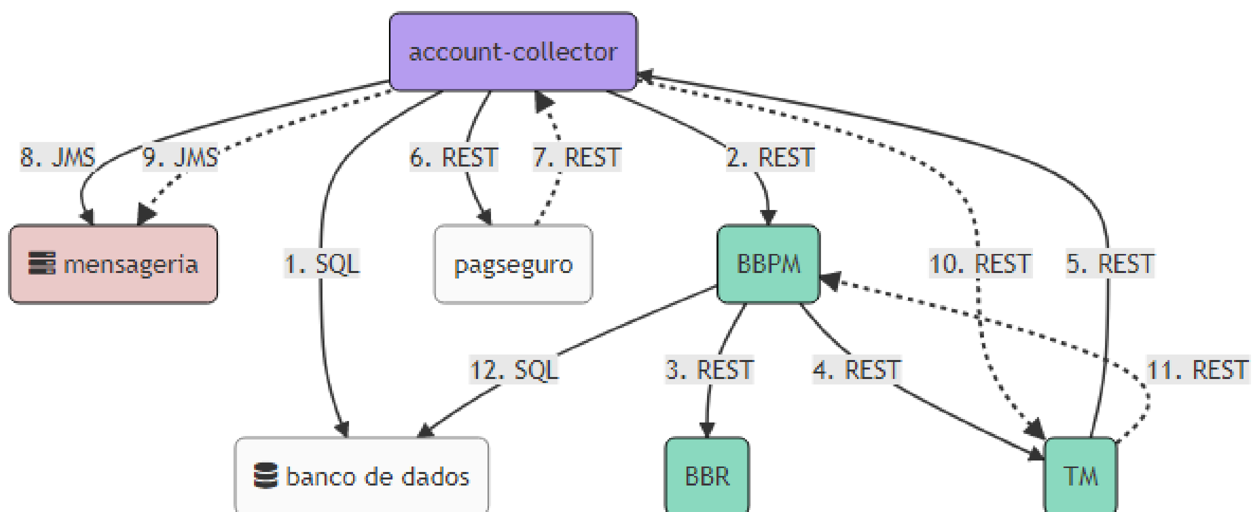

Figura 4.6: *Fluxo de recorrência após marco um*

Além de todas essas alterações que afetaram o processo de cobrança de recorrência, alguns outros detalhes implementados no account-collector serão discutidos na seção a seguir.

4.3.2 Implementações realizadas

A seção anterior contextualizou o cenário das mudanças e estimulou o surgimento de diversas dúvidas que serão esclarecidas e aprofundadas na explicação de cada novo componente criado, além das alterações realizadas no próprio legado.

BBPM: Billing Business Process Model

O desenvolvimento do novo sistema billing-business-process-model, apelidado por sua sigla BBPM, foi a primeira grande decisão arquitetural tomada. Esse sistema não tem um domínio específico, mas sim a função de orquestrar diversos fluxos da plataforma corporativa através da composição de serviços, padrão de integração apresentado na seção 2.2.2.3. Para isto, utiliza notação BPMN [CT12], *Business Process Model and Notation*, uma espécie de fluxograma estendido com funcionalidades específicas capaz de descrever as regras de um processo de negócio e integrá-las ao código-fonte do sistema, tornando-o transparente e entendível aos desenvolvedores e às áreas de negócio através de uma representação gráfica como o exemplo da figura 4.7.

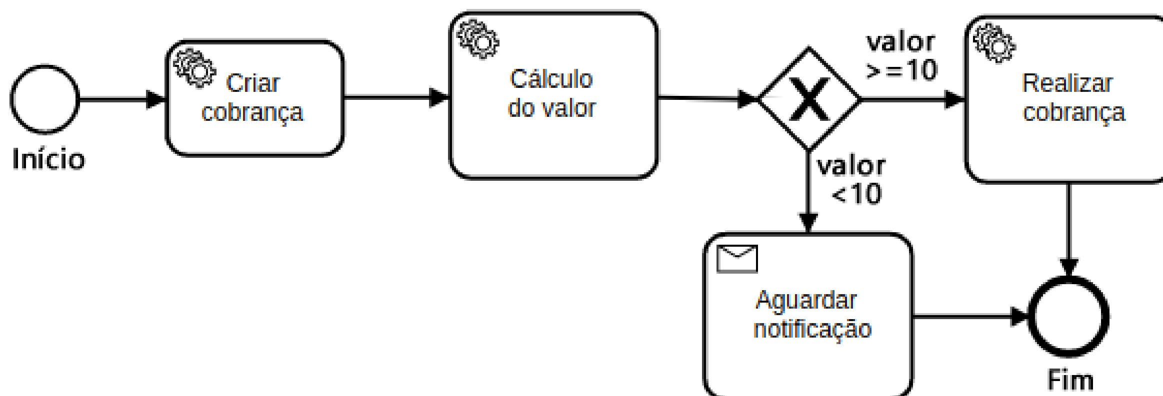

Figura 4.7: *Representação gráfica de um processo BPMN*

A notação BPMN é um somente uma parte da metodologia BPM, *Business Process Management*, ou em português, Gerenciamento de Processos de Negócio, que abrange de forma bem ampla métodos de descoberta, análise, modelagem, monitoração, otimização e automação dos processos de negócio de toda a empresa [Jes14]. Direcionada à questão da modelagem destes processos, a notação BPMN utiliza diversos elementos gráficos na construção dos fluxos, sendo os mais comuns os objetos de eventos, atividades, *gateways* e conectores, representados separadamente na figura 4.8 e detalhados abaixo com base na especificação de BPMN do grupo OMG [CT12]:

- **Eventos**: Compreendem estados lógicos do processo de negócio, sendo os mais comuns o estado inicial e final. Também podem existir estados intermediários que não representam nenhuma ação. No exemplo da figura 4.7, são representados pelos círculos, sendo o evento inicial o primeiro elemento a esquerda e o evento final o mais a direita.

- **Atividades**: Termo genérico que representa as ações executadas pelo processo, podendo iniciar um outro subprocesso ou apenas uma tarefa. Existem diversos tipos de tarefas, sendo as mais utilizadas as tarefas de serviço e de recebimento. A tarefa de serviço faz a ligação entre a notação BPMN e o código-fonte do sistema, que efetivamente efetuará a ação. Já a tarefa de recebimento é assíncrona e tem função única de aguardar uma mensagem antes de prosseguir com o fluxo. São representadas por retângulos arredondados como os quatro presentes na figura 4.7, sendo três deles de tarefas de serviço, cujo símbolo é um engrenagem, e apenas uma de recebimento, cujo símbolo é uma carta.

- **Gateways**: São os elementos responsáveis pela lógica do fluxo do processo. Os mais comuns são os *gateways* exclusivos e paralelos. Enquanto o exclusivo particiona o processo em diferentes caminhos dando sequência ao fluxo em apenas um deles, o *gateway* paralelo permite o paralelismo dentro de um mesmo processo e controla a concorrência e finalização de ambos caminhos. O exemplo 4.7 contém apenas um *gateway exclusivo*, no formato de um losango, que toma uma decisão de qual caminho seguir baseado numa váriavel de contexto chamada 'valor'.

- **Conectores**: Conectores ou conectores de sequência de fluxo são os elementos gráficos representados por uma linha sólida com uma seta preenchida no sentido do destino, similar a uma

flecha, cuja função é conectar todos os demais elementos BPMN e estabelecer a ordem de execução e a dependência dos mesmos. Na figura 4.7 por exemplo, a atividade 'Criar cobrança' será sempre a primeira executada, seguida por 'calcular do valor', que só será iniciada após o término da primeira.

Figura 4.8: *Elementros gráficos da notação BPMN*

Para implementar os conceitos dos processos BPM foi utilizado a plataforma Activiti BPMN [ACT]. Este *framework* contém uma base de dados própria e provê uma série de abstrações capazes de realizar a orquestração dos fluxos desenhados através da notação BPMN e integrá-los ao código-fonte da aplicação. O Activiti faz esse papel central de maneira desacoplada, separando-se as questões ligadas à infraestrutura BPM das implementações das funcionalidades do sistema.

A figura 4.9 corresponde a imagem gerada a partir da notação BPMN de parte do processo de negócio criado e entregue em produção ao final do marco um, juntamente com o novo sistema, o BBPM. Este processo faz parte do fluxo de cobranças das assinaturas do UOL conforme descrito na seção de contextualização, figura 4.6. Ele herdou algumas das responsabilidades que antes estavam escondidas dentro dos *batches* do account-collector e passou a orquestrar os novos componentes criados. Devido ao tamanho do fluxo, algumas tarefas do ínicio e do fim removidas da imagem.

TM: Transaction Manager

O transaction-manager, ou TM, foi outra aplicação criada durante o marco um. Intrínseco ao seu próprio nome, a responsabilidade do novo micro-serviço é de gerenciar transações de cobrança, ou seja, fazer parte do que hoje é responsabilidade do account-collector. Dessa forma, o sistema deve prover todos os serviços necessários relacionados às transações, desde simples consultas do status atual até a criação de novas transações de cartão de crédito ou boleto online, contemplando todo o domínio.

Entretanto, para que seu escopo não seja tão grande e para que haja um encapsulamento adequado, não é papel do transaction-manager conhecer os detalhes da integração com os *gateways* de cobrança que possam vir a ser utilizados. Como descrito na seção 3.1, o account-collector comunica-se ao *gateway* do PagSeguro para o processamento das cobranças, numa integração rica de detalhes

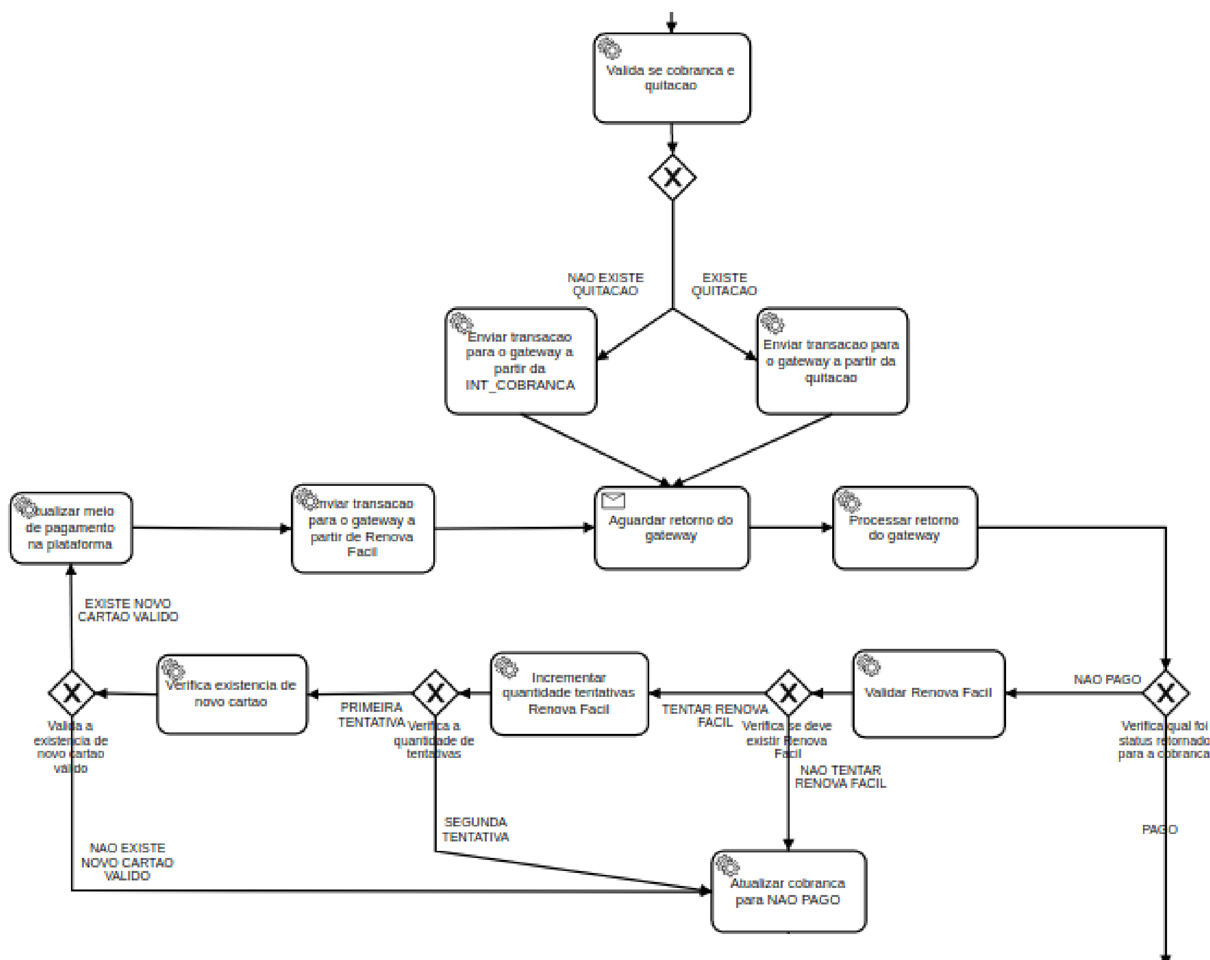

Figura 4.9: *Processo BPMN de cobranças de recorrência*

e que requer uma série de configurações. Para evitar que o mesmo erro fosse cometido, optou-se por não realizar essa integração no transaction-manager mas em outro micro-serviço cuja responsabilidade seria somente conhecer os percalços da integração e garantir a consistência e ordem dos eventos.

Dentre tantos fatores disruptivos que foram discutidos durante todo o desenvolvimento realizado no marco um, o mais significativo foi sem dúvida o de como realizar as cobranças sem utilizar o account-collector. Somente durante a construção do transaction-manager constatou-se que diversos detalhes do fluxo de cobrança ainda eram obscuros às equipes de desenvolvimento devido a alta complexidade do código legado. Percebeu-se então que não seria possível naquele momento transacionar de outra forma que não fosse através do account-collector, sem que grandes impactos fossem gerados. Isto porque, além de ainda não existir o micro-serviço de integração com o PagSeguro, diversos outros papéis exercidos pelo legado também não haviam sido implementados em outra aplicação. Dessa forma, ao invés de ser suprimido, o account-collector adquiriu mais um cliente, o próprio transaction-manager.

Do ponto de vista de arquitetura, o transaction-manager foi construído para atender de forma não blocante um alto número de requisições. Para isto, ele funciona de forma totalmente assíncrona com mecanismos de *callback* e a utiliza filas internas através do sistema de mensageria RabbitMQ [VW12], provendo atomicidade, resiliência e escalabilidade a todo processo. A figura 4.10 apresenta minuciosamente o fluxo de criação de uma transação no transaction-manager, exibindo inclusive o

retorno síncrono das chamadas, conforme detalhado a seguir:

1. Primeiramente o cliente do transaction-manager, BBPM por exemplo, faz uma chamada REST enviando os dados da cobrança e uma URL de *callback* U1.
2. Em seguida, os dados enviados são validados, salvos no domínio da aplicação e enviados para uma fila de processamento via protocolo AMQP.
3. O passo três representa o retorno síncrono da chamada realizada no passo 1. Dessa forma fica evidênciado que, apesar da característica não blocante, o sistema garante resiliência através da publicação na fila interna realizada anteriormente.
4. Na sequência, o próprio transaction-manager consome da sua fila de processamento para prosseguir com o andamento da cobrança.
5. O processamento é realizado através de um *gateway* de pagamentos responsável pela comunicação com os sistemas de adquirência, neste estágio do marco um, ainda o account-collector. Neste ponto, uma URL de *callback* U2 é passada como parâmetro.
6. O *gateway* retorna uma resposta de forma síncrona confirmando que a transação foi inciada. Importante citar que caso haja um erro nessa chamada, devido a algum problema na rede ou indisponibilidade do *gateway*, o transaction-manager efetuará retentativas em períodos de tempo crescentes.
7. O gateway, ao finalizar o processamento da transação, retorna de maneira assíncrona na URL de *callback* U2 o resultado daquela cobrança com o devido detalhamento.
8. A seguir, o transaction-manager interpreta aquele resultado e atualiza seu dados, postando a seguir na fila de notificação.
9. Na sequência, retorna sucesso ao *gateway* de forma síncrona, liberando aquela conexão.
10. O transaction-manager consome da fila de notificação e prepara os dados que serão enviados ao cliente.
11. O retorno final é dado somente no passo 11, quando o transaction-manager efetua o *callback* assíncrono da transação na URL U1, passada como parâmetro pelo cliente inicialmente.
12. O cliente retorna sincronamente que a mensagem foi recebida com sucesso.

Essas integrações providas pelo transaction-manager, para criação e consulta de transações, foram desenvolvidas a partir do padrão REST com os devidos recursos e utilizam dos verbos GET, PUT, POST.

BBR: Billing Business Rule

A aplicação billing-business-rule, ou BBR, foi criada numa tentativa de encapsular e desacoplar os dados relacionados às regras de negócio das novas aplicações. Basicamente, a aplicação é um conjunto de tabelas com dados de diversos domínios expostos através de uma interface REST,

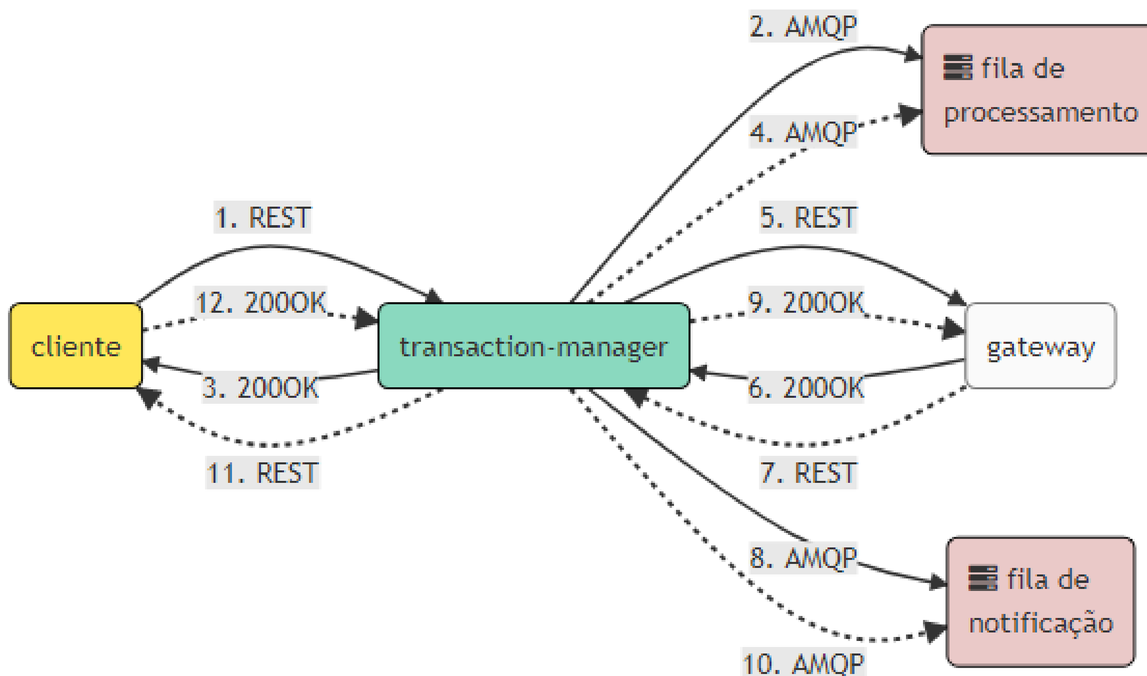

Figura 4.10: *Fluxo de criação de cobranças no transaction-manager*

funcionando de forma similar a um proxy entre a aplicação e o banco de dados, não implementando nenhuma regra em si.

O BBR foi construído baseado num outro sistema da área, o config-adm, responsável por conter as configurações de dados de diversos sistemas através de uma arquitetura muito simples, chave e valor, similar a um *hash*. O config-adm é utilizado para guardar informações de infraestrutura como URL's de serviços e tempo de timeout em requisições HTTP, além de dados relacionados à regras de negócio, como quantidade máxima de parcelas, valor mínimo de pagamento, etc. Entretanto, a utilização desse sistema escalou de forma não planejada e hoje várias regras de negócio estão descritas nele. O account-collector, por exemplo, utiliza esse sistema como máquina de estado ou de controle de fluxo em diversos casos, contendo inclusive os nomes das classes a ser invocadas. Este tipo de implementação, embora permita alterações de fluxos em tempo de execução, afeta a segurança, consistência e legibilidade do sistema.

Dessa forma, o BBR foi implementado sem chave e valor, mas sim com as regras fixas e estruturadas num banco de dados relacional, com a criação de diversas tabelas normalizadas. Além disso, para que fosse suportada alterações do domínio sem impacto aos clientes da aplicação, todos os recursos da API REST criada possuem versionamento.

AC: Account Collector

As implementações realizadas durante este marco relacionadas a alteração do fluxo de recorrência também exigiram que alterações fossem efetuadas no account-collector. Conforme apresentado na figura 4.6, que representa o fluxo de recorrência após marco um, o account-collector adquiriu mais um cliente, o recém criado transaction-manager. A figura 4.11 concentra-se especificamente nessa nova integração, descrita abaixo passo a passo:

1. O transaction-manager faz a chamada para um novo serviço REST criado no account-collector, com a função de processar a cobrança assíncronamente.

2. Integração assíncrona com o PagSeguro.

3. Após o retorno assíncrono do PagSeguro, os dados da transação são processados e persistidos no domínio do account-collector.

4. Na sequência, o account-collector publica o estado da cobrança no tópico JMS de notificação de pagamentos.

5. O account-collector é um dos clientes do seu próprio tópico JMS. Um novo MDB, *message-driven-bean*, foi criado dentro do próprio código legado para consumir as mensagens publicadas e filtrar aquelas pertinentes a serem notificadas ao TM.

6. O account-collector, identificando tais transações iniciadas pelo TM, efetua a chamada REST com os dados da transação, um *callback* assíncrono, último passo da nova integração.

