UNIVERSIDADE DO SUL DE SANTA CATARINA

DÉBORA DA SILVA

BDD: GERAÇÃO DE MASSA DE DADOS VIA API UTILIZANDO *TEMPLATES* DE NEGÓCIOS PARA TESTES AUTOMATIZADOS

Florianópolis

2019

DÉBORA DA SILVA

BDD: GERAÇÃO DE MASSA DE DADOS VIA API UTILIZANDO *TEMPLATES* DE NEGÓCIOS PARA TESTES AUTOMATIZADOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade do Sul de Santa Catarina, como requisito parcial à obtenção do título de Bacharelado em Ciência da Computação.

Orientador: Prof. Dr. Maria Inés Castiñeira.

Florianópolis

2019

DÉBORA DA SILVA

BDD: GERAÇÃO DE MASSA DE DADOS VIA API UTILIZANDO *TEMPLATES* DE NEGÓCIOS PARA TESTES AUTOMATIZADOS

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

Florianópolis, 20 de novembro de 2019.

Prof. Maria Inés Castiñeira, Dra.
Universidade do Sul de Santa Catarina

Prof. Aran B. T. Morales, Dr.
Universidade do Sul de Santa Catarina

Elton João Gubert, Esp.
Agriness

Dedico este trabalho a minha mãe Maria Marlene Hemsing (*in memorian*), que eu sei que sentiria orgulho de mim neste momento.

AGRADECIMENTOS

A Universidade do Sul de Santa Catarina.

Ao ProUni - Programa Universidade Para Todos e a todos os seus idealizadores e responsáveis.

A professora, orientadora e amiga, Maria Inés Castiñeira, que sempre acreditou em mim e se manteve ao meu lado nessa trajetória, em uma linda demonstração de carinho e paciência.

Aos meus irmãos Janir da Silva, Josimari Cristina da Silva e Agnaldo da Silva por estarem ao meu lado em momentos muito difíceis, momentos difíceis meus e momentos difíceis compartilhados nossos. Não importa aonde eu vá, vocês são o meu lar.

A todos os colegas de curso que porventura contribuíram em alguma parte desse processo.

Aos meus sobrinhos André Felipe Beirão e Maria Alice da Silva Leite pelo amor, e por encher meu coração de alegria e de esperança nas pessoas e no mundo.

A todos os funcionários da Agriness, que estiveram ao meu lado, me dando apoio e suporte nesse momento, e que repetiram em uníssono “Vai!”. Agradeço especialmente a Thaiana Lopes por tudo.

A Elisa Ferreira, Alana Durgante e Charles Sartori pelas revisões.

As minhas amigas e amigos que me apoiaram nesse processo.

A Mateus Ventura por ter me ajudado a ter os elementos para começar esse trabalho.

A O. B. F. por ter ficado ao meu lado me dando suporte durante a concepção de parte deste trabalho.

A H. N. M. pela inspiração para terminar esse trabalho.

A todos os membros da banca, Professora Maria Inés Castiñeira, Professor Aran B. T. Morales e Elton Gubert.

E a todos que possam ter alguma responsabilidade pelo início, meio ou fim dessa graduação.

Do fundo do meu coração, sou grata a todos vocês.

“Pasmo sempre quando acabo qualquer coisa. Pasmo e desolo-me. O meu instinto de perfeição deveria inibir-me de acabar; deveria inibir-me até de dar começo. Mas distraio-me e faço. O que consigo é um produto, em mim, não de uma aplicação de vontade, mas de uma cedência dela. Começo porque não tenho força para pensar; acabo porque não tenho alma para suspender. Este livro é a minha cobardia.” (FERNANDO PESSOA, 1982).

RESUMO

A demanda por produtos de qualidade é crescente. Em um ambiente ágil, existe a necessidade de errar rápido para corrigir rápido e, dessa forma, se adequar às exigências dos clientes, que a cada dia se tornam mais minuciosas. Os testes automatizados são um elemento essencial para a garantia dessa qualidade, tanto daquilo que está sendo criado quanto do que já existe, mas que precisa manter-se funcionando. Quando o assunto se trata de testes funcionais via interface, é preciso estratégia para que a execução seja confiável e ágil. Com base nesse contexto, o presente trabalho apresenta um estudo de caso de utilização de técnicas do BDD para geração de massa de dados via API dentro da empresa Agriness – empresa de gestão da informação para o agronegócio. Nessa empresa existe um projeto de testes, que foi desenvolvido utilizando ferramentas disponíveis no mercado hoje, como Git, Bitbucket, Jenkins, Jira, *Robot Framework* e suas bibliotecas. Pensando na otimização desse projeto, foram criados *templates* de negócios para a geração dessa massa, fazendo com que os testes de interface, que são os testes mais caros, estejam focados apenas em validar os elementos da interface, sem a necessidade da preparação desses dados. Como objetivo principal, foi desenvolvida uma solução para a geração de massa de dados utilizando *templates* em formato BDD, de forma que as configurações dos eventos necessários para a execução de cada suíte de testes são enviadas através de requisições via API, dessa forma preparando o ambiente para validação. Este estudo permitiu observar, através da análise dos resultados da execução de um grupo de testes, uma redução no tempo de 02 h 44 min 11 s. E ainda permitiu, zerar a quantidade de erros provocados pelo mau funcionamento de camadas intermediárias. Por fim, os testes que passaram a utilizar essa estratégia obtiveram maior grau de manutenibilidade, independência e reusabilidade. Concluiu-se que a elaboração inteligente de estratégias pode trazer muitas vantagens para um ambiente de desenvolvimento de *software*, diminuindo o tempo de execução de testes, zerando erros provocados por mau funcionamento de intermediários e melhorando a qualidade do código.

Palavras-chave: BDD. *Templates* de negócio. API Requests. Massa de dados. Testes automatizados.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Em um mercado cada vez mais competitivo e exigente é necessário que as empresas de desenvolvimento de *software* elaborem seus produtos com qualidade e no tempo previamente acordado com o cliente. Para que isso aconteça é necessário contar com o apoio de uma equipe de qualidade que acompanhe o processo de desenvolvimento desde a sua especificação.

Segundo Pressman (2006, p.349):

> Qualidade de *software* é a satisfação de requisitos funcionais e de desempenho explicitamente declarados, normas de desenvolvimento explicitamente documentadas e características implícitas que são esperadas em todo *software* desenvolvido profissionalmente.

Algumas das ações necessárias para atingir a qualidade no desenvolvimento de *software* são a validação e a verificação do produto que está sendo desenvolvido (MOLINARI, 2003). Nesse contexto, a aplicação de testes é um processo estratégico para garantir a qualidade. Muitas empresas costumam realizar os testes no tempo que resta para a entrega do projeto, outras nem mesmo conseguem realizar os testes e é o próprio usuário quem identifica os erros. Esse tipo de estratégia costuma trazer grandes problemas e grandes gastos com manutenção e retrabalho.

> Verificação reduz o custo geral do *software* na prevenção, se for usado através do processo de desenvolvimento. Na prática a redução de defeito é 4 para 1, se usarmos verificação, pois um defeito encontrado em produção custa de 20 a 100 vezes mais do que se fosse encontrado antes (MOLINARI, 2003 p. 24).

Molinari (2003) afirma que um *software* de qualidade, do ponto de vista do cliente, é aquele que atende aos requisitos solicitados. Do ponto de vista do usuário, é aquele que faz o que ele precisa. Algumas empresas acreditam que o alto investimento em qualidade desde o princípio do desenvolvimento significa alto custo para o projeto.

A necessidade de dar respostas de *software* com qualidade e rapidez a ambientes de negócios mutáveis levou a geração de novas estratégias de testes que permitem agilidade na criação e execução dos testes. Tais estratégias podem ser responsáveis tanto pelo suporte a equipe, quanto pela crítica ao produto. Ou ainda ser voltados para a validação do negócio ou verificação da tecnologia.

Gregory e Crispin (2015) tratam sobre as características dessas estratégias de testes, e ressaltam que os processos criados a partir disso foram desenvolvidos com o mesmo objetivo

de criar uma compreensão comum para construir a coisa certa da primeira vez. Um desses processos é conhecido como BDD (*Behavior Driven Development*) cujo objetivo é criar cenários em linguagem natural no formato "Dado que, Quando, Então" que expressam o comportamento de uma funcionalidade.

O *template* "Dado que, Quando, Então" é uma das características mais marcantes do BDD, ele é usado na construção de estórias de usuário no momento da análise para descrever os critérios de aceite. North (2006) afirma que quando um sistema atende os critérios de aceite, então ele se comporta corretamente, sendo assim as estórias escritas utilizando este *template* são compreendidas e utilizadas por qualquer pessoa envolvida no processo de desenvolvimento.

O presente projeto tratará da aplicação de algumas técnicas propostas pelo BDD na preparação do ambiente de testes em uma aplicação onde os pré-requisitos são a parte mais importante e complexa do teste.

1.1 PROBLEMÁTICA

A necessidade das empresas de desenvolvimento de *software* de entregar produtos com qualidade em um tempo mínimo fez com que durante anos o período alocado para a execução dos testes fosse negligenciado. Consequentemente a área de testes sempre foi tratada como uma área sem muita importância. Dessa forma, atividades de teste que deveriam ser executadas com atenção e detalhe por profissionais treinados para o exercício das regras de negócio acabavam sendo executadas pelos próprios desenvolvedores.

O tempo passado após a entrega de um produto ou funcionalidade impacta diretamente na dificuldade e no custo dos erros encontrados. Ou seja, a correção dos erros descobertos após a entrega do produto é mais cara e difícil de resolver do que um erro descoberto na fase de desenvolvimento. A frequência, a quantidade, e o impacto desses erros, são negativos de várias formas, pois podem provocar prejuízos financeiros para o cliente, perda de credibilidade na relação empresa e cliente, tempo da equipe gasto para entender novamente todo o contexto da ocorrência do erro e desenvolver uma solução, entre outros.

> Custos de falhas externas consistem nos custos descobertos depois que os produtos foram entregues e liberados. Quanto mais falhas externas forem encontradas, mais desastroso será para a reputação da organização ou resultará em perda de faturamento futuro (MOLINARI, 2003 p. 23).

Para reduzir esses erros encontrados após a aplicação estar em produção é preciso aplicar testes. Tradicionalmente os testes realizados pelos desenvolvedores da aplicação não são dedicados a regra de negócio e sim a garantir o funcionamento técnico da aplicação.

> Testes unitários verificam o comportamento de uma única classe ou método que é uma consequência de uma decisão de design. Este comportamento normalmente não é diretamente relacionado aos requisitos, exceto quando uma parte chave da lógica de negócio é encapsulada dentro da classe ou do método em questão. Estes testes são escritos pelos desenvolvedores para seu próprio uso; eles ajudam desenvolvedores a descrever como "concluído se parece" resumindo o comportamento da unidade na forma de testes (MESZAROS, 2007, tradução nossa).

Uma proposta para superar estes desafios poderia ser abraçar um modelo de desenvolvimento de *software* que incorpore já desde as primeiras etapas as preocupações com qualidade e testes. Assim, a proposta deste trabalho é apresentar um estudo de caso de aplicação prática de técnicas utilizadas no modelo BDD de forma que a geração da massa de dados para testes siga esse modelo e ao mesmo tempo seja criada de forma rápida através do uso de requisições para API.

