

RODRIGO AGUIAR ORDONIS DA SILVA

Abordagem de desenvolvimento e sustentação de  
*software* orientado a negócio

São Paulo  
2020

RODRIGO AGUIAR ORDONIS DA SILVA

**Abordagem de desenvolvimento e sustentação de  
*software* orientado a negócio**

Trabalho apresentado à Escola Politécnica  
da Universidade de São Paulo para a con-  
clusão do MBA de Transformações Digitais.

São Paulo  
2020

RODRIGO AGUIAR ORDONIS DA SILVA

**Abordagem de desenvolvimento e sustentação de  
*software* orientado a negócio**

Trabalho apresentado à Escola Politécnica  
da Universidade de São Paulo para a con-  
clusão do MBA de Transformações Digitais.

Área de concentração:

3141 – Engenharia de Computação

Orientador:

Reginaldo Arakaki

A minha mãe e meu pai, por todo  
o apoio, amor e carinho.

# AGRADECIMENTOS

Ao meu orientador, professor Reginaldo Arakaki, pela atenção, pelo apoio e principalmente pelos conselhos durante a realização deste trabalho.

*“Truth can only be found in one place:  
the code.”*

- [Martin 2008]

# RESUMO

Este trabalho apresenta uma abordagem de como realizar o desenvolvimento e a sustentação de *software* centrado no negócio. A abordagem especifica práticas que devem ser realizadas com o intuito de produzir um sistema com escalabilidade onde o negócio é utilizado para aprendermos como desenvolver o sistema e assegurar a qualidade do que está sendo desenvolvido e o que foi desenvolvido é utilizado para aprendermos como o negócio funciona e como podemos melhorá-lo. É aplicado uma forma de aprendizado contínuo, o processo de desenvolvimento ocorre não apenas para produzir um sistema, mas para descrever o negócio e aplicar formas de assegurar que o produto está aderente. É apresentado como especificar as funcionalidades, identificar riscos e descobrir quais ações tomar com os riscos identificados se baseando em métricas e testes realizados durante o processo de desenvolvimento e de sustentação. A abordagem também apresenta como agir quando um erro em produção é identificado e o que devemos fazer para mitigar ou solucionar o erro garantindo que ele não ocorra novamente ou que seja solucionado de forma mais rápida e eficaz.

**Palavras-Chave** – Qualidade. Negócio. Escalabilidade. Usuário. Testes. Testes Automatizados. Produto. Sistema. Funcionalidade.

# ABSTRACT

This work presents an approach to develop and sustain software based on the business. The approach specifies a set of practices that should be realized to produce software with scalability where the business is used to learn about how to develop the software and assure your quality and, the software is used to learn how the business work and how to improve him. Is applied a form of continuous learning, the process of development occurs don't just to produce software but to describe the business and how apply ways to assure the product is adherent. It shows ways to specify the functionalities, identify the risks and find what actions take with the identified risks based on metrics and tests realized on the software in the development and sustain processes. The approach presents too how to act when a problem is identified in production and what we should do to resolve and assure the error doesn't occur again or that he be resolved faster and effectively.

**Keywords** – Quality. Business. Scalability. User. Tests. Automated Tests. Product. Software. functionality.



# LISTA DE FIGURAS

1	Estrutura de funcionalidade . . . . .	40
2	Fluxo de ambientes . . . . .	42
3	Fluxo de ambientes com CHAOSPROD . . . . .	46
4	Fluxo de entrega . . . . .	48

## LISTA DE TABELAS

1	Definição da classificação de funcionalidades . . . . .	28
2	Exemplo de levantamento de funcionalidades . . . . .	31
3	Exemplo de levantamento dos riscos e dependências por funcionalidade . .	32
4	Definição da classificação de requisitos . . . . .	34
5	Exemplo de classificação de requisito funcional . . . . .	35
6	Exemplo de classificação de requisito não-funcional . . . . .	35
7	Exemplo de classificação de requisito inverso . . . . .	35
8	Exemplo de especificação de critérios de aceite . . . . .	39
9	Exemplo de análise de limite de sistema — limite de processamento . . . .	45
10	Exemplo de análise de limite de sistema — limite de requisições . . . . .	45
11	Especificação para estruturação de estratégias . . . . .	47
12	Especificação para documentação de resolução de problemas . . . . .	52

# SUMÁRIO

<b>Parte I: INTRODUÇÃO E MOTIVAÇÕES</b>	<b>11</b>
<b>1 Introdução</b>	<b>12</b>
1.1 Motivações . . . . .	14
<b>2 Metodologia de pesquisa</b>	<b>16</b>
<b>3 Objetivo</b>	<b>17</b>
<b>Parte II: FUNDAMENTOS CONCEITUAIS</b>	<b>18</b>
<b>4 Manifesto Ágil</b>	<b>19</b>
<b>5 <i>DevOps</i></b>	<b>21</b>
5.1 <i>Continuous Integration</i> . . . . .	22
5.2 <i>Continuous Delivery</i> . . . . .	23
5.3 <i>Test-Driven Development</i> . . . . .	23
5.4 Escalabilidade . . . . .	25
<b>Parte III: PROPOSTA</b>	<b>26</b>
<b>6 Como criar produtos com qualidade?</b>	<b>27</b>
6.1 Assumindo riscos . . . . .	28
6.2 Entendendo os requisitos de uma funcionalidade . . . . .	32
6.3 Arquitetura evolutiva . . . . .	36
6.4 Estruturando o fluxo de entrega . . . . .	38
6.5 Automatizando testes de qualidade . . . . .	42

<b>7</b>	<b>Como adquirir escalabilidade?</b>	<b>43</b>
7.1	Descobrimos os limites do sistema . . . . .	43
7.2	Desenvolvendo estratégias de escalabilidade . . . . .	46
<b>8</b>	<b>Como manter o seu produto?</b>	<b>49</b>
8.1	Ouvindo os usuários . . . . .	50
8.2	Aprendendo com os erros . . . . .	51
8.3	Testes automatizados como ferramenta de aprendizado . . . . .	52
8.4	Utilizando os dados adquiridos . . . . .	53
<b>9</b>	<b>Resultados da proposta</b>	<b>56</b>
	<b>Parte IV: CONCLUSÃO</b>	<b>58</b>
<b>10</b>	<b>Resultados em relação ao objetivo</b>	<b>59</b>
<b>11</b>	<b>Trabalhos futuros</b>	<b>60</b>
	<b>Referências</b>	<b>61</b>

# PARTE I

## INTRODUÇÃO E MOTIVAÇÕES

# 1 INTRODUÇÃO

Acreditamos que um produto escalável e uma equipe que se preocupe em aprender com erros e manter o sistema sempre disponível, pode criar produtos de alta qualidade e grande valor. Como já apresentado no Site Reliability Engineering [Beyer et al. 2016], devemos esquecer a cultura de culpar as pessoas por erros cometidos e adquirir uma cultura de aprendizado, utilizar do erro para entender como ele ocorreu e assim produzir um produto com maior qualidade e adquirir o conhecimento para que o erro não ocorra novamente no nosso produto ou em outros. Adquirir essa maturidade e esse *mindset* é um processo, não é nada imediato, e é necessário entender em qual momento a equipe está e como implementar estes processos.

De acordo com o The DevOps Handbook [Kim et al. 2016], o processo de transformação é dividido em três partes, o fluxo, *feedback* e, aprendizado contínuo e experimentação. A primeira parte é sobre definirmos o nosso fluxo de trabalho, os nossos processos e com isso automatiza-los com a criação do *pipeline*, realização de testes automatizados, estruturação da arquitetura do *software* e desta forma, conseguindo realizar entregas de baixo risco. A segunda parte é adquirir a prática do *feedback*, cujo objetivo é o de aprender com os nossos erros, tentar solucionar o mais rápido possível e logo após, realizar uma reunião com todos os envolvidos e interessados para entender o que ocorreu, quais foram as medidas tomadas e o que fazer para não ocorrer novamente, sempre lembrando que o objetivo não é encontrar um culpado, mas sim aprender com o ocorrido. Outro fator importante é o de investir tempo em melhorar a telemetria que temos sobre o sistema, trabalhar em quais informações precisamos para supervisionar o funcionamento do sistema e evitarmos que problemas ocorram, como, por exemplo, manter o controle do desempenho de cada servidor utilizado para o funcionamento da aplicação, quando um deles estiver com o nível de processamento abaixo do esperado, podemos analisar se ele não está sendo aproveitado, e assim realocar recursos para melhorar o desempenho do sistema, ou se ele está apresentando sinais de mau funcionamento. A terceira parte é o de aprendizado contínuo e experimentação, ela nos incentiva a realizar experimentos no nosso produto de modo a adapta-lo e gerarmos conhecimento para o negócio, um exemplo de experimento,

unificar três telas em uma só e ver como o usuário reage a esta mudança, dependendo do resultado, podemos aumentar a produtividade do usuário no sistema, ou descobrir que ele isso o deixou mais confuso, em ambos os casos, aprendemos sobre o comportamento do nosso usuário e isso pode ser utilizado para futuras demandas e resoluções de erros. Outro ponto importante sobre essa parte é a de gerar conhecimento, assim que algo é descoberto, devemos apresentar para todos da equipe e caso seja algo que possa ser utilizados em outros projetos, escalar esse aprendizado para a empresa, onde devemos considerar demonstrar tudo como conhecimento, seja uma descoberta comportamental do usuário, a resolução de algum defeito, um erro cometido durante o projeto, qualquer aprendizado deve ser considerado e, devido a isso, não podemos culpar as pessoas por erros cometidos. Quando culpamos alguém, inibimos a pessoa de demonstrar o erro, conseqüentemente, perdemos uma importante fonte de aprendizagem, desta forma, perdemos a oportunidade de melhorarmos nosso produto, nossa equipe e nosso trabalho.

Um dos problemas mais urgentes em nosso sistema e que precisamos procurar evitar o máximo possível são os de disponibilidade. Um sistema que não está disponível, perde credibilidade, o negócio é impactado de forma direta e os usuários que necessitam dele, acabam se estressando por não poder fazer nada. Lidar com a disponibilidade do sistema está diretamente relacionada com assumir riscos, entender que haverá momentos críticos no sistema que precisamos nos preparar e que sempre há a hipótese de falharmos e do sistema cair. Precisamos através das necessidades do negócio e da utilização do usuário aprender como podemos manter a disponibilidade. Primeiramente, disponibilidade não pode ser considerado simplesmente como “no ar ou fora do ar”, uma funcionalidade que demore para ser executada, pode motivar o usuário a não utiliza-la, o que prova que ela não está disponível, vamos supor, que estamos em um sistema que sejam realizadas negociações, o usuário está negociando com um cliente e propõe um desconto para o produto que ele está tentando vender, nosso sistema tem uma função de simulação de desconto, então nosso vendedor logo tenta simular, porem, como é um produto grande, essa simulação está sendo processada já a cinco minutos, e o cliente já está cansado de esperar, como o ele não teve o resultado em tempo hábil e o vendedor dependia desse valor para continuar a negociação, decidiram continuar a conversa outro dia, já com o valor em mãos. Após este episódio, nosso vendedor perdeu a confiança no sistema e decidiu não utilizar mais a funcionalidade de negociação, embora a simulação tenha funcionado muito bem para produtos pequenos, quando o nosso vendedor mais precisou, ele não atendeu as expectativas. Cinco minutos foi o suficiente para não disponibilizar essa funcionalidade para esse usuário, que como está aborrecido, vai passar esta frustração para outros usuários. Se não tivermos a informação de que há este problema no sistema, seja por *feedback* ou por

alguma métrica, jamais saberemos deste erro e com o passar do tempo esta funcionalidade será esquecida.

Outro ponto a ser considerado é o preço do *software*, podemos manter nosso sistema sempre disponível se tivermos uma grande quantidade de servidores, armazenamento infinito, e vários domínios de redundância, o custo para manter todos esses recursos é inviável, precisamos sempre controlar o que o sistema precisa e o que a empresa pode pagar. Controlar os custos é uma tarefa desafiadora, precisamos ver qual o retorno que estamos recebendo com a aplicação e quais são os riscos que podemos enfrentar. Não adianta montar um sistema indestrutível se ele não gera valor para o negócio, e não adianta deixar o sistema vulnerável para controlar custos, a queda de um produto acaba por descredibilizar a empresa, pois este erro se torna um símbolo de falta de qualidade. O que devemos fazer é montar um produto escalável, conseguir controlar o quanto de recurso disponibilizar, saber quais são as épocas em que precisamos do sistema funcionando, quais são as funcionalidades mais críticas e quais riscos podemos aceitar. Tomar as nossas decisões com base nos riscos, nos dá segurança e preparo para as mais adversas situações, como planejamos que pode acontecer erros, conseguimos nos antecipar e solucionar estes erros o mais rápido possível, gerando pouco impacto para o negócio. Sabendo onde estão os pontos fracos do nosso produto, podemos prever que um problema ocorra antes que ele aconteça acompanhando as métricas diariamente, quando os primeiros sinais de que vai ocorrer um erro surgirem, podemos reavaliar os riscos e assumir uma postura de mitigação ou solução. Admitindo que nosso produto não é perfeito e aceitando falhas, podemos trabalhar de forma a melhorar o nosso produto continuamente.

## 1.1 Motivações

Nossas motivações se definem em criar e manter produtos escaláveis, possibilitando o controle de seu custo com base na utilização dos usuários e na situação da empresa, demonstrando como utilizar a interação dos usuários para aprender como melhorar e engajar a criação de testes automatizados como um processo de aprendizagem e de controle de qualidade.