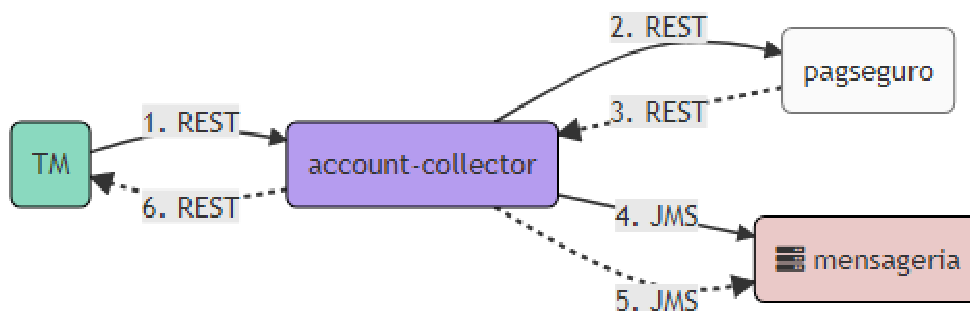

Figura 4.11: *Fluxo de recorrência com foco na integração AC e TM*

Além da criação desse novo serviço REST assíncrono, descrito pelos passos 1 e 6 acima, e do novo *subscriber* criado para alimentar o mecanismo de *callback*, apresentado no passo 5, algumas mudanças no fluxo proporcionaram também que parte do código-fonte do legado pudesse ser eliminado. Entretanto, devido a utilização da técnica de *feature-toggle* no processo de modernização, os códigos não foram fisicamente excluídos nesse primeiro momento, apenas deixaram de ser executados. Dentre eles:

- *Batch* de processamento de recorrência deixou de invocar via RMI o próprio account-collector para efetuar uma chamada REST ao BBPM, deixando de executar algumas regras de negócio migradas para o processo BBPM, como o controle dos estados das cobranças nas tabelas de integração.

- *Batch* de retorno de recorrência deixou de ser necessário, uma vez que a comunicação com as tabelas de integração foram migradas para o processo BBPM, que passará a persistir os dados de forma síncrona dentro do fluxo criado.

Outra importante alteração realizada no account-collector durante este marco, de forma independe às alterações do fluxo de recorrência, foi a migração das filas e tópicos JMS que faziam

parte do código do legado para um novo sistema, o billing-queue. Este sistema não contém código-fonte Java, apenas a configuração das instâncias dos servidores JBoss bem como a configuração dos tópicos e filas existentes através de arquivos XML. Para facilitar a visualização, a figura 4.3 que representa a arquitetura macro do account-collector não deixava claro que todo o sistema de mensageria também estava acoplado ao account-collector, isto é, sob o mesmo código e rodando no mesmo servidor JBoss. A figura 4.12 retrata essa mudança em específico.

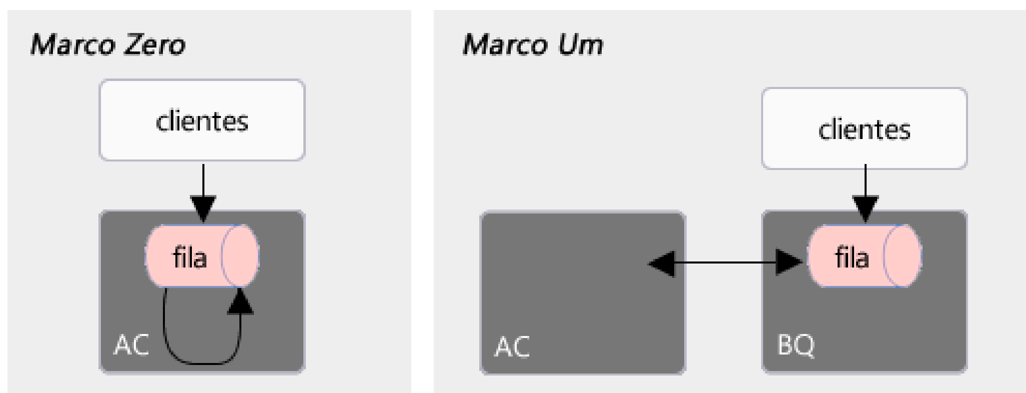

Figura 4.12: *Migração do sistema de mensageria*

4.3.3 Avaliação

Após aproximadamente 18 anos de existência, o marco um do processo de modernização sob estudo neste projeto de pesquisa, foi a primeira grande iniciativa que gerou entregas relevantes e proporcionou mudanças nos processos de cobranças antes totalmente orquestrados pelo account-collector. Entretanto, como evidenciado nas seções anteriores, o marco tomou grandes proporções e abrangeu a entrega não de somente uma, mas três novas aplicações. Caso essas aplicações fizessem sentido sozinhas, este marco poderia ser quebrado em três. Todavia, do ponto de vista de negócio, o fluxo só foi realmente entregue em produção com a finalização de todas as aplicações, o que justifica o marco único.

4.3.3.1 Dívida Técnica

Nesta seção será apresentada e analisada a variação da dívida técnica entre o marco zero e o marco um. Primeiramente serão apresentados individualmente os dados de cada aplicação, o legado account-collector e as novos sistemas criados nesta iteração: TM, BBR e BBPM. Ao final, a somatória de cada métrica de cada sistema, conforme especificado na seção de metodologia 3.2.2, será apresentada para fornecer os dados necessários para a avaliação do processo de modernização como um todo.

As métricas de dívida técnica do account-collector e as respectivas variações podem ser visualizadas na tabela 4.3. Destaca-se a expansão de mais de 5% do número de classes e de mais de 3% no número de linhas do sistema, que proporcionaram um aumento também da métrica *Technical Debt*, de 2,04% ou 30 dias. Estes números são o reflexo das alterações efetuadas para atender a nova integração com o transaction-manager e da não remoção física dos códigos-fontes dos *batches* simplificados. Em relação a migração do sistema de mensageria para um sistema apartado, embora

seja de grande relevância arquitetural, não promoveu alterações signficativas no código-fonte do sistema e dessa forma pouco afetou positivamente a dívida técnica extraída.

Métricas	**Marco zero**	**Marco um**	**Variação**
Technical Debt	1.472 dias	1.502 dias	2,04%
Technical Debt Ratio	11,8%	11,7%	-0,85%
Issues	85.111	86.658	1,82%
Complexidade	29.155	29.935	2,68%
Duplicações	23,9%	23,8%	-0,42%
LOC	199.761	205.832	3,04%
Métodos	13.867	14.348	3,47%
Classes	2.114	2.228	5,39%
Média do tamanho das classes	94,49	92,38	-2,23%
Média do tamanho dos métodos	14,41	14,35	-0,42%
Média de métodos por classe	6,56	6,44	-1,83%
Frequência de issues	2,35	2,38	1,20%
Frequência de complexidade	6,85	6,88	0,35%

Tabela 4.3: *Marco um: dívida técnica do account-collector*

O transaction-manager, por sua vez, teve sua primeira medição extraída no marco um, cujos dados estão disponíveis na tabela 4.4. Em relação ao legado, percebe-se que os números obtidos são muito inferiores. O *Technical Debt*, por exemplo, é de apenas 64 contra 1.502 dias, uma relação de 4,3%. Entretanto, uma simples análise dos números absolutos não produz nenhum resultado significativo dado a diferente magnitude e proporção destes dois sistemas. Neste caso, a métrica *Technical Debt Ratio*, que representa o percentual de *Technical Debt* em relação ao tamanho do sistema, fornece uma base de comparação proporcional da dívida técnica entre duas aplicações diferentes. Assim, ao analisar-se a métrica *Technical Debt Ratio* entre TM e AC, 6% contra 11,7%, percebe-se que a relação da dívida técnica obtida entre os sistemas é de aproxidamente 50%. Esse mesmo valor pode ser confirmado pela quantidade de *issues*, cuja frequência de 4,70 no transaction-manager contra 2,38 no account-collector corresponde à mesma relação proporcional.

Métricas	**Marco zero**	**Marco um**	**Variação**
Technical Debt	-	64 dias	-
Technical Debt Ratio	-	6%	-
Issues	-	3.661	-
Complexidade	-	2.876	-
Duplicações	-	5,8%	-
LOC	-	17.224	-
Métodos	-	1.960	-
Classes	-	489	-
Média do tamanho das classes	-	35,22	-
Média do tamanho dos métodos	-	8,79	-
Média de métodos por classe	-	4,01	-
Frequência de issues	-	4,70	-
Frequência de complexidade	-	5,99	-

Tabela 4.4: *Marco um: dívida técnica do transaction-manager*

Outra revelação significativa pode ser extraída dos números a respeito do tamanho do sistema. Com 489 classes contra 2.228, o transaction-manager tem quase $\frac{1}{4}$ do número de classes do account-collector, 21,9%. Todavia, a proporção muda drasticamente ao analisarmos o número de métodos, 1.960 contra 14.348, apenas 13,7%, ou o número de linhas de código, 17.224 contra 205.832, 8,3%, indicando que nem todas as métricas apresentam um comportamento proporcional entre os sistemas.

Dessa forma, a média do tamanho das classes no transaction-manager é de apenas 35 linhas, contra 92 linhas no sistema legado, uma proporção de 38% apenas. Isso porque além das classes apresentarem menos métodos, 4,01 contra 6,56, na média, os métodos também são menores, 8 linhas por método contra 14. A grande discrepância nestes dados corrobora com o prévio levantamento realizado na seção 3.1, a respeito do tamanho das classes e funções do legado, com características próximas de linguagens procedurais e pouco uso da orientação a objetos.

Quanto ao BBR, billing-business-rule, os valores das métricas de dívida técnica estão dispostas na tabela 4.5. O tamanho do sistema e a porcentagem de dívida técnica assemelha-se ao dados do transaction-manager. Entretanto, nota-se um grande valor na métrica de duplicações de código, 28,2% no BBR contra apenas 5,8% no TM. Isto porque, como citado na seção anterior, o BBR é apenas uma fachada para o banco de dados e apresenta diversas classes similares relacionadas a API REST do sistema. Um termo utilizado com frequência neste caso é *boilerplate code* [BOI], linhas de código-fonte utilizadas para a sintaxe da linguagem ou *framework* mas que não representam nenhuma função para a lógica do sistema.

Métricas	**Marco zero**	**Marco um**	**Variação**
Technical Debt	-	65 dias	-
Technical Debt Ratio	-	5,7%	-
Issues	-	3.608	-
Complexidade	-	3.435	-
Duplicações	-	28,2%	-
LOC	-	18.622	-
Métodos	-	1.977	-
Classes	-	434	-
Média do tamanho das classes	-	42,91	-
Média do tamanho dos métodos	-	9,42	-
Média de métodos por classe	-	4,56	-
Frequência de issues	-	5,16	-
Frequência de complexidade	-	5,42	-

Tabela 4.5: *Marco um: dívida técnica do billing-business-rule*

Já as métricas do BBPM, billing-business-process-model, tem suas métricas apresentadas na tabela 4.6. O código-fonte considerado para a extração dos dados corresponde somente as classes que compõem a infraestrutura geral do sistema e aquelas relacionadas ao processo de recorrência criado, conceituado ao longo deste marco. Embora não seja um micro-serviço, e sim um orquestrador, os números são muito aproximados com os do TM e BBR, inclusive o tamanho do sistema, 16.868 linhas. O valor que mais se diferencia é justamente o *Technical Debt* e *Technical Debt Ratio*, de somente 2,9%, metade do número do transaction-manager e billing-business-rule.

Por fim, apresenta-se na tabela 4.7 as métricas obtidas do processo de modernização como um todo, levando em consideração a somatória de cada métrica de todos os sistemas envolvidos no

Métricas	Marco zero	Marco um	Variação
Technical Debt	-	30 dias	-
Technical Debt Ratio	-	2,9%	-
Issues	-	817	-
Complexidade	-	2.892	-
Duplicações	-	9,7%	-
LOC	-	16.868	-
Métodos	-	2.012	-
Classes	-	463	-
Média do tamanho das classes	-	36,43	-
Média do tamanho dos métodos	-	8,38	-
Média de métodos por classe	-	4,35	-
Frequência de issues	-	20,65	-
Frequência de complexidade	-	5,83	-

Tabela 4.6: *Marco um: dívida técnica do BBPM*

processo de modernização: AC, BBPM, TM e BBR. As métricas não absolutas como *Technical Debt Ratio* e Duplicações foram descartadas pois já são percentuais em relação ao tamanho do próprio sistema correspondente.

Métricas	Total marco zero	Total marco um	Variação
Technical Debt	1.472 dias	1.661 dias	12,84%
Issues	85.111	94.744	11,32%
Complexidade	29.155	39.138	34,24%
LOC	199.761	258.546	29,43%
Métodos	13.867	20.297	46,37%
Classes	2.114	3.614	70,96%

Tabela 4.7: *Marco um: dívida técnica do processo como um todo*

A adição de novos sistemas sem que houvesse a remoção de funcionalidades do legado fez com que os resultados em relação a dívida técnica de código-fonte não fossem positivos neste primeiro momento. O *Technical Debt* acumulado teve um aumento significativo de 12,84%, de 1.472 dias para 1.661 dias, conforme gráfico histórico apresentado na figura 4.13. Este valor foi impactado tanto pelo adição das dívidas de três novos sistemas, no valor 159 dias, quanto pelo aumento da dívida do próprio account-collector, correspondente em 30 dias. No total, 189 dias a mais de esforço foram adicionados para corrigir os problemas relacionados ao código-fonte dos sistemas envolvidos. No entanto, as métricas que mais se destacam são o número de classes e o número de linhas, que cresceram 70,96% e 29,43%, respectivamente. Esses grandes números comprovam o grande escopo de desenvolvimento realizado neste marco, com alterações demasiadamente grandes.

Embora todas as métricas obtidas tenham tido um grande aumento, o mesmo ocorreu de forma de não proporcional. O número de linhas, por exemplo, cresceu 2,4 vezes mais que o *Technical Debt*, indicando que o novo código-fonte implementado, maior parte relacionado às novas aplicações, tem uma melhor qualidade. A análise dos números revelam que, proporcionalmente, a dívida técnica nos novos sistemas é pequena quando comparada à do legado, uma redução de ao menos 50%. A comparação também trouxe informações importantes sobre a coesão das classes do sistema, que

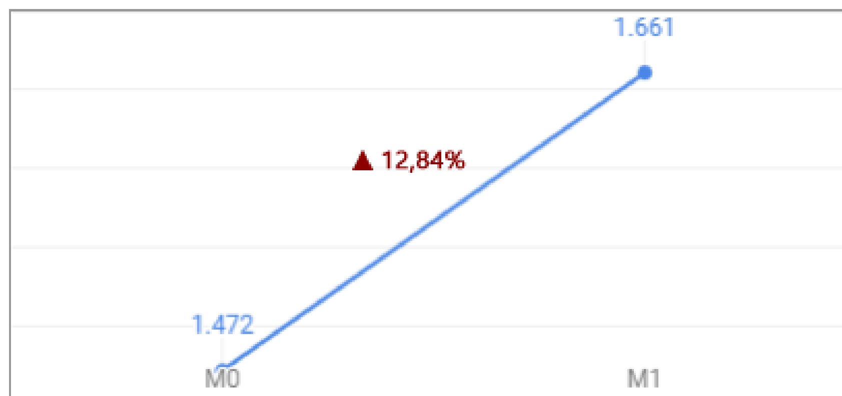

Figura 4.13: *Comportamento da dívida técnica somada de todos os sistemas no marco um*

tendem a ter menos de 40% do tamanho das classes do legado.

Dívida técnica além do código-fonte

Se por um lado as dívidas técnica de código-fonte aumentaram, as dívidas técnicas não relacionadas ao código-fonte tiveram vários pontos significados de melhoria neste primeiro marco:

- **Migracao mensageria**: Embora as alterações não sejam refletidas nas métricas de dívida técnica, já que não houve alteração de código-fonte Java, o impacto para a arquitetura do sistema é de grande relevância. O desacoplamento do account-collector às filas e tópicos permitiu que novas entregas em produção fossem realizadas sem impacto aos cliente consumidores da fila, colaborando para as entregas fora de janela, isto é, a qualquer hora do dia, não somente de madrugada para evitar a indisponibilidade à grande parte dos clientes.
- **Testes**: Tanto no BBPM como no TM e BBR, além de testes unitários, testes ágeis foram implementados para cada funcionalidade específica. Estes testes são caixa-preta e por isso conhecem somente as API's REST dos sistemas em teste, validando os fluxos do início ao fim. As dependências externas são todas substituídas por *mocks* que simulam o corportamento das mesmas, evitando a intermitência dos testes integrados que existia no account-collector, por exemplo.

Origem da dívida técnica

As implementações deste marco foram realizadas baseadas em decisões bem estabelecidas e sem que houvesse qualquer pressão externa ou urgência de negócio, o que descaracteriza a opção da dívida técnica ter sido adicionada intencionalmente, para acelerar o desenvolvimento. Em relação a outra váriavel do quadrante, apesar de não ter sido realizado o gerenciamento da dívida técnica nem sido utilizada qualquer ferramenta de controle de qualidade pelos times de desenvolvimento, as melhores decisões arquiteturais e tecnológicas foram tomadas na construção dos novos componentes, o que caracteriza a nova dívida adicionada como **prudente e não intencional**, uma vez que não houve intenção nem foram ignorados os princípios de desenvolvimento de software.

Reflexo da dívida técnica no dia-a-dia

Além do aumento do *Technical Debt*, o crescimento demasiado de todas as métricas relacionadas ao tamanho do código-fonte e a própria comparação das figuras 4.5 e 4.6, com os fluxos antes e depois deste marco, fornecem evidências que uma maior complexidade foi adicionada aos sistemas de cobrança durante este primeiro marco. Embora a arquitetura dos novos sistemas seja performática e evite a abertura de incidentes, o grande aumento de código-fonte trouxe inevitavelmente aumento ao custo de manutenção, principalmente no entendimento do fluxo como um todo e identificação de possíveis pontos de falha diante de algum problema sistêmico.

4.3.3.2 Arquitetura

As aplicações seguem o padrão de uma arquitetura de micro-serviços?

Neste marco três novas aplicações foram criadas e serão avaliadas em relação ao seu papel numa arquitetura de micro-serviços: BBPM, TM e BBR.

O BBPM, billing-business-process-model, foi criado com a intenção não de ser um micro-serviço, mas sim um orquestrador de serviços. Conforme conceituado na seção 2.2.2, o BBPM é o controlador central dos fluxos de negócio, responsável pela composição dos diversos serviços disponíveis. Dada a alta complexidade dos fluxos de negócio relacionados a cobrança, a adoção da orquestração trouxe maior visibilidade e controle dos fluxos, característica que não teria sido obtida com a coreografia. Dessa forma, o BBPM tem papel essencial na arquitetura de micro-serviços, seguindo os padrões da literatura a respeito da orquestração. Entretanto, alguns detalhes da implementação realizada podem ser considerados *bad smells* arquiteturais, dentre eles:

- Não se comunica somente através de protocolos leves e não intrusivos. Embora a grande maioria dos sistemas permita integração via REST, alguns serviços não foram construídos neste momento e o BBPM teve que integrar-se diretamente via banco de dados. A integração com os sistemas ERP é um exemplo.

- A escalabilidade do orquestrador também é um ponto negativo, uma vez que todas as regras de negócio de diferentes fluxos do sistema estão concentradas no mesmo componente. Além de não permitir que fluxos sejam escalados independentemente, problemas de concorrência entre os fluxos devido a momentos de alta demanda podem comprometer o tempo de resposta.

O TM, transaction-manager, micro-serviço criado para ser o sucessor do account-collector e responsável pelo domínio de cobranças, atende a todos os príncipios de uma arquitetura de micro-serviços, dentre eles vale destacar:

- Atende ao princípio de responsabilidade uma vez que não implementa nenhuma outra responsabilidade que não seja relacionada ao gerenciamento de transações, mantendo alta coesão.

- Possuí apenas uma dependência transitiva, o *gateway* responsável pela efetivação do pagamento, módulo corretamente mantido apartado que permite uma maior flexibilização do sistema e força a separação de duas questões distintas: o gerenciamento dos dados da transação e os detalhes de implementação de integração com os sistemas bancários. Assim, no futuro, o

transaction-manager pode inclusive controlar o fluxo de transações de outros tipos de pagamento, que podem utilizar outros *gateways* de saída.

- Toda integração é via REST, de forma não intrusiva e que permite alterações sem impactos ao clientes.
- Garante tolerância a falhas através da utilização das filas internas que promovem retentativas em tempos crescentes, sendo necessário intervenção manual apenas em casos relacionados a *bugs* nos sistemas envolvidos.
- Fácil de escalar e de entregar em produção. Embora a parte e infraestrura ainda não permita entrega contínua, a aplicação provê alta confiablidade e segurança através dos testes de regressão existentes que são executados de forma automática.

Já o BBR, billing-business-rule, terceira e última aplicação criada neste marco, embora seja relativamente pequeno, escalável e utilize protocolos de comunicação leve e não intrusivos, tem um papel questionável dentro da arquitetura implementada e não pode ser considerado um micro-serviço já que não possuí um domínio específico. Ao invés de ter uma responsabilidade única de negócio, contém somente dados relacionados às regras de cobrança, meios de pagamento e outras informações abrangentes da plataforma corporativa que deveriam estar contidas dentro dos micro-serviços responsáveis por aqueles domínios em específico, seguindo os padrões de *domain-driven-design* para evitar a anemicidade da arquitetura. Além disso, essa prática afetou o encapsulamento dos dados e até mesmo exigiu que as informações fossem duplicadas entre as tabelas do BBR e dos micro-serviços com o domínio específico. No caso do transaction-manager, por exemplo, dados relacionados ao mapeamento do estado das transações foram replicados já que as informações para decidir o estado final da transação estão contidas em tabelas do BBR e não do TM.

Quais paradigmas de comunicação foram adotados?

REST foi o principal padrão de comunicação adotado entre os novos sistemas criados. Tanto o BBPM, TM e BBR tem seus serviços expostos através de uma API RESTful, que mesmo baseando-se no paradigma **requisição-resposta**, apresenta um alto nível de desacoplamento e liberdade entre cliente e servidor. Além disso, tanto BBPM como TM, aproveitando da característica longa-vida de seus processos, foram implementados de maneira assíncrona através de um mecanismo de *callback*, conforme apresentado durante este marco. Já o BBR, capaz de prover respostas de forma muito rápida, foi implemetado sincronamente, de forma blocante.