Segundo Smartbear (2019) uma *Application Programming Interface* (API) permite a interação entre dois sistemas, ela fornece o contrato e a linguagem para que dois sistemas se comuniquem. A extremidade de um canal de comunicação é conhecimento de *endpoint*, que é a forma que dois sistemas se contactam. Para APIs, uma URL de um servidor ou serviço pode estar associadas ao *endpoint*, que ao ser chamado permite acessar o recurso necessário e executar as funções a que está associado.

A respeito do modelo BDD cabem as primeiras indagações: Quais são as características principais desse modelo? Quais as suas vantagens e desvantagens? Como ele deve ser aplicado ao longo do projeto de desenvolvimento? Que tecnologias e ferramentas estão disponíveis para apoiar o processo?

Essas temáticas não são estudadas nos cursos de graduação devido a restrições de tempo, mas seria importante se aprofundar nesses assuntos. Dessa forma o foco deste trabalho é a descrição detalhada da aplicação de algumas técnicas do modelo BDD em testes via API.

1.2 OBJETIVOS

Serão apresentados a seguir o objetivo geral e os objetivos específicos.

1.2.1 Objetivo Geral

Apresentar um estudo de caso com a utilização de técnicas do BDD para geração de massa de dados via API utilizando *templates* de negócios para testes automatizados.

1.2.2 Objetivos Específicos

a) Pesquisar os fundamentos de BDD.

b) Pesquisar a teoria de testes de *software*, automação de testes, arquitetura de *software* e API.

c) Apresentar as ferramentas utilizadas no estudo de caso pela equipe de testes na organização (*Robot framework*, biblioteca Collections, Git, Python, Bitbucket, Jenkins, Atom, biblioteca RequestsLibrary, Jira).

d) Aprofundar o conhecimento nas teorias e tecnologias aplicadas em testes, BDD, e preparação de massas de dados para teste.

e) Elaborar sugestões de melhorias, ou apontar fragilidades, através da análise do estudo de caso, comparando com as teorias apresentadas.

1.3 JUSTIFICATIVA

Molinari (2003 p. 22) afirma que o gerenciamento da qualidade diminui os custos porque os defeitos serão localizados e corrigidos cedo e de forma tal que o custo desse defeito será menor ao longo do tempo.

Segundo Myers (2004, apud MASUDA 2009):

> (...) quanto antes forem descobertos os erros da aplicação, mais barato será a sua manutenção. O custo aumenta exponencialmente a cada etapa do processo no qual ele é encontrado, ou seja, o custo da correção é maior para cada fase do ciclo de desenvolvimento do *software* em que o erro passou.

O impacto da criação de um produto de *software* sem qualidade é muito grande, e será sentido pela empresa, pelo cliente e, dependendo do produto, até pela comunidade. A seguir estão listadas algumas consequências da não aplicação de um processo com qualidade:

- Atraso ou não entrega do produto.

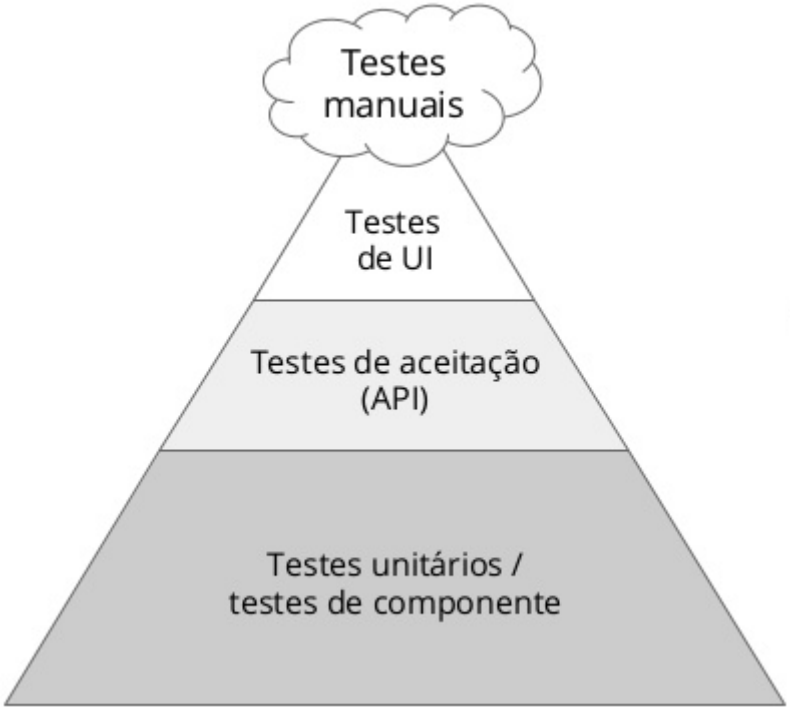
- Problemas financeiros para a empresa desenvolvedora devido a pagamento de multas pelo atraso.
- Impacto negativo no orçamento em consequência de custos não planejados com manutenção.
- Risco de vida para os usuários e terceiros no caso de sistemas críticos, por exemplo, sistemas de controle aéreo.
- Sistemas ou funcionalidades indisponíveis regularmente, ou se comportando anormalmente.
- Impacto negativo nos processos dependentes do funcionamento do sistema.

Os testes automatizados atuam principalmente de forma preventiva. Todos os itens citados anteriormente podem ser reduzidos quando um processo de testes é aplicado de forma organizada e sistemática. Dessa forma pode se verificar a importância da sua aplicação para atingir um produto com qualidade.

Uma das principais vantagens dos testes automatizados é a repetição da execução de testes para a validação periódica do comportamento de um *software*. Funcionalidades críticas, cujo comportamento precisa ser monitorado de perto, são sempre excelentes candidatas a automação, pois de outra forma sua validação estaria suscetível a erros. Esses testes uma vez criados são executados sempre, não sendo uma atividade cansativa, mas sim uma atividade que traz segurança e agilidade para todo o processo de testes.

No entanto, dependendo tipo de teste aplicado, às vezes é necessário pensar em estratégias para maior confiabilidade e aproveitamento dos testes que estão sendo executados. No caso de testes funcionais que são executados via interface, é preciso criar uma independência para que somente o que precisa ser validado seja executado via tela. Uma das estratégias para que isso aconteça é a geração da massa de dados através utilizando outros recursos que não a própria interface.

Figura 1 - Pirâmide da automação de testes



Fonte: Crispin e Gregory (2009).

Segundo Crispin e Gregory (2009) a pirâmide da automação de testes representada da Figura 1 demonstra como é o cenário ideal de testes aplicados sobre um *software*, ou seja, é necessário empurrar o máximo possível de testes para a base da pirâmide, pois é essa camada a fundação que dá suporte a todo resto.

A camada base, que é a camada contendo os testes unitários e de componentes, é a camada que é escrita e executada de forma mais rápida e barata, e, portanto, é a que mais pode ser usada para a criação de testes robustos. Esses testes têm por principal objetivo validar o comportamento da tecnologia.

A camada de testes de aceitação (camada da API) é a camada que realiza os testes de aceitação, validando a funcionalidade sem passar pela interface. Ao contornar a camada de apresentação, eles são menos caros para escrever e manter do que os testes de interface.

A camada de testes de interface, ou camada de testes de GUI (*Graphical User Interface*), é a camada mais cara para escrever, assim como é a camada traz menor retorno de investimento. Essa camada tem por objetivo navegar pela camada de apresentação, operando e manipulando os elementos da interface. E para dessa forma criticar o produto e estes testes vão para as suítes de testes de regressão.

Entende-se assim que apesar de trazerem *feedbacks* muito importantes para o time, eles são testes caros e que precisam ser construídos de forma estratégica, ou seja, cada operação executada deve fazer parte da validação e trazer algum retorno, assim como ser independente em sua execução. Se todos os dados necessários para a execução desses testes estão prontos no momento em que o teste irá iniciar sua execução, então isso trará independência entre cada operação que precisa ser executada.

1.4 ESTRUTURA DA MONOGRAFIA

O capítulo 1 apresenta o assunto, o problema, os objetivos e a justificativa.

O capítulo 2 aborda o referencial teórico sobre qualidade de *software*, processo de testes, tipos de testes e arquitetura de *software*.

No capítulo 3 encontra-se a metodologia utilizada no desenvolvimento deste trabalho.

O capítulo 4 descreve o estudo de caso.

Finalmente é apresentado o capítulo com as conclusões e trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta a fundamentação teórica do trabalho. Serão apresentados conceitos relacionados com qualidade, processo de desenvolvimento de *software* e testes de *software*.

2.1 PROCESSO DE DESENVOLVIMENTO DE *SOFTWARE*

O desenvolvimento de *software* trata-se do incremento de um projeto baseado em funções, ações e práticas organizadas pelo processo. Segundo Sommerville (2003) este processo também pode ser definido como "um conjunto de atividades e resultados associados que levam a produção de um produto de *software*".

A definição de um processo de desenvolvimento de *software* deve ser baseada nas necessidades e recursos de uma equipe, tornando-o mais complexo conforme as variáveis envolvidas (número de indivíduos, tamanho e complexidade do produto, tempo do cronograma, entre outros). Sommerville (2003) assegura que esses processos, bem como todos os outros processos intelectuais, devem sofrer um julgamento humano, uma vez que é necessária a apreciação e a criatividade para decidir o método ideal para cada realidade.

2.1.1 Atividades essenciais ao desenvolvimento de *software*

Os processos de desenvolvimento de *software* sofreram grandes alterações desde a sua concepção, houve aprimoramentos e adaptações, tudo para alcançar um modelo de *software* ideal. Parreiras e Oliveira (2004) dizem que apesar das grandes mudanças nos processos de desenvolvimento de *software*, as atividades fundamentais se mantêm, sendo elas a especificação de *software*, projeto e implementação, validação, e evolução de *software*.

Segundo Soares (2011) as definições de cada fase fundamental para o desenvolvimento de *software* são:

- **Especificação de *software*:** É a etapa na qual é realizado o levantamento junto ao cliente dos requisitos de *software* e de suas restrições, ou seja, é definido o que o *software* deverá fazer e como ele o fará.
- **Projeto e implementação:** Essa é a fase de construção do *software*. A construção do *software* baseia-se em modelos, criados a partir das solicitações

dos clientes, para codificar esses modelos é necessário escolher uma linguagem de programação que melhor se adeque a resolução do problema.

- **Validação do *software*:** Nessa fase o *software* passa por validação para verificar se o mesmo atende às especificações.
- **Evolução do *software*:** O *software* precisa evoluir para acompanhar as necessidades do cliente.

Os processos de *software* foram criados para garantir que o projeto de software seja construído de forma a atender às necessidades do cliente. No entanto, para aplicar e identificar as falhas nesses processos, é necessário que os indivíduos estejam capacitados. Infelizmente, empresas de pequeno e médio porte, onde a falta de pessoal qualificado é um problema comum e consequentemente a atenção ao processo perde prioridade em meio a todas as outras atividades inerentes aos papéis dos membros das equipes. Segundo Soares (2011) isso geralmente acontece porque os processos tradicionais não são adequados à realidade dessas empresas, dessa forma é comum que muitas delas não utilizem nenhum processo, podendo resultar em produtos finais de baixa qualidade.

2.1.2 Modelos de processos de *software*

Para garantir a qualidade no desenvolvimento de *software* foram criados muitos modelos e metodologias de desenvolvimento de *software*, que foram consagrados durante os anos no meio do desenvolvimento de *software*.