Pretendemos demonstrar técnicas para aprender com os erros possibilitando que o seu produto evolua e que os aprendizados auxiliem na criação de outros produtos. Desta forma, a busca por valor não se limita a apenas a um sistema, beneficiando o negócio com outras visões e ampliando a inteligência de mercado na empresa. A abordagem apresentada não tem como objetivo criar produtos perfeitos, sem erros, e que vão concluir todos



os objetivos da empresa, mas pretendemos utilizar do produto e da interação do usuário para montar *software* de qualidade, aproveitando o aprendizado obtido para auxiliar na evolução do negócio, gerar valor para a empresa e auxiliar na criação de novos produtos.

## 2 METODOLOGIA DE PESQUISA

Para a realização deste trabalho foi utilizado livros, artigos, *papers* e notas técnicas de criação de sistemas com qualidade. Procuramos por trabalhos que descrevessem o que é qualidade de *software* e como construir produtos com qualidade. Durante a pesquisa foi identificado a importância dos requisitos não-funcionais o que nos levou a procurar por trabalhos que apresentassem e demonstrassem técnicas para assegurar que eles fossem cumpridos.

Foi utilizado como ferramenta de pesquisa o Google Scholar, ResearchGate, Software Engineering Institute da Carnegie Mellon University, Institute of Electrical and Electronics Engineers e a biblioteca digital da Association for Computing Machinery.

### 3 OBJETIVO

Este trabalho tem como objetivo apresentar uma abordagem para a construção de *software* com qualidade, focando em sua escalabilidade. Através da sustentação do produto, definirmos melhorias baseadas nas métricas criadas para supervisionar a saúde do sistema e para verificar como está sendo a utilização do usuário e com estes dados, identificamos as mudanças no negócio e conseguimos tornar as funcionalidades criadas mais aderentes a essas mudanças.

# **PARTE II**

## **FUNDAMENTOS CONCEITUAIS**

## 4 MANIFESTO ÁGIL

No começo do desenvolvimento de *software*, a metodologia mais utilizada para realizar a produção de *software* era a *waterfall*, que consiste em um desenvolvimento faseado, primeiro especificamos todo o sistema, depois desenvolvemos, testamos e por fim colocamos em produção. Muitas vezes quando terminávamos de especificar o sistema todo, as necessidades do negócio havia mudado, gerando uma mudança no desenho do sistema e fazendo a equipe voltar para a fase de especificação. O que também acontecia, era que durante o desenvolvimento, as necessidades do negócio mudavam novamente e a equipe precisava fazer a escolha de voltar para a fase de desenho ou continuar o da forma que foi desenhado. Esta decisão não era fácil e gerava muitos conflitos entre a área de negócio e a equipe de desenvolvimento, pois muito do que foi realizado teria que ser refeito.

Para solucionar este problema em fevereiro de 2001 foi publicado o manifesto ágil, um conjunto de valores e princípios para auxiliar que o desenvolvimento de *software* seja realizado de forma rápida e adaptativa as necessidades do negócio, onde é mais enfatizado a necessidade de comunicação entre as pessoas do que o planejamento excessivo, a adaptação a mudanças ao invés de soluções definitivas e a priorização da realização do produto ao invés de seguir processos e ferramentas [manifesto for agile software development].

Conforme apresentado no Agile Principles, Patterns, and Practices in C# [Martin e Martin 2007], o manifesto ágil valoriza as pessoas que estão desenvolvendo o *software*, mais do que processos e contratos, não é necessário ter desenvolvedores extremamente experientes, mas sim uma equipe que esteja unida e que seja multidisciplinar, ou seja, que além de desenvolvedores, é necessário manter as pessoas com entendimento sobre o negócio para que durante o desenvolvimento, dúvidas e mudanças no negócio possam ser explicadas e a equipe conseguir desenvolver um produto de maior valor para o negócio. É importante notar, que sob a visão ágil, mudanças nos requisitos do projeto são bem-vindas, pois o produto que está sendo construído é mais importante do que documentos e processos, é mais importante atender as necessidades do negócio do que seguir um planejamento, ou seja, embora seja importante planejar, precisamos estar preparados para mudanças e imprevistos. Desta forma, os desenvolvedores devem construir um sistema

que seja fácil de alterar, possa ser alterado conforme as necessidades do negócio.

Com esses valores e princípios, a produção de *software* se tornou mais rápida, o que antes demorava anos para ser produzido começou a ser realizado em meses, pois o sistema começou a ser desenhado conforme era desenvolvido, os desenvolvedores não desenhavam primeiro todo o sistema para só depois começar o desenvolvimento, ao invés, eles já começavam a desenvolver sem se preocupar muito com a documentação, e a cada mudança que ocorre, não é mais necessário desenhar novamente toda a solução, mas sim adaptar o que já foi desenvolvido.

As metodologias advindas das práticas ágeis, promovem o conceito de iterações, que consiste, em um curto espaço de tempo, normalmente semanas, seja produzido *software* em produção, para que desta forma consigamos avaliar os impactos do que foi contruído para o negócio em menos de um mês e seja possível já mensurar o valor que está sendo gerado. Como o tempo de desenvolvimento é curto, está previsto que vamos errar diversas vezes, as soluções que vamos produzir não vão atender todas as necessidades do negócio, porém é através do aprendizado destes erros que vamos entendendo as necessidades do negócio e corrigindo o sistema para que ele se torne cada vez mais aderente a essas necessidades, temos que errar o mais rápido possível para que possamos aprender com os nossos erros e acertar o mais rápido possível.

## 5 *DEVOPS*

Através dos valores e das práticas ágeis e com o surgimento de novas tecnologias, mais práticas foram sendo criadas e novas ferramentas surgiram para auxiliar o desenvolvimento de *software*, já com a perspectiva ágil. Desta forma, surgiu o conjunto de práticas denominado *DevOps*, que consiste em auxiliar a equipe de desenvolvimento e de operação a trabalharem de forma unida [Kim et al. 2016]. Conforme o tempo de desenvolvimento dos produtos foi encurtando, surgiu a necessidade de realizar o *deploy* em produção, diversas vezes ao longo do dia, pois como trabalhamos de forma adaptativa e valorizamos o produto sendo entregue de forma rápida, para gerar valor o mais rápido possível, a infraestrutura montada e a equipe de operações, precisam estar preparados e embora seja importante realizar entregas rápidas, ainda é necessário avaliar a qualidade do que está sendo entregue, para evitar erros em produção e não tornar o que deveria gerar valor, se tornar um problema para o negócio.

As práticas de *DevOps*, nos fornece um conjunto de ações que podemos realizar para que consigamos evitar que os erros aconteçam e que caso venham a acontecer, teremos as informações necessárias para avaliar o ocorrido e resolver da melhor forma o mais rápido possível, neste capítulo vamos abordar rapidamente as principais práticas, durante a proposta elas serão exploradas mais detalhadamente.

Entre as principais práticas está o de *code review*, que consiste que antes de um *deploy* ocorrer, o código que foi produzido será verificado em busca de falhas, se está seguindo as boas práticas, erros de lógica, brechas de segurança, ou seja, qualquer coisa que possa causar algum erro em produção. Outra prática importante é a de estratégias de ambientes, quando estamos produzindo um *software* não é aconselhável realizar alterações direto em produção sem antes ela ter sido testada e o código ter sido avaliado, desta forma, é importante ter um ambiente separado de produção, onde seja possível os desenvolvedores realizarem testes das funcionalidades que estão sendo produzidas, para assegurar a qualidade do que foi desenvolvido e conseguir avaliar se está aderente a necessidade do negócio. O *DevOps* apresenta que o processo de desenvolvimento deve levar em consideração todos os processos de verificação de qualidade e efetivamente o *deploy* em produção. A equipe

de operação, que é responsável por realizar o *deploy*, faz parte da equipe de desenvolvimento e o tempo que será utilizado para realizar o *deploy* considerando o *code review*, alterações na infraestrutura do sistema, o tempo do *deploy*, análises de segurança, entre outro qualquer processo que deva ser realizado para assegurar um *deploy* com qualidade, verificando a presença de possíveis *bugs*, deve ser considerado durante o planejamento da iteração, e caso venha a aparecer algum ponto que deve ser ajustado pelos desenvolvedores, eles devem ser notificados o mais rápido possível, para que seja realizado o ajuste e o *deploy* possa acontecer.

Para otimizar esta comunicação, e facilitar adicionar estes processos de verificação do código possibilitando o *deploy*, foi criado os conceitos de CI (*Continuous Integration*) e CD (*Continuous Delivery*).

## 5.1 *Continuous Integration*

Quando estamos trabalhando em equipes, é natural que em determinados momentos seja necessário que dois desenvolvedores tenham que alterar o mesmo arquivo, se não trabalharmos utilizando o conceito de CI, só vamos descobrir esse conflito quando um dos dois desenvolvedores perceberem que o seu código foi sobrescrito. Uma das principais ferramentas utilizadas para auxiliar no CI são os sistemas de controle de versões (Git, Mercurial, SVN, etc.) que consegue criar versões do seu código em sua máquina local e essas versões podem ser compartilhadas para um servidor, chamado de repositório (GitHub, Bitbucket, GitLab, etc.) e compartilhado com outras máquinas, apresentando os conflitos nos arquivos e possibilitando que o desenvolvedor realize as mudanças necessárias [Chacon e Straub 2014].

Com a utilização de um sistema de controle de versões, devemos separar o nosso trabalho em *branches* [Kim et al. 2016], que possibilita separar as alterações que realizamos em outra linha de mudanças, onde o que for realizado nesta *branch* não terá impacto na linha principal do código até que seja solicitado realizar o *merge* (juntar às duas *branches*) [Chacon e Straub 2014], desta forma garantimos maior controle sobre o que está sendo implementado em nossas funcionalidades, pois cada *branch* deve tratar somente de uma implementação, podendo ser uma funcionalidade nova ou uma melhoria em uma funcionalidade já existente. Com as *branches*, somente disponibilizamos que o nosso código seja integrado com os desenvolvedores que estão atuando em outras *branches*, depois de finalizado os nossos desenvolvimentos e realizar o *merge* para a *branch* principal, desta forma evitamos integrar um código que pode impactar o desenvolvimento de outras pes-



soas na nossa equipe por cause dele estar inacabado.

## 5.2 *Continuous Delivery*

Agora que conseguimos integrar o nosso código de forma que os conflitos são apresentados para nós e as implementações são realizadas em *branches* separadas, conseguimos aplicar essas implementações de forma contínua [Kim et al. 2016]. Com o conceito de CD, podemos realizar a criação de um *pipeline*, para cada implementação, conseguimos realizar diversos testes, verificar cobertura de código e fazer o *deploy* de forma automatizada, diminuindo as ações que a equipe teria que executar para cada demanda, consequentemente otimizando o tempo de entrega [Humble e Farley 2010]. Quando aplicamos CD para o nosso sistema, para cada automatização, precisamos implementar uma ferramenta diferente, por exemplo, uma ferramenta de gerenciamento de dependências (Nexus), para verificarmos as dependências entre as implementações, uma ferramenta para a elaboração de testes automatizados (Selenium), para cada necessidade que o seu produto necessite é interessante pesquisar alguma ferramenta para realizar esta automatização e aplicá-la ao seu *pipeline*.

Com o auxílio do *pipeline* podemos aplicar estratégias de criação de ambientes separados para o processo de desenvolvimento, com o intuito de possibilitar que os desenvolvedores consigam testar as funcionalidades que já eles estão desenvolvendo e os *testers* testarem as funcionalidades que já foram desenvolvidas sem que as alterações dos desenvolvedores atrapalhem, é interessante criar ambientes separados para cada etapa do fluxo de entrega. Cada projeto possui uma combinação diferente de ambientes, cada uma sendo a estratégia tomada pela equipe do projeto, onde o mais importante nessa criação é a de realizar entregas rápidas garantido a qualidade daquilo que foi desenvolvido e caso venha a acontecer algum impacto negativo na implementação, podemos utilizar os ambientes para identificar e mitigar os erros antes que entre em produção [Kim et al. 2016].

## 5.3 *Test-Driven Development*

Para assegurar a qualidade do nosso *software*, é necessário realizar diversos testes antes que uma funcionalidade seja disponibilizada em produção, eles devem ser considerados durante o período de desenvolvimento, todas as nossas funcionalidades devem ser

testadas e os testes não devem somente cobrir o código criado, mas deve garantir que os requisitos da funcionalidade estejam sendo cumpridos, desta forma, verificamos se o código está funcionando de forma automatizada [Martin 2008].

Pensando nisso, foi criada a prática do TDD (*test-driven development*) [Beck 2002], que consiste em primeiro criar os testes, ver eles falharem, pois a funcionalidade ainda não foi desenvolvida, e conforme o desenvolvimento, ir verificando se os testes estão passando e se é necessário mais teste. Conforme vamos criando cenários de teste e verificando o resultado do nosso desenvolvimento, nos deparamos com diversos requisitos, e para garantir que eles estão sendo cumpridos, devemos adaptar os nossos testes para que estes requisitos sejam verificados e caso haja uma mudança nos requisitos, podemos alterar algum teste que desenvolvemos e verificar se caso o teste continue passando, a funcionalidade desenvolvida está contemplando a mudança e caso o teste falhe, teremos que ajustar o nosso desenvolvimento.