Apesar de utilizarem o paradigma requisição-resposta, tanto BBPM como TM fazem uso de sistemas de mensageria internas, capazes de prover resiliência e tolerância a falhas numa arquitetura de micro-serviço, tanto devido a erros sistêmicos ou problemas de rede, funcionando como anteparos na propagação de erros como aconselhado na literatura [New15].

Outro aspecto determinante neste marco foi a escolha do modelo de **orquestração** em detrimento ao modelo de coreografia, conceitos apresentados na seção 2.2.2.3. O desenvolvimento e utilização da infraestrutura do BBPM foram questões de importante relevância dentro da companhia por prover uma maior visibilidade das regras de negócio e facilitar a monitoração e *troubleshooting* em caso de erro. Este preceito de transparência surgiu de forma antagônica aos anos de obscuridade

intrínsecos à arquitetura do account-collector. Dentre os ganhos ao optar-se por essa estrutura de orquestração, vale citar:

- **Documentação viva e sempre atualizada**: a notação BPMN é capaz de auto-documentar todo o fluxo do processo através do desenho gerado, favorecendo uma maior visibilidade das implementações das regras de negócio tanto para os desenvolvedores como para as outras áreas da companhia, contribuindo para a distribuição do conhecimento.
- **Monitorações avançadas**: a orquestração central através do BBPM permitiu a criação de uma monitoração única que garante o tempo de resposta das aplicações clientes. Para isto, para cada chamada externa pode ser configurada uma SLA, *service level agreement*, que determina o tempo máximo de resposta que dado serviço deve prover, além de outras condições. Na prática, as monitorações geram alertas descrevendo qual etapa ou task do processo foi comprometido e não respeitou a SLA, diminuindo consideravelmente o esforço e tempo na resolução de problemas.

Em relação ao legado, a alteração realizada neste marco fez uso do sistema de mensageria que já era utilizado pelo account-collector, paradigma **baseado em eventos**. Além de ser o publicador das mensagens no tópico, o AC também já possuia alguns MDB's inscritos para tomada de ações e outras comunicações com clientes externos. Neste marco, mais um MDB foi criado para satisfazer a chamada de *callback* para o transaction-manager. Embora mais código tenha sido adicionado ao legado, fazê-lo dessa maneira, com a utilização do tópico assíncrono, fez com que a comunicação ficasse apartada de todo o resto do código-fonte do account-collector e do fluxo de pagamentos em si, evitando um acoplamento ainda maior.

A comunicação entre as aplicações atende as premissas de integração?

Neste marco, somente o padrão de comunicação REST foi utilizado como forma de integração entre diferentes sistemas. Tanto a implementação no BBPM, TM e BBR suportam versionamento e expõem os serviços através de recursos específicos do domínio, satisfazendo as três premissas de integração, conforme conceituado na seção de avaliação realizada no marco zero, 4.2.2.2.

***Bad Smells* de micro-serviços**

Em relação aos *bad smells* ou anti-padrões de arquiteturas de micro-serviços apresentados na seção 2.2.5, alguns deles puderam ser identificados nas novas aplicações, conforme apresentado abaixo:

- **Intimidade inapropriada**: Anti-padrão de comunicação que pode ser encontrado no orquestrador BBPM quando integra-se aos sistemas ERP através de um banco de dados compartilhado durante o processo de recorrência de pagamentos.
- **Corte errado**: Anti-padrão de decompsição que pode ser identificado no BBR e expressa, em outras palavras, um dos motivos pelo qual essa aplicação não segue os padrões de uma arquitetura de micro-serviço. A implementação no BBR seguiu a decomposição baseada em camadas da aplicação, no caso, camada de dados, uma vez que o sistema é apenas uma

fachada para expor informações do banco de dados, sem que tenha uma funcionalidade de negócio específica.

Embora dependências transitivas possam ser consideradas anti-padrões, a dependência do transaction-manager em relação a um *gateway* de pagamentos não caracteriza um problema uma vez que não existe uma longa cadeia de transitividade síncrona, apenas um salto. Além disso, pensando na arquitetura final, com a criação de um *gateway* específico para a integração com o PagSeguro, somente o transaction-manager será o cliente dessa aplicação, que embora fisicamente separadas, estarão dentro do mesmo contexto.

4.3.3.3 Modernização de software

Nesta seção será apresentada a avaliação do marco segundo o ponto de vista do processo de modernização.

Estratégia de modernização adotada

As estratégias de modernização de sistemas foram conceituadas na seção 2.3.1 e divididas em três abordagens distintas: redesenvolvimento ou *Big-Bang*, *Strangler Application* ou migração e empacotamento de software. Destas abordagens, a técnica de *Strangler Application* é a que mais se proxima da estratégia utilizada neste primeiro momento já que foi migrada somente uma das capacidades de negócio do legado e também houve a criação infraestruturas com o intuito de atender aos próximos passos da modernização. Além disso, as alterações foram feitas para suportar a coexistência do sistema legado e das novas aplicações criadas, sempre tentando preservar e isolar o domínio dos novos micro-serviços. Entretanto, muitas das boas práticas presentes hoje na literatura em relação a *Strangler Application* não foram seguidas, como o critério de priorização dos módulos migrados, a granularidade da mudança, a utilização de técnicas de isolamento dos dados, dentre outros pontos que serão abordados nesta seção.

Critério para a priorização de migração dos módulos

A abordagem adotada na escolha do que seria primeiramente modernizado no sistema legado não seguiu as boas práticas propostas pela literatura atual [Ric16][Bro], que aconselha a migração de módulos mais simples e menos arraigados às funcionalidades centrais da aplicação, através de pequenos passos, conforme conceituado na seção 2.3.1.1. Embora tenha sido escolhida somente uma das capacidades de negócio do legado, o processamento de cobranças de recorrência, as alterações efetuadas tiveram uma maior magnitude e abrangeram inclusive o cerne do sistema legado, o seu domínio de transações, com a criação desnecessária do transaction-manager neste primeiro momento, duplicando parte das regras do account-collector. Essas decisões responsáveis por inflar este passo da modernização podem ter sido influenciadas por diversos fatores, dentre eles:

- A ausência de referências e estudos aprofundados relacionados a modernização de sistemas, principalmente à metodologia de *Strangler-application*, de certa forma recente e pouco explorada, ainda mais naquele momento.
- A desconfiança do time de desenvolvimento de que novas mudanças não seriam priorizadas

tão cedo novamente, fazendo-os optar por efetuar mudanças de grande impacto de uma só vez.

- A ausência de um planejamento e levantamento prévio detalhado sobre todo o domínio e responsabilidades do legado, que fez com que, por exemplo, o fluxo de cobranças de recorrência ainda utiliza-se o account-collector como *gateway* de cobrança.

Adição de novas funcionalidades no legado

Embora o costume de implementar todas as novas funcionalidades no account-collector tenha sido restringido, não foram apenas implementações relacionadas a modernização do sistema que foram entregues neste período do marco um. Uma nova demanda de negócio com alta prioridade fez com que uma funcionalidade já existente no account-collector, o pagamento via boleto, fosse estendida para também efetuar o envio da cobrança via e-mail aos clientes. Além disso, pequenas correções também foram necessárias neste período.

Entrega de valor para o negócio

Como citado anteriormente na avaliação arquitetural, a infraestutura do BBPM foi uma entrega significativa que propiciou a migração e construção de novos fluxos de negócio já na nova arquitetura. Somente neste período do marco um, cinco outros processos foram implementados no BBPM além do processo de recorrência, migrado do account-collector. Muitas das regras destes processos seriam possivelmente implementadas no próprio sistema legado caso o BBPM não tivesse sido criado. Dessa forma, embora o processo de recorrência não tenha entregue ganhos para o negócio neste primeiro momento, outras entregas importantes puderam ser realizadas devido ao processo de modernização.

Técnicas de decomposição de micro-serviços

As técnicas de decomposição de arquiteturas, apresentadas na seção de *Strangler Application*, 2.3.1.1, fazem parte de um assunto extenso, complexo e muitas vezes téorico da literatura, com poucas evidências e exemplos que auxiliem a interpretação de cada uma das técnicas e como aplicá-las em diferentes domínios de negócio.

As alterações realizadas neste marco corresponderam a migração das cobranças recorrentes, ou cobranças de assinaturas, escopo que pode ser considerado tanto um caso de uso do sistema como uma *business capability*, já que pode ser interpretado também como uma funcionalidade de negócio do sistema. Essa técnica de decomposição muitas vezes se assemelha a segmentação por contextos limitados ou subdomínios, em aplicações mais simples. Entretanto, no account-collector, é possível encontrar diversas capacidades de negócio que estão espalhadas entre diversos subdomínios da aplicação, uma matriz complexa de relacionamento que dificulta o processo de estrangulamento. A capacidade de cancelamento de tansações, por exemplo, é uma funcionalidade de negócio cujas regras estão espalhada entre diferentes partes de código do legado que correspondem a vários subdomínios: transações, saldo, impostos, etc.

Dessa forma, a opção decomposição por funcionalidades do sistema ou casos de uso pode não ser a melhor decisão em sistemas monolíticos com estas características devido ao alto acoplamento entre os subdomínios. A abordagem de identificar esses domínios e seus contextos, começando a

migração por aqueles menos coesos é uma estratégia muito menos complexa neste caso.

Técnicas de isolamento de dados do domínio

Das aplicações construídas, tanto BBPM como TM possuem algum tipo de integração com o sistema legado. O BBPM foi construído para ser o novo orquestrador e sua relação com o account-collector é de servidor, de forma unidirecional, ou seja, é o legado que depende do BBPM. A integração é realizada através de uma API REST disponibilizada pelo BBPM, já no seu próprio domínio. Dessa forma, cabe ao account-collector fazer a tradução dos seus dados legados para essa nova interface na hora de realizar a comunicação, evitando que o domínio do novo componente seja contaminado pelo domínio legado.

No caso do transaction-manager, a relação é inversa. É o TM que depende do account-collector, seu *gateway* de pagamentos. Para evitar que o domínio do novo sistema se acoplasse ao domínio legado, novas interfaces de entrada e saída foram criadas no account-collector com o intuito único de atender ao transaction-manager, conforme apresentado anteriormente neste marco. A interface de entrada foi construída através de um serviço REST que faz a tradução dos dados do TM para o domínio do legado, processando com sucesso as transações de cobrança. Já a interface de saída utilizou a tópico de mensageria, cujo MDB é responsável pelo mapeamento dos dados legados para os dados do transaction-manager, que serão enviados através de outra chamada REST. Dessa forma, além das interfaces fazerem a tradução dos dados desses dois mundos distintos, elas também padronizam a comunicação através do protocolo HTTP, evitando que o novo sistema utilize tecnologias intrusivas.

Essas interfaces criadas no account-collector funcionam exatamente como *anti-corruption layers* pois são fronteiras que isolam o novo sistema do domínio e dos detalhes de implementação e tecnologia do monolítico legado, permitindo que o novo serviço possa ser criado de maneira totalmente desacoplada. Além disso, essas interfaces foram criadas de maneira temporária e com o propósito único de atender ao transaction-manager enquanto perdurar a coexistência desses dois sistemas, não sendo expostas externamente para outros clientes. No entanto, alguns detalhes das implementações realizadas não encontram referências na literatura, tais como:

- Devido ao maior custo para criação de uma *anti-corruption layer* em um sistema apartado, essas interfaces foram criadas no código do próprio account-collector.
- Dependência entre novo serviço e legado não é de apenas dados do legado, mas sim de um serviço assíncrono.
- Utilização de sistema de mensageria como fonte de dados ao invés dos tradicionais bancos de dados.

Técnicas de migração e validação das alterações

Conforme constatado neste marco, é possível que em alguns passos do processo de modernização haja a coexistência de funcionalidades tanto no código do legado como nas novas aplicações, resultando em um aumento da quantidade total de código dos sistemas. Esse desfecho é uma consequência direta da utilização da técnica de *feature-toggle*, utilizada para garantir a convivência das

novas e velhas implementações. Esta técnica é responsável pelo roteamento do fluxo em tempo de execução, evitando-se uma virada abrupta dos sistemas e provendo mecanismo de fácil *rollback*, sem que seja necessário novas entregas de código. Esta técnica cautelosa vem sendo quase que obrigatoriamente aplicada em grandes mudanças, principalmente em sistemas de cobrança onde os impactos são sempre financeiros e muitas vezes imperceptíveis num curto espaço de tempo.

4.4 Marco dois

O marco dois corresponde ao segundo conjunto de entregas significativas do processo de modernização do account-collector, na data de setembro de 2016, seis meses após a finalização do marco um. Esta iteração teve um escopo pequeno, focado na criação de um novo micro-serviço, o creditcard-authorizer.

4.4.1 Contextualização

Com a migração do *gateway* de pagamento da plataforma UOL, conforme contextualizado no na seção de domínios 3.1.3 e na seção de arquitetura 3.1.4, todas as cobranças via cartão de crédito de transações online e recorrência passaram a ser realizadas através do PagSeguro ao invés da antiga plataforma Sitef. Este desenvolvimento, anterior ao início do estudo de caso, reduziu drasticamente a dependência do sistema externo Sitef, mas não o removeu completamente. Restou ainda o fluxo de validação de cartão de crédito, utilizado para confirmar a identidade e validade de um novo cartão antes que seja inserido na base de meio de pagamentos.

Este fluxo de validação de cartão é utilizado pelo checkout-online durante o processo de compra de um produto UOL e também pelos sistemas de SAC, serviços de atendimento ao cliente, que permitem ao cliente que seu meio de pagamento seja alterado conforme necessidade. A figura 4.14 representa a arquitetura e o fluxo de mensagens antes das mudanças realizadas neste marco:

1. Os clientes externos conectam-se via RMI no account-collector de forma síncrona e blocante.

2. O account-collector comunica-se com o Sitef através de bibliotecas externas realizando uma autorização no cartão do cliente. O resultado, sucesso ou não, é retornado ao account-collector, que interpreta os dados do vindos do Sitef e traduz para os clientes como meio de pagamento válido ou inválido.

Após as implementações realizadas no marco dois, a nova arquitetura removeu completamente o Sitef das dependências do account-collector e adicionou um novo micro-serviço, o creditcard-authorizer, conforme observado na figura 4.15, cujo fluxo está descrito abaixo:

1. Os clientes continuam a se conectar via RMI no AC de forma síncrona e blocante.

2. O account-collector conecta-se ao datafortress, responsável pela criptografia e sigilo dos dados de cartão de crédito conforme padrão PCI. Neste momento, os dados do cartão são transformados num token na notação hexadecimal. Vale citar que a integração com o datafortress, embora ainda não estivesse sido citada, já existia desde o momento em que o account-collector passou a transacionar via PagSeguro, que não trafega os dados puros de cartão de crédito.

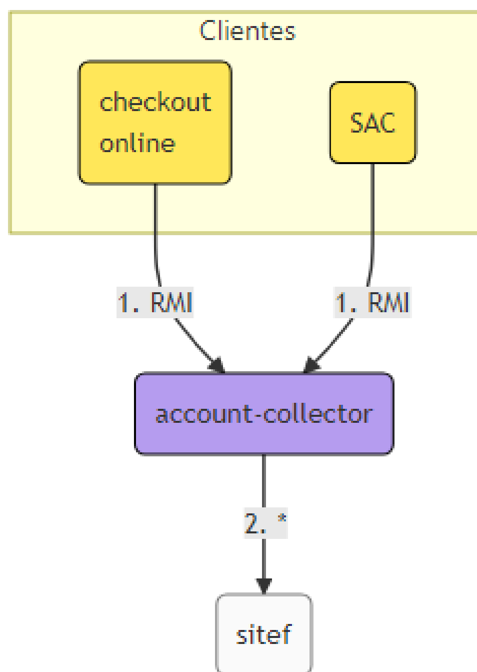

Figura 4.14: *Fluxo de validação de cartão antes do marco dois*

3. Em seguida, o cartão tokenizado gerado no passo 2 é enviado ao novo micro-serviço CCA via REST.

4. O CCA integra-se diretamente a um serviço específico do PagSeguro, desta vez de forma síncrona, para fazer a autorização e validação do cartão de crédito. Somente após o retorno do PagSeguro que os dados são retornados ao AC e na sequência para os clientes iniciais do passo 1.

4.4.2 Implementações realizadas

Neste marco as implementações foram restritas às mudanças no fluxo de validação no código do account-collector e na criação do novo micro-serviço creditcard-authorizer.

CCA: CreditCard Authorizer

O CCA tem como única responsabilidade a validação do meio de pagamento cartão de crédito. Para isto, ele abstrai toda a comunicação com um dos sistemas do PagSeguro responsável pela autorização, ou pré-autorização, de uma transação. Na pré-autorização os dados da transação são efetivamente enviados para a adquirente que valida os dados do cartão de crédito e o limite disponível junto ao banco e emissor do cartão. Quando a pré-autorização é bem sucedida, o valor da transação é reservado junto ao limite do cliente, normalmente por um período de 5 dias, mas sem que ocorra cobrança.

A pré-autorização é um mecanismo muito utilizado para evitar fraudes, entretanto, no caso do CCA, não se deseja fazer a captura do valor reservado, a cobrança em si. Desta forma, o CCA efetua também o cancelamento da pré-autorização, evitando que o limite do usuário fique bloqueado por

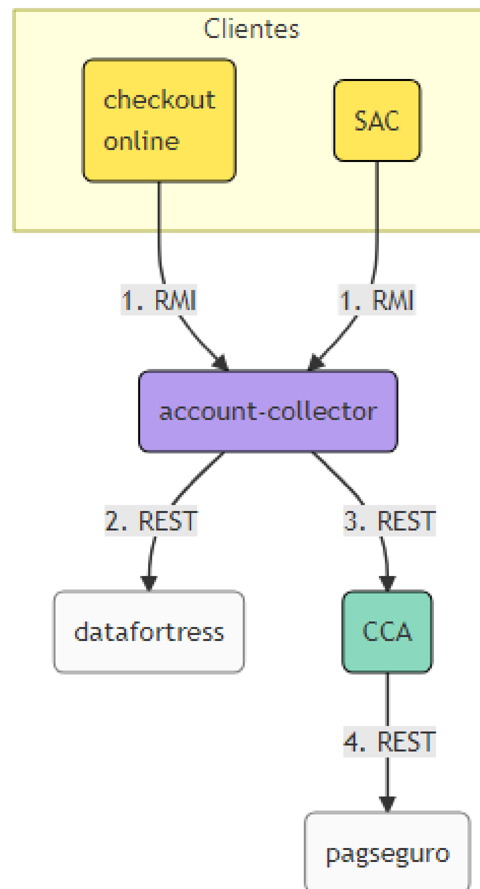

Figura 4.15: *Fluxo de validação de cartão após marco dois*

alguns dias. A arquitetura do CCA está representada na figura 4.16, que expõe inclusive os retornos síncronos das aplicações. Segue a descrição detalhada:

1. Cliente inicia validação do cartão de crédito passando dados tokenizados.
2. CCA faz chamada HTTPS ao PagSeguro para realizar a pré-autorização de um valor pré-estabelecido para todas as validações.
3. PagSeguro retorna o resultado da pré-autorização de forma síncrona.
4. CCA publicação os dados da pré-autorização numa fila de cancelamento de um sistema de mensageria RabbitMQ caso a autorização tenha sido realizada com sucesso.
5. Em seguida, a chamada síncrona efetuada no passo 1 é finalizada, sendo retornado o resultado da validação do cartão de crédito.
6. Paralelamente, o CCA consome as mensagens da fila de cancelamento, que correspondem à transações previamente pré-autorizadas com sucesso.
7. CCA efetua chamada ao PagSeguro para realizar o cancelamento da pré-autorização.
8. PagSeguro retorna o resultado do cancelamento da pré-autorização de forma síncrona.

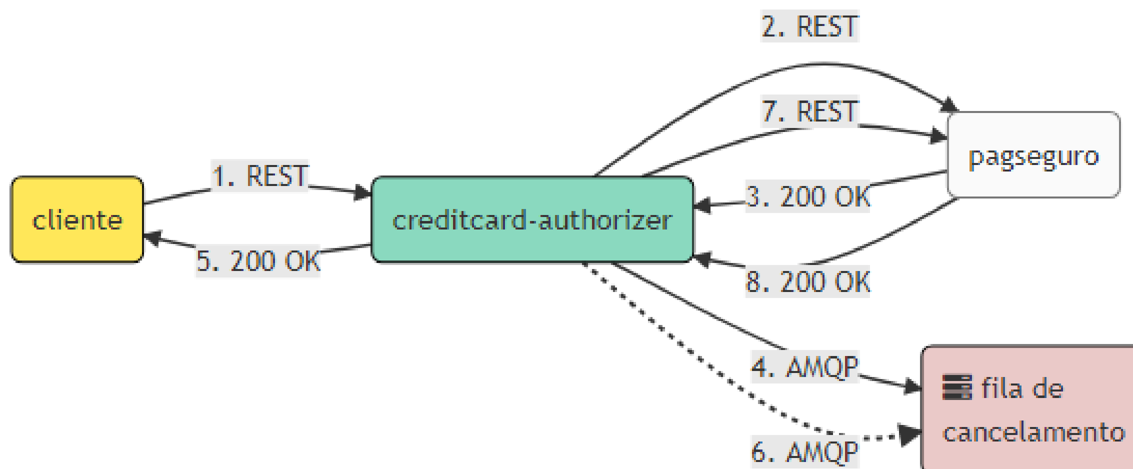

Figura 4.16: *Arquitetura do creditcard-authorizer*

A arquitetura do CCA foi construída tentando atender duas premissas: retorno síncrono de maneira rápida e resiliência aos erros do PagSeguro ou de rede. Desta forma, a utilização da fila de cancelamento garante que haverá retentativas até que efetivamente ocorra o cancelamento, mesmo em cenários de falha, além de contribuir para o processamento em paralelo, permitindo que a chamada síncrona do cliente seja respondida o mais rápido possível. O sistema de mensageria se encontra apartado da aplicação.