Sommerville (2011) classifica a forma de desenvolver *software* da seguinte maneira:

- **Modelo em Cascata:** Neste modelo as atividades fundamentais do processo de desenvolvimento de *software* são representadas como fases distintas.
- **Desenvolvimento Incremental:** As atividades fundamentais são intercaladas liberando uma série de versões, sempre adicionando novas funcionalidades à versão anterior. Diversos modelos de processo de *software* fazem parte desta categoria, por exemplo o RUP ou vários modelos ágeis.
- **Desenvolvimento orientado ao reuso:** Temos aqui uma forma de desenvolver na qual o sistema de *software* não é iniciado do zero. Ou seja, o *software* é componentizado, e cada nova funcionalidade é adicionada de forma modular ao que já existe até alcançar o objetivo final.

Utilizar somente as características de um dos modelos não é uma regra. É perfeitamente aceitável mesclar os modelos existentes para definir a estrutura e o sistema de entregas ideal para cada sistema de *software* e para cada equipe ou empresa. Essas características foram usadas como base para a definição dos modelos ou das metodologias de desenvolvimento de *software* disponíveis no mercado.

Os modelos também podem ser classificados como sendo ágeis ou tradicionais, Pressman (2006) define estes últimos como prescritivos, e Sommerville (2011) define como dirigidos a planos. Entre os modelos tradicionais destacam-se: cascata, espiral, prototipação, RUP. Algumas desvantagens dessas metodologias tradicionais provocaram o surgimento de abordagens que prezavam pela flexibilidade durante o processo de desenvolvimento, para dessa forma atender e entender de fato as necessidades do usuário e das alterações nos requisitos de *software*. Surgiram, então, as metodologias ágeis.

Paula Filho (2012) diz que os métodos ágeis mais populares atualmente são *Scrum, Crystal Clear*, *Adaptive Software Development*, *Feature Driven Development* (**FDD**), *Dynamic Systems Development Method* (**DSDM**), e *Extreme Programming* (**XP**), entre outras.

2.1.3 Metodologias ágeis de desenvolvimento

As metodologias de desenvolvimento ágil surgiram na década de 1990 como uma solução para os problemas existentes nas metodologias dirigidas a planos, nas quais o processo era moroso, longo e demorado, mesmo para sistemas corporativos de pequeno e médio porte. Sommerville (2011) afirma que nos anos 1980 e início dos anos 1990 havia a percepção de que o melhor *software* é aquele obtido por meio de um processo rigoroso e controlado. Quando esse tipo de metodologia é aplicada para sistemas de pequeno e médio porte o overhead gerado é tal que prevalece sobre todo o processo de desenvolvimento de *software*.

Dessa forma, a cultura ágil mostrou-se uma alternativa ao processo de desenvolvimento de *software*.

Sommerville (2011) afirma que todos os métodos ágeis são baseados no modelo incremental, dessa forma, há a ideia de entrega contínua permitindo ao cliente a mudança de requisitos durante o projeto. Sendo assim, as equipes sempre voltam seus esforços para definir apenas o que será entregue em cada estória, e não em desenhar todo o sistema desde o início, uma vez que é difícil prever qual será exatamente o resultado final. O objetivo dessa metodologia é minimizar a burocracia e entregar *software* executável rapidamente de forma que

o cliente possa assim propor incrementos para as próximas fases do processo. Esta filosofia foi descrita no manifesto ágil:

Estamos descobrindo melhores maneiras de desenvolver *software*, fazendo-o nós mesmos e ajudando os outros a fazerem o mesmo. Através deste trabalho, passamos a valorizar:

> Indivíduos e interações mais que processos e ferramentas.
> *Software* em funcionamento mais que documentação abrangente.
> Colaboração com o cliente mais que negociação de contratos.
> Responder a mudanças mais que seguir um plano.
> Ou seja, mesmo havendo valor nos itens à direita, valorizamos mais os itens à esquerda. (BECK et al., 2001).

Embora as metodologias ágeis sejam todas baseadas no modelo incremental, elas possuem suas próprias características e processos. Além do modelo em que são baseadas, as metodologias ágeis compartilham do mesmo princípio, conforme descrito no manifesto ágil.

Segundo o manifesto ágil as metodologias ágeis priorizam indivíduos e interações, *software* funcionando, colaboração com o cliente, e reação à mudança mais do que processos e ferramentas, documentação abrangente, negociação de contratos, e seguir um plano. Isso significa que os primeiros itens são priorizados, mas não significa que os últimos serão desconsiderados. No entanto enquanto que nos modelos dirigidos à plano são priorizadas a formalidade e as documentações, no modelo ágil temos equipes focadas no trabalho cooperativo. No Quadro 1 a seguir o autor Paula Filho (2012) faz uma comparação entre os modelos ágeis e os modelos tradicionais ou dirigidos a planos.

Quadro 1 – Métodos ágeis x Métodos dirigidos a planos

Área	Característica	Métodos ágeis	Métodos dirigidos a planos
Desenvolvedores	Cultura	Mais informal	Mais formal
	Tamanho da equipe	Menor	Maior
	Experiência	Maior	Menor
	Localização	Única	Espalhada
Clientes	Relacionamento	Mais cooperativo	Mais formal
	Localização	Próxima	Distante
Requisitos	Natureza	Emergentes	Bem conhecidos
	Variabilidade	Alta	Baixa
Arquitetura	Foco	Solução imediata	Expansibilidade
	Refatoramento	Fácil	Difícil

Projeto	Tamanho	Menor	Maior
	Objetivo	Valor rápido	Alta garantia

Fonte: Paula Filho, 2012

As metodologias SCRUM e XP são duas das metodologias mais populares atualmente. Ambas seguem o manifesto ágil, no entanto possuem algumas diferenças entre si. Segundo Pressman (2011) o XP é baseado num conjunto de cinco valores, que são a base de todo o trabalho executado enquanto aplicando esta metodologia. São esses os valores:

- Comunicação.
- Simplicidade.
- *Feedback* (realimentação ou retorno).
- Coragem.
- Respeito.

Como comunicação entende-se a colaboração estreita, mesmo que informal entre todos os membros da equipe, aplicando o uso de metáforas eficazes, ou seja, histórias criadas a partir de linguagem ubíqua, compreensível a todos, de forma que todos os membros do processo possam entender até mesmo as peças mais complexas do sistema.

Para manter a simplicidade, o XP sugere que os desenvolvedores trabalhem apenas nos requisitos planejados para a estória atual, aplicando o refatoramento se necessário em uma fase seguinte, ou seja, melhorando o código, mas mantendo o foco nos objetivos atuais, buscando sempre códigos simples, utilizando o mínimo possível de classes e métodos.

O XP busca o *feedback* contínuo, seja do *software*, da equipe de desenvolvimento ou do próprio cliente. O *software* provê *feedback* através dos resultados dos testes, e do comportamento dos casos de usos. O cliente provê *feedback* em cada iteração, solicitando alterações e melhorias, e sugerindo novas funcionalidades. A equipe de desenvolvimento provê *feedback* através dos impactos, sejam nos custos ou no cronograma.

Uma equipe de desenvolvimento de *software* precisa de coragem enquanto aplicando XP, pois é fácil durante o processo de desenvolvimento se desviar dos caminhos traçados pela metodologia. Uma equipe com coragem continua aplicando os valores, mesmo sofrendo pressões externas ou internas.

Por último, uma equipe precisa de respeito para com seus membros, com outros envolvidos e para o próprio *software*. Segundo Pressman (2011) uma equipe que segue todas

as fases do processo de desenvolvimento XP e assim como as fases segue também os valores propostos passa a ter respeito pela própria metodologia.

Medeiros (2006) afirma que os testes devem ser criados antes do início do desenvolvimento, sejam os testes de aceitação, sejam os testes unitários. Além disso, a base para a criação dos testes unitários deve ser os critérios de aceitação levantados.

Pensando nessa mudança de paradigma, houve também uma mudança de mentalidade do profissional de testes. Pois o seu papel deixou de ser caçador de erros e culpados, e passou a ser de protetor do cliente. Crispin e Gregory (2009) afirmam que as responsabilidades de um testador ágil incluem minerar informações, trabalhar com o cliente e com os membros da equipe fazendo com que os requisitos fossem expressados adequadamente, e providenciando uma resposta rápida e contínua a cada novo fragmento de *software*.

Nos últimos anos as metodologias ágeis têm tomado conta do mercado de desenvolvimento de *software*, consequentemente influenciando diretamente a forma de testar *software*. Dentro dessas metodologias se desenvolveu o BDD que será apresentado mais adiante, após o conceito do que é qualidade de *software*. Este modelo será então abordado neste trabalho.

2.2 QUALIDADE

Um dos principais objetivos da Engenharia de *Software* é garantir a qualidade do *software*. Para Pressman (1995, p. 724) qualidade de *software* é definida como: “Conformidades a requisitos funcionais e de desempenho explicitamente declarados, a padrões de desenvolvimento claramente documentados e a características implícitas que são esperadas de todo *software* profissionalmente desenvolvido”.

Apesar da necessidade de entregar um produto de *software* com qualidade, muitas empresas focam apenas na codificação do *software*. Pressman (1995, p. 723 - 724) diz que muitas vezes a preocupação com a garantia de qualidade do *software* surge somente depois que o *software* foi codificado. Esse tipo de atitude faz com que muitas vezes o *software* não atenda aos requisitos do cliente.

A garantia de qualidade de *software* (*Software Quality Assurance* - SQA) é uma “atividade de guarda-chuva” que é aplicada ao longo de todo o processo de engenharia de *software*. A SQA abrange (1) métodos e ferramentas de análise, projeto, codificação e teste; (2) revisões técnicas formais que são aplicadas durante cada fase da engenharia de *software*; (3) uma estratégia de teste de múltiplas fases; (4) controle da documentação de *software* e das

mudanças feitas nela; (5) um procedimento para garantir a adequação aos padrões de desenvolvimento de *software* (quando aplicáveis); e (6) mecanismo de medição e divulgação (PRESSMAN, 1995 p. 723 - 724).

O teste de *software* (Validação e Verificação de *software*) é uma parcela do processo de qualidade de *software* e é desta parcela que trata esta monografia.

2.2.1 Características da qualidade de *software*

Segundo McConnell (2005 p. 498-499) um *software* pode ter características de qualidade internas e externas. As características externas são as características percebidas tanto pelos desenvolvedores quanto pelo usuário final. Os fatores que podem ser considerados características externas são:

- Correção.
- Usabilidade.
- Eficiência.
- Confiabilidade.
- Integridade.
- Adaptabilidade.
- Precisão.
- Robustez.

Cada uma dessas características externas será descrita na próxima seção.

Os programadores por sua vez preocupam-se com as características internas, são elas:

- Manutenibilidade.
- Flexibilidade.
- Portabilidade.
- Reusabilidade.
- Legibilidade.
- Capacidade de Teste.
- Inteligibilidade.

Destas características algumas se sobrepõem, no entanto cada uma tem um papel importante dentro do desenvolvimento de *software* e estes papéis não podem ser ignorados.

2.2.1.1 Características Externas

Esse tipo de característica é o único tipo com a qual o usuário final tem contato, pois o interesse do usuário está em se o sistema é fácil de usar e se funciona corretamente. A dificuldade para modificar o sistema não é percebida pelo usuário, portanto as características internas não são do seu interesse.

A primeira característica externa associada à qualidade de *software* é a Correção. McConnell (2005 p. 498) diz que a Correção caracteriza o grau em que um *software* está livre de falhas em todo o contexto (especificação, design e implementação), ou seja, caracteriza o grau em que um *software* atende as especificações e os requisitos definidos.