A utilização do TDD garante segurança na hora do desenvolvimento, como confiamos nos testes que desenvolvemos, se realizarmos alguma alteração no código, conseguimos verificar nos testes se nossa alteração gerou um erro em alguma funcionalidade e com base nos testes, conseguimos tratar este erro antes de liberar esta alteração para os membros da equipe, já que estamos utilizando um sistema versionador de código. Os testes também conseguem demonstrar como o código testado funciona [Martin e Martin 2007], como os requisitos estão sendo representados através dos testes que realizamos, a descrição de como funciona a nossa funcionalidade, quais são os parâmetros que ela recebe e qual é o seu objetivo, pode ser compreendido através dos testes montados, com eles, conseguimos enxergar o cenário que a funcionalidade está sendo executada e o que deve acontecer com o sistema após a sua execução.

A execução dos testes deve ser automatizada, um trabalho executado pelo *pipeline*, desta forma conseguimos que todas as nossas funcionalidades sejam testadas e os erros de cada alteração que realizamos no código, caso aconteça, seja verificado e corrigido para rodar novamente o *pipeline* e verificar se a nossa implementação está passando nos testes após o ajuste [Humble e Farley 2010]. Cada classe desenvolvida deve ter uma classe de teste em que devemos verificar se o objetivo da classe está sendo atingido e os requisitos que estamos trabalhando na classe estão sendo cumpridos. Para aumentar a quantidade de testes automatizados podemos criar testes que executem automaticamente operações no sistema, simulando cliques na tela, desta forma podemos construir *scripts* de testes, de forma a validar que o que foi desenvolvido está funcionando da maneira esperada, simulando a utilização do usuário.

## 5.4 Escalabilidade

Agora que implementamos o nosso sistema com qualidade em produção, devemos nos assegurar que o nosso sistema continue em funcionamento. Como a utilização de um sistema varia muito com a época em que ele está sendo utilizado, muitas vezes não conseguimos prever quais os impactos que uma grande quantidade de acessos pode causar, um ataque a segurança do nosso sistema, um servidor que se encontrou fora do ar, diversos imprevistos podem ocorrer em produção que pode impactar negativamente o nosso produto.

Para nos prepararmos para esses imprevistos, devemos nos preocupar com que o nosso sistema tenha escalabilidade, esteja disponível, se mantenha seguro e que sua manutenção seja simples de ser realizada, mas para isso devemos investir na sustentação do nosso sistema. Através da utilização de métricas, conseguimos verificar como o nosso sistema está funcionando e onde está apresentando sinais de problemas [Kim et al. 2016], desta forma conseguimos nos preparar para problemas que possam ocorrer em produção. Os problemas que ocorrerem em produção e os dados que coletarmos no nosso sistema, deverão servir como meio de aprendizagem para que possamos descobrir como nos preparar para eventuais ocorrências em produção e manter nosso sistema disponível e confiável [Beyer et al. 2016].

# PARTE III

## PROPOSTA

## 6 COMO CRIAR PRODUTOS COM QUALIDADE?

Quando produzimos um *software*, focamos sempre nas funcionalidades que o sistema deve conter, pretendemos entregar o mais rápido possível e respeitar as datas alinhadas com o negócio. Porém, na pressa de colocar as funcionalidades em produção, desconsideramos os requisitos não-funcionais e por mais que a funcionalidade tenha sido entregue conforme o especificado, o usuário não consegue utilizar, conseqüentemente, está entrega não está gerando valor.

Como assim, os requisitos foram atendidos, mas o usuário não consegue utilizar? Como ignoramos os requisitos não-funcionais, o usuário pode estar sofrendo por diversos problemas, por exemplo, um *layout* confuso, um desempenho baixo, o sistema está forçando ele a realizar uma tarefa que ele não está acostumado a fazer, ou pelo menos não no momento em que apareceu no sistema. Como entregamos visando a data da entrega e não o valor para o usuário, aceitamos o risco de entregar algo que o usuário não vê valor, e por diversas vezes cometemos erros. Ao testar uma funcionalidade, não costumamos ter a visão total do negócio, focamos na funcionalidade, então questões como tempo de resposta, ordem lógica no processo, localização das informações, nos passam despercebidos, não estamos utilizando o sistema diariamente para entender estas questões sozinhos, e o negócio muda frequentemente, então quando finalmente entendemos a realidade do usuário, ela já mudou, e voltamos a estar suscetíveis ao erro.

Como não atendemos a expectativa do usuário, e a funcionalidade não está gerando valor, implementamos diversas melhorias, todas o mais rápido possível, o que acaba gerando *bugs*, como erramos uma vez, precisamos corrigir o erro o mais rápido possível, o que nos faz ignorar novamente requisitos não-funcionais e, no que lhe concerne, ignoramos questões de arquitetura de *software* o que acaba deixando o sistema complexo, dificulta a sua manutenção, e deixa o sistema suscetível a erros.

Não queremos mais cometer estes erros, queremos criar o produto perfeito que será totalmente aderente ao usuário e que irá gerar valor para o negócio.

O primeiro passo para gerar um produto de qualidade, que auxilia a empresa a concluir

os seus objetivos e que traga valor ao negócio e entender que não existe produto perfeito. Somos humanos, erramos constantemente, realizamos escolhas erradas, nos equivocamos. Para realizar uma entrega de pouco risco e extremamente assertiva, é necessário muito tempo de planejamento e de estudo, como, por exemplo, a construção de um avião, ou de um prédio, ou de um equipamento hospitalar, este produto não pode ter falhas, pois caso tenha, é um risco para a vida das pessoas. Porém, nosso negócio é mais dinâmico, as necessidades mudam com mais frequência do que as necessidades de um avião, se utilizarmos tanto tempo para planejar como podemos seguir as mudanças do negócio?

## 6.1 Assumindo riscos

Para conseguirmos acompanhar o negócio e gerar valor para a empresa, devemos escolher os riscos que vamos enfrentar. Mesmo um avião encara riscos no seu projeto, por isso que é implementado diversas redundâncias nas funcionalidades mais importantes. Devemos pensar de forma semelhante, qual o objetivo do nosso projeto? Qual é sua funcionalidade mais importante? Quais são os riscos que vamos encarar?

Com base nisso, podemos focar a maior parte de nosso tempo no *core* do produto, descobrindo o objetivo do sistema, podemos alinhar com os objetivos da empresa, assim gerando o retorno esperado. Para realizar o levantamento da funcionalidade é importante considerar o nome da funcionalidade, a sua importância, os objetivos em que ela vai auxiliar a alcançar, os riscos que vamos enfrentar com ela e as dependências para disponibilizar a funcionalidade ao usuário.

<b>Funcionalidade</b>	Nome da funcionalidade que será desenvolvida.
<b>Importância</b>	A importância que a funcionalidade representa para o negócio, separa entre alta, média e baixa.
<b>Objetivos</b>	Os objetivos que esta funcionalidade irá auxiliar a alcançar.
<b>Riscos</b>	Quais os riscos que essa funcionalidade pode gerar para o negócio.
<b>Dependências</b>	Quais são as dependências para a utilização e desenvolvimento dessa funcionalidade.

Tabela 1: Definição da classificação de funcionalidades

Toda funcionalidade deve ter um nome que represente o que será desenvolvido, pois, durante o desenvolvimento, é através do nome que os desenvolvedores vão se comunicar, sem um nome claro, que represente bem o negócio, os desenvolvedores não vão conseguir se comunicar de forma clara para entender as necessidades do negócio. Termos e

nomenclaturas devem ser estabelecidas, para que o usuário entenda como o sistema deve ser utilizado e para os desenvolvedores compreenderem como o sistema será utilizado. Através desta comunicação, a equipe de desenvolvimento consegue extrair os requisitos de forma fácil e formular um padrão de qualidade.

É necessário colocar a importância da funcionalidade que será desenvolvida, pois, com base nela podemos entender a ordem que vamos iniciar o desenvolvimento e quais riscos podemos enfrentar e através da importância, definir o rigor dos testes que devemos realizar no desenvolvimento da funcionalidade e o padrão de qualidade para cada nível de importância. Embora coloquemos somente três níveis de importância (alto, médio e baixo), cada negócio pode ter mais níveis, porém recomendamos não exagerar e passar de cinco, pois o principal objetivo desta classificação é forçar escolhermos quais funcionalidades são realmente mais importantes, se colocarmos muitos níveis, a menor classificação será “alta”.

As funcionalidades que vamos desenvolver tem que estar relacionada com um objetivo da empresa, pois é isso que dá propósito para ela, e a nossa base para verificar se ela está gerando valor. Com base no uso da funcionalidade, podemos verificar se o objetivo está sendo alcançado, e com base nesse retorno, podemos decidir quais serão os nossos próximos passos.

Quando pensamos em uma funcionalidade, devemos sempre considerar os riscos para o negócio, a utilização de um sistema implica em como a operação da empresa trabalha, e toda mudança gera riscos para a operação. Tudo que é novo, deve ser ensinado, por mais que a funcionalidade reflita exatamente o que os usuários já faziam, eles vão começar a executar essas atividades em outro lugar, o que no começo pode gerar dúvidas e frustrações. Devemos sempre levantar os riscos com base na perspectiva do usuário, pois assim podemos entender quais serão as reações dele e quais estratégias devemos utilizar para engajar o uso da ferramenta e buscar por melhorias.

Com base nos objetivos, nas funcionalidades e nos riscos, podemos mapear as pendências das funcionalidades, onde conseguimos traçar quais os passos que devemos tomar para iniciar o desenvolvimento até disponibilizar a funcionalidade para o usuário.

Para o levantamento dessas informações recomendamos utilizar o formato de tabela. A tabela possibilita consolidar de forma clara cada tópico e sempre que a empresa tiver um novo objetivo, é mais fácil analisar o que já foi listado, possibilitando ver se o sistema se encaixa com essa nova necessidade ou se será necessária uma nova funcionalidade, ou se será necessário um novo produto, ou uma reformulação do negócio. Quando listamos tudo que estamos fazendo, com o que o negócio pretende alcançar, podemos tomar decisões mais assertivas, entender dependências e otimizar o valor que será desenvolvido.

Muitas vezes focamos em desenvolver uma funcionalidade que não irá gerar muito valor para o negócio, o que nos faz aceitar um risco que não precisamos no momento, toda funcionalidade gera riscos, então é melhor nos concentrarmos somente naqueles que podem gerar mais valor.

Uma vez definido nossas funcionalidades, sua importância e os objetivos que pretendemos com elas, podemos escolher quais vamos implementar primeiro e quais riscos nós vamos enfrentar.

Para exemplificar a utilização da tabela proposta vamos considerar uma empresa que possui vendedores que entram em contato com outras empresas para negociar a venda de computadores. Estes computadores podem ser customizados pelo cliente, solicitando maior espaço de memória, se deseja um computador com *SSD* ou *HD*, qual o processador, entre outras escolhas. Embora já existam computadores pré-montados, eles podem ser customizados somente aproveitando a base do produto.

Hoje a venda é feita sem a utilização de um sistema, a especificação que o cliente deseja é feita de forma individual por cada vendedor e controlado através de planilhas, onde cada vendedor possui a sua forma de organizar. Somente o pedido final é colocado em um sistema de controle de pedidos, para emitir a nota. A comunicação entre os vendedores e a área técnica é feita através de telefone e e-mail, os produtos base e os produtos vendidos são compartilhados através de planilhas enviadas por e-mail. Como cada vendedor negocia de uma forma, não há Padronização nas planilhas, o que acaba gera confusões na área técnica na hora da montagem dos produtos. Outro ponto a destacar é que a área administrativa não tem visibilidade de como as negociações estão sendo realizadas o que dificulta o entendimento do negócio na totalidade e a criação de estratégias.

Com base nessas necessidades, a empresa decidiu começar um projeto para a criação de um sistema para os vendedores realizarem as suas negociações. Onde o produto base seria disponibilizado na ferramenta e os pedidos finais seriam montados conforme a negociação avança, outra necessidade, é que a negociação deve ser faseada, para que seja possível rastrear os passos do vendedor e ter uma melhor visão do negócio.

Neste cenário, o primeiro passo é levantar as principais funcionalidades para atender o negócio, já colocando a sua importância e os objetivos que pretendemos alcançar.

Na tabela dois, representamos três funcionalidades, o cadastro de produtos, a venda de produtos, e o *chat* interno. Podemos notar que a maior necessidade da empresa é a de padronizar a venda do produto, onde a negociação deve ser feita de forma uniforme e que facilite os vendedores a venderem produtos mais aderentes com o que a área técnica consegue produzir. Com base nestas necessidades, foi levantado as funcionalidades de cadastro



Funcionalidade	Importância	Objetivos
Cadastro de produtos	Alta	Padronização na criação de produtos; Reduzir o cadastro incorreto de produtos.
Venda de produtos	Alta	Venda faseada do produto; Melhor rastreabilidade da venda; Padronização da negociação.
<i>Chat</i> interno	Baixa	Otimização da comunicação entre a área técnica e os vendedores.

Tabela 2: Exemplo de levantamento de funcionalidades

de produtos e venda de produtos, onde o cadastro está relacionada com a padronização da criação do produto e a venda está relacionada com a padronização da venda. Porém, a funcionalidade de *chat* interno, não é uma prioridade para o negócio, as áreas estão se comunicando, o problema é que cada vendedor trabalha de um jeito, o que dificulta as decisões da área técnica. Embora um dos objetivos seja melhorar a comunicação entre as áreas, este não é o maior dos problemas que precisamos resolver, então os riscos que desta funcionalidade não precisam ser corridos até o desenvolvimento das outras duas.

Outro ponto de destaque é a identificação das dependências, quando mapeamos as dependências, conseguimos ver qual funcionalidade precisamos realizar primeiro, como listado no exemplo, não é possível vender os produtos sem antes cadastrá-los. Além das dependências entre funcionalidades, podemos listar o que precisamos fazer para disponibilizar a funcionalidade para o usuário, como apresentado no exemplo, todas as funcionalidades precisam de treinamento, como o sistema é novo e cada vendedor realiza as negociações do seu jeito, é importante explicar como foi montado o processo de vendas e apresentar como realizar as atividades no sistema.