AC: Account Collector

O account-collector, por sua vez, também sofreu alterações neste marco. A substituição do Sitef pelo CCA no fluxo de validação pôs fim de forma definitiva à dependência deste sistema externo e todas as bibliotecas privadas que o AC utilizava como forma de integração. Além disso, pôde ser desligado o servidor que hospedava o sitef-client, parte da solução de comunicação com o Sitef. Entretanto, apesar da alteração do fluxo, boa parte do código legado relacionado ao Sitef não foi removido neste marco do account-collector devido ao alto acoplamento com outras partes do sistema.

4.4.3 Avaliação

Este marco teve um impacto positivo significativo, nem tanto pela diminuição da dívida técnica, mas pela independência completa do Sitef e toda infraestrutura extra aplicação que era necessária, como as máquinas servidoras do sitef-client. A remoção das bibliotecas proprietárias do Sitef, dependências externas do account-collector, também não são contempladas através das métricas de dívida técnica, o que pode mascarar os ganhos obtidos no processo.

Diferentemente do marco anterior, este marco teve um escopo reduzido e não trouxe impactos para a dívida técnica de código. Embora boa parte da lógica de validação de cartão de crédito tenha migrada do account-collector para o creditcard-authorizer, o legado manteve parte do fluxo já que as aplicações clientes, outros sistemas legados de outras áreas, não tiveram priorização neste momento para realizar a integração diretamente com o CCA.

4.4.3.1 Dívida Técnica

Nesta seção será analisada a variação da dívida técnica entre o marco um e o marco dois, primeiramente das aplicações envolvidas de forma individual e por fim do processo de modernização como um todo.

A dívida técnica do account-collector, apresentada na tabela 4.8, permaneceu estável durante esta iteração já que novamente, embora o fluxo tenha sido desviado do Sitef para o CCA, está operação foi lógica através de uma *feature-toggle* e o código-fonte não foi removido até a finalização deste marco. Não há nenhuma métrica que se destaque durante esse processo.

Métricas	**Marco um**	**Marco dois**	**Variação**
Technical Debt	1.502 dias	1.500 dias	-0,13%
Technical Debt Ratio	11,7%	11,6%	-0,85%
Issues	86.658	86.611	-0,05%
Complexidade	29.935	30.142	0,69%
Duplicações	23,8%	23,6%	-0,84%
LOC	205.832	206.338	0,25%
Métodos	14.348	14.453	0,73%
Classes	2.228	2.240	0,54%
Média do tamanho das classes	92,38	92,12	-0,29%
Média do tamanho dos métodos	14,35	14,28	-0,48%
Média de métodos por classe	6,44	6,45	0,19%
Frequência de issues	2,38	2,38	-
Frequência de complexidade	6,88	6,85	-0,44%

Tabela 4.8: *Marco dois: dívida técnica do account-collector*

O micro-serviço criado, creditcard-authorizer, apresenta *Technical Debt* de somente 7 dias, conforme os dados apresentados na tabela 4.9. Entretanto, devido ao pequeno escopo do micro-serviço, esse valor equivale a 4,5% de *Technical Debt Ratio*, não ficando muito abaixo do TM e BBR, que apresentam por volta de 6%, mas ainda bem distante do account-collector, que apresenta mais de 11%.

As métricas relacionadas ao tamanho da aplicação também corroboram o escopo reduzido e bem definido deste micro-serviço, uma vez que possui pouco mais de 2.500 linhas de código distribuídas em 71 classes, apenas 14% do tamanho do transaction-manager, outro micro-serviço, e somente 1,2% do tamanho do legado account-collector. Quanto às questões de coesão e acoplamento, este micro-serviço apresenta o mesmo padrão de distribuição em relação ao tamanho de classes e métodos dos outros micro-serviços criados, indicando uma possível faixa de referência para este tipo de arquitetura e tecnologia.

A dívida técnica total do processo de modernização, calculada a partir da somatória das métricas de todos sistema envolvidos: AC, BBPM, TM, BBR e agora também CCA, sofreu ligeiro aumento neste marco, 0,30%, cujos dados estão disponíveis na tabela 4.10. O histórico do *Technical Debt*, por sua vez, do marco zero ao marco dois, está representado na figura 4.17.

Dívida técnica além do código-fonte

A migração do fluxo de valição de cartão do Sitef para o CCA garantiu uma maior visibilidade das

Métricas	Marco um	Marco dois	Variação
Technical Debt	-	7 dias	-
Technical Debt Ratio	-	4,5%	-
Issues	-	412	-
Complexidade	-	461	-
Duplicações	-	1,0%	-
LOC	-	2.537	-
Métodos	-	307	-
Classes	-	71	-
Média do tamanho das classes	-	35,73	-
Média do tamanho dos métodos	-	8,26	-
Média de métodos por classe	-	4,32	-
Frequência de issues	-	6,16	-
Frequência de complexidade	-	5,50	-

Tabela 4.9: *Marco dois: dívida técnica do creditcard-authorizer*

Métricas	Total marco um	Total marco dois	Variação
Technical Debt	1.661 dias	1.666 dias	0,3%
Issues	94.744	95.109	0,39%
Complexidade	39.138	39.806	1,71%
LOC	258.546	261.589	1,18%
Métodos	20.297	20.709	2,03%
Classes	3.614	3.697	2,30%

Tabela 4.10: *Marco dois: dívida técnica do processo como um todo*

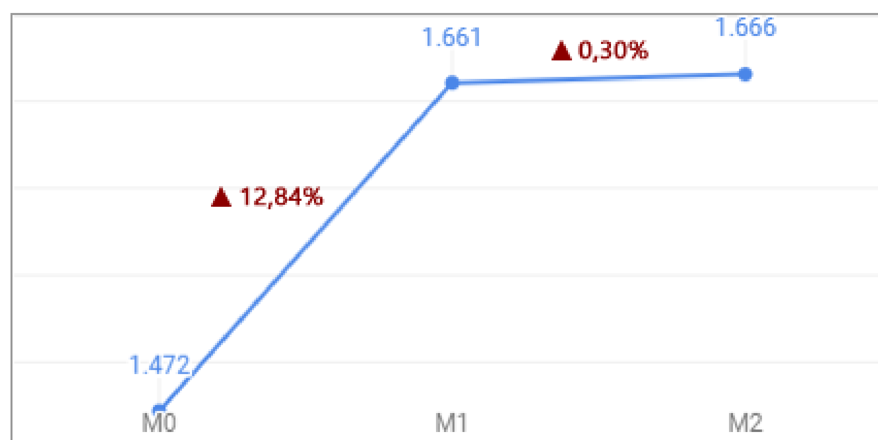

Figura 4.17: *Comportamento da dívida técnica somada de todos os sistemas no marco dois*

regras e da forma de comunicação, características antes restritas às implementações das bibliotecas externas. Além disso, essa indepedência é positiva do ponto de vista tecnológico pois desacopla o account-collector de utilizar a mesma versão de linguagem compatível com essas bibliotecas. Por último, facilita-se também a criação de testes e a evolução desse fluxo de forma independente da plataforma Sitef.

Reflexo da dívida técnica no dia-a-dia

A alteração realizada proporcionou alguns ganhos em relação a manutenibilidade dos sistemas, entre eles:

- Queda no número de incidentes relacionados a essa funcionalidade, que além de regulares eram de díficil *troubleshooting* devido a obscuridade da integração.
- Desligamento dos servidores Sitef que exigiam constante manutenção e atualização.

4.4.3.2 Arquitetura

Nesta seção os aspectos de arquitetura serão avaliados com base na literatura segundo as questões propostas na seção 3.2.2.

As aplicações seguem o padrão de uma arquitetura de micro-serviços?

O CCA, creditcard-autorizer, micro-serviço criado para ser encapsular a funcionalidade de validação de cartão de crédito, além de abstrair a comunicação com serviços de validação, possuí algumas entidades e regras específicas relacionadas ao seu próprio contexto, de forma completa e coesa, totalmente desacoplado das outras funcionalidades do account-collector. Além disso, alguns outros pontos podem ser citados em relação aos príncipios de microserviços:

- Atende ao princípio de responsabilidade única.
- A API REST exposta garante desacoplamento dos clientes.
- Garante tolerância a falhas através da utilização da filas interna de cancelamento, ou retorna um erro síncrono para o cliente.
- Fácil de escalar e de entregar em produção.
- Fácil de evoluir e substituição.

Quais paradigmas de comunicação foram adotados?

Novamente REST foi o estilo arquitetural de comunicação adotado, seguindo novamente o paradigma **requisição-resposta**. Entretanto, neste caso a requisição é sincrona e blocante, uma vez que o serviço de validação precisa responder da forma mais rápida possível para os sistemas clientes, já que esses efetuam operações em tempo real com o cliente.

Além disso, assim como o transaction-manager, o sistema utiliza uma fila de mensageria interna responsável pela resiliência dos cancelamentos e pela sua execução de forma paralela, liberando a conexão mais rapimente para os clientes. Dessa forma, apesar da integração com os sistemas extenos ser via REST, a utilização da mensageria faz com que o CCA seja um sistema hibrido, adotando internamente o paradigma **baseado em eventos**. Os eventos de cancelamento poderiam ser consumidos por outro sistema, por exemplo, de forma desacoplada, caso fizesse sentido essa separação de funções.

A comunicação entre as aplicações atende as premissas de integração?

Conforme conceituado na seção de avaliação realizada no marco zero, 4.2.2.2, a integração via REST satisfaz a todas essas premissas, diferentemente da comunicação que existia anteriormente com o Sitef. Essa integração por meio de bibliotecas externas, independemente da forma como a comunicação ocorre, atrela o cliente a uma linguagem específica na qual foi desenvolvida a biblioteca.

4.4.3.3 Modernização de software

Nesta seção será apresentada a avaliação do marco segundo o ponto de vista do processo de modernização.

Estratégia de modernização adotada

Neste marco, a migração ocorrida foi de uma funcionalidade pequena e apartada do domínio principal do sistema, escopo reduzido que facilitou que evitou que grandes alterações fossem realizadas no legado, conforme aconselhado pela literatura [Deh18][Ric16], um passo significativo do processo de *Strangler Application*.

Critério para a priorização de migração dos módulos

A migração deste módulo foi priorizada devido aos ganhos obtidos com a remoção dessa antiga dependência, o Sitef, sem que fosse necessário grande esforço ou aumento da complexidade do legado, ou seja, um ótimo custo benefício. Dessa forma, apresentou-se ganhos tanto para o negócio como para a maior qualidade do software.

Entrega de valor para o negócio

A entrega realizada neste marco possibilitou uma redução dos custos para companhia e também uma padronização em relação ao seus *gateways* de pagamento, encapsulados através dos sistemas do PagSeguro e não mais responsabilidade do plataforma-corporativa. Além disso, essa modificação permitirá em breve que todas as aplicações clientes façam as chamadas diretamente ao CCA utilizando cartão tokenizado ao invés dos dados puros, atendendo outra demanda de negócio, o *PCI Compliance*.

Técnicas de decomposição de micro-serviços

Diferentemente da análise realizada no marco um, na qual observou-se que a funcionalidade de cobranças de recorrência emaranhava-se em diversos subdomínios do legado, desta vez a relação é de um pra um, ou seja, a validação de cartão de crédito é tanto um subdomínio único como uma capacidade de negócio única, o que facilitou o processo de extração.

Técnicas de isolamento de dados do domínio

Não há compartilhamento de dados entre os domínios do AC e do CCA, somente uma dependência transitiva em relação ao serviço em si, no sentido do AC para o CCA, garantindo um total desacoplamento do novo micro-serviço criado, conforme aconselhado na literatura [Deh18].

Técnicas de migração e validação das alterações

Mais uma vez foi utilizado *feature-toggle* para uma migração segura e sem impactos ao clientes. Entretanto, assim como no marco um, não houve remoção do código legado após a completude da validação, apenas o desligamento do servidor Sitef.

Em relação aos testes, que antes não existiam devido a obscuridade da integração com o Sitef, novos foram criados tanto no account-collector como no creditcard-authorizer. No AC, foram criados testes unitários com a utilização de um *mock* do serviço exposto pelo CCA. Já no CCA, além de testes unitários, foram criados também testes ágeis caixas-pretas executados com a aplicação rodando através do *framework* Spring [SPR].

4.5 Marco três

O terceiro marco foi finalizado em fevereiro de 2017, cinco meses após o marco anterior. Neste período, uma grande reestruturação na organização dos times de desenvolvimento de software foi realizada com a divisão da área de *billing* e cadastro, conhecida como plataforma corporativa, em domínios específicos. Além disso, do ponto de vista de software, mais um micro-serviço foi criado, o pagseguro-gateway.

4.5.1 Contextualização

Diferentemente dos marcos anteriores, está iteração vai além do código-fonte das aplicações envolvidas no processo de modernização e envolve também mudanças organizacionais dentro de toda área da plataforma corporativa, questão intrínseca ao processo de modernização e construção dos micro-serviços, conforme apresentado em alinhamento organizacional na seção 2.2.1.

A plataforma corporativa, antes divididas entre duas grande áreas, *billing* e cadastro, unificou-se em uma grande área com times de menor escopo, focados em apenas um domínio da plataforma, por exemplo: vendas, usuário, assinatura, cobrança, integração-financeira, etc. Na prática, buscou-se uma melhor organização para evitar retrabalho de tarefas comuns e aumentar a capacidade e foco no desenvolvimento de grandes projetos, que muitas vezes envolviam essas duas áreas, cada uma com diferentes *backlogs* e priorização.

Os sistemas, novos ou legados, também foram incluídos nessa reestruturação. Cada aplicação foi delegada para o time cujo domínio parecia mais adequado, de acordo com as funcionalidades do sistema. Estes times tornaram-se responsáveis por todo desenvolvimento e correção de bugs daqueles sistemas, bem como a evolução do domínio em si.

Do ponto de vista de desenvolvimento de software, um novo micro-serviço foi criado, o pagseguro-gateway, PSGW, com o propósito de encapsular toda a comunicação com o *gateway* de pagamentos do pagseguro. No entanto, neste marco, apenas uma pequena fatia da integração com o PagSeguro foi migrada do account-collector, o fluxo de renova-fácil. Este fluxo tem como responsabilidade realizar a atualização automática dos cartões de créditos vencidos, cuja data de expiração foi ultrapassada, cenário muito comum em cobranças recorrentes de assinatura.

Antes das alterações realizadas neste marco, toda orquestração do renova-fácil era responsabilidade do account-collector, conforme exposto na figura 4.18. Tal figura é uma extensão da figura 4.6

apresentada no marco um, com a adição do fluxo renova-fácil, omitido anteriormente. Desta maneira, a maior parte do fluxo de mensagens exposto na figura 4.18 já foi detalhado, com exceção dos passos 5 e 6, descritos abaixo:

5. AC, após identificar que o pagamento foi negado devido ao cartão estar vencido, realiza uma consulta a outro serviço do sistema do PagSeguro para buscar o novo cartão válido.

6. Dado que exista um novo cartão de crédito para este cliente, o AC realiza uma chamada HTTPS ao sistema Cubus com os novos dados, realizando a atualização deste meio de pagamento. O sistema Cubus é o responsável pelos dados gerais dos clientes de toda a plataforma UOL.

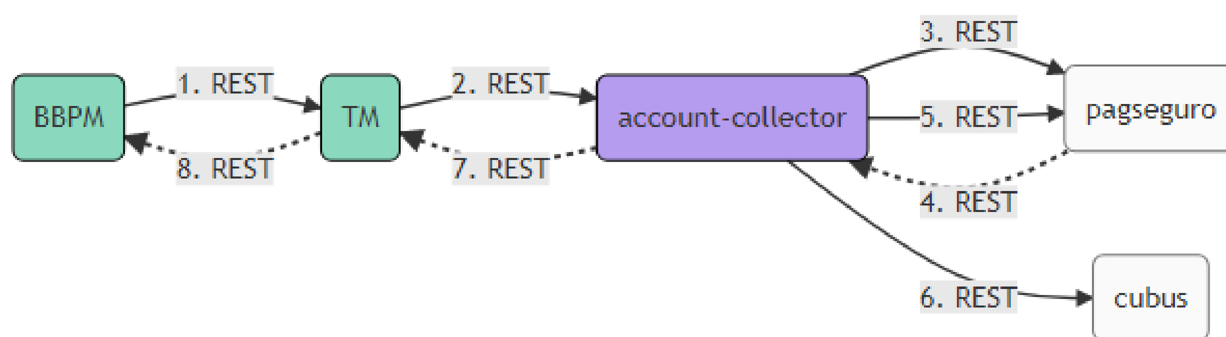

Figura 4.18: *Fluxo do renova-fácil antes do marco três*

Após a implementação realizada neste marco, o fluxo de renova-fácil foi migrado do account-collector para outras duas aplicações, o recém criado PSGW e o BBPM, orquestrador desenvolvido no marco um. O processo de negócio desenvolvido no BBPM para atender as cobranças de recorrência, descrito na seção 4.3.1, foi estendido para também orquestrar o renova-fácil, identificando transações cujo retorno indicam que o cartão já não é mais válido e realizando a consulta e atualização do mesmo. O PSGW, por sua vez, encapsula e abstrai os detalhes de implementação dos serviços do PagSeguro. A figura 4.19 detalha este novo comportamento pós migração do fluxo de renova-fácil:

1. BBPM inicia transação de pagamento através do TM.

2. TM inicia transação de pagamento através do AC.

3. AC realiza a transação de pagamento através do PagSeguro.

4. PagSeguro, de forma assíncrona, retorna um resultado final para a transação.

5. AC, de forma assíncrona, comunica o resultado ao TM.

6. TM, de forma assíncrona, comunica o resultado ao BBPM.

7. BBPM interpreta os dados do resultado da transação e caso identifique que a cobrança foi negada devido ao cartão de crédito ter expirado, realiza a consulta do novo cartão através do PSGW de forma síncrona.

8. PSGW realiza uma chamada síncrona ao serviço específico de renovação de cartão de crédito do PagSeguro, encapsulando detalhes de comunicação e segurança.

9. BBPM, após retorno síncrono do PSGW, atualiza o cartão de crédito do usuário caso um novo cartão válido exista, através do sistema Cubus.

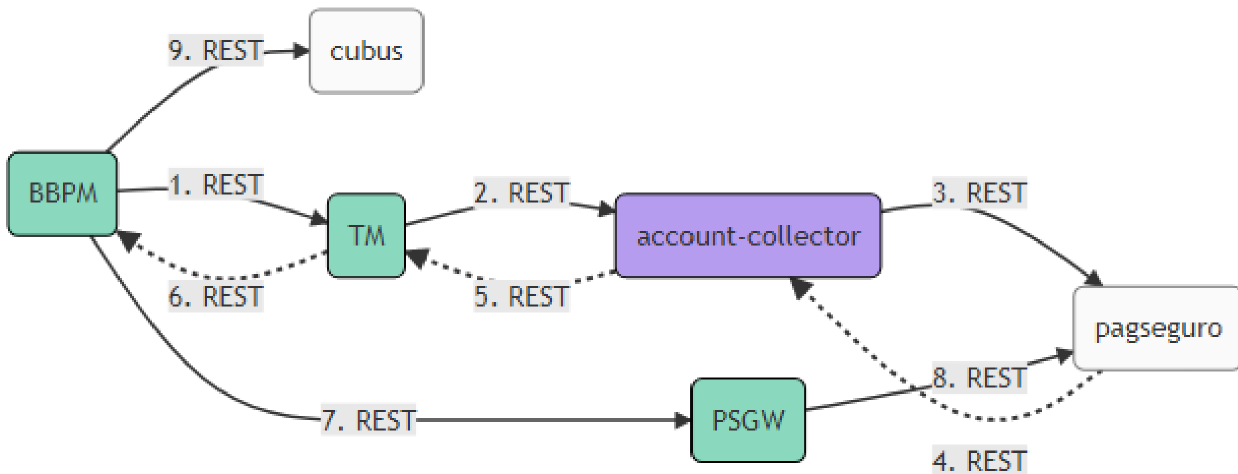

Figura 4.19: *Fluxo do renova-fácil depois do marco três*

Além dos passos anteriores que contribuíram ao processo de modernização do sistema legado, algumas implementações urgentes do ponto de vista de negócio tiveram que ser desenvolvidas no account-collector. A mais relevante foi o suporte ao boleto registrado [FEB], conforme nova norma estabelecida pela FEBRABAN, Federação Brasileira de Bancos. Para suportar está funcionalidade, além das alterações pontuais no account-collector, um novo micro-serviço foi criado para comunicar-se com os bancos através de arquivos EDI. Entretanto, por ser uma nova implementação, não existente anteriormente no legado, este micro-serviço não será envolvido no processo de modernização deste projeto de pesquisa.

4.5.2 Implementações realizadas

Neste marco as implementações foram restritas às mudanças no fluxo de renovação de cartão de crédito, conhecido como renova-fácil. Conforme contextualizado, a orquestração dessa funcionalidade foi migrada do account-collector para o BBPM, que passou a utilizar o novo micro-serviço criado, o pagseguro-gateway.

PSGW: PagSeguro Gateway

O pagseguro-gateway foi criado com o propósito de abstrair toda integração com o sistema de pagamentos do PagSeguro, cuja complexidade foi em parte já exposta nos marcos anteriores. Entretanto, dada a grande dimensão dessa alteração, optou-se neste marco pela migração de somente um dos serviços expostos pelo PagSeguro, o renova-fácil. Diferentemente do serviço assíncrono de cobrança, o serviço de renova-fácil é síncrono e de baixa complexidade, o que facilitou esta etapa do processo.

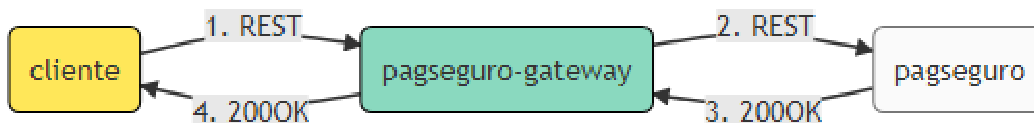

Figura 4.20: *Arquitetura do pagseguro-gateway*

Dessa forma, foi implementado um serviço REST no PSGW que atua como uma fachada para o serviço de renova-fácil do PagSeguro, de forma síncrona e blocante, retornando o resultado ao cliente somente após o retorno do PagSeguro, conforme exposto na figura 4.20.