Durante a especificação de um sistema é comum os clientes exigirem que o sistema seja usável, ou seja, possua Usabilidade, no entanto há certa dificuldade de mensurar se um *software* possui ou não essa característica. Quando um produto possui alto grau de Usabilidade os usuários aprendem a usá-lo rapidamente e não se sentem inseguros durante a navegação no sistema. Durante a validação e verificação de um *software* é necessário avaliar certos critérios práticos. Pezzè e Young (2008) afirmam que existem pontos específicos, que quando em conformidade, garantem um mínimo de qualidade ao sistema:

- Baixo tempo de resposta às solicitações do usuário.
- Quantidade reduzida de operações para execução de uma tarefa.
- Tratamento à operações inválidas realizadas pelo usuário.
- Entre outros.

Além desses itens, Pezzè e Young (2008) alegam que para garantir a Usabilidade de um sistema é fundamental que alguns itens relacionados a outras características de qualidade do *software* sejam atendidas, como:

- Fidelidade aos requisitos funcionais.
- Bom desempenho do sistema.
- Bom tratamento de problemas referentes à segurança.
- Entre outros.

Eficiência refere-se ao uso dos recursos computacionais e ao seu aproveitamento, McConnell (2005 p. 498) diz que a eficiência é referente ao uso mínimo de recursos como memória e tempo de execução.

Um sistema confiável é aquele que continua executando suas operações normalmente mesmo em circunstâncias anormais, tais circunstâncias devem ser previamente definidas para que sejam prevenidas. Segundo Sommerville (2003) os itens que devem ser atendidos, para que um sistema possua Confiabilidade, são:

- Tempo médio para falhar.
- Baixa probabilidade de indisponibilidade.
- Baixa taxa de ocorrência de falhas.
- Disponibilidade.

Integridade é referente à Segurança do sistema, um sistema íntegro não permitirá a manipulação indevida dos seus componentes e dos seus dados. Sommerville (2003) escreve sobre problemas de segurança “Se puder ser demonstrado que esses defeitos não podem ocorrer ou que, se eles ocorrerem, o perigo associado não resultará em acidentes, então o sistema é seguro.”.

Adaptabilidade é referente ao abalo que a alteração de um requisito pode provocar em outro requisito. Sommerville (2003) sustenta que durante a revisão dos requisitos, os revisores devem validar sua adaptabilidade realizando os seguintes questionamentos:

- O requisito é adaptável?
- Um requisito pode ser alterado sem que outros requisitos sejam impactados?

Precisão é referente ao quão bem o *software* realiza o trabalho para o qual ele foi construído, principalmente relacionado às saídas quantitativas. Pezzè e Young (2008) sustentam que é necessário que a equipe de qualidade faça a verificação do sistema, garantindo que o resultado final desenvolvido seja consistente com o resultado final esperado e especificado no projeto.

Robustez é a capacidade de um sistema continuar funcionando mesmo em condições anormais (entradas inválidas ou condições ambientais adversas). Sommerville (2003) sustenta

que um sistema deve possuir certas características que garantem o funcionamento distinto de um *software* robusto:

- Rápido reinício do sistema após a ocorrência de uma falha.
- Baixa porcentagem de eventos que causam falha.
- Baixa probabilidade de que os dados sejam corrompidos por falhas.

2.2.1.2 Características Internas

Características internas são referentes aos fatores que são vistos somente pelos desenvolvedores do *software*, elas levam em consideração questões como se o *software* é fácil de modificar, se o código é legível e bem estruturado, entre outros.

O primeiro fator referente às características internas é a Manutenibilidade, que leva em consideração o quão fácil é modificar um *software*. Pressman (1995) define Manutenibilidade como "A facilidade com que você pode modificar um sistema de *software* para alterar ou acrescentar recursos, melhorar o desempenho ou corrigir falhas".

Segundo McConnell (2005), o fator conhecido como Flexibilidade descreve o quanto um sistema de *software* pode ser modificado para que opere em ambientes diferentes ou que se adeque a usos diferentes.

A Portabilidade é o fator que descreve a facilidade com que um sistema pode ser migrado para um ambiente diferente daquele que ele foi projetado. De acordo com Sommerville (2003) isso pode ser mais facilmente atendido quando durante o desenvolvimento, os desenvolvedores atentem para os seguintes pontos:

- Cautela para que a porcentagem de declarações dependentes dos sistemas-alvo seja reduzida.
- Que o número de ligações aos sistemas-alvo seja mínimo.

Segundo Pressman (1995) Reusabilidade é a forma como as partes de um sistema podem ser aproveitadas, isso pode ser possível através da catalogação eficiente dessas partes, bem como a sua padronização. A catalogação facilita a consulta, e a padronização facilita a integração e isso permite que a reusabilidade possa ser aplicada de eficientemente.

A Legibilidade é descrito por McConnell (2005) como a facilidade de ler e entender o código-fonte de um sistema.

O fator que descreve o grau com que se pode verificar que um sistema atende aos requisitos através da execução de testes de unidade e de sistema é conhecido como Capacidade de Teste. Sommerville (2003) afirma que durante a revisão dos requisitos os revisores deverão verificar se um requisito é passível de ser testado, ou seja, se de alguma forma é possível testá-lo. Um sistema testável deve ter seus requisitos redigidos clara e detalhadamente de forma que cada ponto escrito seja testado.

McConnell (2005, p. 499) descreve Inteligibilidade como:

> A facilidade com que você pode compreender um sistema em termos da sua organização geral e das instruções detalhadas. A inteligibilidade está relacionada com a coerência do sistema em um nível mais geral do que a legibilidade proporciona.

2.3 TESTE DE *SOFTWARE*

Rios e Moreira (2006) dizem que nas décadas de 1960 e 1970, os desenvolvedores dedicavam-se na maior parte de projeto em codificar e realizar os testes unitários. Acredita-se que nessa época cerca de 80% dos esforços eram dedicados às essas atividades. Os testes de sistema tinham o único objetivo de provar aos usuários que o sistema funcionava.

Testar sistemas de grande porte é de fato uma tarefa muito cansativa, pois muitos testes são refeitos diversas vezes até que se tenha certeza de que o módulo a que o teste se refere forneça a saída esperada às entradas fornecidas. Refazer os testes, repetitivamente, até que se tenha certeza de que o erro foi solucionado faz com que a equipe desperdice muito do seu tempo hábil. A solução para esse problema é a automação dos testes, que permite a otimização do tempo disponível para testes, que com uma gestão inteligente possibilita a equipe de testes um foco maior na elaboração de testes, permitindo a equipe de qualidade usar a criatividade em testes mais produtivos e elaborados.

Os processos de testes que fazem parte do desenvolvimento de *software* passaram a ser mais valorizados a partir dos anos 1980. Rios e Moreira (2006) afirmam que:

> A partir dos anos 1980, durante o processo de desenvolvimento passou a ser dada maior importância à análise de requisitos, ao desenho funcional e técnico dos novos sistemas. Um esforço maior passou a ser dedicado à integração das diversas partes que compunham os *software*s e ao teste destes para funcionarem como um sistema. As atividades de testes passaram a ser tratadas como processo formal, aparecendo Metodologias de Testes que evoluíram até os dias de hoje.

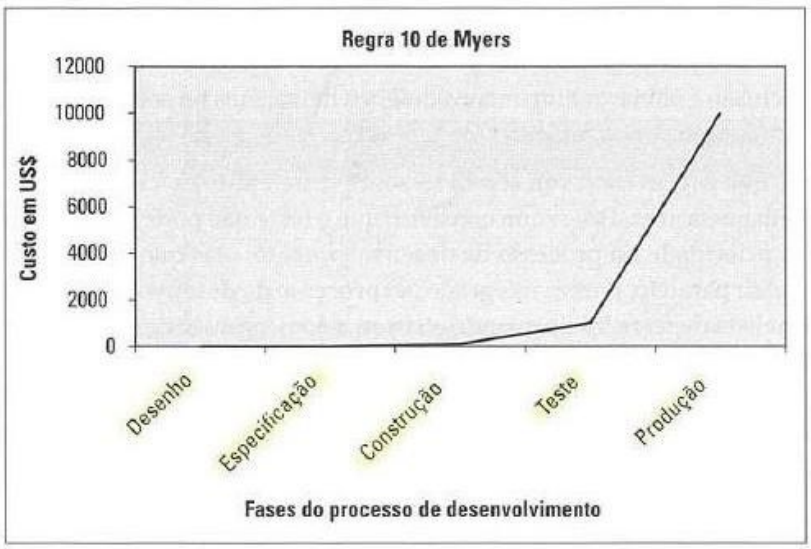
Os profissionais da área de testes estão se profissionalizando cada vez mais, suas tarefas giram exclusivamente em torno do teste de *software*, que é realizado de forma padronizada e com objetivo. O desenvolvimento dos testes é baseado nos requisitos especificados, que são o norte do documento de teste. Rios e Moreira (2006) dizem que testar um *software* é provar que o *software* está fazendo o que deveria fazer, seguindo os requisitos levantados, e provar que não está fazendo o que não deveria fazer.

Um profissional de teste, durante a elaboração do documento de teste deve levar em consideração as entradas de dados esperadas e também as entradas de dados não esperadas. Segundo Hetzel (1988, apud RIOS e MOREIRA) qualquer atividade usada para provar que o *software* alcança os resultados desejados também é considerada teste de *software*. Myers (1979, apud RIOS e MOREIRA 2006) afirma que uma parte do processo de teste de *software* é executar um programa ou sistema para procurar defeitos, e esse tipo de atividade também é conhecido como teste negativo.

A Figura 2 representa o gráfico que mostra a regra de Myers. Segundo Rios (2006) encontrar um erro antes ou durante o desenvolvimento é bem mais barato do que encontrar um erro depois que o *software* é colocado em produção.

Analisando o gráfico é possível notar que quanto mais cedo um erro for encontrado, mais barato ele irá custar. Depois que o *software* é colocado em produção o erro pode custar até 1000 vezes mais do que se ele fosse encontrado nas fases iniciais. O gráfico permite uma visualização clara de que o custo da manutenção aumenta gradativamente conforme as fases de desenvolvimento se passam.

Figura 2 - Regra 10 de Myers



Fonte: Rios, 2006. p.29

Rios (2007) concorda com a regra de Myers e reforça que é mais vantajoso descobrir os erros nas fases iniciais do desenvolvimento, por isso é necessário fazer os testes desde o início, aplicando metodologias de teste de forma estruturada e criticando os requisitos desde o princípio.

2.3.1 Fases do processo de testes

Rios e Moreira (2006) afirmam que o processo de testes está dividido em fases, como pode ser visto na Figura 3. As fases de um processo de testes são:

- **Procedimentos Iniciais:** Os procedimentos iniciais devem se dar antes de qualquer fase do processo de testes, nessa fase é definida a forma de abordagem dos testes, quem serão os envolvidos (usuários, desenvolvimento, teste e produção), definição do objetivo, definição das responsabilidades de cada um, definição do plano preliminar de trabalho, avaliação de riscos, esclarecimento dos níveis de serviço acordados e definição dos itens considerados relevantes para garantir o sucesso do projeto.
- **Planejamento:** O Planejamento ocorre durante todo o processo de testes. No Planejamento é necessário, inicialmente, fazer uma definição prévia da Estratégia de Testes e do Plano de Testes, no entanto essas definições podem e devem ser alterados durante o processo de testes. A revisão delas deve ser periódica para que não se tornem obsoletas.
- **Preparação:** Assim como o Planejamento, a Preparação ocorre durante todo o processo de teste. A Preparação é a fase que trata da preparação do ambiente de teste para o desenvolvimento dos trabalhos, isso inclui preparação dos equipamentos, da rede, do pessoal, do *software* e das ferramentas que auxiliarão o processo.
- **Especificação:** A Especificação ocorre logo após os procedimentos iniciais. Na Especificação são elaborados todos os critérios de aceitação e, no caso de testes automatizados, os scripts. Nessa fase também são realizados os testes estáticos, ou seja, são realizados os testes de verificação dos requisitos do sistema.
- **Execução:** A Execução ocorre logo após a Especificação dos testes. A Execução usa de todos os planejamentos realizados na fase de Especificação,

pois é nessa fase que são executados todos os testes planejados seguindo os critérios de aceite definidos. É nessa fase também que os scripts são executados. Todos os resultados obtidos devem ser registrados.