Cada funcionalidade tem os seus riscos, porém é através de sua importância que podemos criar estratégias para enfrentar estes riscos e marcar estas ações que definimos em nossa estratégia como dependências para liberar o uso da funcionalidade, é importante atrelar estas ações como dependências, pois desta forma os usuários não são surpresos, e é através dos treinamentos e das apresentações que podemos pegar o *feedback* dos usuários e começar a mapear novas funcionalidades e melhorias. Podemos ver o mapeamento dos riscos e das dependências das funcionalidades exemplo na tabela três.

Funcionalidade	Riscos	Dependências
Cadastro de produtos	Processo de cadastro de produtos mais demorado; Criação de produtos menos flexível.	Treinamento para cadastrar os produtos no sistema; Explicação das regras utilizadas para o cadastro dos produtos.
Venda de produtos	Venda menos flexível; Mudança na forma de negociação dos vendedores;	Funcionalidade de cadastro de produtos; Treinamento para realizar vendas no sistema; Explicação das fases na venda.
<i>Chat</i> para clientes	Não adoção do <i>chat</i> como principal meio de comunicação; Utilização do <i>chat</i> para assuntos sem relação ao trabalho;	Treinamento para a utilização do <i>chat</i> ; Treinamento sobre como se comunicar por <i>chat</i> .

Tabela 3: Exemplo de levantamento dos riscos e dependências por funcionalidade

## 6.2 Entendendo os requisitos de uma funcionalidade

Após entendermos os riscos, escolher quais vamos aceitar e definir a prioridade das funcionalidades que queremos desenvolver, devemos começar a refinar cada uma e definir quais são seus requisitos. É importante salientar que os requisitos, assim como foi com os riscos, devem partir do negócio, para assim fazer parte do sistema, a funcionalidade deve solucionar um problema e facilitar a vida dos usuários, agregando valor para o cotidiano deles e consequentemente agregando valor para o negócio.

A funcionalidade deve ser desenvolvida com base no que os usuários já fazem no dia a dia, temos que mapear todas as tarefas que o usuário executa, identificando quais incomodam e quais não incomodam. Identificando quais tarefas o usuário não gosta de fazer, já é um ponto de partida no que o sistema automatizar, uma vez que o usuário não precise fazer determinada funcionalidade, já facilitou o seu cotidiano. Outro ponto de atenção, são com as tarefas que o usuário não se importa em executar, automatizar ou mudar a forma que ele executa pode ser um problema em um primeiro momento, pois ele terá que se acostumar com uma nova forma de trabalho. Muitas vezes essa mudança pode se originar em uma nova fase da empresa, que está revendo os seus processos e mudando a forma como de trabalho dos colaboradores, neste caso, é importante implementar a funcionalidade de forma que seja um meio-termo entre o que os usuários realizavam e a nova forma da empresa, desta forma os usuários começam a se adaptar a nova perspectiva e conforme o projeto for avançando, podemos ir adaptando a funcionalidade a reestruturação

da empresa. Devemos lembrar que nenhuma mudança acontece do dia para a noite, e uma mudança brusca na forma de trabalho pode gerar desconforto nos usuários e deixar eles confusos em como executar suas atividades.

Uma funcionalidade deve sofrer melhorias constantes, como o negócio está constantemente sofrendo mudanças, o sistema deve se adaptar para os novos requisitos do mercado, por este motivo, os requisitos devem ser identificados, mas não podem ser considerados como certeza, o sistema deve estar pronto para sofrer qualquer alteração e a qualquer hora porem, sabemos que certas mudanças levam muito tempo e até a alteração, o requisito já sofreu mais uma alteração, por este motivo, devemos classificar nossos requisitos ao nível de importância e ao nível de mutabilidade. Enquanto nos riscos nós classificamos a funcionalidade na totalidade, aqui vamos classificar os requisitos que formam essa funcionalidade, devemos sempre pensar, o que é essencial para a funcionalidade atingir o objetivo? O que pode se tornar uma melhoria para o futuro? Desta forma, conseguimos entregar valor mais rápido para diferentes áreas do negócio, pois se focarmos todo o nosso tempo na funcionalidade mais importante, nunca vamos realizar as outras funcionalidades. Para cada funcionalidade devemos também atribuir um nível de mutabilidade, ou seja, qual é a hipótese dessa funcionalidade sofrer alterações conforme o tempo vai passando, embora o negócio esteja em constante mudança, sabemos que certos processos dificilmente sofrem alterações, pois faz parte do *core* da empresa, é a essência do negócio. A forma de classificar a mutabilidade é através de porcentagem, onde as primeiras classificações serão intuições com base na perspectiva do negócio, mas as seguintes, é importante que conforme o sistema vá sofrendo alterações e melhorias, estas taxas sejam atualizadas por funcionalidade. A primeira regra na hora de classificar é que não existe zero por cento, toda funcionalidade pode sofrer alterações, mesmo que demore anos para isso acontecer, a sociedade muda, novas tecnologias vão surgindo e o que era comum é substituído por algo novo, que acaba se tornando o novo comum, e se a empresa não estiver preparada para acompanhar essas mudanças, ela acaba se perdendo no tempo. Realizando essa classificação podemos identificar quais processos estão consolidados e quais processos ainda devem ser estruturados, com essa visão sobre os processos, podemos identificar o novo *core* do negócio e desta forma trabalhar em uma transformação na empresa para atender as novas necessidades do mercado, assim evoluindo o sistema ou até mesmo desenvolvendo um novo, garantindo que a empresa se mantenha sempre atualizada com as tendências do mercado. Na tabela quatro está os atributos que devem ser considerados quando classificarmos um requisito.

<b>Funcionalidade</b>	Nome da funcionalidade que o requisito faz parte.
<b>Requisito</b>	Nome do requisito que faz parte da funcionalidade.
<b>Importância</b>	O nível de importância do requisito separado em baixo, médio e alto.
<b>Mutabilidade</b>	O quanto este requisito pode sofrer alterações em porcentagem.
<b>Stakeholders</b>	Quais áreas são impactadas com esta funcionalidade.
<b>Descrição</b>	A descrição do requisito da funcionalidade.

Tabela 4: Definição da classificação de requisitos

Agora que temos mapeado os processos que fazem parte de determinada funcionalidade, e sabemos como classificar, podemos começar a levantar eles. Todo requisito deve possuir uma descrição, algo que instrua a equipe de desenvolvimento a entender o seu propósito e auxilie eles a desenvolver a funcionalidade. Quando analisamos um requisito, outros requisitos vão surgindo, pois, a um primeiro momento, quando levantamos os requisitos, ficamos sempre na funcionalidade, e como o sistema deve se comportar, mas além de estruturar como determinada funcionalidade deve se comportar, devemos utilizar os requisitos para garantir a qualidade do que será desenvolvido. É através de como os usuários vão utilizar o sistema que podemos identificar o que precisamos garantir que funcione durante a utilização do usuário. Questões como quantidade de acessos, tempo de resposta, usabilidade, métricas, devem ser requisitos do sistema, algo diretamente relacionado ao que vai ser desenvolvido e estabelecer um critério de aceitável ou não para a utilização em produção. Quando identificamos os requisitos, devemos também identificar quais áreas da empresa será impactada pelo requisito, para os requisitos funcionais, devemos identificar quais são as áreas que vão utilizar esta funcionalidade, para os requisitos não-funcionais, identificamos as áreas que caso o requisito não seja atendido, serão impactadas, caso o requisito não seja atendido, quais áreas terão seu processo afetado, podendo criar um bloqueio ou deixando ele mais longo. Para exemplificar o levantamento dos requisitos, vamos utilizar a funcionalidade de “Cadastro de produtos”, utilizado no exemplo de identificação dos riscos.

Para realizar o cadastro de produtos, vamos supor que envolva três áreas, o *marketing*, a equipe técnica e vendas. A equipe de *marketing*, com base em suas análises de mercado e nas solicitações dos vendedores, analisam as tendências dos pedidos do cliente e montam projetos para novos produtos, este projeto é enviado para a área técnica que avalia se é viável ou não a sua execução, podendo aprovar ou recusar a montagem desse produto. Caso o produto o seja recusado, a solicitação é devolvida para a equipe técnica, com os motivos da recusa, onde eles podem realizar as devidas alterações ou cancelar o projeto. Caso aceite, a equipe técnica começa a montagem, realizam seus testes e validam se está

aderente com o que foi proposto no projeto, uma vez encerrado está etapa, é anunciado aos vendedores que há um novo produto no catálogo e caso tenha relação à algum pedido de algum vendedor, ele é notificado que determinado produto foi produzido. Há casos, em que o vendedor pega um produto base e este deve ser customizado com base nas especificações do cliente.

<b>Funcionalidade</b>	Cadastro de produtos
<b>Requisito</b>	Notificação de projeto de produto base
<b>Importância</b>	Média
<b>Mutabilidade</b>	45%
<b>Stakeholders</b>	Área técnica
<b>Descrição</b>	Quando um projeto de produto base for cadastrado, o sistema deve enviar uma notificação para a área técnica, que um novo produto foi cadastro e aguarda aprovação.

Tabela 5: Exemplo de classificação de requisito funcional

<b>Funcionalidade</b>	Cadastro de produtos
<b>Requisito</b>	A notificação deve ser enviada imediatamente para a área técnica
<b>Importância</b>	Média
<b>Mutabilidade</b>	10%
<b>Stakeholders</b>	<i>Marketing</i> , área técnica
<b>Descrição</b>	A notificação da criação do projeto deve ser enviada em um intervalo de menos de um segundo para a área técnica.

Tabela 6: Exemplo de classificação de requisito não-funcional

<b>Funcionalidade</b>	Cadastro de produtos
<b>Requisito</b>	O cadastro é destinado somente para produtos base
<b>Importância</b>	Alta
<b>Mutabilidade</b>	7%
<b>Stakeholders</b>	Vendas
<b>Descrição</b>	Os produtos que serão cadastrados nesta funcionalidade deve somente ser produtos base, customizações serão feitas durante a venda.

Tabela 7: Exemplo de classificação de requisito inverso

Embora a funcionalidade de “Cadastro de produtos” tenham muitos outros requisitos, vamos exemplificar somente os que estão nas tabelas cinco, seis e sete. O requisito da tabela cinco se trata de um requisito funcional, o requisito de enviar a notificação para a área técnica quando um projeto for criado. Podemos notar que sua mutabilidade é média,

embora o processo dificilmente possa mudar, a área técnica sempre deve ser informada quando um projeto for criado, pois, são eles que aprovam se o projeto é válido ou não porem, no futuro é possível que seja criada uma área somente para análise de projetos, hoje está concentrado dentro da mesma área que monta o produto, outra alteração é a adição de outra área, hoje o negócio estende que somente a área técnica deve ser informada porem, os vendedores também fazem parte do processo, uma vez que o projeto pode se tratar de uma solicitação deles, por este motivo, a área de vendas pode ser incluída no recebimento da requisição. Embora os vendedores também participem desse processo, a notificação é destinada somente para a área técnica, desta forma, somente a eles são um *stakeholder* desse requisito. Na tabela seis, temos um requisito não-funcional, é um requisito que está atrelado diretamente a qualidade da funcionalidade, ela está definindo que a notificação deve ser disparado com no máximo um *delay* de um segundo da criação do produto. Para o negócio é importante que a área seja informada que um novo projeto foi criado imediatamente após sua criação, este requisito deve ser utilizado como critério de qualidade e utilizado como cenário de teste, não somente da criação de um projeto, mas para a criação de vários projetos de uma vez só. Na tabela sete, possuímos um requisito inverso, algo que o sistema não deve fazer, que no caso, é que a funcionalidade de “Cadastro de produtos” se destina somente para produtos base, ou seja, as customizações dos vendedores não estão incluídas nessa funcionalidade, é interessante na descrição deste requisito, sempre colocar em qual funcionalidade este requisito se aplica, caso ele se aplique a alguma funcionalidade. Como a customização dos produtos base está sempre atrelada a venda do produto e não ao seu cadastro, dos três exemplos, este é o de menor mutabilidade, pois este processo dificilmente será mudado e atribuir a responsabilidade ao vendedor de cadastrar um produto, somente por causas das customizações é uma mudança muito grande no processo de como é realizada a venda.

## 6.3 Arquitetura evolutiva

Conforme o produto vai evoluindo, mudanças deverão ser feitas para acompanhar as necessidades do negócio e para isso, devemos construir o nosso sistema de forma que seja fácil implementar novas funcionalidades e aplicar mudanças e melhorias em funcionalidades já desenvolvidas. Para que isso seja possível, devemos desenvolver um *software* com alto nível de abstração, cada parte do sistema deve funcionar de forma quase individual, para que desta forma uma alteração realizada em alguma parte do sistema, não afeta as demais e assim, minimizamos *bugs* e tempo de manutenção e desenvolvimento de melho-

rias.