BBPM: Billing Business Process Model

O BBPM, por sua vez, teve seu processo de recorrência de assinaturas estendido para orquestrar também o fluxo de renova-fácil. Dessa maneira, ao identificar que uma transação de pagamento foi negada e que o código de retorno corresponde a cartão vencido, um novo fluxo dentro do processo BPMN é iniciado para consultar, atualizar o meio de pagamento e efetuar a retentativa da cobrança. A figura 4.21 corresponde à imagem extraída de parte do processo BPMN alterado para suportar esta funcionalidade.

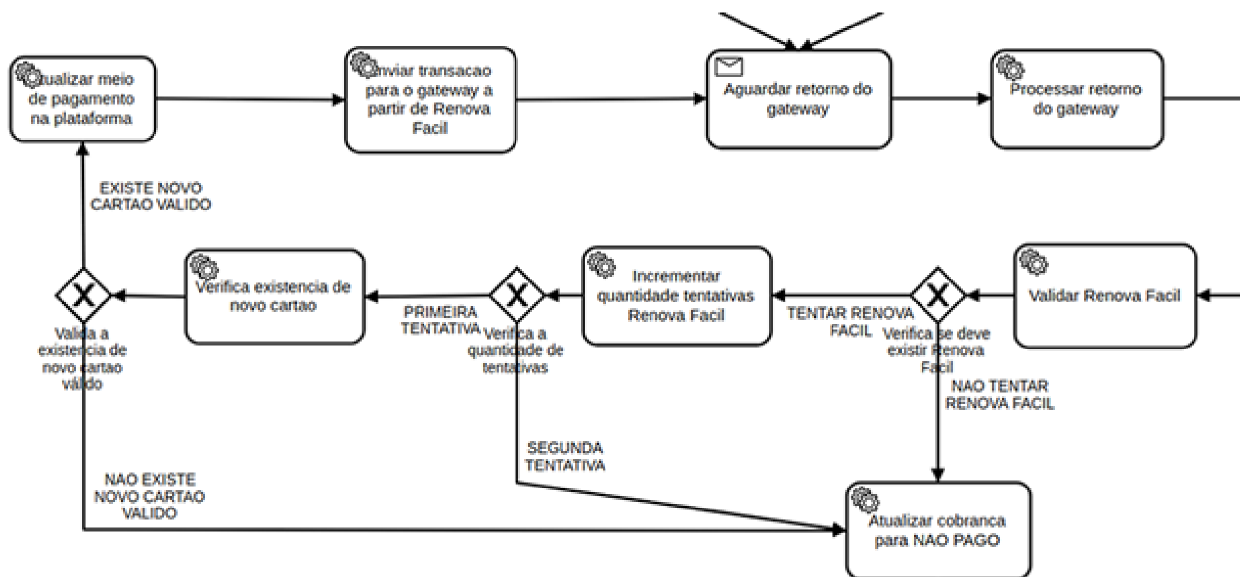

Figura 4.21: *Fluxo de renova-fácil dentro do processo de recorrência*

AC: Account Collector

Quanto ao monolítico, neste marco foi removido todo código-fonte relacionado a orquestração do fluxo de renova-fácil e a integração com o serviço relacionado do PagSeguro. Além disso, conforme citado na contextualização, algumas novas implementações tiveram que ser realizadas no código legado para atender a tempo a nova funcionalidade de boleto registrado.

4.5.3 Avaliação

O escopo reduzido e bem definido do marco três colaborou para que mais uma funcionalidade pudesse ser migrada do account-collector para os novos sistemas. Remover o renova-fácil, um fluxo desacoplado das demais funcionalidades do AC, foi uma decisão acertada no processo de modernização que se aproveitou de mudanças já realizadas anteriormente, como a criação do orquestrador BBPM, Além disso, a criação do micro-serviço PSGW contribuirá com os próximos passos da modernização, possibilitando a migração de todos os detalhes da comunicação com o PagSeguro.

Em relação a mudança estrutural da área plataforma corporativa, a reorganização das equipes seguiu exatamente os princípios da *Conway's Law*, forjados por Melvin Conway em 1968 [Con68]. Conforme citado na seção 2.2.1, em alinhamento organizacional, a divisão da área em domínios menores, com times pequenos e independentes favorece a criação de boas interfaces entre seus sistemas e um menor acoplamento. Na prática, facilita a adoção dos micro-serviços em detrimento aos sistemas monolíticos legados, como o account-collector.

Além disso, a nova organização parece trazer ganhos ainda maiores. A imersão do time em um único domínio favorece seu aprofundamento no mesmo e facilita a manutenção e evolução do domínio. A necessidade de lidar com os problemas diários relacionados a aquele escopo incentiva a equipe de desenvolvimento a adotar boas práticas e também corrigir na raiz os problemas já existentes no seu domínio, até mesmo, incentivando a modernização de seus próprios sistemas legados. Está noção de maior responsabilidade do time por todo ciclo de vida das aplicações, desde o desenvolvimento até a entrega final ao cliente, é conhecida na comunidade pelo termo *'you build you run it'*, tendência global seguida por grandes empresas como a Amazon [Gra06]. Com isto, esta mudança combate um dos problemas históricos da plataforma corporativa, a falta de governança das aplicações, que trouxe impactos visíveis principalmente nos sistemas mais antigos, como o account-collector.

Entretanto, apesar dos diversos ganhos obtidos, algumas dificuldades foram encontradas durante este realinhamento organizacional da área. Como qualquer mudança brusca, inicialmente houve certa resistência e levou certo tempo até que as equipes pudessem realmente entender seu papel dentro da plataforma corporativa como um todo. Além disso, a mudança é constante visto que os domínios definidos podem não ser os mais corretos ou estar em constante transformação, sendo necessário constante discussão para melhor adequação dos nomes e papéis de cada time. Outro ponto adverso da mudança foi a divisão dos sistemas monolíticos legados. Muitos destes não se comportam ou respeitam os limites recém criados pelos domínios, dificultando a governança deste legado, que pode ter mais de um time responsável.

4.5.3.1 Dívida Técnica

Nesta seção será analisada a variação da dívida técnica entre o marco dois e o marco três, resultado das implementações realizadas no account-collector, billing-business-process-rule e pagseguro-gateway.

A métrica *Technical Debt* do account-collector, apresentada na tabela 4.11, teve uma pequena queda de 0,80%, pois apesar da remoção do fluxo de renova-fácil novas implementações foram realizadas parar suportar o boleto registrado. A métrica mais impactada foi o número de classes, reduzidas em aproximadamente 3%.

Métricas	**Marco dois**	**Marco três**	**Variação**
Technical Debt	1.500 dias	1.488 dias	-0,80%
Technical Debt Ratio	11,6%	11,7%	0,86%
Issues	86.611	85.758	-0,98%
Complexidade	30.142	29.654	-1,62%
Duplicações	23,6%	23,9%	1,27%
LOC	206.338	203.861	-1,20%
Métodos	14.453	14.120	-2,30%
Classes	2.240	2.173	-2,99%
Média do tamanho das classes	92,12	93,82	1,85%
Média do tamanho dos métodos	14,28	14,44	1,13%
Média de métodos por classe	6,45	6,50	0,71%
Frequência de issues	2,38	2,38	-
Frequência de complexidade	6,85	6,87	0,43%

Tabela 4.11: *Marco três: dívida técnica do account-collector*

Já o pagseguro-gateway tem suas métricas disponíveis na tabela 4.12. Como mostram os números, o sistema tem praticamente zero de dívida técnica e apenas 15 classes, por volta de apenas 3% do tamanho de outro micro-serviço, o transaction-manager, com 489 classes. Sua relação a respeito do tamanho de classes e métodos é novamente similar aos outros micro-serviços criados, com média máxima de 40 linhas por classe. Entretanto, o domínio desse micro-serviço tende a crescer uma vez que apenas um dos serviços e o mais simples deles foi migrado do account-collector. Ao final da modernização, espera-se concentrar no pagseguro-gateway todos os detalhes da integração com o PagSeguro, abstraindo essa responsabilidade dos outros sitemas de forma coesa.

Métricas	**Marco dois**	**Marco três**	**Variação**
Technical Debt	-	2 dias	-
Technical Debt Ratio	-	5,1%	-
Issues	-	85	-
Complexidade	-	85	-
Duplicações	-	0%	-
LOC	-	511	-
Métodos	-	60	-
Classes	-	15	-
Média do tamanho das classes	-	34,07	-
Média do tamanho dos métodos	-	8,52	-
Média de métodos por classe	-	4,00	-
Frequência de issues	-	6,01	-
Frequência de complexidade	-	6,01	-

Tabela 4.12: *Marco três: dívida técnica do pagseguro-gateway*

Em relação ao BBPM, o *Technical Debt* se manteve constante uma vez que as alterações realizadas foram pontuais. Entretanto, houve uma grande uma diminuição do número de linhas e duplicações devido a um *refactoring* relacionado a arquitetura do sistema realizado durante este período, o que levou a um pequeno aumento no *Technical Debt Ratio*, conforme apresentado na tabela 4.13.

Métricas	**Marco zero**	**Marco um**	**Variação**
Technical Debt	30 dias	30 dias	-
Technical Debt Ratio	2,9%	3,1%	6,90%
Issues	817	848	3,79%
Complexidade	2.892	2.916	0,83%
Duplicações	9,7%	6,5%	-32,99%
LOC	16.868	15.596	-7,54%
Métodos	2.012	2.043	1,54%
Classes	463	448	-3,24%
Média do tamanho das classes	36,43	34,81	-4,45%
Média do tamanho dos métodos	8,38	7,63	-8,94%
Média de métodos por classe	4,35	4,56	4,94%
Frequência de issues	20,65	18,39	-10,92%
Frequência de complexidade	5,83	5,35	-8,30%

Tabela 4.13: *Marco três: dívida técnica do BBPM*

Por fim, apresenta-se na tabela 4.14 as métricas de dívida técnica do processo como um todo, com a adição de mais um novo micro-serviço criado. Dado o pequeno escopo das alterações, o impacto obtido foi pequeno porém o melhor resultado até agora entre entre todas iterações já que tanto o *Technical Debt* como o tamanho total do sistema diminuiu, ao invés de aumentar como nos marcos um e dois, conforme registrado históricamente na figura 4.22.

Métricas	**Total marco dois**	**Total marco três**	**Variação**
Technical Debt	1.666 dias	1.656 dias	-0,63%
Issues	95.109	94.341	-0,81%
Complexidade	39.806	39.403	-1,01%
LOC	261.589	258.351	-1,24%
Métodos	20.709	20.467	-1,17%
Classes	3.697	3.630	-1,81%

Tabela 4.14: *Marco três: dívida técnica do processo como um todo*

Figura 4.22: *Comportamento da dívida técnica somada de todos os sistemas no marco três*

Dívida técnica além do código-fonte

A figura 4.21, que corresponde a imagem extraída de um dos fluxos do processo de recorrência, evidência os ganhos relacionados a documentação e distribuição de conhecimento que podem ser obtidos através da utilização de técnicas BPM. É possivel, indenpendente de ter acesso ao código-fonte, identificar e entender facilmente as regras de negócio implementadas.

4.5.3.2 Arquitetura

Nesta seção os aspectos de arquitetura serão avaliados com base na literatura segundo as questões propostas na seção 3.2.2.

As aplicações seguem o padrão de uma arquitetura de micro-serviços?

O pagseguro-gateway, construído de maneira similar ao creditcard-authorizer, apresenta as mesmas características em relação às tecnologias de desenvolvimento e comunição, atendendo as boas práticas de arquiteturas de micro-serviços. Além disso, apresenta um papel coeso na arquitetura, com responsabilidade única de abstrair todos os detalhes da complexa integração com o PagSeguro. São esses detalhes do ponto de vista de comunicação e também de dados que fazem com que seja necessário a extração dessa integração para um micro-serviço apartado, que terá um subdomínio com entidades próprias relacionadas aos cada evento de troca de mensagem entre os sistemas, capaz de fazer a tradução da linguagem do PagSeguro para o contexto da plataforma corporativa, funcionando neste caso de modo semelhante a uma *anti-corruption layer*.

Quais paradigmas de comunicação foram adotados?

O padrão de comunicação adotado foi novamente REST e, por conseguinte, também o paradigma **requisição-resposta**. Embora não haja necessidade de um tempo de resposta curto nesse caso, optou-se pela comunicação síncrona devido a simplicidade da mesma e facilidade de desenvolvimento.

A comunicação entre as aplicações atende as premissas de integração?

A integração REST que antes já exista no account-collector foi migrada para o pagseguro-gateway, com as mesmas características, satisfendo a três premissas novamente.

4.5.3.3 Modernização de software

Nesta seção será apresentada a avaliação do marco segundo o ponto de vista do processo de modernização.

Estratégia de modernização adotada

Este marco da pesquisa correspondeu a mais um pequeno passo do processo de *Strangler Application*, que aproveitou-se das ações previamente realizadas no marco um e também facilitará os passos futuros da modernização através da infraestrutura criada para pagseguro-gateway.

Critério para a priorização de migração dos módulos

Vislumbrava-se, desde o final do marco um, a criação do pagseguro-gateway, um sistema com a finalidade única de abstrair a comunicação com o PagSeguro. Dessa forma, seria possível remover essas integrações do account-collector e migrar outras funcionalidades orquestradas também pelo legado, como o fluxo de renova-fácil. Esse fluxo, somente utilizado durante as cobranças de recorrência cujo processo já havia sido migrado para o BBPM, não havia sido alterado anteriormente justamente para evitar mais uma chamada do BBPM para o legado. Dessa forma, definiu-se o escopo desse passo da modernização a partir desses dois pontos, colaborando tanto na redução da complexidade do legado como nos próximos passos.

Adição de novas funcionalidades no legado

Neste marco, algumas alterações foram realizadas no legado para que suportasse boletos registrados, requisito de negócio imprescindível para que o UOL se adequasse as normas da FEBRABAN. Entretanto, a maior parte da implementação foi realizada em um novo micro-serviço, com regras específicas para tratar detalhes desse tipo de boleto. Esse comportamento confirma que a estratégia de evitar novas implementações no legado esta sendo corretamente seguida, sendo pequenos ajustes e criação de interfaces de comunicação com novos sistemas inevitáveis. Consequentemente, mesmo diante de incrementos de funcionalidades, a dívida técnica ainda foi reduzida neste marco.

Técnicas de decomposição de micro-serviços

As alterações realizadas durante este marco podem ser divididas em dois pontos: a migração do renova-fácil e a criação do PSGW, cada um com uma técnica diferente de decomposição. Na construção do PSGW foi considerado que o mesmo faz parte de um contexto limitado apartado do domínio central de transações, devendo ser desacoplado fisicamente das aplicações como AC e TM. Já o renova-fácil não é considerado um subdomínio, apenas uma funcionalidade do processo de recorrência cuja nova implementação divide-se entre a orquestração do BBPM e o serviço criado no PSGW.

Técnicas de migração e validação das alterações

Novamente, foram criados testes ágeis e unitários no novo micro-serviço e o chaveamento da migração realizado através de uma *feature-toggle* no account-collector. Dessa vez, diferentemente dos marco um e dois, a *feature-toggle* e o código não mais utilizado foram removidos logo após a validação.

4.6 Marco quatro

O quarto marco foi finalizado em outubro de 2017, nove meses após o marco três. Neste período, diferentemente do ocorrido desde o início do processo de modernização, não se buscou realizar a migração de funcionalidades do legado, mas sim avaliar as mudanças efetuadas até o momento e atuar em possíveis correções e *refactorings*.

4.6.1 Contextualização

O marco quatro simboliza a data de dois anos desde que as primeiras mudanças foram implementadas com o objetivo de modernizar o sistema de cobranças, estrangulando aos poucos o account-collector. Entretanto, como demonstrado através das métricas de dívida técnica apresentadas até o marco três, o número de linhas de código, classes e até mesmo o *Technical debt* acumulado só aumentou. Apesar das implementações realizadas não gerarem novos incidentes a complexidade de toda arquitetura também cresceu, como pôde ser notado já na figura 4.6, que esboça a arquitetura pós marco um. Esta maior complexidade foi percebida pelas equipes durante o desenvolvimento de novas histórias, uma vez que boa parte do código-fonte já migrado para os novos micro-serviços ainda não havia sido removido do sistema legado, confundindo o entendimento dos fluxos e o papel de cada aplicação dentro do domínio.

Dessa forma, o grande foco deste marco foi realizar *refactorings* para remover ou adaptar o código-fonte das implementações já não mais utilizadas no sistema account-collector, migradas nos marcos anteriores. Além disso, revisitou-se também os novos micro-serviços TM e BBR, cujos domínios poderiam ser simplificados.

4.6.2 Implementações realizadas

As implementações realizadas neste marco abrangem o sistema legado e outros dois micro-serviços, o transaction-manager e billing-business-rule. Não houve criação de novos micro-serviços.

AC: Account Collector

Durante o desenvolvimento dos últimos três marcos, apesar de parte do código-fonte migrado já ter sido removido do account-collector, como é o caso do fluxo de renova-fácil migrado no marco três, muitos outros ainda não foram totalmente excluídos. Os *batches* de comunicação com o ERP, expostos na figura 4.5 são exemplos de códigos ainda remanescentes. Estes *batches* foram modificados durante o marco um para utilizar a técnica de *feature-toggle*, tornando possível um *rollback* lógico do fluxo caso fosse identificado algum problema em produção. Entretanto, mesmo após a validação completa deste fluxo, a remoção deste e outros códigos legados ainda não haviam sido priorizadas.

O marco dois também deixou alguns vestígios pra trás. A criação do CCA pôs fim a dependência do sistema externo Sitef, juntamente com as bibliotecas proprietárias necessárias. Entretanto, um grande número de classes relacionadas a esta integração implementadas no account-collector também não foram removidas. Além da simples exclusão de diversas classes, muitas outras tiveram que ser readaptadas pois eram utilizadas em outros fluxos, fruto do alto acoplamento presente no legado.

No entanto, a remoção de código-fonte já não mais utilizado não ficou restrito aos fluxos já migrados para outras aplicações. Foi removido também todo o código-fonte relacionado aos testes integrados do account-collector, apresentados na seção 3.1.3. Estas centenas de testes intermitentes passaram a ser ignorados e não mais atualizados pelas equipes de desenvolvimento mesmo antes do início do processo de modernização, já que demoravam em torno de 24 horas rodar e não garantiam

nenhuma segurança na entrega. Entretanto, mesmo não sendo utilizados, esses testes tomavam tempo de desenvolvimento já que precisavam ser corrigidos mediante alterações nas classes do sistema para que pelo menos compilassem. Dessa forma, apesar da cobertura dos testes unitários não ser satisfatória, decidiu-se removê-los completamente.

TM: Transaction Manager

O transaction-manager, por sua vez, teve seu domínio reorganizado. A motivação deu-se a partir de um requisito de negócio relacionado ao parâmetro utilizado para determinar o estado final de uma transação. Na prática, deixou-se de utilizar um parâmetro que era específico para cada adquirente com o qual o PagSeguro poderia se conectar para que fosse utilizado um outro parâmetro cujo valor é normalizado, ou seja, não depende do adquirente no qual foi realizada a cobrança.

Essa alteração fez com que a equipe de desenvolvimento revisitasse o transaction-manager e constata-se a mistura de domínios entre o mesmo e o billing-business-rule. Detalhe abstraído até então, o TM, durante o processamento final de um transação, consultava o BBR para determinar o estado final da mesma, paga ou não paga, já que as regras relacionadas ao mapeamento de estados haviam sido criadas no BBR. Diante disso, optou-se por refatorar ambos domínios, sendo necessária a criação de duas novas tabelas no TM, que passaram a ser consultadas diretamente ao invés da chamada remota que era realizada para o BBR.

Além disso, aproveitou-se a alteração nas entidades de banco de dados para que parte do domínio fosse remodelado e simplificado, resultando na remoção ou normalização de outras tabelas ou campos específicos.

BBR: Billing Business Rule

No billing-business-rule houveram somente remoções. Uma vez identificado que parte de suas tabelas eram na verdade pertencentes a outro domínio, toda API REST criada, repositórios e demais camadas de código foram removidos, assim como sete tabelas, que foram substituídas pelas já existentes e recém criadas no transaction-manager.

4.6.3 Avaliação

O marco quatro teve um escopo totalmente diferente dos marcos anteriores, focado não na migração de funcionalidades do legado mas sim na revisão do processo como um todo ocorrido até o momento. As mudanças realizadas ocorreram de forma natural e contribuíram para finalizar algumas partes do processo de modernização iniciados em outros marcos. A remoção do código responsável pelos *batches* do processo de recorrência, por exemplo, fluxo descrito e migrado no marco 1, seção 4.3.2, veio a ocorrer somente dois anos depois. Estes e outros códigos legados que poderiam ter sido removidos ou adaptados anteriormente foram mantidos por todo esse período, afetando a dívida técnica do processo de modernização como um todo e consequentemente o juros dela resultante. Os números expostos até agora na seção de dívida técnica corroboram com está análise. A métrica *Technical Debt*, por exemplo, passou de 1.472 dias para 1.656, do marco zero ao marco três, um aumento acumulado de mais de 12%.

Algumas razões influenciaram para que esses *refactorings* ou simples remoções de código tardassem para acontecer, dentre elas:

1. A necessidade da utilização de *feature-toggle* durantes essas migrações, que possibilitou uma maior convivência entre os códigos novos e legados. Está técnica, que tem caráter provisório, deve ser utilizada com cautela e através de um acompanhamento que permita sua remoção o mais rápido possível, juntamente com o código não mais utilizado, assim que a validação tiver sido realizada.

2. A alta complexidade e alto acoplamento do código legado também dificultaram a remoção de parte dos códigos. Muitas classes são compartilhadas por mais de um fluxo, embora estes não possuam qualquer semelhança. Este problema foi constatado na remoção do fluxo do Sitef, onde percebeu-se que parte das classes legadas estavam sendo reutilizadas, de forma incorreta, também no novo fluxo do PagSeguro. A necessidade de uma maior *refactoring* desencoraja que essas mudanças ocorram.