- **Conclusão:** A Conclusão é a última fase de todo o processo. O processo de testes deve ser concluído com a entrega da funcionalidade para o ambiente de produção.

Figura 3 - Fases de um processo de testes

Fonte: Rios e Moreira (2006)

As fases do desenvolvimento do projeto de *software* devem seguir uma ordem cronológica passando por todas as etapas. Rios (2007) afirma que a metodologia de teste deve ser compatível com a metodologia de desenvolvimento de *software* para que o início dos dois seja ao mesmo tempo e que o projeto seja avaliado desde o início.

2.3.2 Validação e Verificação

Existem duas formas de testar um *software*, uma é conhecida como Validação e a outra é conhecida como Verificação. Sommerville (1997, p. 446) diz que as atividades de Verificação e Validação (V & V) têm por objetivo garantir que o *software* atenda o que foi especificado e esteja de acordo com os requisitos do cliente.

Segundo Pressman (1995) a Verificação verifica se o *software* realiza corretamente as funções codificadas, enquanto que a Validação verifica se o *software* atende aos requisitos previamente levantados com o cliente.

Rios (2007) diz que é possível fazer a diferenciação entre Verificação e Validação fazendo as seguintes perguntas:

- Estamos fazendo corretamente o sistema? (Verificação)
- Estamos fazendo o sistema correto? (Validação)

Rios (2007) também diz que existem diferentes atividades para cada tipo de teste. As atividades para testes do tipo verificação podem ser:

- Revisões de requisitos.
- Revisões de modelos.
- Inspeções de código.
- Revisões e inspeções técnicas em geral.

As atividades para testes do tipo validação podem ser:

- Teste unitário.
- Teste de integração.
- Teste de sistemas.
- Teste de aceitação.
- Homologação.
- Teste de regressão.

Para que o processo de teste seja eficiente é necessário realizar primeiro os testes de verificação depois os testes de validação, pois a verificação avalia a documentação gerada a partir dos levantamentos de requisitos e é necessário testar se toda essa documentação está correta. Segundo Rios (et al. 2007):

(...) as atividades de verificação e validação estão distribuídas nas diversas etapas do processo de teste e de uma maneira genérica, a verificação é executada antes da validação. Ou seja, verificamos os documentos antes de testar o *software*.

Segundo Pressman (1995) os testes não podem ser vistos como a garantia de que o *software* está sem erros, como dizem “Não se pode testar a qualidade. Se ela não estiver lá antes de você começar a testar, não estará lá quando você tiver terminado de testar”. É necessário usar de princípios de qualidade de *software* durante todo o processo de engenharia de *software*.

2.3.3 Documentação

Os testes devem ser executados seguindo uma documentação que deve ser confeccionada antes de iniciados os testes. Os documentos de testes devem atender a todos os estágios do *software*, começando com a revisão dos requisitos, continuando através do design até a revisão do código para o teste do produto.

O objetivo da documentação é, além de manter um registro de todos os problemas encontrados, organizar a forma que os testes são realizados para que estes atendam à metodologia da empresa. Rios e Moreira (2006) dizem que: "A metodologia de testes deve ser o documento básico para organizar a atividade de testar aplicações no contexto da empresa. Como é indesejável o desenvolvimento de sistemas sem uma metodologia adequada, também acontece o mesmo com a atividade de testes.".

É quase impossível desenvolver um *software* livre de erros, no entanto é possível minimizá-los. Através dos testes de *software* é possível reduzir muito os erros existentes antes de ser colocado em produção. Para que os testes de *software* tenham sucesso, é necessário que a correção do erro seja garantida.

A Figura 4 representa a forma como deve ser tratado um erro encontrado. É possível perceber que após a correção do erro, o programa deve ser retestado para garantir que o erro não existe mais. O reteste envolve, além do módulo em que o erro foi encontrado, tudo aquilo que de alguma forma está envolvido com o erro, pois um erro corrigido pode gerar muitos outros.

Figura 4 - O processo de depuração

Fonte: Pressman (1995, p. 447)

Durante o processo de criação e execução de testes são gerados vários artefatos de testes. Esses documentos são importantes pois fazem parte da análise e definição de quais e como cada teste será executado, ajudam também na definição de prioridades, assim como a fazer o rastreamento de possíveis problemas encontrados. Cada um desses documentos possui objetivos e características distintas, conforme IBM (2019):

- ***Script* de testes:** Cada caso de testes é associado com um script de testes. Um script de testes contém as instruções para implementar um caso de testes que pode ser executado de forma manual ou automatizada. Nem todos os scripts de testes são referentes a testes funcionais, eles podem se referir também a testes de performance e testes de segurança.
- **Casos de teste:** Um caso de testes é um cenário de teste, e este tem o objetivo de executar o mesmo script de testes para validar uma ou várias configurações diferentes. Por exemplo, um script de testes de login pode ser executado em vários navegadores diferentes conforme a configuração da execução.
- **Suíte de testes:** Suítes de teste agrupam um conjunto de casos de testes que são os elementos que simulam uma transação completa dentro do *software*, cada caso de testes contém os seus respectivos scripts de testes. É possível dentro de uma suíte de testes criar dependências entre cada caso de testes, de forma que se um caso de testes falhar, toda a execução irá falhar também.

2.4 DESENVOLVIMENTO GUIADO POR COMPORTAMENTO

O Desenvolvimento Guiado por Comportamento parte do princípio que para validar uma funcionalidade é necessário prever e verificar os comportamentos possíveis. North (2006) diz que se os métodos não descrevem de forma compreensível o comportamento do sistema, então eles dão uma falsa sensação de segurança.

A seguir está descrito o funcionamento do processo Behavior-Driven Development (BDD), que foi desenvolvimento baseado na ideia de que a melhor forma de validação é através do desenvolvimento focado no comportamento do sistema ou funcionalidade.

2.4.1 BDD – Behavior-Driven Development

Antes de falar sobre Behavior-Driven Development é importante falar sobre Test-Driven Development (TDD), ou ainda desenvolvimento guiado por testes, uma vez que para identificar a importância de se desenvolver um processo que torne o comportamento o cerne do desenvolvimento, é necessário entender o funcionamento do desenvolvimento guiado por testes e suas lacunas.

Crispin e Gregory (2009) colocam que a escrita de testes de alto nível no princípio do desenvolvimento é uma forma de fornecer informações o suficiente para os programadores escreverem o seu próprio TDD. Seguindo a linha de que os insumos para criação de testes estão sempre sendo fornecidos para os programadores, é necessário se mover rapidamente para que o desenvolvimento do código não se distancie da criação de testes, que pode, eventualmente, se mover na direção errada.

North (2006) foi a figura chave para a criação do que conhecemos hoje como Behavior-Driven Development (BDD) e ele descreveu todo o processo de desenvolvimento e amadurecimento. Sendo assim, ele afirma que o ato de testar é intrínseco ao TDD, ou seja, quando os programadores estão aplicando TDD um conjunto de métodos pode garantir que o código funciona, mas que se esses métodos não descrevem de forma compreensível o comportamento do sistema, então os programadores estão vivendo uma falsa sensação de segurança.

Todo o processo de amadurecimento do BDD se deu a partir das definições do que é e como funciona o TDD. North (2006) deixa claro que ficou muito mais fácil delimitar o objetivo do processo quando compreendeu que o que precisava ser validado era o comportamento e não o teste. Pois na verdade a definição do que é teste passou a ser o comportamento, ou seja, o teste é criado e executado baseado em uma sentença descrevendo o próximo comportamento que se espera que o sistema ou a funcionalidade tenha.

Neste momento ocorreu o entendimento de que a criação dos testes deveria incorporar não só os desenvolvedores, mas todos os outros indivíduos que fazem parte do processo de desenvolvimento de *software* (analistas, testadores, desenvolvedores, e equipe de negócio).

Figura 5 - *Template* para criação de estórias de usuário

```
1. Como um [X]
2. Eu quero [Y]
3. Para [Z]
```

Fonte: da autora, 2019

Foi a partir de um *template* para criação de estórias de usuário, que pode ser observado na Figura 5. Onde X é a funcionalidade, Z é o benefício ou valor da funcionalidade, e X é a pessoa ou papel que se beneficiará. North (2006) então desenvolveu um *template* em linguagem ubíqua para criação de cenários que contenham os critérios de aceitação de uma estória. Sendo

que se um sistema atende os critérios de aceitação, então ele está se comportando corretamente. Na Figura 6 é possível verificar o resultado desse desenvolvimento.

Figura 6 - *Template* para criação de critérios de aceitação

```
1. Dado que [X]
2. Quando [Y]
3. Então [Z]
```

Fonte: da autora, 2019

A Figura 6 descreve o *template* base que é a principal característica do BDD. Além de "Dado que", "Quando" e "Então" também é possível usar o "E" que conecta os fragmentos de um cenário entre si. Cada fragmento de um cenário gerado deve ser detalhado o suficiente para ser representado diretamente no código. E fica então sob responsabilidade da ferramenta de automação, que irá ler esse cenário, conectar os fragmentos e os executar.

Dentro do contexto de um cenário de testes, os fragmentos associados a "Dado que" tem a função de preparar o ambiente para a execução dos fragmentos associados a "Quando", que executa as ações do cenário em si, essas ações resultam em um retorno que é lido e avaliado pelos fragmentos associados a "Então".

Resumindo, cada item do *template* refere-se a:

- **Dado que:** *setup* dos dados do teste.
- **Quando:** passo a passo que irá gerar um retorno para validação.
- **Então:** comportamento esperado após a execução de todos os passos.

Segundo *Robot Framework Foundation* (2019) a ferramenta *Robot Framework* é uma ferramenta de automação guiada por palavras-chave (*keywords*). Então um cenário criado usando a ferramenta *Robot Framework* com o objetivo de validar um critério de aceitação, nada mais é do que um conjunto de palavras-chave escritas no formato BDD. E cada palavra chave representa um fragmento de um cenário. E isso é parte do que trata a presente monografia.

2.5 AUTOMAÇÃO DE TESTE DE *SOFTWARE*

A automação de testes de *software* trata-se da criação de *scripts* com o objetivo de agilizar o processo de testes. Segundo Bernardo e Kon (2008) o teste automatizado pode ser definido como "(...) programas ou *scripts* simples que exercitam funcionalidades do sistema sendo testado e fazem verificações automáticas nos efeitos colaterais obtidos.". Dessa forma os testes transformados em *scripts* são reproduzíveis e consequentemente os erros são replicáveis.

Além disso, os *scripts* dos testes automatizados precisam ser simples de forma que a manutenção não seja custosa, é necessário também garantir que eles sejam reutilizáveis para que a sua vida útil se prolongue. Segundo Nogueira (2010) é necessário criar padrões de desenvolvimento de testes e garantir que sejam seguidos para que os testes automatizados sejam reutilizáveis e de fácil manutenção.