O mundo perfeito, realmente é desenvolver um sistema como o descrito anteriormente, com alto nível de abstração e cada parte do sistema funcionando de forma individual, mas para montar determinado sistema, o produto e a equipe de desenvolvimento deve estar madura, no começo do projeto ainda não sabemos quais são as partes do sistema, não conseguimos separar o que deve funcionar de forma individual e o que faz parte desse individual. Para descobrirmos é importante quando desenharmos a arquitetura de nosso sistema, nos atentarmos na mutabilidade de cada requisito, devemos desenvolver cada funcionalidade como se ela pudesse mudar a qualquer hora, mas quando tivermos que tomar a decisão em qual parte deverá ficar mais abstraída do sistema, a mutabilidade dos requisitos deverá ser considerada. Com base na mutabilidade, podemos montar a base de nosso sistema, as funcionalidades que muito dificilmente deverão ser modificadas, e a partir delas evoluir o sistema para as outras funcionalidades, que dependendo do nível de mutabilidade, deverá funcionar de forma quase independente, desta forma, conforme o tempo for passando, podemos ir isolando as partes do sistema cada vez mais, a medida que a mutabilidade dos requisitos vão ficando mais assertivas com a realidade e novos requisitos vão surgindo, pois, desta forma, a equipe de desenvolvimento terá um melhor entendimento do negócio e conseguirá abstrair o sistema de forma mais condizente de como o negócio funciona.

Após tivermos um sistema extremamente abstraído e um equipe de desenvolvimento com um grande entendimento do negócio, podemos evoluir nossa arquitetura para micros-serviços, de forma que, cada parte do sistema é verdadeiramente um sistema apartado, com seu próprio equipe e requisitos, onde todas essas funcionalidades se encontram na mesma plataforma. Quando conseguimos evoluir nosso produto para este nível, ele deixa de ser um sistema para se tornar uma plataforma, um lugar onde vários sistemas vão ser executado para cumprir diversos Objetivos relacionados. Lembrando, que embora agora temos vários sistemas, realizando diversas tarefas diferentes, o objetivo da plataforma deve ser o mesmo de quando iniciamos o projeto, nunca devemos esquecer o objetivo de o produto ter sido criado, porque é com base nele que as melhorias e os requisitos devem ser levantados, uma vez que esquecemos de esse objetivo, nosso produto começa a realizar diversas ações que não estão relacionadas, sua arquitetura fica cada vez mais complexa e nosso produto perde seu propósito.

Uma vez que nosso produto esteja separado em vários sistemas, podemos reaproveitar as funcionalidades que desenvolvemos em outros produtos, e como cada sistema da plataforma tem a sua própria equipe, as análises realizadas no produto serão realizadas em cada parte de valor do que foi desenvolvido, com esses dados, podemos analisar qual parte

do negócio está sofrendo maior alteração, qual está gerando mais valor, qual precisa ser reestruturada, qual precisa de mais investimento, quais áreas podem ser divididas, entre outras. Como cada parte da nossa plataforma está se preocupando com uma determinada parte do negócio e como, nossa equipe já conseguiu separar essas partes de uma forma que esteja completamente aderente a como o negócio funciona, conseguimos ter a visão de todas as atuações da empresa no mercado. Conseguimos ter a visão do todo separado em várias equipes diferentes, preocupadas em evoluir a sua parte e contribuir com as outras. Com essa visão, podemos evoluir nossos sistemas para outras plataformas, com objetivos diferentes, e agregando valor para o negócio de forma distinta. Podemos através das necessidades do mercado, e com os requisitos do sistema, reformular a forma que a empresa atua no mercado, identificar atuações secundárias da empresa, que tem potencial para se tornar primárias, fazer os sócios e diretores enxergarem cada pedaço que gera valor para a empresa e definir estratégias não tradicionais, algo diferente, para adquirir uma possível vantagem no mercado.

## 6.4 Estruturando o fluxo de entrega

Para alcançarmos nossos objetivos e desenvolvermos as nossas funcionalidades em tempo ágil, com qualidade e se preocupando em manter nossa arquitetura simples e evolutiva, precisamos estruturar o nosso fluxo de entrega, ou seja, precisamos definir como será o processo de desenvolvimento, verificação de qualidade, homologação e subida em produção.

Para cada requisito, devemos escrever os critérios de aceite, para que eles sejam considerados cumpridos, para que desta forma, quem for testar as funcionalidades, consigam compreender o que devem verificar e para que os desenvolvedores, compreendam como devem desenvolver a funcionalidade. Desta forma, além de termos desenvolvedores e testadores cientes de como deve o sistema deve funcionar, conseguimos criar um padrão de qualidade, que será documentado e revisado conforme o produto for evoluindo, pois, conforme o tempo for passando, novos requisitos forem surgindo e os requisitos já mapeados forem mudando. Com os critérios de aceite escritos, podemos mapear quais testes deverão ser feitos para cada funcionalidade, teste de regressão, teste de fumaça, teste de integração, teste em massa, teste unitário, cada requisito vai exigir um determinado tipo diferente de teste, e com o critério de aceite é importante mapearmos não somente quais testes, mas também em quais cenários e como eles devem ser executados, pois, é nos testes que devemos procurar simular os momentos mais importantes para o negócio,



os momentos em que o sistema não pode falhar. Embora cada requisito tenha o seu próprio conjunto de testes, é importante padronizar que todas as novas funcionalidades tenham sempre testes unitários e em massa, e para melhorias ou mudanças seja sempre realizado testes de regressão, caso uma nova funcionalidade esteja relacionada com outra, é importante também realizar os testes de regressão nas funcionalidades relacionadas. Os critérios de aceite devem ser escritos se baseando diretamente nos riscos que aceitamos, quando mapeamos as funcionalidades, nossos testes, além de garantir que os requisitos da funcionalidade sejam cumpridos, deve direcionar os nossos testes para que realizemos as situações que mapeamos como risco, embora os riscos tenham sido aceite, precisamos verificar como o sistema vai se comportar nestes cenários, desta forma conseguimos antes de disponibilizar para o usuário, averiguar se realmente faz sentido correr esse risco ou não.

Para exemplificar a escrita dos critérios de aceite, vamos utilizar o requisito da tabela seis.

<b>Funcionalidade</b>	Cadastro de produtos
<b>Requisito</b>	A notificação deve ser enviada imediatamente para a área técnica
<b>Critério</b>	Deve ser disparado a notificação para todos os produtos que forem criados.
<b>Critério</b>	Na notificação deve conter o nome do produto e quando ele foi criado.
<b>Critério</b>	A área técnica deverá identificar se uma notificação já foi lida ou não.
<b>Critério</b>	A notificação somente deve ser marcada como lida caso o usuário da área técnica acesse a notificação.

Tabela 8: Exemplo de especificação de critérios de aceite

Na tabela oito, podemos verificar os critérios de aceite para o requisito de notificação da funcionalidade de “Cadastro de produtos”. Podemos notar que cada requisito descreve como a funcionalidade deve funcionar, assim como auxilia como este requisito deve ser testado, conseguimos identificar também que este requisito deve ser testado de forma massiva, pois um dos critérios é de que cada produto deve disparar obrigatoriamente uma notificação, desta forma, se eu criar nove produtos de uma vez, deve ser gerada uma notificação para os nove produtos.

Agora que definimos como nosso requisito será desenvolvido e como ele será testado, precisamos definir em qual momento cada uma dessas ações deverá ser realizada e, em qual ambiente. Quando estamos desenvolvendo uma funcionalidade devemos sempre tes-

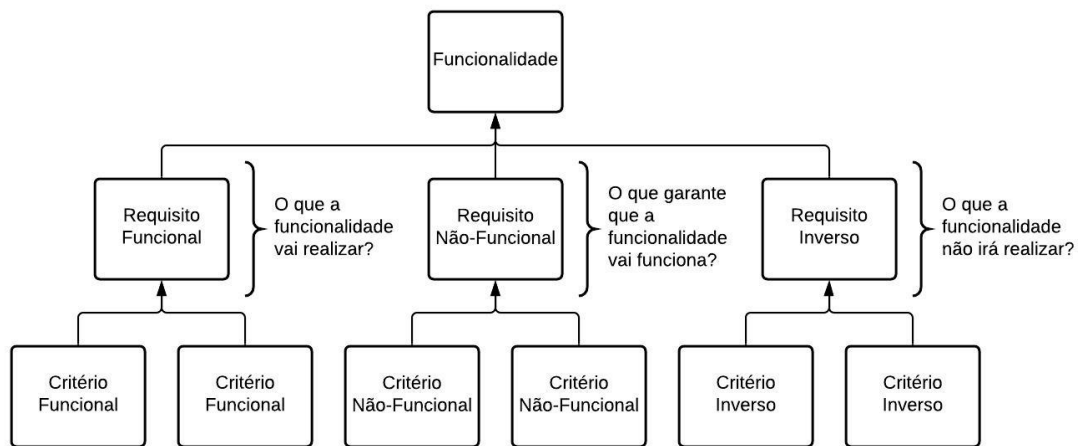


Figura 1: Estrutura de funcionalidade

tar ela por completo, mas para sermos ágeis, não podemos esperar todos os requisitos serem desenvolvidos para iniciarmos os testes, da mesma forma que os desenvolvedores não podem ficar aguardando todos os testes serem realizados para iniciarem um novo desenvolvimento. Enquanto os testadores só podem iniciar um teste assim que um desenvolvedor finalizar uma atividade e os desenvolvedores não podem alterar a funcionalidade para não influenciar nos testes, é interessante realizar estas ações em ambientes separados. Os desenvolvedores realizam os seus desenvolvimentos em um ambiente exclusivo para desenvolvimento, enquanto os testadores realizam seus testes em um ambiente exclusivo para testes. Cada critério, cada cenário deverá ser simulado neste ambiente, devido a isso a qualidade dos dados deste ambiente deverá ser o mais fiel possível ao que os usuários irão utilizar, por outro lado, os desenvolvedores necessitam de dados somente para verificar se a funcionalidade está funcionando e os critérios foram atendidos, devido a isso, é importante ter uma boa qualidade nos dados, mas não é preciso investir tanto tempo nos dados deste ambiente. Vamos chamar o ambiente dos desenvolvedores de **DEV** (desenvolvimento) e o ambiente dos testadores de **QA** (*Quality Assurance*).

Os desenvolvedores têm a obrigação de garantir que todos os critérios do requisito desenvolvido seja atendido, ou seja, só é permitido disponibilizar o requisito para ser testado no ambiente de **QA** após o desenvolvedor testar todos os critérios no ambiente de **DEV**. O requisito disponibilizado em **QA**, o testador deverá testar todos os critérios nos mais diversos cenários possíveis, dando foco nos riscos que foram mapeados na especificação do requisito e conforme mais requisitos vão sendo disponibilizados, ele deverá testar novamente todos os cenários novamente, mas agora validando os critérios dos outros requisitos. Desta forma, conseguimos garantir que as situações de maior necessidade do negócio está sendo contemplada. Para que possamos gerar valor de forma ágil, é importante que na

primeira iteração do equipe, seja desenvolvida somente as funcionalidades de maior importância, pois dessa forma conseguimos integrar uma funcionalidade de forma rápida, embora ela realizando o mínimo planejado, já podemos como será a reação dos usuários e identificar se há a necessidade de mudanças ou melhorias para serem desenvolvidas nas próximas iterações, além de reavaliar a importância dos requisitos mapeados.

Logo após todos os testes em todos os cenários de risco da funcionalidade, podemos disponibilizar a funcionalidade para homologação, ou seja, devemos realizar todas as atividades que o usuário irá executar, com o objetivo de verificar se o que foi desenvolvido está aderente com o negócio, podemos também realizar apresentações para os usuários, de modo a adiantar para eles o que será entregue e preparar uma documentação de como determinada funcionalidade funciona, com o intuito de disponibilizar esta informação para os usuários e para novos integrantes do projeto. Todo esse processo deve ser realizado em um ambiente separado que vamos nomear de **HOMOL** (homologação). Este ambiente deve estar separado, pois, os dados que contem nele, devem ser preparados para apresentações, como ele será utilizado para compreender o que será entregue para o usuário e para apresentar uma prévia para eles, as funcionalidades não devem ser influenciadas por funcionalidades em desenvolvimento e os dados presentes nelas, não devem ser influenciadas pelos testes realizados pelos testadores. Os dados neste ambiente deve ser o mais limpo possível, ele deve ser o mais próximo possível do que os usuários irão utilizar em produção, e as funcionalidades que estão neste ambiente devem estar completamente testadas e com todos os critérios garantidos.

Após realizarmos a homologação da nossa funcionalidade, devemos nos preparar para entregar essa funcionalidade para os usuários, para isso, vale realizarmos uma última validação, em um ambiente que vamos nomear de **PREPROD** (pré-produção). Este ambiente deve ser uma cópia perfeita de produção, nele devemos realizar novamente todos os principais testes que já realizamos em **QA** e **HOMOL**, para verificar que todos os critérios continuam sendo cumpridos, além de aplicarmos treinamentos para os usuários da nova funcionalidade, permitindo que eles utilizem esta funcionalidade de forma controlada, para que eles aprendam a utilizar o sistema antes de colocar ele no ar. Com essa abordagem, antes de disponibilizar o sistema em produção, conseguimos verificar a reação deles com o que será entregue, além de ter uma prévia de como o sistema será utilizado, desta forma já conseguimos identificar melhorias para as próximas iterações.

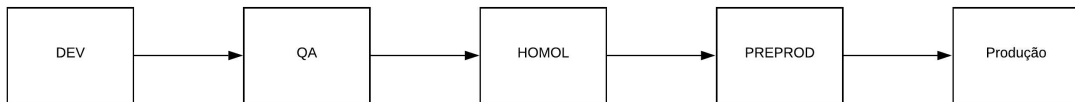


Figura 2: Fluxo de ambientes

## 6.5 Automatizando testes de qualidade

Para otimizar a execução dos testes, e facilitar o trabalho dos testadores, podemos automatizar os testes que realizamos no sistema, para que sempre que uma funcionalidade nova, uma melhoria, uma alteração ou uma correção for implementada podemos realizar novamente todos os testes no sistema automaticamente. Desta forma, não consumimos tanto tempo dos testadores e garantimos que o tudo no nosso sistema continua funcionando da forma que esperamos. A fase de automação dos testes, é importante ser realizada assim que o requisito for homologado, com a homologação, estamos mais certos que o requisito está aderente com o negócio e por causa da etapa de testes, temos todos os principais cenários que precisam ser garantidos, desta forma podemos montar testes automatizados para após homologado, qualquer alteração que seja realizada nos ambientes de **QA** e adiante, inicie os testes e garanta que os cenários de risco continuem não apresentando problemas para o sistema.