3. O alto risco devido a falta de testes e incerteza sobre a utilização ou não de certos fluxos e classes devido a alta complexidade também trazem receio aos desenvolvedores durante estas mudanças. Muitas vezes opta-se pela cautela para evitar que problemas sejam gerados em razão do *refactoring*.

4. A atuação de diferentes equipes num mesmo domínio e no processo de modernização também colaborou para que parte do conhecimento não fosse compartilhado. As mudanças de alinhamento organizacional e divisão de domínios realizadas no marco três, por sua vez, vieram justamente para combater este problema e já trouxeram resultados positivos como pode ser observado neste marco.

Desta forma, as alterações implementadas neste marco foram cruciais para o bom andamento do processo de modernização. Não havia mais espaço para que fossem postergadas tais mudanças que contribuíram de forma significativa para a redução da complexidade de todo os sistemas resultantes da modernização do account-collector.

4.6.3.1 Dívida Técnica

Diferentemente dos marcos anteriores, as alterações efetuadas no marco quatro trouxeram impactos significativos nas métricas de dívida técnica dos projetos e do processo como um todo. A tabela 4.15 apresenta as drásticas mudanças nas métricas do account-collector. A maior redução percentual pode ser notada na métrica *Technical Debt*, com uma diminuição de 65,73%, de 1.488 dias para 510. As métricas relacionadas ao tamanho do software, como LOC, também reduziram drasticamente, 49,19%. O *Technical Debt Ratio*, por sua vez, teve uma queda de 32,48%, de 11,7% para 7,9%, indicando que os códigos legados removidos neste marco contribuíam mais para a dívida técnica que o código ainda existente, já que aproximadamente 49% do código era responsável por 65% da dívida total.

Como diversos pedaços de código foram removidos neste marco, resultado possível após as implementações realizadas nos marcos anteriores, fica difícil determinar qual o principal fluxo excluído

que trouxe tamanho ganho para as métricas. Além disso, a remoção dos testes integrados não utilizados, que também afetaram significativamente esses números, podem contribuir no mascaramento do resultado, já que a simples remoção desses códigos mortos não indica o sucesso da modernização em si.

Métricas	**Marco três**	**Marco quatro**	**Variação**
Technical Debt	1.488 dias	510 dias	-65,73%
Technical Debt Ratio	11,7%	7,9%	-32,48%
Issues	85.758	30.925	-63,94%
Complexidade	29.654	18.129	-38,86%
Duplicações	23,9%	7,8%	-67,36%
LOC	203.861	103.576	-49,19%
Métodos	14.120	8.445	-40,19%
Classes	2.173	1.288	-40,73%
Média do tamanho das classes	93,82	80,42	-14,28%
Média do tamanho dos métodos	14,44	12,26	-15,05%
Média de métodos por classe	6,50	6,56	0,90%
Frequência de issues	2,38	3,35	40,89%
Frequência de complexidade	6,87	5,71	-16,89%

Tabela 4.15: *Marco quatro: dívida técnica do account-collector*

A reorganização do domínio do transaction-manager também afetou positivamente suas métricas, apresentadas na tabela 4.16. Tanto *Technical Debt* como LOC tiveram quedas similares, na casa dos 6%, não afetando desta forma a métrica *Technical Debt Ratio*. Essas diminuição corresponde justamente a remoção da integração com o BBR, dependência que foi substituida por entidades no próprio domínio da aplicação, e da remodelagem de outras tabelas domínio.

Métricas	**Marco três**	**Marco quatro**	**Variação**
Technical Debt	64 dias	60 dias	-6,25%
Technical Debt Ratio	6%	6%	-
Issues	3.661	3.398	-7,18%
Complexidade	2.876	2.679	-6,85%
Duplicações	5,8%	4,9%	-15,52%
LOC	17.224	16.222	-5,82%
Métodos	1.960	1.839	-6,17%
Classes	489	468	-4,29%
Média do tamanho das classes	35,22	34,66	-1,59%
Média do tamanho dos métodos	8,79	8,82	0,38%
Média de métodos por classe	4,01	3,93	-1,96%
Frequência de issues	4,70	4,77	1,47%
Frequência de complexidade	5,99	6,06	1,11%

Tabela 4.16: *Marco quatro: dívida técnica do transaction-manager*

O billing-business-rule também passou por um grande *refactoring* neste marco, afetando positivamente suas métricas, disponíveis na tabela 4.17. As reduções de *Technical Debt*, *Issues* e LOC foram signicativas e proporcionais, mais de 20%, referentes a remoção das entidades e dos serviços relacionados.

Métricas	Marco três	Marco quatro	Variação
Technical Debt	65 dias	51 dias	-21,54%
Technical Debt Ratio	5,7%	5,7%	-
Issues	3.608	2.784	-22,84%
Complexidade	3.435	2.847	-17,12%
Duplicações	28,2%	26,6%	-5,67%
LOC	18.622	14.439	-22,46%
Métodos	1.977	1.486	-24,84%
Classes	434	329	-24,19%
Média do tamanho das classes	42,91	43,89	2,28%
Média do tamanho dos métodos	9,42	9,72	3,16%
Média de métodos por classe	4,56	4,52	-0,85%
Frequência de issues	5,16	5,19	0,49%
Frequência de complexidade	5,42	5,07	-6,45%

Tabela 4.17: *Marco quatro: dívida técnica do billing-business-rule*

A tabela 4.18, por sua vez, expõe as métricas do processo como um todo após a finalização do marco quatro. Pela primeira vez, o soma agregada das dívidas técnicas de todos os projetos criados em função da modernização do account-collector foi menor do que a dívida inicial, apresentada no marco zero. Além da redução das dívidas no TM e BBR, a drástica redução no AC foi a grande responsável pela diminuição de 60,16% da métrica *Technical Debt* em relação ao marco três, ou 55,16% em relação ao marco zero. O comportamento das outras métricas seguiu de forma similar às métricas do account-collector, todas com grandes reduções.

Métricas	Total marco três	Total marco quatro	Variação
Technical Debt	1.656 dias	660 dias	-60,16%
Issues	94.341	38.513	-59,18%
Complexidade	39.403	27.117	-31,18%
LOC	259.623	152.881	-41,11%
Métodos	20.436	14.180	-30,61%
Classes	3.645	2.619	-28,15%

Tabela 4.18: *Marco quatro: dívida técnica do processo como um todo*

Origem da dívida técnica

Este marco trouxe outra nova constatação: mesmo novos serviços podem precisar de *refactorings*. As simplificações e correções realizadas nos domnínios do transaction-manager e billing-business-rule, apenas dois anos após seu desenvolvimento, indicam a existência de uma dívida técnica **prudente e não intencional**, quando mesmo através de um planejamento, percebe-se que a solução desenvolvida poderia ser melhor do que o que realmente foi implementado.

Outro ponto interessante percebido neste marco foi que as alterações do marco um foram além de uma simples migração de domínio. Em relação ao transaction-manager, alguns novos requisitos de negócio foram vislumbrados pela equipe de desenvolvimento na época e implementados no código, embora não tivessem utilidade naquele momento. Todo esse tempo passou e essas im-

Figura 4.23: *Comportamento da dívida técnica somada de todos os sistemas no marco quatro*

plementações, além de ainda desnecessárias, foram consideradas um transbordo do domínio do transaction-manager e removidas do código neste quarto marco. Dessa forma, deve-se começar simples o domínio e só evoluir em termos de regras de negócio quando for realmente necessário, evitando a adição de uma complexidade prematura. Este pensamento vai de acordo com o princípio KISS [KIS], difundido no mercado de desenvolvimento de software mas muitas vezes ignorado.

Reflexo da dívida técnica no dia-a-dia

Como apresentado durante a contextualização deste marco, a migração parcial de algumas implementações e a não remoção de muitos dos códigos legados aumentava a complexidade e trazia confusão para os desenvolvedores que não sabiam exatamente quais serviços era responsáveis por quais atribuições e se determinados códigos-fontes ainda eram realmente necessários. Esse resultado é amplificado pela não distribuição do conhecimento, já que grande parte dos integrantes da equipe responsável por este domínio do marco quatro não são os mesmos desenvolvedores que atuaram nas alterações anteriores.

4.6.3.2 Arquitetura

Em relação a arquitetura, somente uma das questões tem algum aspecto que pode ser avaliado neste marco.

Bad Smells de micro-serviços

Como identificado no marco um, o BBR não segue os padrões de uma arquitetura de micro-serviços e também apresenta um *anti-pattern* de decomposição, o corte errado. Neste marco, parte desse problema arquitetural foi resolvido com a reorganização dos dados em seus próprios domínios. Assim como o processo de modernização de um monolítico legado, a modernização da arquitetura como um todo pode também ser realizada através de pequenos incrementos, como realizado neste marco. Dessa forma, embora o BBR ainda não tenha sido eliminado, está mais próximo do fim.

4.6.3.3 Modernização de software

Quanto a modernização, novamente os dados obtidos só permitem a avaliação de único ponto.

Entrega de valor para o negócio

Conforme apresetado na seção de implementação do transaction-manager, algumas alterações foram realizadas para que o código normalizado das transações passassem a ser utilizados em detrimento de um código específico por adquirente, permitindo uma maior flexibilidade ao PagSeguro que também passa por diversas evoluções.

4.7 Marco final

O quinto e último marco foi estabelecido em julho de 2018, nove meses após o marco quatro, com o intuito de obter os dados finais dos sistemas para que a pesquisa fosse consolidada. Neste período, assim como no marco quatro, não houve um foco em migrações de módulos, apenas algumas alterações no sistema legado.

4.7.1 Contextualização

No período entre o marco quatro e cinco somente algumas alterações no legado foram realizadas. Além da correção de diversos *bugs* relacionados a integração com o PagSeguro que foram descobertos, resquícios de implementações anteriores relacionadas ao fluxo do renova-fácil também foram encontradas e removidas do código. Além disso, identificou-se que uma das capacidades de negócio do sistema, o pagamento via *voucher* implementado vários anos atrás, nunca tinha sido utilizado e não atendia mais os requisito de negócio da companhia. Essa implementação, que emaranhava-se com todas as camadas de código do sistema, foi cuidadosamente removida.

4.7.2 Avaliação

Este último marco do processo de modernização poderia na verdade ter sido o primeiro. A presença de códigos-mortos ou implementações não utilizadas aumenta o tamanho do código-fonte e sua complexidade, ainda mais em sistemas altamente acoplados como o account-collector. Como resultado, impacta também a dívida técnica e o juros pago na manutenção e evolução do sistema, conforme resultado observado ao longo dos anos três desta pesquisa.

Dessa forma, somente a avaliação da dívida técnica pode ser realizada neste marco, não existindo novos dados para a avaliação de aspectos arquiteturais ou em relação ao processo de modernização.

4.7.2.1 Dívida Técnica

As métricas do account-collector deste último marco estão representadas na tabela 4.19. Houve uma redução significativa de -7,25% da métrica *Technical Debt* mas não proporcional ao número de linhas, que permaneceu praticamente constante, resultando numa queda de mais de 8% *Technical Debt Ratio*.

Métricas	Marco três	Marco quatro	Variação
Technical Debt	510 dias	473 dias	-7,25%
Technical Debt Ratio	7,9%	7,2%	-8,86%
Issues	30.925	28.702	-7,19%
Complexidade	18.129	16.842	-7,10%
Duplicações	7,8%	7,0%	-10,26%
LOC	103.576	104.381	0,78%
Métodos	8.445	7.746	-8,28%
Classes	1.288	1.182	-8,23%
Média do tamanho das classes	80,42	88,31	9,81%
Média do tamanho dos métodos	12,26	13,48	9,87%
Média de métodos por classe	6,56	6,55	-0,05%
Frequência de issues	3,35	3,64	8,58%
Frequência de complexidade	5,71	6,20	8,48%

Tabela 4.19: *Marco final: dívida técnica do account-collector*

A tabela 4.20, por sua vez, expõe as métricas do processo como um todo após a finalização deste marco final, consolidando as métricas resultantes desta pesquisa. Os números tiveram comportamento semelhantes ao do account-collector, mas de forma mais tênue, uma vez que não houve alteração em nenhum dos outros sistemas.

Métricas	Total marco quatro	Total marco cinco	Variação
Technical Debt	660 dias	623 dias	-5,61%
Issues	38.513	36.290	-5,77%
Complexidade	27.117	25.830	-4,75%
LOC	152.881	153.686	0,53%
Métodos	14.180	13.481	-4,93%
Classes	2.619	2.513	-4,05%

Tabela 4.20: *Marco final: dívida técnica do processo como um todo*

A figura 4.24 apresenta graficamente o comportamento da métrica *Technical Debt* durante todos os marcos desta pesquisa, com base no valor acumulado entre todos os sistemas relacionados ao processo de modernização. Após duas altas consecutivas, no marco um e dois, os três marcos subsequentes apresentaram quedas expressivas que colaboraram para uma redução total de 57% do valor da métrica quando comparado o ínicio e o final do processo, 1.472 e 623 dias, respectivamente.

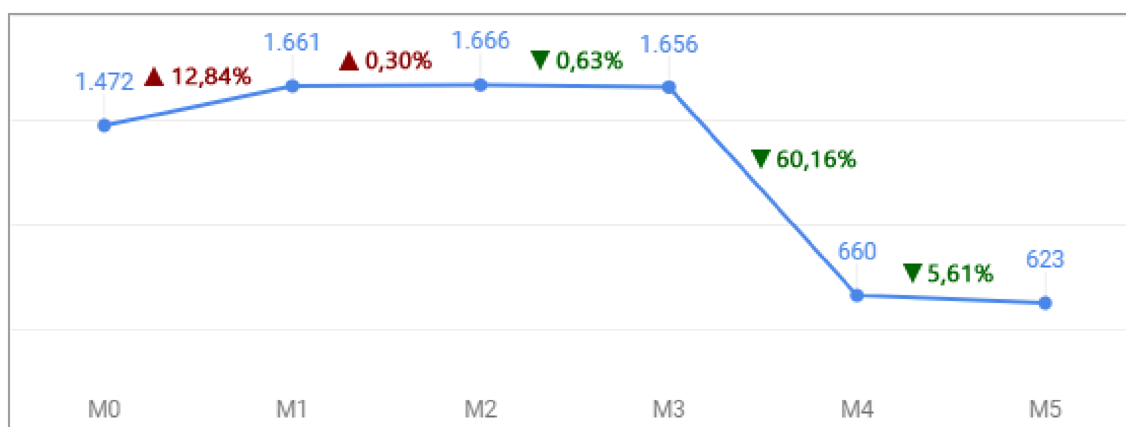

Figura 4.24: *Comportamento da dívida técnica somada de todos os sistemas no marco final*

Capítulo 5

Análise dos resultados e conclusões

Neste capítulo serão apresentados os resultados e as conclusões obtidas a partir do desenvolvimento do estudo de caso, detalhado no capítulo 4.

5.1 Resultados

Embora diversas questões menores tenham sido avaliadas durante cada etapa do processo de modernização, as três questões centrais dessa pesquisa, detalhadas na seção de objetivos, 1.2, serão apresentadas a seguir.

5.1.1 Variação da dívida técnica durante o processo de modernização

Responder ao objetivo 'analisar a variação da dívida técnica durante o processo de modernização do monolítico legado' foi possível após cinco marcos de desenvolvimento durante o processo de modernização.

Apesar do termo dívida técnica ter mais de 25 anos, ainda hoje há algumas más interpretações do seu significado, amplificado e modificado ao longo dos anos com a expansão do seu uso pela indústria. Logo, antes da análise da variação, é necessário um entendimento histórico que colabore com a desambiguação de todos os significados desse termo:

- O primeiro significado de dívida técnica remete justamente a origem do termo, seu aspecto financeiro, criado por Ward Cunningham em 1992. Cunningham entendia a dívida técnica como uma ferramenta estratégica que deveria ser gerenciada para que a empresa obtivesse vantagens no curto-prazo e não deteriorasse a qualidade de seus sistemas no longo-prazo. Essa ideia, que introduz os aspectos de principal e juros, é uma visão pouco explorada na indústria e desconhecida por boa parte dos desenvolvedores. Alem disso, outro ponto as vezes não claro na literatura é o papel do juros da dívida, que não somente se aplica no desenvolvimento de novas atividades, mas também no dia-a-dia quando intervenções ou atividades manuais são necessárias devido a baixa qualidade da implementação realizada.
- O segundo significado da dívida técnica nos dias atuais relaciona-se à qualidade interna de software, representada pelas métricas estáticas de código-fonte. Essa visão é comum na indústria, habituada a utilizar as ferramentas justamente para esse propósito. Muitas vezes,

não há noção de que a dívida técnica se estende além do código-fonte para outros aspectos relacionados a arquitetura, tecnologia, testes, ambiente de desenvolvimento, distribuição do conhecimento, etc.

- O terceiro significado representa a métrica *Technical Debt*, conforme apresentado ao longo deste projeto. O termo foi justamente utilizado em inglês para que não se confundisse com a expressão dívida técnica em si. O *Technical Debt* representa uma forma de mensurar quantitativamente a dívida técnica total, sendo o *SQALE quality index* uma das formas de implementação deste cálculo. Entretanto, devido a uma limitação das ferramentas atuais somente as dívidas de código-fonte fazem parte do mesmo.

Dessa forma, a variação da dívida técnica corresponde justamente ao comportamento da métrica *Technical Debt*, seu terceiro significado, cujo valor foi extraído de cada sistema ao longo de todo processo de modernização. A figura 5.1, apresentada novamente abaixo, evidência graficamente o histórico do comportamento da dívida técnica em todo processo.

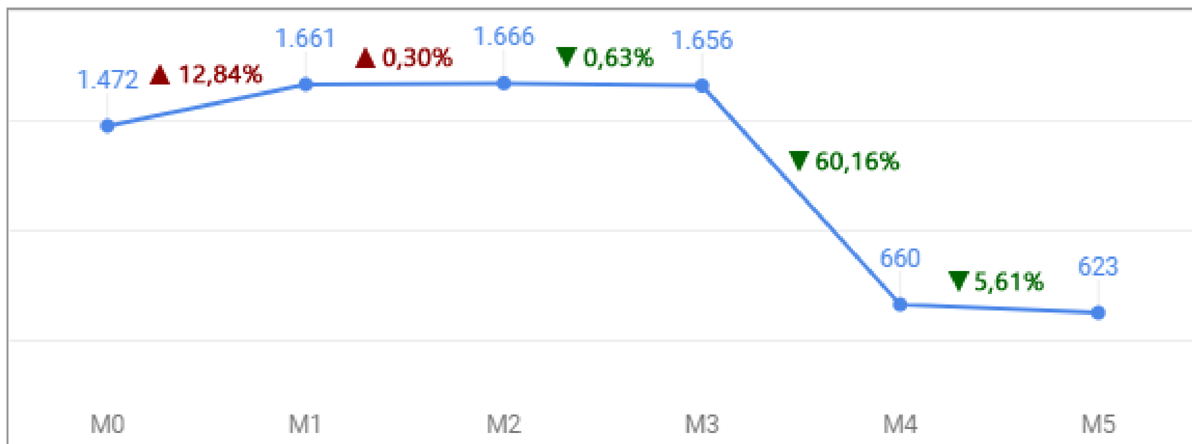

Figura 5.1: *Variação da dívida técnica durante o processo de modernização*

Pode-se obervar uma significativo aumento da dívida do marco zero para o marco um, que se estabilizou durante os marcos dois e três, apresentando uma queda vertiginosa somente no marco quatro, quando enfim retornou para um patamar inferior ao do ínicio do processo. Alguns fatores indicam tal comportamento da dívida:

- **Granularidade das mudanças**: Passos grandes demais não são aconselhados na literatura [Ric16][Deh18], que favorecem ganhos pequenos e constantes, com o objetivo a cada passo chegar a um estado mais próximo da arquitetura ideal. No marco um, por exemplo, além da migração de parte do fluxo de recorrência, a criação em paralelo de um novo domínio de transações e uma nova aplicação com regras gerais fez que a dívida técnica, a complexidade do fluxo e o próprio tamanho de código-fonte tivessem um expressivo aumento. Se somente a migração do fluxo de recorrência para o BBPM tivesse sido realizada naquele momento, o impacto na dívida técnica poderia ter sido positivo. Entretanto, de forma geral, devido a necessidade de coexistência dos sistemas e a criação de camadas de anti-corrupção, pode ser natural que ocorra um aumento da dívida em casos específicos dependendo do acoplamento do sistema.

- **Adição de novas funcionalidades**: Apesar das implementações no legado estarem congeladas, pequenas alterações são sempre priorizadas quando existe uma demanda de negócio urgente. Mesmo quando novas fucionalidades já são construídas fora do domínio, em micro-serviços próprios, é natural que alterações no legado sejam necessárias para permitir uma comunicação entre os mesmos. Correções de *bugs* e pequenas mudanças ocorreram em todos os marcos do estudo de caso, sendo a mais importante a implementação do boleto registrado.

- **Conhecimento do legado**: A falta de conhecimento dos subdomínios do legado levou a decisões iniciais que não surtiram ganhos imediatos à arquitetura. Um erro na priorização de quais módulos migrar pode determinar o aumento da dívida técnica e da complexidade como um todo.

- ***Feature-toogle* utilizada de forma indiscriminada**: Essa técnica deve ser utilizada com cautela para evitar que código não utilizado continue indefinidamente na aplicação, prática identificada em três dos cinco marcos do processo. A não remoção desses códigos pode exigir um custo muito maior no futuro, quando outros desenvolvedores, fora do contexto, gastarão muito mais tempo com a análise e remoção desses códigos, uma dívida técnica. Esse foi um dos motivos pelo qual a curva da dívida técnica apresentou um comportamento longe de ser linear. Caso as remoções de código tivessem sido realizadas dentro de seus marcos, a dívida técnica não teria apresentado tamanho crescimento inicial e queda do marco quatro teria sido mais sutil.

- **Governança da aplicação**: A alteração da estrutura da plataforma corporativa durante o marco três refletiu diretamente nos resultados da dívida técnica obtidos. Com a alteração, um único time passou dar manutenção e atuar na evolução do sistema, identificando ao acaso diversas melhorias que poderiam ser realizadas, como a remoção das *features-toggles* e códigos não mais necessários.