2.5.1 Vantagens e Desvantagens

A automação de testes possui muitas vantagens, apresentando aparente resolução de problemas de qualidade, no entanto para implantar a automação em uma empresa que não está preparada é necessário planejamento, disciplina e conhecimento para que haja um retorno desejado. Caetano (2012b) afirma que muitas empresas buscam garantir a qualidade dos *softwares* a qualquer preço, no entanto não fazem uma preparação para o processo de automação, adquirindo uma ferramenta sem qualquer maturidade e dando abertura para o fracasso do projeto.

A principal vantagem da automação de testes é a facilidade de reproduzir os erros encontrados. Bernardo e Kon, (2008) dizem que na automação de testes todos os casos de teste podem ser facilmente e rapidamente repetidos a qualquer momento e com pouco esforço, facilitando o trabalho dos profissionais de testes quando houver a necessidade de reprodução dos eventuais erros encontrados.

Outra grande vantagem da automação de testes é a possibilidade de gerar scripts de requisitos complexos como testes de cálculos matemáticos e casos onde são necessários muitos passos, esses scripts poderão ser reutilizados diversas vezes com a garantia de que o resultado gerado está correto, não estando tão suscetível a erros.

> (...) como os casos para verificação são descritos através de um código interpretado por um computador, é possível criar situações de testes mais bem elaboradas e complexas do que as realizadas manualmente, possibilitando qualquer combinação de comandos e operações. A magnitude dos testes pode também facilmente ser alterada.

> Por exemplo, é trivial simular centenas de usuários acessando um sistema ou inserir milhares de registros em uma base de dados, o que não é factível com testes manuais. (BERNARDO e KON, 2008)

Os testes funcionais automatizados podem ser baseados na interface gráfica ou na lógica de negócio. Caetano (2012b) diz que ambos possuem vantagens e desvantagens específicas de seu tipo.

A principal vantagem da automação de testes baseados na interface gráfica é que o sistema não precisa ser alterado para que os testes sejam realizados. Caetano (2012b) afirma que não é necessário modificar a aplicação, nem tornar a aplicação mais fácil de testar para que o teste automatizado seja realizado porque os testes se baseiam na mesma interface utilizada pelo usuário.

Por outro lado, esse tipo de teste automatizado possui muitas desvantagens, desvantagens essas que podem tornar os testes úteis apenas por pouco tempo. Sobre essa questão Caetano (2012b) esclarece que esse tipo de teste é dependente da interface e como ela é muito mutável qualquer pequena alteração torna-o obsoleto. Além disso, eles não são úteis quando é necessária a realização de qualquer tipo de teste que exija muitas repetições, como testes de regras que usam cálculos complexos, teste de integração entre sistemas e assim por diante.

A interface gráfica é a forma de interação entre o usuário e o sistema, ela é responsável pela entrada e saída dos dados de forma que a lógica de negócio seja abstraída pelo usuário.

Quando a automação de testes é baseada na lógica de negócio, ao contrário da automação baseada na interface gráfica, os scripts são construídos em cima da lógica de negócio, fazendo-se necessária a alteração da aplicação para facilitar a construção do teste. Caetano (2012b) afirma que essas alterações expõem as funcionalidades da aplicação, o que permite a criação dos testes em cima dessa camada.

A principal vantagem da automação de testes baseada na lógica de negócio é que os erros identificados durante a execução desses testes provocam maior impacto ao negócio do cliente, para se ter uma ideia cerca de 80% das falhas ocorridas em produção são consequência de erros nessa camada. Esses testes podem ser independentes da interface gráfica, pois as regras de negócio devem possuir uma independência da interface. Caetano (2012b) afirma sobre as vantagens desse tipo de automação:

> Independência das mudanças de interface gráfica. Alto desempenho para testes automatizados que exigem centenas de milhares de repetições, testes de

> funcionalidades que realizam cálculos complexos, integração entre sistemas diferentes e assim por diante.

As desvantagens da automação baseada na lógica de negócio são sobre as alterações na aplicação e a criação dos testes automatizados. Caetano (2012b) diz que para ser possível esse tipo de automação são necessárias grandes alterações na aplicação para expor as funcionalidades ao mundo exterior, assim como são necessários profissionais especializados em programação.

Crispin e Gregory (2009) descrevem algumas vantagens da automação de testes quando estes são aplicados em um ambiente ágil. Em um ambiente ágil a implementação ou não da automação pode ser a diferença entre o sucesso e o fracasso. Além disso, há várias razões para implementar a automação de testes, e essas razões estão listadas a seguir:

- Testes manuais demoram muito.
- Processos manuais são propensos a erros.
- Automação deixa as pessoas livres para fazer o seu melhor trabalho.
- Testes de regressão fornecem uma barreira de segurança.
- Testes automatizados fornecem *feedbacks* rápidos e frequentes.
- Testes e exemplos que guiam a codificação podem fazer mais.
- Testes fornecem documentação.
- Automação podem ser um bom retorno de investimento.

Além das desvantagens apresentadas anteriormente, Crispin e Gregory (2009) dizem que há outros problemas que podem impactar a automação de testes. Ao contrário da lista de vantagens listadas anteriormente, parte destes itens não se aplicam às equipes que aplicam e seguem as práticas definidas pelas metodologias ágeis. Veja a seguir:

- Dar atenção a automação de testes apenas no tempo livre não é o suficiente.
- Falta de objetivos claros de automação.
- Falta de experiência com automação.
- Equipes com alta rotatividade, pois toda a experiência adquirida com a automação é perdida.
- Escolher a automação como uma reação desesperada, tornando a automação um desejo e não uma proposta realística.

- Pode haver certa relutância ao pensar na criação de testes, pois a parte mais interessante é a automação e não a criação de testes.
- Foco em resolver o problema tecnológico pode causar à equipe perder de vista se os resultados encontram as necessidades de teste.
- Programadores que não estão acostumados a um processo que inclui automação de testes podem ter dificuldades de se adaptar.
- A curva de aprendizagem ao iniciar o processo de automação de testes pode ser dolorosa.
- Falta de investimento inicial.
- Mudanças frequentes no *software*, seja na interface, na API ou mesmo no banco de dados.
- É muito mais difícil criar testes em *softwares* com código legado do que criar testes para códigos novos.
- Para aqueles que não dominam a automação de testes pode ser assustador abraçar o desafio de dar o pontapé inicial.
- Dificuldades para se adaptar pode induzir a equipe a não desenvolver os testes automatizados corretamente.

Para escolher o tipo de teste ideal para a empresa é necessária uma avaliação das condições e necessidades, assim como uma preparação através de treinamento de pessoal, e disciplina por parte de todos os membros da equipe.

2.5.2 Tipos de automação de teste de *software*

Segundo Molinari (2010) existem diversos tipos de automação de teste de *software* e os principais são:

- Automação de Testes Unitários.
- Automação de Gerência e Planejamento de Testes.
- Automação de Testes de Performance.
- Automação de Testes Funcionais.

Os testes unitários são testados em nível de componente ou classe. Esses testes são executados por desenvolvedores durante o período de desenvolvimento. Assim como as outras formas de automação, a elaboração da automação de Testes Unitários é baseada na documentação. Pressman (1995) fala sobre a automação de Testes Unitários:

> Usando a descrição do projeto detalhado como guia, caminhos de controle importantes são testados para descobrirem erros dentro das fronteiras do módulo. A complexidade relativa dos testes e os erros detectados como resultado deles são limitados pelo campo de ação restrito estabelecido para o teste de unidade. O teste de unidade baseia-se sempre na caixa branca, e esse passo pode ser realizado em paralelo para múltiplos módulos.

A automação de Gerência e Planejamento de Testes é automação responsável pela gerência dos documentos dos testes, mantendo-os organizados. Molinari (2010) afirma que "Automatizar a gerência de testes significa também automatizar o processo de testes, ou o controle do processo que gira em torno dos casos de testes".

Quando falamos sobre os testes de performance, a automação é a melhor saída, pois o objetivo desse teste é verificar o desempenho do sistema quando submetido a uma carga muito grande de acessos concorrentes. Quando o cliente solicita que o sistema seja capaz de suportar o acesso mútuo de um determinado número de pessoas é necessário recorrer a automação, pois essa é a forma garantida de ter um resultado consistente e confiável. Sobre essa questão Molinari (2010) afirma que:

> Em testes de performance não há meio termo no que se refere à automação. Fazer manualmente é loucura ou incompetência. Um gerente tentou fazer um teste de performance manual com 50 pessoas juntas em um sábado pela manhã. Não foi possível realizar. É impossível sincronizar 100% das ações e reações de 50 pessoas juntas.

Os testes de automação funcional são simplesmente a automação dos casos de testes funcionais gerados pelos profissionais da área de qualidade, assim como o teste funcional, o objetivo da automação de teste funcional é validar a interface para garantir que os requisitos definidos foram devidamente implementados e estão de acordo com a solicitação do cliente.

> A automação de testes funcionais visa automatizar um ou mais casos de testes funcionais. Testes funcionais visam testar as principais funcionais [sic] de um sistema no qual se destacam os testes de interface. (MOLINARI, 2012, p. 103)

A presente monografia trata especificamente da geração da massa de dados que será usada pelos testes de automação funcional.

2.5.3 Massa de Dados

Conforme o exposto anteriormente, a estrutura de um cenário de testes usando BDD é formado por “Dado que”, “Quando”, e “Então”, que representam a preparação do ambiente, a execução do passo a passo e a validação do retorno, respectivamente. Seguindo essa linha, os dados que serão usados nos testes são informados, configurados e preparados no “Dado que”.

Esses dados que são usados na execução dos testes são chamados de massa de dados, pré-requisitos ou *setup*. A cada novo teste é preciso chegar ao ponto exato necessário para o início da execução do teste e essa é a parte mais morosa da execução dos testes. Crispin e Gregory (2009) dizem que é comum testes serem reexecutados inúmeras vezes e que se a cada execução de um desses testes os pré-requisitos são cadastrados e configurados novamente, então esse processo deveria ser automatizado.

Em um cenário de execução de testes funcionais via tela, por exemplo, se o *setup* de dados estiver todo dentro dos testes, o processo de cadastro, configuração e preparação desses dados, mesmo sendo automatizado, irá retardar demais a execução dos testes, pois terá a interferência de um elemento intermediário.

Esse *setup* de dados não pode ser removido do processo, pois não importa a estratégia de testes adotada, os testes precisam de dados para processar. Crispin e Gregory (2009) apresentam uma série de pontos que podem facilitar o *setup* de dados. São elas:

- **Ferramentas de geração de dados:** existe no mercado hoje uma série de ferramentas para geração de dados para testes. Essas ferramentas podem gerar dados diretamente no banco, e podem gerar uma série de dados como nomes e endereços. As autoras citam ainda uma série de ferramentas disponíveis no mercado hoje que podem ser usadas, são elas Data Generator, databene benerator, testgen, Datatect, e Turbo Data.
- **Evitar acesso ao banco de dados:** acesso ao banco de dados significa operações de entrada e saída sendo executadas, e essas operações atrasam os testes. Se o objetivo do teste é validar a lógica, algoritmos ou cálculos no código, então o objetivo é o comportamento do código dado certas entradas. Essas entradas podem ser simuladas através de um banco de dados em memória, simulando o acesso ao banco de dados e atentar-se ao no propósito dos testes. O uso dessa abordagem pode precisar de uma reavaliação e reestruturação da arquitetura.