Os testes devem estar diretamente vinculados com o *deploy* para os ambientes, ou seja, é importante que seja um processo do *pipeline*, onde todo *deploy* realizado, rode novamente todos os testes, em cada um dos ambientes, pois desta forma, conseguimos averiguar se alguma alteração no sistema impactou em alguma parte que não estávamos prevendo e com essa descoberta, conseguimos uma visão mais detalhada do quanto o nosso sistema está estruturado, auxiliando nas decisões de arquitetura.

## 7 COMO ADQUIRIR ESCALABILIDADE?

Conforme o nosso sistema vai aumentando, mais usuários vão utilizando ele, mais funcionalidade ele possui, e mais poder computacional ele necessita. Garantir que um sistema se mantenha no ar é uma tarefa complicada, embora estejamos desenvolvendo o sistema o melhor possível, realizando testes na maioria dos cenários e evoluindo a arquitetura para se adequar a atual realidade do sistema, não conseguimos prever tudo, e muitas vezes, quando mais precisamos que ele funcione, ele apresenta algum problema. Muitas vezes podemos subestimar alguma situação ou superestimar outra situação, e se não estivermos preparados para os imprevistos, a qualidade de nosso produto pode cair e a confiança dos nossos usuários diminuir.

Vamos imaginar, que a empresa que utilizamos como exemplo anteriormente, conseguiu criar um computador base, que atendem muitas necessidades dos seus clientes, devido a isso, o número de pedidos aumentou, e a empresa teve que contratar mais vendedores, mais colaboradores da área técnica e a equipe de *marketing* está com várias ideias para novos produtos. O sistema nessa situação vai estar constantemente executando vendas, assim como vários projetos serão criados, e várias notificações deverão ser disparadas. Certo dia, a utilização no sistema foi tão intensa, que o sistema não enviou mais notificações durante dois minutos, estes dois minutos foi o suficiente para que a área técnica não analisasse três projetos, estes projetos que a equipe de *marketing* tinha uma grande expectativa, o tempo passou foi gerado o conflito entre as duas áreas, e foi identificado este erro no sistema, agora o momento para a produção dos produtos base se perdeu, este conflito abalou as duas áreas e a confiança com o sistema pelos usuários diminuiu.

### 7.1 Descobrimos os limites do sistema

Para nos prepararmos para as situações inesperadas, precisamos descobrir os limites do nosso sistema, precisamos tentar fazer o sistema falhar em um ambiente controlado, antes que ele falhe em produção, quando os nossos usuários estiverem utilizando. Entender que nosso sistema tem limites e que ele está suscetível a erros é o primeiro passo

para descobrirmos como podemos encontrar estes erros, devemos utilizar os riscos que mapeamos, e simular diversas vezes estes cenários, o que levantamos em nossos requisitos não-funcionais, devem ser levado ao extremo, então caso levantamos que um determinado requisito deve ter um tempo de resposta de no máximo dois segundos, devemos forçar o processamento do nosso sistema para que o tempo de resposta supere dois segundos e neste cenário, rodar novamente todos os testes novamente, desta forma nós conseguimos ver se o fato do requisito não ser atendido tem algum impacto nos critérios que levantamos. Uma vez conseguido simular este cenário, podemos mapear os limites do sistema, como nos esforçamos para forçar o sistema a funcionar no seu limite, agora podemos documentar qual é o seu limite e analisar se devemos aumentar este limite ou se aceitamos o risco que ele representa.

Definir limites para o sistema é definir novamente os riscos que pretendemos aceitar, quando estávamos especificando as nossas funcionalidades, os riscos que mapeamos era com a perspectiva de negócio, nosso objetivo era garantir que o que iria ser desenvolvido, gerasse valor e não teria impactos negativos para os usuários, agora devemos utilizar estes riscos, em conjunto com os nossos requisitos, para mapearmos os limites do sistema. Estamos utilizando o negócio para aprender até onde o sistema deve aguentar, e, por outro lado, estamos utilizando o sistema para aprender o que o negócio precisa e quais riscos podemos enfrentar.

Quando mapeamos os limites do sistema, é importante separarmos os limites por funcionalidade e não buscar valores absolutos, mas sim valores estatísticos. Por mais testes que realizamos e por mais cenários que simulamos, não podemos garantir que em uma situação extrema, o sistema vai se comportar sempre da mesma forma. Como estamos buscando o extremo, e esperando que o que desenvolvemos pare de funcionar, por mais que tenhamos uma ideia de qual parte do sistema vai falhar, não é garantido que as nossas expectativas sempre se cumpram, realizando o mesmo teste mais de uma vez, podemos ter diversos retornos diferentes. Por este motivo, é interessante, com base na importância da funcionalidade, estipular uma quantidade de testes para cada situação extrema, com base nos retornos gerados, podemos documentar o que aconteceu e qual foi a frequência deste ocorrido, pois caso algum dia venha a ocorrer em produção, já temos mapeado os principais problemas ou uma vez que, produção comece a trabalhar no limite do sistema, já sabemos o que é provável de ocorrer. Para armazenarmos os testes que ocorreram, é importante colocar anotar a funcionalidade que testamos, o cenário que estava o sistema e quais foram resultados com as porcentagens. Nas tabelas nove e dez, exemplificamos está análise utilizando a funcionalidade, “Cadastro de produtos”.

<b>Funcionalidade</b>	Cadastro de produtos
<b>Cenário</b>	O processamento do servidor utilizado para a funcionalidade se encontra com 95% de uso.
<b>Resultado</b>	Em 89% do cadastro de novos projetos, a notificação demorou cerca de três segundos para aparecer.
<b>Resultado</b>	Em 96% do cadastro dos novos projetos, levou cerca de dois segundos para realizar a criação do projeto.

Tabela 9: Exemplo de análise de limite de sistema — limite de processamento

<b>Funcionalidade</b>	Cadastro de produtos
<b>Cenário</b>	Realizado 500 criações de novos projetos ao mesmo tempo.
<b>Resultado</b>	Em 98% dos cadastros a conexão com o banco de dados foi perdida ao tentar inserir os produtos.

Tabela 10: Exemplo de análise de limite de sistema — limite de requisições

As análises devem ser feitas em um ambiente que seja uma cópia de produção, sempre que algo for entregue em produção, deve ser automaticamente replicado para este ambiente, vamos chamá-lo de **CHAOSPROD** (produção caótica). Os dados deste ambiente devem ser uma representação dos dados que contém as estruturas mais complexas, neste ambiente deve ser possível simular cenários que complexos para o nosso sistema, que force as nossas funcionalidades a situações que consideramos difíceis de ocorrer, praticamente impossíveis. Este ambiente deve nos fornecer a possibilidade de testar situações em que não estamos preparados, identificar riscos que não mapeamos e analisar erros que ainda não aconteceram em produção.

Estas análises são prevendo cenários adversos, imaginando situações inesperadas, neste ambiente devemos nos concentrar nos requisitos não-funcionais, pois, é através das situações que vamos simular nestes ambientes que vamos conseguir avaliar se os limites que aceitamos em nosso sistema realmente está aderente as necessidades do negócio, e caso comece a aparecer situações semelhantes as que simulamos neste ambiente, conseguimos nos preparar e criar um plano para que o erro que foi gerado em **CHAOSPROD** não ocorra em produção e por este motivo, é interessante realizar estas análises sempre após que um requisito for entregue em produção, desta forma, conseguimos maior produtividade na entrega dos requisitos em desenvolvimento, e uma vez que ele tenha sido homologado, e temos o retorno de como os usuários estão utilizando o sistema, conseguimos realizar as análises nos baseando não somente no que mapeamos como risco, mas também com base em como o usuário está utilizando o sistema e no *feedback* que estamos tendo com eles, seja esse *feedback* dito diretamente pelo usuário ou através de alguma análise realizada em produção.

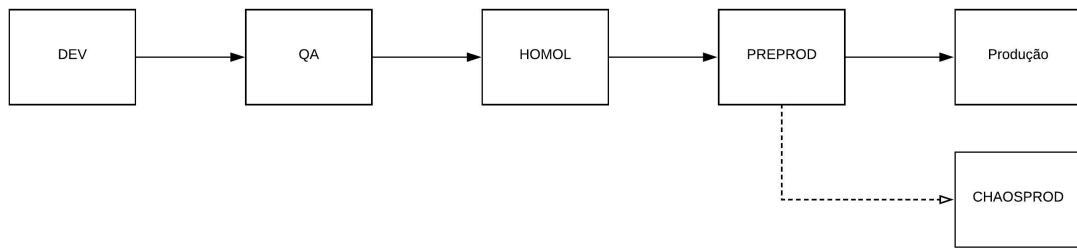


Figura 3: Fluxo de ambientes com CHAOSPROD

## 7.2 Desenvolvendo estratégias de escalabilidade

Uma vez identificado os nossos limites, devemos criar estratégias para lidar com elas, ser aumentar o poder de processamento dos nossos servidores, migrar a nossa aplicação para um banco de dados mais robusto, melhorar o desempenho da funcionalidade desenvolvida através do código, existem várias ações que podemos tomar, para podermos enfrentar os riscos que mapeamos.

Para cada cenário que mapearmos precisamos criar um plano, para caso esta situação venha a ocorrer. Este plano, deve estar atrelado diretamente a importância da funcionalidade e os impactos que sua execução irá gerar, devemos considerar os gastos financeiros, o tempo que o plano vai demorar para ser executado, os equipes que irão atuar nele, as áreas de negócio que vão ser impactadas, como devemos avisar os usuários, entre outras ações. Com a realização deste plano, nos preparamos para o imprevisto, caso algum dos cenários que mapeamos aconteça, já vamos estar preparado, embora não sabemos exatamente o que vai acontecer, temos a probabilidade de cada situação. Podemos também nos antecipar a esse limite, caso a funcionalidade seja muito importante para o negócio, e identificamos que o limite identificado pode ser atingido, podemos trabalhar para aumentá-lo, como sabemos aonde precisamos melhorar, basta realizar uma exploração mais aprofundada da melhor forma que podemos aumentar este limite. Outro ponto de destaque, é que conseguimos adaptar o nosso sistema conforme as necessidades de negócio, caso uma determinada funcionalidade necessite de mais processamento, podemos alocar mais servidores para esta funcionalidade, caso um não tenhamos os servidores disponíveis, podemos aumentar nossos servidores de forma horizontal, caso não haja verba disponível, para aumentar a quantidade de servidores, podemos desativar uma funcionalidade menos importante para realocar os recursos para a funcionalidade de maior importância, desta forma, assim que este período de maior necessidade passar, reativamos a funcionalidade antiga.

Para cada caso devemos estruturar uma estratégia e devemos utilizar de métricas no am-



biente de produção para supervisionar a saúde do ambiente e anteciparmos algum tipo de erro. Devemos monitorar o armazenamento já utilizado pelo nosso sistema, para caso ele esteja no fim, podemos aumentar o nível de armazenamento ou excluir registros antigos, devemos monitorar os nossos servidores, para caso algum deles estejam começando a apresentar defeito, ou algum conjunto de servidores estejam sendo mais utilizados que outros, assim podemos realocar as nossas funcionalidades em servidores diferentes e aprendemos qual parte do sistema utiliza maior quantidade de recursos dos nossos servidores. Podemos utilizar diversas acompanhamentos e implementar diversas métricas em nosso produto, mas é importante sempre lembrar, que devemos gerar as nossas métricas e basear os nossos acompanhamentos com base em como o sistema está sendo utilizado, ou seja, é através das necessidades do negócio, que aprendemos aonde devemos olhar. Na tabela dez, especificamos como estruturar uma estratégia.

<b>Funcionalidades</b>	Nome das funcionalidades que foram impactadas
<b>Cenário</b>	Descrição do cenário em que ocorreu os erros.
<b>Stakeholders</b>	Quais são as áreas de negócio que serão impactadas no cenário descrito.
<b>Times</b>	Quais são os equipe relacionados para a execução deste plano.
<b>Plano</b>	O que deverá ser feito caso produção se encontre neste cenário.
<b>Tempo de Execução</b>	Tempo para a execução do plano.

Tabela 11: Especificação para estruturação de estratégias

Com o ambiente de **CHAOSPROD**, conseguimos descobrir os limites do sistema, com o ambiente de produção, conseguimos descobrir como o sistema é utilizado, é através dos usuários, e da percepção de valor gerado para eles e para o negócio que devemos basear as nossas prioridades e ajustar a importância das nossas funcionalidades. Com esta visão podemos tomar a decisão aonde precisamos testar mais, e também onde precisamos de maior supervisão, embora estejamos constantemente procurando por erros, não estamos livres de falhas, erros em produção vão acontecer. É nestas horas que as ferramentas que implementamos para supervisionar o sistema tem sua maior utilidade, é através delas que vamos descobrir o que aconteceu e uma vez descoberto, vamos utilizar este ocorrido como aprendizado, além de resolver o problema, vamos identificar o porquê ele aconteceu, se realmente resolvemos o problema ou mitigamos, se ele pode acontecer e como podemos nos preparar para ele. Este erro pode se tornar mais um teste automatizado para ser realizado durante os desenvolvimentos ou pode se tornar mais uma análise a ser feita em **CHAOSPROD**. De toda forma, ele se tornou um aprendizado para o produto e agora

faz parte de um cenário que estamos prevendo, pode ser que ele aconteça novamente, mas desta vez estaremos preparados.