- **Maior qualidade dos micro-serviços**: Outro ponto determinante para o resultado é a maior qualidade dos micro-serviços em relação ao sistema legado. Apresentam dívida técnica proporcional inferior a metade da dívida do legado, o que colaborou para uma redução de 57% da dívida total durante todo o processo enquanto o numero de linhas de código caiu somente 23%.

- **Limpeza de códigos desnecessários**: A grande remoção de códigos desnecessários realizada no marco quatro foi grande responsável pela queda expressiva da dívida técnica e das outras métrica em geral. Apesar dos ganhos visíveis, os números da queda mascaram o que realmente havia sido migrado e o que era somente um código-morto, que inevitavelmente contribuem para a redução total da dívida.

5.1.2 Padrões de comunicação em arquitetura de micro-serviços

Em relação ao objetivo 'identificar os padrões de comunicação utilizados na arquitetura de micro-serviços e a motivação para cada uma deles, comparando com os demais existentes na literatura', diversos aspectos foram observados a partir da criação de cinco novas aplicações durante o estudo de caso.

REST, *Representational State Transfer*, conceito definido por Roy Fielding nos anos 2000 [Fie00], é uma estratégia arquitetural baseada num conjunto de princípios a partir do protocolo HTTP para o desenvolvimento de arquiteturas distribuídas de software. Com a expansão dos micro-serviços, REST consolidou-se como um dos padrões mais utilizados já que atende as três premissas de integração conceituadas na seção 2.2.2. Basicamente, REST define como expor os subdomínios de um sistema através de recursos, cujas ações são determinadas a partir dos verbos HTTP utilizados. Todos os sistemas criados durante a modernização tiveram seus serviços expostos através de recursos REST, implementando o modelo de maturidade [Fow10] nível 3 na maioria dos casos.

Outro aspecto relevante da comunicação é o tipo de sincronia adotado. Embora o modo síncrono seja o mais comumente aplicado no paradigma requisição-resposta, o modo assíncrono também pode ser implementado através do uso de *polling* ou mecanismos de *callback*. A tabela 5.1 apresenta de forma sumarizada esses detalhes em relaçao ao serviços criados.

Sistemas	**Serviços REST**	**Mensageria**	**Tipo de sincronia**	**Filas internas**
BBPM	Sim	Não	Assíncrono	Sim
TM	Sim	Não	Assíncrono	Sim
BBR	Sim	Não	Síncrono	Não
CCA	Sim	Não	Síncrono	Sim
PSGW	Sim	Não	Síncrono	Não

Tabela 5.1: *Detalhes da comunicação dos sistemas criados*

Os sistemas que utilizam comunicação assíncrona, BBPM e TM, foram construídos dessa forma pois suportam operações de longa duração, isto é, não são capazes de processar e retornar um resultado num curto espaço de tempo. Optou-se pela implementação de mecanismos de *callback* em detrimento de *polling* nesses casos, já que o tempo de duração das operações podem demorar diversos dias, como é o caso das transações de boletos bancários, evitando-se um grande disperdício de recursos computacionais [Bat15]. A implementação do modo assíncrono difere-se do modo síncrono em alguns pontos:

- A resposta do servidor REST passa a ser *202 Accepted* ao invés de *201 Created* [Far16], indicando que a requisição foi aceita e será processada, não retornando neste momento o *payload* com os dados finais da operação.
- O envio de um parâmetro extra na requisição que corresponde ao endereço de *callback* para o qual será enviado o resultado final da operação.
- A requisição de *callback*, um POST HTTP para determinado endereço no qual será enviado o *payload* com os dados do processamento da operação.

Para que a implementação fosse também resiliente, adotou-se filas internas aos sistemas assíncronos, como apresentado na tabela 5.1. Dessa forma, mesmo em caso de longas indisponibilidades de rede, o processamento completo das operações é garantido através de retentativas de comunicação em tempos exponenciais, tanto com os clientes e dependências, tornando o sistema um anteparo no caso de grandes adversidades, já que não propaga os erros e é capaz de reestabelecer-se automaticamente.

Contudo, essa implementação de REST assíncrona leva a seguinte indagação: por que utilizar REST assíncrono com filas internas ao invés de usar mensageria diretamente? As seguintes inferências podem ser feitas em relação a essa comparação:

- Permite maior controle do fluxo, isto é, cliente sabe em tempo real se o servidor está apto a atender a requisição.
- Permite validação dos parâmetros de forma síncrona e retorno em casos de erro de forma mais rápida.
- Permite o retorno síncrono de dados iniciais, caso necessário.
- Permite a padronização da comunicação, síncrona ou assíncrona, através de recursos REST. Imagine um micro-serviço que disponibilize tanto operações síncronas e assíncronas. Caso mensageria fosse utilizado para as assíncronas e REST para síncronas, o conceito de recursos poderia ser comprometido.
- Não garante total disponibilidade como mensageria, isto é, em caso de quedas de todas as instâncias do sistema, o cliente final é impactado. Entretanto, disponibilidade em entregas de novas versões não é um problema para REST já que mecanismos de *load balance* devem ser utilizados.
- Requer uma maior implementação, já que a diferença pra mensageria é justamente a presença de uma camada REST inicial.

Uma vez que o padrão requisão-resposta foi escolhido em detrimento a colaboração baseado em eventos, consequentemente optou-se pela orquestração ao invés da coreografia. A orquestração é essencial nesse caso para evitar uma arquitetura *spaguetti*, quando os micro-serviços se comunicam diretamente uns aos outros e formam longas dependências transitivas e até mesmo cíclicas, levando a um alto acoplamento arquitetural também chamado de *dependency hell*. Para fazer a implementação do orquestrador BBPM, adotou-se ferramentas de BPM, *Business Process Management*, capazes de definir complexos fluxos do processo de negócio ou simplesmente realizar a composição de serviços de forma paralela ou sequêncial, estendo assim as características de ferramentas mais simples como *API-Gateways*. A escolha se mostou satisfatória e atualmente a orquestração já é utilizada para diversos fluxos de negócio dentro da plataforma corporativa. Dentros os ganhos:

- **Maior controle das regras de negócio**: A centralização das regrais gerais de negócio e do fluxo de cada processo mostrou-se essencial em fluxos de cobrança, geralmente demasiadamente complexos e que dependem de diversos serviços distintos de outros domínios. A utilização de coreografia, abordagem descentralizada, traria demasiada complexidade para o sistema e dificultaria o entendimento dos complexos fluxos.
- **Documentação viva e sempre atualizada**: A notação BPMN é capaz de auto-documentar todo o fluxo do processo através do desenho gerado, favorecendo uma maior visibilidade das implementações das regras de negócio tanto para os desenvolvedores como para as outras áreas da companhia, contribuindo para a distribuição do conhecimento.

- **Monitorações avançadas**: A orquestração central através do BBPM permitiu a criação de uma monitoração única que garante o tempo de resposta das aplicações clientes. Para isto, para cada chamada externa pode ser configurada uma SLA, *service level agreement*, que determina o tempo máximo de resposta que dado serviço deve prover, além de outras condições. Na prática, as monitorações geram alertas descrevendo qual etapa do processo foi comprometida e não respeitou a SLA, proporcionando um rápido descobrimento de erros e gargalos no sistema, diminuindo consideravelmente o esforço e tempo na resolução de problemas.

5.1.3 Guia de boas práticas de *Strangler Application*

Em relação ao objetivo 'desenvolver um guia de boas práticas da estratégia de *Strangler Application*, com os principais aspectos e premissas a serem seguidas, consolidando a literatura existente', muitas informações puderam ser observadas e coletadas ao longo do processo, confirmando e estendendo parte das afirmações presentes na literatura. A lista dessas boas práticas está listada abaixo, já ordenada de acordo com a sequência nas quais essas práticas devem ser adotadas durante o processo:

1. **Monitore a dívida técnica**: Utilizar uma ferramenta como o SonarQube[CP13] é simples, rápido e garantirá uma maior visibilidade das métricas de código-fonte durante o processo de modernização, colaborando na tomada de decisões e identificando se o caminho seguido está trazendo resultados satisfatórios. O estudo de caso desenvolvido nesta pesquisa demonstra a importância desse tipo de monitoração.

2. **Conheça bem o sistema legado**: Entender o domínio do sistema, seus subdomínios, a relação e as dependências entre cada um é um processo vital para o sucesso da modernização. Além disso, importante identificar todas os clientes e dependências externas do sistema e todos os serviços ou casos de uso pelos quais a aplicação é responsável. Para isso, o time deve coletar toda a documentação existente, conversar com os integrantes mais antigos da área e até mesmo com os clientes para entender todo o contexto. Após isso, uma análise manual do código é necessária para confrontar as informações obtidas com as implementações reais no código, que podem ter sofrido mudanças cujo conhecimento não foi propagado.

3. **Faça correções iniciais**: Colocar a casa em ordem é uma boa prática muito aconselhada na literatura e que também colabora para o ganho de conhecimento do sistema legado. Dentre as correções iniciais, vale citar:

    (a) **Remoção de código morto**: A utilização de ferramentas para a detecção pode facilitar o processo de identificação desses códigos, que muitas vezes aumentam o acoplamento do sistema, sem que haja necessidade. Além disso, as migrações durante o processo podem exigir que alterações sejam necessárias nesses códigos para que a compilação não quebre, como foi o caso dos testes integrados do estudo de caso.

    (b) **Remoção de fluxos não mais usados**: Remova também partes de códigos não mais necessárias que as ferramentas não são capazes de detectar. Isso porque são fluxos completos do sistema mas que não são utilizados em tempo de execução, como no caso da funcionalidade de *voucher*, removida do legado somente no marco cinco.

4. **Analise se a modernição é necessária**: Independente da modernização ser ou não necessária, todos os passos anteriores já geraram pequenos ganhos para a diminuição da dívida técnica do sistema, inclusive documental, sem que houvesse um grande esforço. Analisar se a modernização é necessária é importante para evitar que a companhia gaste seus recursos em algo que não gerará retorno esperado, mesmo no caso de *Strangler Application* onde os custos e riscos são baixos. Além dos indícios obtidos de outras literaturas em relação a necessidade de modernização, alguns novos indícios foram adicionados na lista abaixo:

    - Longos ciclos de desenvolvimento.
    - Não há separação dos módulos e há um alto acoplamento no código.
    - Tecnologias utilizadas ultrapassadas.
    - Baixa escalabilidade.
    - Dificuldade de entendimento do sistema por parte de novos desenvolvedores.
    - **Custo de manutenção e necessidade de evolução**: Mesmo que todos os outros indícios anteriores tenham sido satisfeitos, qual a vantagem de modernizar um sistema que não exige manutenção nem vai ser evoluído tão cedo? Em alguns casos, quando o legado não for um importuno no dia-a-dia das equipes nem for ter seus requisitos alterados constantemente, pode ser mais aconselhado que a técnica de *wrapping*, ou encapacotamento, descrita na seção 2.3.1 seja utilizada. Essa técnica é mais simples e menos custosa, permite que o legado seja exposto através tecnologias não intrusivas e também que seus domínio seja isolado, uma vez que uma fachada pode funcionar de maneira similar a uma camada anti-corrupção. Dessa forma, todo o esforço pode ser concentrado em atividades de maior retorno, como a evolução de outros sistemas que podem utilizar a fachada criada sem se preocupar com o código legado.

5. **Faça uma decomposição correta do domínio**: Decompor o sistema monolítico em micro-serviços da forma incorreta pode levar a uma arquitetura monolítica distribuída, que é a pior das soluções arquiteturais possíveis [RIC]. Dessa forma, decompor o sistema vaí alem da simples extração de blocos de código em serviços diferentes. É necessário conhecer os domínios ou subdomínios que compõem seu sistema legado, isto é, quais os conjuntos de conhecimentos distintos com o qual seu sistema deve saber lidar. Dessa forma, geralmente são aplicadas técnicas de *domain-driven-design* para encontrar os *bounded-contexts*, que correspondem à implementação do modelo idealizado pelo conceito de subdomínio. Entretanto, a implementação do sistema legado provavelmente apresenta um alto acoplamento e não segue esses limites definidos pelos domínios, o que torna mais difícil a extração. Embora outra práticas sejam descritas na literatura, essa visão global e ideal do sistema, de dentro pra fora, favorece as mudanças em relação as técnicas que estimulam uma visão de fora para dentro, como a utilização de casos de uso, que geralmente não correspondem a um micro-serviço específico. Para facilitar o entendimento dos próximos itens, será denominado de módulo essas unidades de negócio mínimas identificadas que devem corresponder a um micro-serviço extraído.

6. **Crie um dígrafo dos módulos do sistema**: Para documentar e consolidar o conhecimento e as descobertas da equipe em relação ao legado e seus domínios, construa um grafo direcionado,

ou dígrafo, com todos os módulos do sistema legado e seus relacionamentos. Neste grafo, os nós representam os módulos e as arestas direcionadas as comunicações entre os mesmos. No exemplo abaixo, figura 5.2 o módulo **A** possue três dependências eferentes e por isso grau de saída igual a três, enquanto o módulo **E** possuí duas dependências aferentes, logo grau de entrada igual a dois. Embora conhecimento do negócio seja necessário, algumas ferramentas como as conceituadas nos trabalhos citados na seção de técnicas de decomposição, 2.3.1.1, podem ser úteis nessse momento.

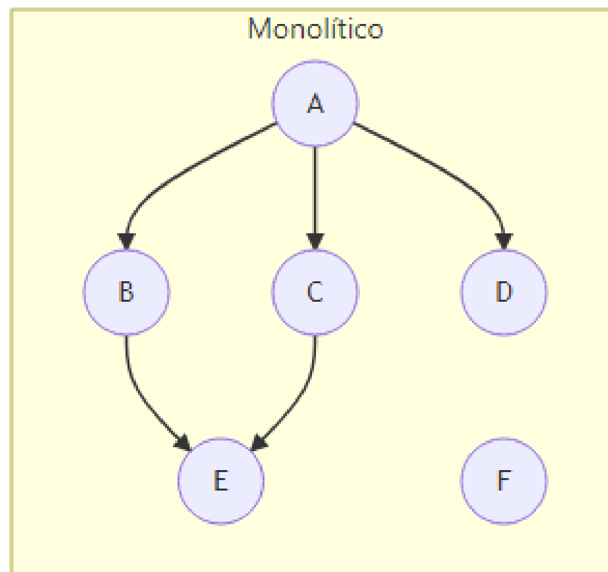

Figura 5.2: *Dígrafo dos módulos do monolítico*

7. **Priorize a migração corretamente**: Uma vez identificados os módulos ou *bounded-contexts* do seu sistema, a priorização do que será extraído deve ocorrer de forma a minizar os custos, o tempo e o acoplamento entre os sistemas, fazendo com que cada passo possa ser uma pequena melhora do estado anterior, evitando qualquer tipo de risco caso a modernização não seja continuada. Dessa forma, utilizando o grafo de dependências construído anteriormente, priorize a extração de acordo com a seguinte ordem:

   (a) **Nós com menor grau de saída**: Priorize aqueles módulos que não dependem de outros do monolítico, dessa forma será minimizada a dependência dos novos micro-serviços em relação ao monolítico e evitada a construção de camadas anti-corrupção. Enquanto [VAC] considera um anti-padrão o roteamento do legado para o novo micro-serviço, [Deh18] tem exatamente a mesma percepção de que é inevitável muitas vezes que não haja nenhuma dependência, logo, prioriza-se os mais desacoplados, iniciando por aqueles com menor grau de saída.

   (b) **Nós com menor grau de entrada**: Caso haja um empate em relação ao critério anterior, a decisão é dada de acordo com o nó de menor grau de entrada, que pelo fato de estar mais desacoplado dos demais módulos, pode ser mais simples de ser migrado, evitando a modificação em diversas parte do legado. Entretanto, essa condição pode nem sempre ser verdadeira, sendo responsabilidade dos desenvolvedores avaliarem dentre as possibilidades quais os módulos de menor complexidade de migração ou que podem

agregar mais valor ao negócio. Essa condição pode variar de acordo com o tamanho dos módulos e da forma como estão dispostos no sistema.

Priorizar os nós com menor saída e não com menor entrada, além de diminuir as dependências dos novos micro-serviços, colaborando para a criação de domínios isolados, mantém o acoplamento baixo e evita dependências cíclicas entre o monolítico e os micro-serviços. Além disso, é muito mais fácil e eficiente construir testes ágeis nos novos serviços do que no sistema legado, aumentando a confiabilidade da arquitetura como um todo.

Seguindo essas regras de priorização, a sequência de migração dos módulos do exemplo da figura 5.2 seria **F, D, E, B ou C, A**, não sendo necessário em nenhum momento a integração de volta do micro-serviço para o monolítico. Esse padrão evita a cada etapa a escolha de nós com maior acoplamento, como os que representam o módulo **B** ou **C**, por exemplo. Estes nós dependem de dados do módulo **E**, e caso fossem migrados prioritariamente, levariam a uma dependência cíclica entre entre os sistemas e exigiriam a construção de uma camada anti-corrupção de acesso aos dados do módulo **E**, sem nenhuma serventia para o futuro.

Entretanto, muitas vezes deseja-se evoluir um dos módulos em específico devido a questões de manutenção ou evolução. Neste caso, ao invés de realizar a extração do mesmo prioritariamente, que pode resultar numa piora do acoplamento e num esforço extra com a construção de uma camada anti-corrupção, aconselha-se encontrar o caminho mínimo de uma das raízes do grafo até o mesmo, dada as priorizações acima. Por exemplo, caso o módulo escolhido seja o **C**, será necessário a migração do módulo **E** anteriormente, uma das raízes do grafo. Neste caso, a sequência da migração seria **E, C, F, B ou D, A**.

8. **Implemente cada etapa atomicamente**: A não remoção dos códigos legados já migrados para outros sistemas foi uns dos problemas encontrados no estudo de caso desta pesquisa. Tal problema ocorreu justamente pela falta de atomicidade na etapa de migração, que como levantado em [Deh18], deve ser composta das seguintes atividades:

   - Construção do novo micro-serviço.
   - Redirecionamento de todos clientes para o novo serviço criado.
   - Remoção do código legado do monolítico.

   A utilização da técnica de *feature-toggle* não altera nenhuma dessas premissas.

9. **Isole os dados dos novos domínios**: Para evitar que os novos serviços tenham seus domínios contaminados por dados legados, evita-se diretamente a comunicação entre os mesmos através da utilização de uma camada anti-corrupção. Na maioria dos casos, essa camada pode estar contida dentro do próprio sistema monolítico caso a tradução necessária não seja tão complexa. Dessa forma, quando a comunicação for unidirecional no sentido do monolítico para o micro-serviço, construa a API do micro-serviço da forma desejada e com tecnologias não intrusivas. No monolítico, implemente uma camada que abstraia e traduza os dados do legado para a nova API criada. Quando a comunicação no sentido micro-serviço para o monolítico for inevitável, mesmo seguindo as regras de priorização, construa uma camada anti-corrupção num sistema apartado ou no próprio legado caso seja possível, expondo uma API de acordo com os dados do micro-serviço, que serão convertidos para o domínio legado.

5.2 Ameaças à validade

Procurou-se, ao longo do desenvolvido desta pesquisa, triangular dados quantitativos e qualitativos para que fosse minimizado o viés e as limitações do estudo de caso. No caso da ameaça à validade interna, por se tratar de um caso do mundo real, não houve controle por parte do pesquisador de como as variáveis iriam se comportar ou quais novas variáveis poderiam surgir. Neste contexto, procurou-se contextualizar durante cada marco todos os fatos observados, mas não sendo possível afirmar de maneira isolada o impacto de cada um no resultado da pesquisa. Dentre eles:

- Implementações de novas funcionalidades no legado.
- Correções de bugs no legado.
- Remoção de código-morto, que simplesmente não indica o sucesso de um processo de modernização.
- Diferença de conhecimento das equipes de desenvolvimento do passado e durante o processo de modernização.
- Diferença de tecnologia utilizada que pode influenciar no comportamento da dívida técnica.

Além disso, por se tratar de somente um estudo de caso, a ameaça externa, ou capacidade de generalização do estudo pode ser questionada. Entretanto, devido a duração e profundidade da pesquisa dentro de um contexto corporativo, com a observação de um dos principais sistemas legados da companhia, acredita-se que os resultados da pesquisa sejam representativos e possam generalizados para outras populações ou contextos, uma vez que não divergiram da literatura.

O último aspecto de ameaça à validade refere-se ao relacionamento entre a teoria e a observação, ou seja, se as medições do estudo realmente refletem a teoria apresentada. Um dos problemas dessa ameaça de construção do projeto é a medição da dívida técnica, que devido as limitações ferramentais compreendem somente o código-fonte sistema. Para que isso fosse minimizado, utilizou-se aspectos etnográficos de pesquisa com a imersão do pesquisador e coleta contínua de dados qualitativos. Nesse aspecto, não foi utilizado entrevistas pois a visão da modernização constrói-se ao longo de um processo faseado, que teve duração de três anos, nos quais boa parte das equipes de desenvolvimento foram alteradas e poucos foram os participantes que colaboraram do início ao fim do processo.

5.3 Conclusões

Neste projeto de mestrado foi realizado um trabalho exploratório procurando incorporar diversos elementos presentes no dia-a-dia do desenvolvimento do estudo de caso, fornecendo um detalhamento aprofundado sobre um caso de modernização de um sitema legado dentro de um contexto real, numa grande empresa brasileira.

Para que os dados obtidos durante o estudo de caso pudessem ser avaliados significativamente, um amplo estudo da literatura foi realizado em relação aos três temas pilares desta pesquisa: dívida técnica, arquitetura de micro-serviços e modernização de software, temas justificadamente

importantes tanto na acadêmica como na indústria. Além da exploração dos temas em si, identificou-se as principais características de correlação entre os mesmos.

Dessa forma, o contexto histórico levantado tornou perceptível como vários conceitos da literatura de diferentes épocas apresentam características similares. A própria técnica de *Strangler Application*, cunhada por Martin Fowler em 2004, apresenta semelhanças com o estudo de David Parnas da década de 70, relacionado à decomposição de software. Este mesmo estudo apresenta também diversas características que hoje ainda podem ser aplicadas no conceito de micro-serviços, mostrando a importância da literatura acadêmica e de como o conhecimento vai sendo aproveitado e evoluído.