- **Quando o acesso ao banco de dados é necessário/desejável - *setup*/*teardown* de dados para cada teste:** nesse caso é preferível que cada teste adicione os dados que serão usados a um esquema de testes, utilize-os, verifique os resultados a partir do que foi criado na base de dados e então apague todos os dados de teste, ou seja, execute o *teardown* dos dados. Dessa forma os testes poderão ser executados novamente sem impactar nos testes subsequentes, criando assim uma independência entre os testes.
- **Quando o acesso ao banco de dados é necessário/desejável - modelos canônicos:** é possível montar a massa de dados a partir de esquemas de testes que são alimentados por um banco de dados canônico ou seed, que é basicamente um banco de dados que representa uma amostra dos dados reais de produção, sendo muito pequeno então pode ser reconstruído a cada uso. A grande desvantagem de usar dados canônicos ou seeds é manter os dados atualizados de acordo com os dados que estão em produção.
- **Quando o acesso ao banco de dados é necessário/desejável - dados similares a produção:** o ideal quando se fala em teste de *software* é testar sobre um banco de dados que é o mais parecido com o banco de dados de produção possível. Porém, executar testes sobre uma cópia de produção pode ser moroso demais, de forma que talvez os resultados dos testes não sejam mais úteis quando prontos. Além disso, manter uma cópia de produção, assim como manter essa cópia atualizada pode ser caro e desgastante dependendo do tamanho do banco de dados. Sem contar que uma cópia dos dados de produção precisa ser tratada de forma que omita dados confidenciais.

Cada um dos pontos citados anteriormente pode ser usado durante a execução dos testes, no entanto é preciso entender quais os objetivos dos testes antes de escolher qual estratégia vai se adequar mais às necessidades. Quando há muitos objetivos em uma execução de testes, o ideal é torná-los independentes de forma que cada elemento possa ser avaliado individualmente, e também vai tornar os retornos dos testes mais rápidos e esse deve ser a meta de todos os testes, uma vez que a reação a problemas será igualmente rápida.

3 MÉTODO

Esse capítulo irá apresentar a caracterização do tipo da pesquisa assim como suas etapas e atividades.

3.1 CARACTERIZAÇÃO DO TIPO DE PESQUISA

Segundo Minayo (1993 apud SILVA e MENEZES, 2005) pesquisa é "atividade básica das ciências na sua indagação e descoberta da realidade. É uma atitude e uma prática teórica de constante busca que define um processo intrinsecamente inacabado e permanente. É uma atividade de aproximação sucessiva da realidade que nunca se esgota, fazendo uma combinação particular entre teoria e dados".

Silva e Menezes (2005) afirmam que do ponto de vista da natureza uma pesquisa pode ser classificada como Básica ou Aplicada. A presente monografia tem por objetivo gerar conhecimento e material para aplicação prática tendo em vista a solução de problemas específicos, portanto trata-se de uma pesquisa do tipo Aplicada.

Do ponto de vista da abordagem uma pesquisa pode ser Quantitativa e/ou Qualitativa. Este projeto apresenta uma abordagem Qualitativa, seguindo, portanto, a seguinte definição:

> Pesquisa Qualitativa considera que há uma relação dinâmica entre o mundo real e o sujeito, isto é, um vínculo indissociável entre o mundo objetivo e a subjetividade do sujeito que não pode ser traduzido em números. A interpretação dos fenômenos e a atribuição de significados são básicas no processo de pesquisa qualitativa. Não requer o uso de métodos e técnicas estatísticas. O ambiente natural é a fonte direta para coleta de dados e o pesquisador é o instrumento-chave. É descritiva. Os pesquisadores tendem a analisar seus dados indutivamente. O processo e seu significado são os focos principais de abordagem. (MENEZES; SILVA, 2005, p. 20).

Silva e Menezes (2005) dizem que do ponto de vista dos objetivos uma pesquisa pode ser Exploratória, Descritiva ou Explicativa. A presente pesquisa pode ser classificada como exploratória, pois tem por objetivo estudar um problema tornando-o explícito, envolvendo levantamento bibliográfico e análise de exemplos.

Dentre as inúmeras definições de procedimentos técnicos a presente pesquisa pode ser classificada como Pesquisa Bibliográfica e Estudo de Caso, pois de acordo com Silva e Menezes (2005) que será elaborada analisando materiais que já foram publicados (livros, artigos e materiais da internet) e envolvendo o estudo de um objeto é assim descrita como desse tipo.

3.2 ETAPAS E ATIVIDADES

A pesquisa irá realizar as seguintes atividades nas próximas etapas para atingir os objetivos deste trabalho:

1. Descrição da empresa estudo de caso, descrição dos processos de desenvolvimento e qualidade de *software*.
2. Seleção de um projeto da empresa estudo de caso para aplicação da automação de testes.
3. Descrição das ferramentas utilizadas para automação de testes.
4. Descrição do processo automatizado.
5. Apresentação dos resultados da automação de testes.

Nesse contexto pode-se destacar que a anuência da empresa para desenvolvimento do presente trabalho encontra-se no Apêndice C.

3.3 DELIMITAÇÕES

Para atingir os objetivos propostos será feita a descrição de um processo de testes, assim como os *templates* e os casos de testes para auxiliar e documentar a elaboração do trabalho.

Como delimitação da presente pesquisa pode se mencionar a não aplicação das recomendações geradas após análise do processo de testes na empresa estudo de caso.

Além disso, não é proposta deste trabalho desenvolver um *software*, apenas fazer a descrição da automação dos testes.

4 ESTUDO DE CASO

O capítulo 4 irá apresentar o estudo de caso, desde a problemática, até a aplicação das técnicas apresentadas nos capítulos anteriores para resolver o problema.

O estudo de caso foi aplicado em uma empresa de desenvolvimento de *software* voltado para o agronegócio, a Agriness. O estudo de caso foi aplicado em um de seus produtos, que consiste em uma solução para gestão de granjas de suínos que visa o máximo potencial produtivo de cada granja.

4.1 DESCRIÇÃO DA EMPRESA

Segundo Agriness (2019), em 2001 foi fundada a empresa Agriness, que surgiu a partir da necessidade de soluções que ajudassem o suinocultor a gerenciar as atividades das granjas de forma organizada e confiável. A Agriness é uma referência em soluções e modelos de gestão da informação para o agronegócio, onde tem forte atuação na suinocultura e é líder de mercado no setor.

De 2001 pra cá as soluções oferecidas pela Agriness foram evoluindo, de forma que hoje o foco passou a ser a produção animal, e não mais somente a suinocultura.

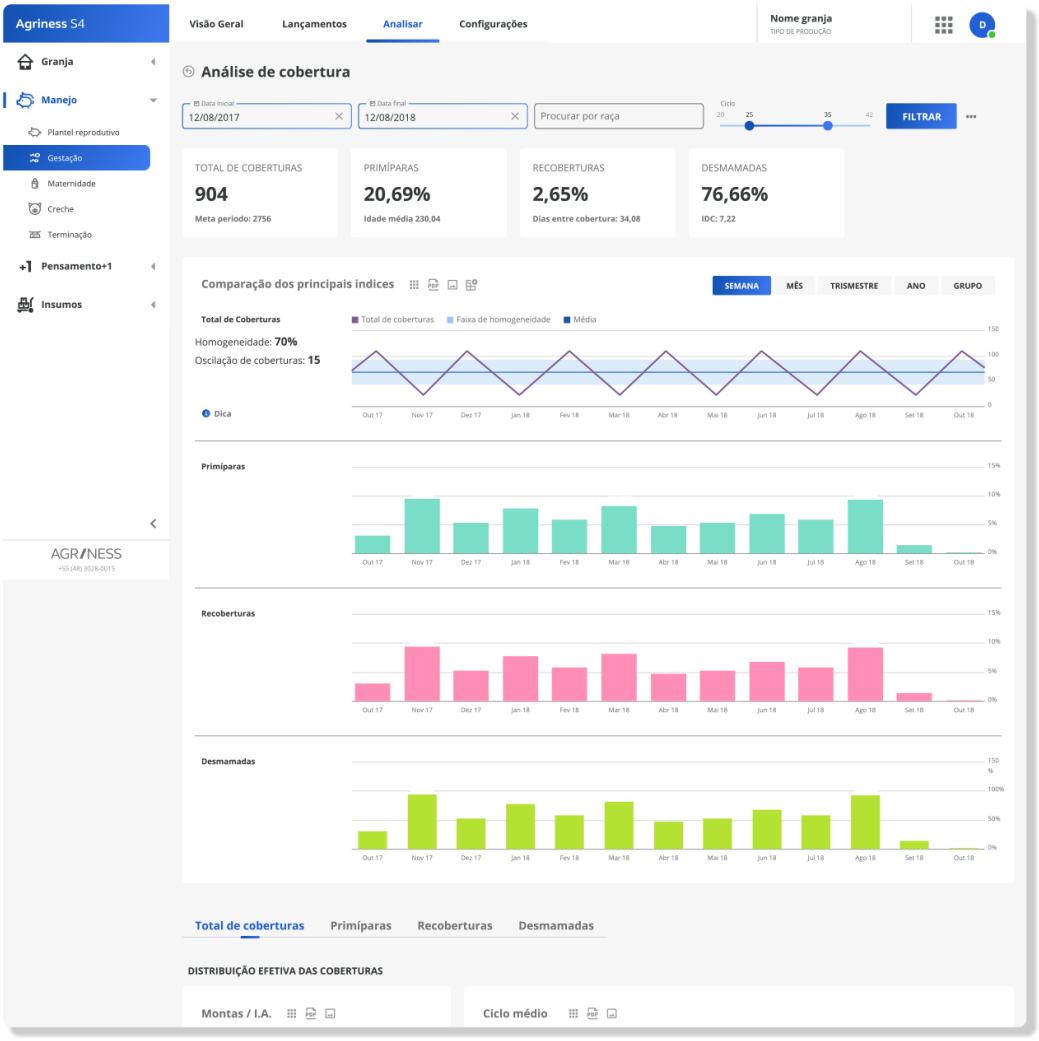
A Agriness oferece hoje não somente uma solução em tecnologia voltada para a produção animal, mas também cursos e consultorias que auxiliam o produtor a utilizar análises e conhecimentos para obter resultados de excelência através da aplicação de uma metodologia de aceleração de produtividade pioneira no mundo.

O processo de desenvolvimento de uma solução tecnológica se inicia dentro da Fábrica de Inovação da Agriness, sendo que, segundo a Agriness (2019b), todo o processo, desde a proposta até a entrega para o cliente, é dividido em três fases, que são:

1. **Innovation:** Fase presente dentro da Fábrica de Inovação, e é responsável pela concepção de novas soluções pensadas a partir de uma proposta.
2. **Zero Code:** Fase também presente na Fábrica de Inovação, e cabe a essa fase a produção das soluções desenhadas na fase Innovation. É nessa fase que os testes automatizados são desenvolvidos.
3. **Unique Customer:** Esta fase não está presente na Fábrica de Inovação. Fase que se dedica a operação da solução e todo o processo de prosperidade do cliente.

Dentro desse processo foi desenvolvido o Agriness S4, que é a solução para gestão de granjas de suínos, totalmente na cloud, que associa recursos de análises de dados e tecnologias que garantem a gestão da produção em tempo real. Na Figura 7 é possível visualizar a tela de Análise de cobertura do Agriness S4.

Figura 7 - Tela de Análise de cobertura do Agriness S4



Fonte: Agriness (2019c)

O processo de gerir a produção de suínos é sistemático e baseado em vários ciclos. Toda a estrutura descrita anteriormente existe para garantir que, em um cenário que é tão complexo e que a cada dia investe mais em especialização, as mudanças sejam identificadas e mapeadas, gerando soluções inovadoras, construindo e as mantendo dentro do contexto da suinocultura atual.