Para evitar que os cenários que identificamos em **CHAOSPROD** aconteçam em produção e conseguirmos monitorar problemas que não mapeamos é importante monitorar se os critérios que definimos para cada requisito de cada funcionalidade continua assegurado, precisamos criar métricas em produção para monitorar o sistema e verificar se nossas funcionalidades continuam funcionando como o esperado, é importante destacar que precisamos verificar se a funcionalidade está sendo utilizada, se o tempo de resposta continua sendo o que tínhamos mapeado em nosso requisito não-funcional, se a atividade que o usuário está executando, está levando o sistema à algum limite que definimos, se este cenário foi testado em **CHAOSPROD**, precisamos averiguar se o produto continua gerando valor ou se vamos precisar implementar alguma melhoria. Para realizar está análise, precisamos voltar para as funcionalidade e rever tudo o que havíamos definidos para elas, voltando para a fase de sua escrita, desta forma, vamos rever sua importância, os objetivos que ela está atrelada e consequentemente identificar novos requisitos e ajustar requisitos já existentes, modificando critérios e adicionando novos e assim que a funcionalidade for redefinida para as novas necessidades do negócio, voltamos para a fase de desenvolvimento, testes, comunicação, testes em **CHAOSPROD** e criamos métricas para garantir que os critérios continuem garantidos.

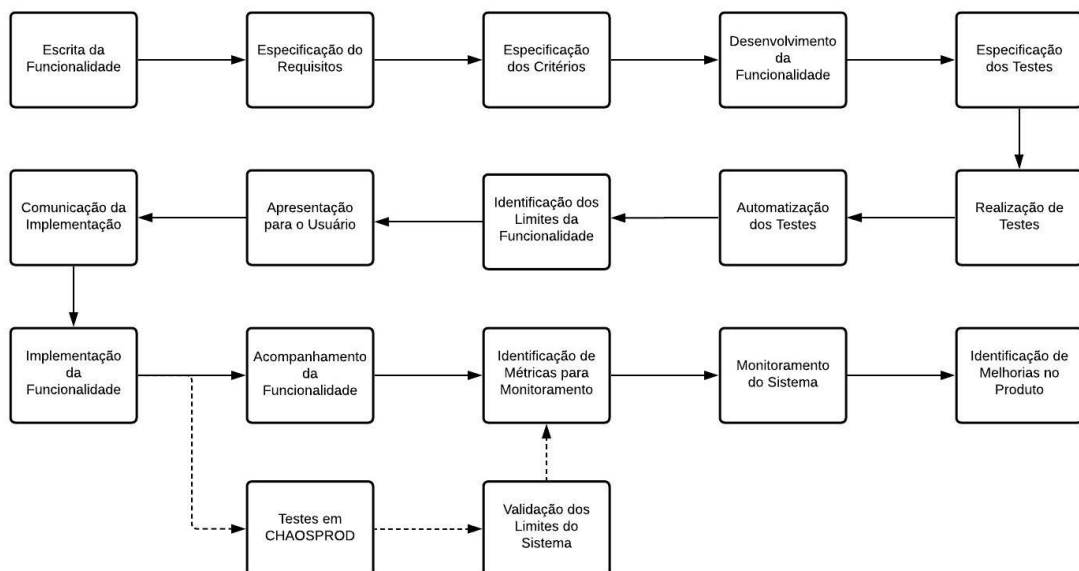


Figura 4: Fluxo de entrega

## 8 COMO MANTER O SEU PRODUTO?

Para assegurar que o produto continue disponível, gerando valor, confiável para o usuário e que ele continue evoluindo com o negócio, nós precisamos enxergar a saúde do produto. Enxergar a saúde do produto não quer dizer apenas analisar questões do sistema como, desempenho, *bugs*, quantidade de armazenamento e entre outros. Nós precisamos visualizar como o sistema está impactando o negócio, qual o valor que ele está produzindo e qual a percepção do usuário durante a sua utilização, precisamos além de captar se o sistema ainda está funcionando, de um ponto de vista operacional, temos que verificar se ele ainda está funcionando para o negócio, temos que garantir todo dia que ele seja um facilitador para os usuários e não um problema e principalmente, precisamos sempre verificar se o sistema está indo de encontro com os objetivos do negócio.

Para conseguirmos estes insumos, precisamos primeiramente garantir a confiança do usuário, eles precisam estar seguros nas ações que eles vão realizar no sistema e caso uma melhoria mude a forma que eles realizam as suas atividades, é importante informar os usuários o motivo desta alteração e apresentar o objetivo da empresa com a mudança do processo. Um usuário informado com confiança que o sistema funciona e atende as necessidades dele, se torna um colaborador do sistema, é através do *feedback* dele que conseguimos mapear as melhorias que devemos realizar e como realizar as comunicações de novas implementações. Com base no como o usuário está utilizando o sistema é que conseguimos melhorar os nossos testes automatizados e garantir cada vez mais qualidade para o nosso produto, o negócio está sempre em constante mudança, então precisamos sempre adaptar as nossas funcionalidades para essas mudanças. Através dessas mudanças, novos requisitos vão surgindo, novos limites serão encontrados e novos riscos irão surgir, com isso mais escolhas e mais testes serão necessários para garantir que o sistema continue disponível e aderente ao negócio.

## 8.1 Ouvindo os usuários

Há diversas ações que podemos realizar para conseguirmos captar as percepções do usuário. A primeira delas é perguntar diretamente para eles, utilizar a opinião de cada usuário para tentar identificar as suas opiniões, buscando entender se o produto está aderente ao negócio, melhorias que devem ser implementadas e novas funcionalidades que devem ser desenvolvidas, todos lembrando de classificar a importância de cada nova funcionalidade ou de cada novo requisito. Nestas perguntas é essencial que seja identificado também questões de usabilidade do que já foi desenvolvido, identificar se as operações que eles realizam estão com um bom tempo de resposta, se o *layout* não está confuso, se a forma que eles estão interagindo com as funcionalidades faz sentido para eles, se houve algum *bug* ou situação que alguma operação não funcionou como o esperado. Através destas perspectivas é que conseguimos garantir que os usuários estão satisfeitos ao utilizar o sistema.

Outra ação que devemos realizar para verificar a interação do usuário com o sistema é criar métricas no sistema que informe como ele está sendo utilizado. Podemos medir o tempo que um usuário leva para realizar uma operação, para verificar se determinada operação está com um bom desempenho, quantidade de cliques em tela, para verificar se o usuário precisa realizar muitas atividades para concluir o seu objetivo, campos em um formulário, para identificar se todas as informações ainda fazem sentido o usuário preencher, assim podemos ou automatizar o preenchimento de alguns campos, ou até mesmo remover eles. Para cada funcionalidade devemos elaborar quais métricas produzir para conseguir supervisionar a interação do usuário com o sistema e identificar um *bug* durante a utilização do usuário antes mesmo que ele reporte o problema para o suporte. Outra análise que podemos fazer com esses dados é a de verificar se determinada funcionalidade está apresentando sinais de defeito, podemos averiguar, por exemplo, que determinada funcionalidade nos últimos três dias, está apresentando maior lentidão em sua execução. Com essa informação podemos investigar se esta lentidão está ocorrendo devido a alguma implementação recente, se um servidor está apresentando defeito, se os usuários estão realizando mais requisições para esta funcionalidade, se a distribuição de processamento nos servidores está aderente a utilização dos usuários para cada funcionalidade, entre outras análises. Notamos que somente este dado, nos abriu diversos pontos de análise, que combinando com outras métricas, podemos encontrar a origem do problema, antes mesmo que o usuário perceba. Conseguimos assegurar que o sistema se mantenha disponível e conseguimos manter a confiança do usuário no produto.

## 8.2 Aprendendo com os erros

Com o frequentemente acompanhamento das funcionalidades, vamos conseguir diversos erros cometidos no produto, um *layout* confuso, algum *bug* devido a alguma implementação, um requisito levantado de forma errada, gerando uma funcionalidade funcionando de forma errada, um limite que foi alcançado, um risco que não deveríamos ter aceitado, pode acontecer diversos erros, mesmo com nosso monitoramento contínuo, e o nosso desenvolvimento centrado no negócio. O fato de estarmos monitorando o nosso produto e de desenvolvermos as funcionalidades centrado no negócio, é o que vai nos fazer perceber os nossos erros mais rápido, e também vai auxiliar a encontrar a solução de forma rápida. Uma equipe que entende o negócio, sabe os requisitos de cada funcionalidade, sabe os riscos que aceitamos, conhece os limites do sistema e consegue monitorar a saúde do produto, tem completa capacidade de resolver os problemas e aprender com eles.

Quando nos deparamos com algum problema em produção, devemos analisar ele imediatamente, envolvendo todas as pessoas da equipe que tem relação com o acontecido, seja porque trabalhou na funcionalidade ou as métricas apontam que é necessário a participação de um equipe específica, como, por exemplo, a equipe de infraestrutura, é necessário juntar todos os envolvidos para que possamos realizar a análise de forma conjunta e identificar a tratativa de uma forma ágil. No momento da análise, é importante lembrar que não há culpados, devemos focar na solução do problema, depois de solucionado, é importante documentar o ocorrido com a solução que foi realizada e a origem do problema. Quando aprendemos com os nossos erros e não repreendemos o equipe por um erro cometido, conseguimos garantir que qualquer problema que ocorra, os membros do nosso equipe não vão sentir medo de reportar e todos vão trabalhar em conjunto para resolver o ocorrido. Após resolvido o problema, devemos mapear o que podemos desenvolver para supervisionar o sistema com o objetivo de identificar sinais que o erro vai voltar a ocorrer e que caso ele ocorra novamente, facilite a nossa análise para a sua resolução.

Vale lembrar que um erro, não é somente um mau funcionamento de alguma funcionalidade, mas sim qualquer impacto negativo para o negócio. Uma funcionalidade que não esteja aderente com as necessidades do usuário, deve ser tratada conforme descrevemos anteriormente, todos os envolvidos devem se reunir e discutir o que deve ser feito para atender as necessidades do usuário. Muitas vezes neste caso, os requisitos irão sofrer alterações e o teremos que refazer alguma determinada parte de uma funcionalidade, esta

<b>Funcionalidades</b>	Nome das funcionalidades que foram impactadas.
<b>Cenário</b>	Descrição do cenário em que ocorreu os erros.
<b>Stakeholders</b>	Quais são as áreas de negócio que foram impactadas no cenário descrito.
<b>Problema</b>	Descrição do problema que ocorreu.
<b>Origem do Problema</b>	O que foi que causou o problema descrito.
<b>Equipes Envolvidas</b>	Quais equipes foram envolvidas para a resolução.
<b>Solução</b>	O que foi realizado para resolver o problema.
<b>Novos Desenvolvimentos</b>	O que deve ser desenvolvido para monitorar o cenário descrito com o objetivo de nos alertar ou facilitar a análise caso o erro volte a acontecer.

Tabela 12: Especificação para documentação de resolução de problemas

alteração deverá gerar novos testes automatizados e a arquitetura construída terá que se adaptar a estas mudanças, podendo ter que mudar um componente inteiro para garantir a aderência ao negócio.

### 8.3 Testes automatizados como ferramenta de aprendizado

Conforme vamos realizando mudanças no produto, seja proveniente de algum erro, por mudanças no negócio ou na implementação de alguma melhoria, devemos produzir novos testes automatizados para garantir que os novos requisitos dessas mudanças estejam sendo testados e garantidos.

Além de garantir a qualidade de nossos, os testes têm a função de servir como base de conhecimento, é a partir deles que podemos analisar todos os processos que desenvolvemos para o negócio e explicar o funcionamento das funcionalidades. Nos testes automatizados temos sempre como o usuário vai interagir com o sistema, onde verificamos se todos os tipos requisitos estão sendo garantidos, funcionais, não-funcionais e inversos, conseguimos demonstrar tudo o que foi feito para garantir a qualidade das funcionalidades assim como o seu funcionamento. Desta forma, conseguimos utilizar os testes automatizados, como apresentação do produto como todo, seja para um novo membro da equipe ou para um novo usuário, podemos demonstrar o funcionamento de uma nova funcionalidade, para os usuários envolvidos, podemos explicar como funciona um determinado processo da empresa. Conseguimos através destes testes, uma grande fonte de conhecimento onde com a utilização de um versionador de código, conseguimos apresentar as mudanças que foram realizadas na utilização das funcionalidades. Através dos testes, conseguimos já prever se uma determinada implantação pode gerar algum erro para os usuários, como podemos

observar todo um processo sendo realizado, conseguimos já na etapa de desenvolvimento averiguar se a solução está aderente as necessidades do negócio, verificando se esta nova implementação não dificultou o processo que os usuários já estão realizando ou se gerou algum impacto em outra parte do sistema, que pode ser considerado um erro para o usuário. Conseguimos validar nos testes automatizados, se as mudanças realizadas estão seguindo as mudanças que o negócio está sofrendo, como a equipe tem um grande entendimento do negócio e está realizando seus desenvolvimentos com foco nos objetivos do negócio, podemos verificar o fluxo que será executado pelos usuários nos testes automatizados e validar se as mudanças que ocorreram no negócio foram refletidas corretamente para o sistema.