O estudo procurou também contextualizar os aspectos relacionados aos sistemas legados, sua caracterização, sua relação com qualidade de software, dívida técnica e envelhecimento de software, temas presentes na literatura desde a década de 70. Essa análise é importante para perceber que a modernização de software não é um tema pontual, sempre foi e será uma questão relevante da engenharia de software, cujas estratégias vão evoluindo de acordo com as tendências arquiteturais da época.

Em relação as estratégias de modernização, consolidou-se a tendência atual pela utilização de técnicas faseadas, de menor custo e risco, como *Strangler Application*. O guia de boas práticas desenvolvido converge diversas práticas espalhadas pela literatura além de outras questões importantes constatadas durante os três anos do estudo de caso.

Em relação a arquitetura de software, analisou-se diversas características dos micro-serviços, desde resiliência a granularidade. Além disso, explicitou-se a motivação e os ganhos na adoção de determinados padrões de comunicação, determinados principalmente pelo contexto no qual estão inseridos os sistemas. Quanto às métricas de código obtidas, embora não seja possível comparar diretamente os valores obtidos dos micro-serviços com os do monolítico, já que o mesmo se trata de um sistema legado de mais de 18 anos, alguns padrões relacionados a coesão e acoplamento de classes puderam ser percebidos já que os números pouco variaram entre as cinco novas aplicações criadas.

Constatou-se também que o comportamento da dívida técnica é influenciado pela estratégia de modernização adotada, principalmente de acordo com a escolha de quais módulos do sistema serão prioritariamente migrados. Embora a variação tenha apresentado um resultado satisfatório, com uma queda total de 57%, o aumento inicial explicitou algumas decisões incorretas tomadas durante o processo. Com o desenvolvimento do guia de boas práticas de *Strangler Application*, espera-se que o efeitos possam ser positivos desde o primeiro passo, tornando a arquitetura sempre mais próxima do ideal a cada iteração da modernização.

De forma geral, o estudo de caso forneceu uma ampla gama de dados ao longo do processo, tanto positivos como negativos, que puderam ser confrontados com a literatura e validados através da avaliação qualitativa e quantitativa, contribuindo para a variedade de fundamentos debatidos neste projeto de pesquisa.

5.3.1 Trabalhos futuros

O aspecto amplo e exploratório deste projeto trouxe algumas novas dúvidas e limitações que serão de grande valia quando exploradas. Abaixo, alguns dos trabalhos futuros que podem ser

desenvolvidos:

- Aplicar o guia de boas práticas de *Strangler Application* desenvolvido em outros sistemas reais para confirmar sua eficácia.
- Aprofundar no assunto *domain-driven-design* e *bounded-contexts* no âmbito de *Strangler Application* dentro de sistemas e domínios reais: os exemplos na literatura são poucos e simplificados.
- Constuir ferramentas para a identificação da dívida técnica arquitetural em arquiteturas de micro-serviços: com a mudança das arquiteturas monolíticas para arquiteturas de micro-serviço, cada vez mais as métricas de código-fonte serão insuficientes para detectar *bad smells* já que os códigos estão cada vez mais espalhados e os maiores problemas concentrados em questões de domínios e arquitetura.

Referências Bibliográficas

[AAE16] Nuha Alshuqayran, Nour Ali e Roger Evans. A systematic mapping study in microservice architecture. *IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, 2016. 2, 10, 47

[ACD10] Asil Almonaies, James Cordy e Thomas Dean. Legacy system evolution towards service-oriented architecture. *International Conference on Advanced Information Networking and Applications*, 2010. 25

[ACT] Activiti bpmn engine. https://www.activiti.org. Online; último acesso 01-Ago-2018. 61

[ADQ] Adquirentes, subadquirentes e gateways. https://blog.vindi.com.br/adquirentes-subadquirentes-e-gateways. Online; último acesso 08-Mar-2018. 36

[ALA] Seven microservices anti-patterns. https://www.infoq.com/articles/seven-uservices-antipatterns. Online; último acesso 01-Jul-2018. 22, 23

[Bat15] Amandeep Batra. Asynchronus rest api: A case study. https://amandeepbatra.wordpress.com/2015/12/21/asynchronus-rest-api-a-case-study, 2015. Online; último acesso 01-Ago-2018. 106

[BBB] Beyond big bang - public ceo q2 2016. https://www.purestorage.com/content/dam/purestorage/pdf/whitepapers/Q2SpecialReportLegacyModernization.pdf. Online; último acesso 01-Jul-2018. 2, 24, 28, 31

[BBB+01] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt e R. Jeffries et al. Manifesto for agile software development. http://agilemanifesto.org, 2001. Online; último acesso 15-Out-2016. 34

[BC68] Tom Barnett e Larry Constantine. *Modular Programming: Proceedings of a National Symposium*. Information Systems Institute, 1968. 19

[BCG+10] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan e N. Zazworka. Architectural technical debt identification based on architecture decisions and change scenarios. *12th Working IEEE/IFIP Conference on Software Architecture*, páginas 47–52, 2010. 6

[Ben95] K. Bennett. Legacy systems: coping with success. *IEEE Software*, 12:19–23, 1995. 1, 24, 25, 30

[BLW+97] J. Bisbal, D. Lawless, Bing Wu, J. Grimson, V. Wade, R. Richardson e D. O'Sullivan. An overview of legacy information system migration. *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*, 1997. 25

[BLWG99] J. Bisbal, D. Lawless, Bing Wu e J. Grimson. Legacy information systems: issues and directions. *IEEE Software*, 16:103–111, 1999. 1, 24, 25, 26, 30

[Bob09] Uncle Bob. A mess is not a technical debt. https://sites.google.com/site/unclebobconsultingllc/a-mess-is-not-a-technical-debt, 2009. Online; último acesso 15-Out-2016. 6

[Boe91] B.W Boehm. Software risk management: Principles and practices. *IEEE Software*, 8:32–41, 1991. 6

[BOI] Boilerplate code. https://en.wikipedia.org/wiki/Boilerplate_code. Online; último acesso 01-Ago-2018. 68

[Bro] Apply the strangler application pattern to microservices applications. https://www.ibm.com/developerworks/cloud/library/cl-strangler-application-pattern-microservices-apps-trs/index.html. Online; último acesso 01-Jul-2018. 26, 27, 29, 74

[BSJH14] Belfrit Batlajery, Amir Saeidi, Slinger Jansen e Jurriaan Hage. How do professionals perceive legacy systems and software modernization? *Proceedings of the 36th International Conference on Software Engineering*, 2014. 2, 24, 30

[CAAA15] Alexander Chatzigeorgiou, Apostolos Ampatzoglou, Areti Ampatzoglou e Theodoros Amanatidis. Estimating the breaking point for technical debt. *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*, 2015. 6

[CMAM11] Zenon Chaczko, Venkatesh Mahadevan, Shahrzad Aslanzadeh e Christopher Mcdermid. Availability and load balancing in cloud computing. *International Conference on Computer and Software Modeling IPCSIT, Singapore*, 2011. 37

[COG] Cognitive complexity - a new way of measuring understandability. https://www.sonarsource.com/docs/CognitiveComplexity.pdf. Online; último acesso 01-Jul-2018. 10

[Con68] Melvin Conway. How do committees invent? *Datamation magazine*, páginas 30–31, 1968. 14, 89

[CP13] G. Ann Campbell e Patroklos Papapetrou. *SonarQube in action*. Manning, 2013. 6, 9, 108

[Cre08] John W. Creswell. *Research design: qualitative, quantitative, and mixed methods approaches*. Sage Publications, 2008. 41

[CT12] Michele Chinosi e Alberto Trombetta. Bpmn: An introduction to the standard. *Computer Standards and Interfaces*, 34:124 – 134, 2012. 59, 60

[Cun92] Ward Cunningham. The wycash portfolio management system. *Proceeding of OOPSLA '92 Addendum to the proceedings on Object-oriented programming systems, languages, and applications*, páginas 29–30, 1992. 1, 5

[Dea12] Nigel Deakin. Java message service. *Sun Microsystems*, 2012. 38

[Deh18] Zhamak Dehghani. How to break a monolith into microservices. https://martinfowler.com/articles/break-monolith-into-microservices.html, 2018. Online; último acesso 01-Jul-2018. 28, 29, 84, 104, 110, 111

[DGL+17] Nicola Dragoni, Saverio Giallorenzo, Alberto Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin e Larisa Safina. Microservices: Yesterday, today, and tomorrow. *Domain Language, Inc.*, 2017. 10, 11

[EBO+15] Neil Ernst, Stephany Bellomo, Ipek Ozkaya, Robert Nord e Ian Gorton. Measure it? manage it? ignore it? software practitioners and technical debt. *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015. 5, 6

[Eva03] Eric Evans. *Domain-Driven Design*. Addison-Wesley, 2003. 27, 29

[Eva13] Eric Evans. Getting started with ddd when surrounded by legacy systems. *Domain Language, Inc.*, 2013. 25, 26, 27

[Far16] Victor Farazdagi. Rest and long-running jobs. https://farazdagi.com/2014/rest-and-long-running-jobs, 2016. Online; último acesso 01-Ago-2018. 106

[FEB] Boleto registrado febraban. https://portal.febraban.org.br/pagina/3150/1094/pt-br/servicos-novo-plataforma-boletos. Online; último acesso 01-Ago-2018. 87

[Fie00] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. diploma thesis, Univerisity of California, Irvine, 2000. 106

[FL14] Martin Fowler e James Lewis. Microservices. http://martinfowler.com/articles/microservices.html, 2014. Online; último acesso 15-Out-2016. 10, 11, 14, 15, 19, 20, 21

[Fow03] Martin Fowler. Technicaldebt. http://www.martinfowler.com/bliki/TechnicalDebt.html, 2003. Online; último acesso 15-Out-2016.

[Fow04] Martin Fowler. Strangler application. http://www.martinfowler.com/bliki/StranglerApplication.html, 2004. Online; último acesso 15-Out-2016. 2, 25, 26

[Fow06] Martin Fowler. Continuous integration. http://martinfowler.com/articles/continuousIntegration.html, 2006. Online; último acesso 15-Out-2016. 13, 37

[Fow09] Martin Fowler. Technicaldebtquadrant. http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html, 2009. Online; último acesso 15-Out-2016. 6

[Fow10] Richardson maturity model. https://martinfowler.com/articles/richardsonMaturityModel.html, 2010. Online; último acesso 01-Ago-2018. 106

[Fow15] Martin Fowler. Monolithfirst. http://martinfowler.com/bliki/MonolithFirst.html, 2015. Online; último acesso 15-Out-2016. 22

[FSGVY15] Fernández-Sánchez, Juan Garbajosa, Carlos Vidal e Agustín Yagüe. An analysis of techniques and methods for technical debt management: A reflection from the architecture perspective. *Software Architecture and Metrics (SAM), 2015 IEEE/ACM 2nd International Workshop on*, páginas 22–28, 2015.

[GHVJ94] Erich Gamma, Richard Helm, John Vlissides e Ralph Johnson. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. 36

[GMNK16] Rajeev Gupta, Prabhulinga Manikreddy, Sandesh Naik e Arya K. Pragmatic approach for managing technical debt in legacy software project. *ISEC'16 Proceedings of the 9th India Software Engineering Conference*, 2016.

[GPEM09] Joshua Garcia, Daniel Popescu, George Edwards e Nenad Medvidovic. Toward a catalogue of architectural bad smells. *QoSA'09 Proceedings of the 5th International Conference on the Quality of Software Architectures*, páginas 146–162, 2009.

[Gra06] Jim Gray. A conversation with werner vogels. https://queue.acm.org/detail.cfm?id=1142065, 2006. Online; último acesso 15-Out-2016. 14, 89

[GRI+14] Isaac Griffith, Derek Reimanis, Clemente Izurieta, Zadia Codabux, Ajay Deo e Byron Williams. The correspondence between software quality models and technical debt estimation approaches. *Sixth International Workshop on Managing Technical Debt*, 2014. 5, 6

[HAM] Strangler application case studies. https://paulhammant.com/2013/07/14/legacy-application-strangulation-case-studies. Online; último acesso 01-Jul-2018. 25, 28

[HB16] Sara Hassan e Rami Bahsoon. Microservices and their design trade-offs: A self-adaptive roadmap. *Services Computing (SCC), 2016 IEEE International Conference on*, 2016.

[HIJ] Dependency hell in microservices and how to avoid it. https://www.linkedin.com/pulse/dependency-hell-microservices-how-avoid-nabil-hijazi/. Online; último acesso 01-Jul-2018. 22, 23

[HLH14] Johannes Holvitie, Ville Leppänen e Sami Hyrynsalmi. Technical debt and the effect of agile software development practices on it - an industry practitioner survey. *Sixth International Workshop on Managing Technical*, páginas 35–42, 2014. 37

[HRJ+16] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K. Reiter e Vyas Sekar. Gremlin: Systematic resilience testing of microservices. *IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016. 13

[HSS09] Carl Hinsman, Neeraj Sangal e Judith Stafford. Achieving agility through architecture visibility. *QoSA'09 Proceedings of the 5th International Conference on the Quality of Software Architectures*, 2009.

[IBD95] Charalambos Iacovou, Izak Benbasat e Albert Dexter. Electronic data interchange and small organizations: Adoption and impact of technology. *Management Information Systems Research Center, University of Minnesota*, 19:465 – 485, 1995. 35

[ISM] Challenges of micro-service deployments. http://techtraits.com/microservice.html. Online; último acesso 01-Jul-2018. 22

[JBO04] The jboss 4 application server guide. https://docs.jboss.org/jbossas/jboss4guide/r1/html, 2004. Online; último acesso 15-Out-2016. 34

[Jes14] John Jeston. *Business Process Management*. Routledge, 2014. 60

[KIS] Kiss principle. https://en.wikipedia.org/wiki/KISS_principle. Online; último acesso 01-Ago-2018. 99

[Kno16] Holger Knoche. Sustaining runtime performance while incrementally modernizing transactional monolithic software towards microservices. *ICPE'16 Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, 2016.

[KNOF13] Philippe Kruchten, Robert Nord, Ipek Ozkaya e Davide Falessi. Technical debt: Towards a crisper definition. *4th international workshop on managing technical debt*, 2013. 5, 6

[Kuz14] Michail Kuznetcov. Measuring archtectural technical debt. diploma thesis, Faculty of Science of Radboud University in Nijmegen, 2014.

[Leh80] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1980. 1, 25

[Leh96] Manny Lehman. Laws of software evolution revisited. *European Workshop on Software Process Technology*, páginas 108–124, 1996. 6, 35

[LI12] Jean-Louis Letouzey e Michel Ilkiewicz. Managing technical debt with the sqale method. *IEEE Software*, 29, 2012. 9

[LLA13] Zengyang Li, Peng Liang e Paris Avgeriou. Architectural debt management in value-oriented architecting. *Economics-Driven Software Architecture*, páginas 183–204, 2013. 37

[LLA15] Zengyang Li, Peng Liang e Paris Avgeriou. Managing technical debt in software-reliant systems. *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, 2015.

[LTV15] Alessandra Levcovitz, Ricardo Terra e Marco Tulio Valente. Towards a technique for extracting microservices from monolithic enterprise systems. *3rd Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM)*, páginas 97–104, 2015. 29

[MBC14] Antonio Martini, Jan Bosch e Michel Chaudron. Architecture technical debt: Understanding causes and a qualitative model. *Software Engineering and Advanced Applications (SEAA), 40th EUROMICRO Conference on*, 2014.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2, 1976. 9

[MCL17] Genc Mazlami, Jürgen Cito e Philipp Leitner. Extraction of microservices from monolithic software architectures. *IEEE International Conference on Web Services (ICWS)*, 2017. 29

[MGP+12] Isela Macia, Joshua Garcia, Daniel Popescu, Alessandro Garcia, Nenad Medvidovic e Arndt von Staa. Are automatically-detected code anomalies relevant to architectural modularity? *AOSD'12 Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, 2012.

[MH08] Carlos Matos e Reiko Heckel. Migrating legacy systems to service-oriented architectures. *Proceedings of the Doctoral Symposium at the International Conference on Graph Transformation*, 2008. 29

[MIC] Comunicação em uma arquitetura de microsserviço. https://docs.microsoft.com/pt-br/dotnet/standard/microservices-architecture/architect-microservice-container-applications/communication-in-microservice-architecture. Online; último acesso 01-Jul-2018. 22

[Mye97] Michael Myers. Qualitative research in information systems. *Management Information Systems Quarterly*, 1997. 40, 41

[Mye99] Michael Myers. Investigating information systems with ethnographic research. *Communications of the AIS*, 2, 1999. 40

[NAM] Naming convention. https://en.wikipedia.org/wiki/Naming_convention_(programming). Online; último acesso 01-Ago-2018. 37

[New15] Sam Newman. *Building Microservices*. O'Reilly, 2015. 10, 11, 12, 14, 15, 16, 17, 19, 20, 29, 34, 35, 72

[New16] Sam Newman. Desmistificando a lei de conway. https://www.thoughtworks.com/pt/insights/blog/demystifying-conways-law, 2016. Online; último acesso 15-Out-2016. 2, 14

[PA11] Danielle Pontes e Reginaldo Arakaki. Evolução de software baseada em avaliação de arquitetura de software. *XVII Congreso Argentino de Ciencias de la Computación*, páginas 780 – 788, 2011.

[Par72] David Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053 – 1058, 1972. 19

[Par94] David Parnas. Software aging. *ICSE '94 Proceedings of the 16th international conference on Software engineering*, páginas 279–287, 1994. 1, 6, 19, 30, 35, 52

[PdP16] Alessandro Pizzoleto e Antonio do Prado. An approach for analysis and flexibility of business rules in legacy systems. *SERP'16, 14th International Conference on Software Engineering Research and Practice*, 2016. 29

[PER] Walking the wire: Mastering the four balancing acts in microservices architecture. https://medium.com/systems-architectures/walking-the-microservices-path-towards-loose-coupling-few-pitfalls-4067bf5e497a. Online; último acesso 01-Jul-2018. 22, 23

[PZA+17] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis e Nicolai Josuttis. Microservices in practice, part 1: Reality check and service design. *Domain Language, Inc.*, 2017. 10

[RIC] Microservice architecture by chris richardson. http://microservices.io. Online; último acesso 01-Jul-2018. 23, 29, 109

[Ric14] Chris Richardson. Microservices: Decomposing applications for deployability and scalability. https://www.infoq.com/articles/microservices-intro, 2014. Online; último acesso 15-Out-2016. 15

[Ric16] Chris Richardson. Refactoring a monolith into microservices. https://www.nginx.com/blog/refactoring-a-monolith-into-microservices, 2016. Online; último acesso 15-Out-2016. 2, 26, 27, 29, 74, 84, 104

[RQRA16] Md Tajmilur Rahman, Louis-Philippe Querel, Peter Rigby e Bram Adams. Feature toggles: Practitioner practices and a case study. *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories*, 2016. 28

[RW06] Robert Rowlingson e Richard Winsborrow. A comparison of the payment card industry data security standard with iso17799. *Computer Fraud and Security*, 2006:16 – 19, 2006. 36

[SCG+12] Arndt Staa, Christina Chavez, Alessandro Garcia, Roberta Arcoverde e Isela Macia. On the relevance of code anomalies for identifying architecture degradation symptoms. *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, 2012. 37

[SF12] Pramod Sadalage e Martin Fowler. *NoSQL Distilled*. Addison-Wesley, 2012. 16

[SG11] Carolyn Seaman e Yuepu Guo. Measuring and monitoring technical debt. *Advances in Computers*, páginas 25–46, 2011. 1, 6

[Som10] Ian Sommerville. *Software Engineering*. Addison-Wesley, 2010. 37

[SPR] Spring.io. https://spring.io. Online; último acesso 15-Out-2016. 35, 85

[SQA] The sqale method definition document. http://www.sqale.org/download. Online; último acesso 01-Jul-2018. 9

[SSS16] Ganesh Samarthyam, Girish Suryanarayana e Tushar Sharma. Refactoring for software architecture smells. *Proceedings of the 1st International Workshop on Software Refactoring*, páginas 1–4, 2016. 37

[Ste10] Chris Sterling. *Managing software debt: building for inevitable change*. Addison-Wesley, 2010. 6

[TAV13] E. Tom, A. Aurum e R. Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86:1498 – 1516, 2013. 1, 7

[Tho15] Johannes Thones. Microservices. *IEEE Software*, 32:116 – 116, 2015.

[TL18] Davide Taibi e Valentina Lenarduzzi. On the definition of microservice bad smells. *IEEE Software*, 35, 2018. 22, 23, 24

[UHU03] Elisabeth Umble, Ronald Haft e Michael Umble. Enterprise resource planning: Implementation procedures and critical success factors. *European Journal of Operational Research*, 2003. 38

[VAC] Strangler application pattern and anti-patterns. https://pt.slideshare.net/xpmatteo/strangler-application-patterns-and-antipatterns. Online; último acesso 01-Jul-2018. 24, 25, 26, 28, 110

[VGC+15] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas e S. Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. *Computing Colombian Conference (10CCC)*, 2015.

[VW12] Alvaro Videla e Jason Williams. *RabbitMQ in Action*. Manning, 2012. 17, 62

[Wal98] J. Waldo. Remote procedure calls and java remote method invocation. *IEEE Concurrency*, 6:5 – 7, 1998. 37

[WCKD11] Sunny Wong, Yuanfang Cai, Miryung Kim e Michael Dalton. Detecting software modularity violations. *ICSE'11, 33rd International Conference on Software Engineering*, 2011.

[WPR10] Jim Webber, Savas Parastatidis e Ian Robinson. *REST in Practice*. O'Reilly, 2010. 15, 37

[Yam17] Aiko Yamashita. Modernization from a maintenance process perspective: Challenges and lessons learned. *Journal of Advances in Information Technology*, 2017. 2, 24, 30, 31

[Yin08] Robert K. Yin. *Case Study Research: Design and Methods*. Sage Publications, 2008. 40

[Zay17] Mikulas Zaymus. Decomposition of monolithic web application to microservices. diploma thesis, JAMK University of Applied Sciences, 2017. 24, 29

[Zim15] Olaf Zimmermann. Architectural refactoring. *IEEE Software*, 32:26 – 29, 2015. 24, 25