Devemos ficar atentos as mudanças no negócio, um requisito pode mudar conforme o tempo for passando, e nestes casos, ele pode conflitar com o *core* do que desenvolvemos, quando isto ocorrer, devemos analisar se realmente faz sentido realizar esta alteração no sistema ou se devemos começar a construção de um novo produto, pois, dependendo da mudança, podemos ter que alterar muito a forma que o produto funciona, assim gerando muitos riscos na implementação dessa nova mudança, os requisitos que mapeamos até então os testes que geramos não estão mais aderentes ao que o negócio está se tornando e muitas vezes adaptar o sistema por completo é mais trabalhoso do que desenvolver um novo, desta forma conseguimos já na concepção do novo produto, atender os novos requisitos e desenvolver funcionalidades mais aderentes com o caminho que negócio está trilhando. Para identificar se a algum ocorrido se trata de um erro ou de uma mudança, devemos analisar quanto tempo implementamos os requisitos que estão mudando, uma mudança em uma implementação recente, pode significar um erro ao levantarmos os requisitos, devemos analisar a situação em que a sociedade se encontra, pois, pode ocorrer que algum fator externo, como uma nova lei, redefinição de processo, fusão ou aquisição de alguma empresa, possa ocasionar em uma mudança, onde nesses casos os requerimentos mudaram de repente.

## 8.4 Utilizando os dados adquiridos

Agora que sabemos quais dados precisamos procurar para garantir a qualidade do nosso produto, no processo de desenvolvimento, no monitoramento que realizamos em produção e nos testes em **CHAOSPROD**, precisamos armazenar estes dados em um local que todos os membros do projeto tenham visibilidade para utilizar as informações que eles trazem e com base neles agregar valor ao negócio. As especificações das funcio-

nalidades podemos controlar através de um sistema de gerenciamento de projetos (Jira, Trello, Github, etc.), que nos permite desenhar todo o nosso fluxo de entrega e acompanhar a funcionalidade fase a fase, sempre adicionando os novos requisitos e alterando os requisitos que já foram concluídos, sempre que uma alteração for necessária, podemos voltar a funcionalidade para a fase de escrita e iniciar o fluxo novamente. Devemos também possuir um sistema que possamos mapear as *issues* que venham a ocorrer com o monitoramento do sistema e com os testes em **CHAOSPROD**, o ideal é que este sistema seja o mesmo que o de gerenciamento de projeto, pois, conseguimos consolidar no mesmo sistema, as funcionalidades, os requisitos, os critérios e as ocorrências, facilitando a nossa análise e tomada de decisões. É importante registrarmos as ocorrências e marcar as *issues* geradas, é através destas informações que conseguimos identificar melhorias no nosso produto, conseguindo especificar uma funcionalidade nova, melhorar uma funcionalidade já existente, identificar mais métricas para supervisionarmos o nosso produto, é através das *issues* que conseguimos iniciar o nosso fluxo de entrega.

Todo o código gerado, qualquer documentação que auxiliou o desenvolvimento (lembrando re respeitar a política de proteção de dados da empresa e do país), e os dados gerados através dos testes automatizados em conjunto com a cobertura do código, desta forma, todos os membros do projeto tem acesso ao código que foi desenvolvido, as informações que auxiliaram na construção da informação e os resultados dos testes automatizados, podendo através dos *pull requests* realizar diversas análises para aferir a qualidade da solução implementada. Através do *pull request* devemos classificar o que estamos implementando se o que estamos implementando se trata de um *bug fix*, uma melhoria, uma nova funcionalidade, um novo monitoramento, e além desta classificação, devemos relacionar o nosso *pull request* com as *issues*, os requisitos que solucionamos, os critérios que garantimos, desta forma, podemos mapear exatamente o motivo desta implementação e assegurar que os testes que foram realizados foram suficientes para assegurar que a funcionalidade continua em funcionamento e aderente ao negócio, pois, neste *pull request* é importante não só adicionar o código desenvolvido mas também os resultados dos testes automatizados.

Os dados provenientes do nosso monitoramento em produção e dos testes realizados em **CHAOSPROD**, devem ser armazenados em alguma ferramenta de *analytics* (Google Analytics, Grafana, etc.), estes dados devem ser armazenados para identificarmos como está a saúde de nosso ambiente, ou seja, se estamos nos aproximando dos limites que identificamos ou se o sistema está apresentando sinais que vai ocorrer algum defeito, e também devemos utilizar estes dados para analisar o histórico do passado de modo a definirmos nossas estratégias para assegurar o funcionamento do sistema e a qualidade de



nossas funcionalidades. É através destes dados que conseguimos visualizar o que precisamos melhorar e o que podemos “piorar”, como conseguimos ver quais funcionalidades estão utilizando mais processamento, quais páginas os usuários estão mais utilizando, qual o tempo de resposta médio de cada operação, conseguimos tomar decisões se devemos aumentar a quantidade de recursos computacionais para determinada funcionalidade que está sendo muito utilizada pelos usuários e diminuir de uma funcionalidade não está sendo muito utilizada. Podemos através destes dados identificar como melhorar o processo que os usuários estão executando, automatizando alguma operação e diminuindo a quantidade de acessos de uma determinada página. Através da análise destes dados identificamos novas *issues*, construir mais planos para manter a nossa escalabilidade e iniciar novamente o nosso fluxo de entrega para rever alguma funcionalidade e desenvolver uma melhoria que baseado em dados, vai gerar valor para o negócio.

## 9 RESULTADOS DA PROPOSTA

Ao desenvolver um produto, muitas vezes focamos apenas em requisitos funcionais, nos concentramos em entregar o máximo número de funcionalidades para os usuários e esquecemos dos requisitos não-funcionais. Conforme apresentado no trabalho *Quality Attributes and Service-Oriented Architectures* [O'Brien, Bass e Merson 2005], os requisitos não-funcionais estão diretamente ligados com a qualidade de nosso *software*, questões como desempenho, usabilidade, segurança e entre outros, são requisitos que garantem que o nosso produto funciona e que as nossas funcionalidades desenvolvidas, estão aderentes com o negócio.

Um produto de qualidade deve estar sempre aderente ao negócio, os usuários devem conseguir utilizar o sistema e sentir que as necessidades deles estão sendo atendidas, eles devem sentir segurança a utilizar o sistema e as funcionalidades devem ter um tempo de resposta razoável, para que eles se sintam a vontade ao realizar as suas atividades e para que a experiência do usuário não se torne frustrante. É importante sabermos que não vamos conseguir atender todos os requisitos e que teremos que enfrentar riscos no nosso produto, e conforme apresentado no *The Architecture Tradeoff Analysis Method* [Kazman et al. 1998] com base nos riscos que analisamos, devemos buscar remodelar a arquitetura que desenvolvemos para o nosso produto para que o nosso produto continue com a mesma qualidade e possamos mitigar, resolver ou conviver com esses riscos. A experiência do usuário e o valor que o produto está gerando para o negócio, está diretamente atrelado a qualidade que cada funcionalidade foi desenvolvida e quais os riscos que estamos convivendo, é importante sempre revalidar com o negócio, se o que está desenvolvido continua aderente, se algum risco que estamos convivendo deveria ser mitigado e com essas análises, se devemos investir em reformular a arquitetura montada, para se adaptar as novas definições, desenvolver uma nova funcionalidade, atender mais alguma necessidade do negócio ou se devemos começar a desenvolver um novo produto, pois, o negócio está sofrendo diversas alterações que estão descaracterizando o produto desenvolvido e que o objetivo do sistema que construímos foi cumprido e agora a empresa possui um novo objetivo.

Quando desenvolvemos nosso produto em conjunto com o negócio, conforme apresentado

no Domain Driven Design [Evans 2003] conseguimos criar uma linguagem que será utilizada para que as áreas de negócio e a equipe de desenvolvimento consigam se comunicar e que desta forma, a equipe de desenvolvimento consegue captar melhor os requisitos das funcionalidades e os usuários conseguem compreender mais facilmente como o sistema funciona. Além deste entendimento entre as duas partes, é mais criar um código mais limpo, pois os padrões e os nomes dos objetos e das entidades serão nomeados baseado em um entendimento que todos na equipe adquiriram ao se comunicar com o negócio. Padrões e estilo de desenvolvimento bem definidos entre os membros do equipe de desenvolvimento facilita a compreensão e manutenção do código, como apresentado no Clean Code [Martin 2008], a maior parte de nosso tempo utilizamos lendo o nosso código do que escrevendo linhas novas, e se o código estiver representando o negócio, tanto em sua funcionalidade quanto em sua escrita, ele se torna a documentação do projeto, demonstrando como cada parte do sistema funciona e quais requisitos ele está atendendo.

Além de garantir que o nosso sistema está aderente com o negócio, e que seja possível adaptar a sua arquitetura conforme as necessidades do negócio precisamos garantir que ele continue em funcionamento em todas as situações, precisamos utilizar de métricas, produzir *logs*, conversar com o usuário, precisamos garantir que caso um problema venha a acontecer, conseguimos supervisionar a saúde do ambiente e encontrar a raiz do problema de forma rápida, e após a sua resolução, aprender com ele e desenvolver novas formas de supervisão, para que consigamos verificar se o sistema está apresentando sinais que o erro vai voltar a ocorrer. Com o aprendizado que obtivemos com o erro, agora podemos elaborar estratégias para nos preparar para futuras situações, seja aumentar a quantidade de servidores durante um determinado período, seja particionar mais o processamento de alguma funcionalidade do sistema, ou seja, até mesmo desativar por completo uma funcionalidade que não está mais sendo utilizada, mas ainda consome recursos computacionais, através do negócio e com as situações que enfrentamos dia a dia, podemos traçar estratégias, redefinir a nossa arquitetura e adquirir maior consciência sobre o negócio na totalidade, pois desta forma estamos utilizando o negócio para definir o sistema e o sistema para aprender sobre o negócio.

# PARTE IV

## CONCLUSÃO

## 10 RESULTADOS EM RELAÇÃO AO OBJETIVO

Nosso objetivo foi que com a utilização desta abordagem possibilite que o desenvolvimento do produto seja realizado com qualidade e com escalabilidade, onde as estratégias de melhoria sejam tomadas com base nas mudanças no mercado e na sustentação realizada em produção, com base nas métricas adquiridas e na forma que o usuário utiliza o sistema, através da análise desses dados, seja possível se preparar para problemas, compreender as necessidades dos usuários e direcionar as equipes nas tomadas de decisões sobre como o produto deve ser desenvolvido, como garantir que os requisitos estejam sendo assegurados e que os riscos em cada implementação estejam evidentes para todos os envolvidos no projeto.

O sistema deve trazer novas visões do negócio, e é através dos dados que são gerados que conseguimos captar como o negócio funciona e desenvolver estratégias para adquirir maior vantagem competitiva mapeando novas necessidades para o nosso produto, fazendo surgir novos requisitos e alterando requisitos existentes, onde desta forma, conseguimos avaliar se o nosso produto continua aderente ou se ele já cumpriu o seu objetivo assim descobrindo a necessidade de começar o desenvolvimento de um novo.

A escalabilidade de nosso produto antigo permite que nossas funcionalidades que foram separadas em sistemas, nosso controle sobre os objetivos de cada uma delas, nossa infraestrutura adaptativa e nossa arquitetura evolutiva, possibilite que as partes que ainda estão aderentes com os objetivos da empresa se mantenham e as partes que não estão mais aderentes sejam trocadas, mantendo a mesma infraestrutura e focando que o novo produto seja desenvolvido mais rapidamente e mantendo os aprendizados do produto anterior e verificando se o que vai ser reutilizado, quando implementado novamente, se mantém funcionando através dos testes automatizados e do monitoramento em produção.

## 11 TRABALHOS FUTUROS

Como futuro trabalho, podemos evoluir esta abordagem como um método focando nas questões de arquitetura e em requisitos de qualidade, aonde vamos especificar, com mais detalhes como garantir a escalabilidade e detalhando os fatores de qualidade como segurança, disponibilidade, entre outras. Podemos evoluir os processos de desenvolvimento especificados com mais detalhes apresentando a abordagem sendo utilizada em projetos reais.

## REFERÊNCIAS

- Beck 2002 BECK, K. *Test Driven Development: By Example*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0321146530.
- Beyer et al. 2016 BEYER, B. et al. *Site Reliability Engineering: How Google Runs Production Systems*. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2016. ISBN 149192912X.
- Chacon e Straub 2014 CHACON, S.; STRAUB, B. *Pro Git*. 2nd. ed. USA: Apress, 2014. ISBN 1484200772.
- Evans 2003 EVANS. *Domain-Driven Design: Tackling Complexity In the Heart of Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321125215.
- Humble e Farley 2010 HUMBLE, J.; FARLEY, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2010. ISBN 0321601912.
- Kazman et al. 1998 KAZMAN, R. et al. *The Architecture Tradeoff Analysis Method*. Pittsburgh, PA, 1998. Disponível em: <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=13091>.
- Kim et al. 2016 KIM, G. et al. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. [S.l.]: IT Revolution Press, 2016. ISBN 1942788002.
- manifesto for agile software development MANIFESTO for agile software development. Disponível em: <https://agilemanifesto.org/>.
- Martin e Martin 2007 MARTIN, R.; MARTIN, M. *Agile Principles, Patterns, and Practices in C#*. [S.l.]: Prentice Hall, 2007. (Robert C. Martin series). ISBN 9780131857254.
- Martin 2008 MARTIN, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1. ed. USA: Prentice Hall PTR, 2008. ISBN 0132350882.
- O'Brien, Bass e Merson 2005 O'BRIEN, L.; BASS, L.; MERSON, P. *Quality Attributes and Service-Oriented Architectures*. Pittsburgh, PA, 2005. Disponível em: <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7405>.