> Se por um lado a crescente especialização da suinocultura, obtida pela segmentação do processo produtivo nas granjas, tem contribuído para uma maior eficiência técnica, por outro, tem aumentado o desafio para a coordenação técnica e econômica dessas atividades segmentadas. Ou seja, o desafio encontra-se no modo de governança mais adequado, de modo que garanta maior sintonia entre os componentes e maior eficiência nas transações. (ASSOCIAÇÃO BRASILEIRA DE CRIADORES DE SUÍNOS, 2014, p. 39).

Assim sendo, os cenários de testes envolvem uma série de eventos que são pré-requisitos para a validação das funcionalidades em si. Esses eventos costumam ser dependentes entre si e, portanto, devem ser cadastrados em uma ordem específica. Conforme será descrito na sequência.

4.2 DESCRIÇÃO DA PROBLEMÁTICA

Toda a estrutura de testes está organizada seguindo a estrutura do sistema, sendo que cada suíte de testes se refere a um estágio no processo de produção de suínos. A seguir estão descritos os estágios de interesse no estudo considerado:

- **Plantel reprodutivo:** estágio responsável pela gestão da informação dos animais que estão ou estiveram em algum momento na granja.
- **Gestação:** estágio responsável principalmente pela cobertura/inseminação das fêmeas, essa fase é responsável pelo gerenciamento de todos os eventos ocorridos até o momento antes do parto. Segundo a Associação brasileira de criadores de suínos (2014) o período de tempo que as fêmeas costumam ficar em gestação é de 114 dias, podendo variar em dois dias para mais ou para menos.
- **Maternidade:** estágio responsável por gerenciar todos os eventos ocorridos a partir do parto até o momento do desmame dos leitões. O período de tempo desde o nascimento até o desmame varia de acordo com o peso adquirido pelo leitão enquanto na maternidade, no entanto a Associação brasileira de criadores de suínos (2014) sinaliza que é de grande importância que os leitões sejam desmamados com pelo menos 5,5 quilos aos 20 dias de idade.
- **Creche:** essa é a fase de engorda até o momento da venda ou do envio dos leitões para a terminação. A Associação brasileira de criadores de suínos (2014) afirma que leitões que saem desta fase com muito menos do que 25 quilos, tendem a não obter o peso mínimo desejado na fase da terminação.

- **Terminação:** essa fase também se trata de um período de engorda, no entanto ele é voltado para a terminação desses animais. A Associação brasileira de criadores de suínos (2014) afirma que há vantagens na produção de suínos de 100 a 130 quilos. Sendo assim, essa é ao obter esse peso que os animais saem dessa fase.

Cada estágio possui uma série de cadastros e configurações que permitem o controle próximo e customizado da produção dos animais em cada fase. Cada evento tem uma data prevista para ocorrer baseada na data do evento anterior, e um evento não cadastrado, cadastrado na data incorreta, ou ainda cadastrado com informações incorretas, pode prejudicar todo o fluxo de produção.

Dentro daquilo que foi descrito anteriormente, no estágio Maternidade há o cadastro de mãe de leite, esse cadastro se dá quando uma fêmea ainda está lactante, e portanto, ainda pode amamentar leitões provenientes de leitegadas[1] de outras fêmeas. Dentro do sistema existe uma série de eventos que precisam ser cadastrados até que uma fêmea se torne uma mãe de leite. Conforme segue:

1. **Cadastrar compra de leitoa:** cadastro de compra de uma fêmea que não passou por uma gestação.
2. **Cadastro de localização:** Cadastro do local onde ficará alojada a fêmea.
3. **Cadastrar cobertura de leitoa:** cadastro de cobertura[2], ou seja, cadastro de cruzamento através de inseminação artificial ou de monta, de uma matriz vazia.
4. **Cadastro de parto:** depois de, em média, 114 dias de período de gestação ocorrerá o parto da matriz.
5. **Cadastro de morte de leitão:** após o parto podem ocorrer eventos de morte de leitão, por exemplo, e essas mortes são todas registradas no sistema.
6. **Cadastro de localização na creche:** Cadastro da localização onde ficarão alojados os leitões após o desmame.
7. **Cadastro de lote na creche:** Cadastro do lote onde ficarão alojados os leitões após o desmame.

---

[1] Leitegada: Segundo Michaelis (2019) uma leitegada são os leitões nascidos vivos de um só parto; leitoada.

[2] Cobertura: De acordo com Michaelis (2019) a cobertura trata-se da cópula entre animais com fins de procriação.

8. **Cadastro de desmame:** após 21 dias, ou até que os leitões atinjam um peso mínimo, os leitões da leitegada podem ser desmamados da fêmea.
9. **Cadastro de movimentação de leitão:** os leitões podem ser movimentados entre as matrizes na maternidade conforme a necessidade.
10. **Cadastro de mãe de leite:** caso a matriz fique vazia, ou seja, sem nenhum leitão, no mesmo dia do último evento, o sistema permite que a matriz seja cadastrada como mãe de leite, podendo receber leitões de outras matrizes para que estes continuem sendo amamentados por ela.

Sempre que um teste de mãe de leite precisa ser executado os eventos descritos anteriormente devem ser cadastrados. Além disso há outros eventos adversos que podem ocorrer no meio desse processo, e eles são então exercitados pelos testes em questão, que foram escritos em linguagem ubíqua, usando a ferramenta *Robot Framework*. Por conta disso esses testes são compreensíveis a qualquer pessoa envolvida no processo que possa se interessar pelo que está sendo testado.

Dentro desse cenário existem hoje 2128 testes cadastrados e ativos que são executados todas as noites validando a possível inserção de novos erros no sistema. Tais erros são identificados através da análise dos logs gerados, e então reportados, através da ferramenta Jira, para a equipe de desenvolvimento que faz a avaliação técnica do problema.

Voltando um pouco para o cenário anterior, todos os testes funcionais do sistema eram executados, do começo ao fim, via interface de usuário, e consequentemente eram muito demorados. O tempo total de execução desses testes no estágio Maternidade, por exemplo, era em torno de cinco horas e meia, sendo cerca de três horas apenas de cadastros de pré-requisitos. No Apêndice A encontra-se o detalhamento do tempo de execução de todos os cadastros de pré-requisitos, dando destaque para o valor total de tempo que é de duas horas, cinquenta e seis minutos e vinte e seis segundos.

As operações executadas por esses testes que tivessem um comportamento levemente fora do normal provocavam quebras de testes, ou seja, uma operação mesmo funcionando corretamente apresentava uma falha de teste. Essas falhas de testes poderiam ser classificadas como falsos negativos. Um dos comportamentos que mais causavam esses problemas eram os atrasos no carregamento de informações. Portanto, no exemplo de cadastro de mãe de leite descrito anteriormente, caso os testes quebrassem no cadastro de matriz vazia, todo o fluxo quebraria e todo o teste teria que ser desconsiderado.

4.3 FERRAMENTAS E TECNOLOGIAS UTILIZADAS

Para poder solucionar o problema as seguintes tecnologias estavam disponíveis e foram utilizadas:

- ***Robot Framework*:** Segundo a organização que mantém a ferramenta, a *Robot Framework Foundation* (2019), trata-se de um *framework open source* para automação de testes de aceitação, ATDD (*Acceptance test driven development*), e RPA (*Robotic Process Automation*) e usa a abordagem de testes orientados a *keywords*.
- **Collections:** Segundo a organização *Robot Framework Foundation* (2019), que mantém a ferramenta *Robot Framework* e suas bibliotecas, a biblioteca Collections é uma biblioteca padrão que fornece um conjunto de *keywords* para manuseio de listas e dicionários em Python.
- **Git:** Conforme Git (2019) a ferramenta Git é um sistema de controle de versão distribuído, ou seja, os clientes fazem cópias completas dos repositórios. Garantindo que se houver alguma falha no servidor, os repositórios dos clientes podem ser usados para restaurá-lo.
- **Python:** De acordo com a organização *Python Software Foundation* (2019), Python é desenvolvido sob licença open source, e é uma linguagem de programação clara e poderosa que usa uma sintaxe elegante e fácil de entender.
- **Bitbucket:** Atlassian (2019) descreve a ferramenta Bitbucket como um serviço para hospedagem e gerenciamento de código Git que facilita o planejamento e colaboração para o desenvolvimento de projetos, além de permitir testar e implementar códigos.
- **Jenkins:** Jenkins (2019) afirma que essa ferramenta é um servidor de automação independente e open source que permite a automação de todo tipo de tarefa relacionadas à construção, execução de testes, entrega e implantação de *software*.
- **Atom:** Github (2019) diz que Atom é um editor open source disponível para as plataformas Linux, Microsoft Windows, e macOS desenvolvido pelo Github.

- **RequestsLibrary:** Na documentação disponibilizada pela organização *Robot Framework Foundation* (2019) está descrito que RequestLibrary é uma biblioteca de *keywords* cliente HTTP que usa um módulo de requisições.
- **Jira:** Atlassian (2019) descreve que Jira é um *software* criado para que cada membro possa planejar, acompanhar e lançar ótimos *softwares* através da criação de estórias de usuário e itens, planejando sprints e distribuindo tarefas entre os membros da equipe.

4.4 DESCRIÇÃO DA SOLUÇÃO

A solução encontrada para reduzir o tempo de execução e os falsos negativos foi executar a parte de cadastros e configurações de pré-requisitos através de requisições via API.

No entanto, uma das características que precisava ser mantida eram os testes escritos em linguagem ubíqua, por conta da complexidade do fluxo, o rastreio e entendimento das operações deveriam ficar claras.

Como era de se esperar, a forma como aplicamos essa solução pela primeira vez não foi a solução final, e todo o processo de evolução da ideia será descrito a seguir.

Em um primeiro momento a solução envolveu a criação de uma palavra chave, ou *keyword*, para cada requisição que seria feita para a API. As informações seriam passadas através da *keyword* e montadas na estrutura dos dados, no momento do envio da requisição.

Para cada uma dessas requisições foi criada também uma *keyword* que seria responsável pela validação do retorno da API. Essa validação seria responsável por validar informações como o código de status retornado, a mensagem, e por fazer a comparação dos dados enviados com os dados persistidos.

Essa estrutura substituiu todas as execuções de cadastro e configurações de pré-requisitos existentes, ou seja, todas as vezes que um teste precisava que um pré-requisito fosse cadastrado por exemplo, a estrutura para cadastro desse pré-requisito (*keyword* da requisição e *keyword* de conferência) era criada, passando por parâmetro as informações necessárias.

Essa solução apresentou alguns problemas, o primeiro deles foi que a estrutura era adicionada aos pré-requisitos todas as vezes que um novo cadastro fosse necessário, dificultando a manutenção de todo o código envolvido no processo. Para resolver esse problema foi criada uma biblioteca de *templates*.

A ferramenta *Robot Framework* permite que ao executar um teste, ao invés de chamar uma sequência de *keywords*, é possível definir um modelo composto por *keywords* criadas, este modelo também pode ser chamado de *template*. Sendo assim, para cada pré-requisito foi criada uma estrutura com as *keywords* necessárias e disponibilizadas em uma biblioteca de *templates*.

Então como o cadastro de um lote, por exemplo, é um pré-requisito para a muitos testes, um *template* para o cadastro de lotes foi criado. Conforme demonstrado na Figura 8.

Figura 8 - *Template* para cadastro de lote em estágio

```
1. Cadastrar lote
2.      [Arguments]   ${NOME_LOTE}   ${ESTAGIO}
3.      Dado que sou um funcionário da granja
4.      Quando cadastro o lote "${NOME_LOTE}" em "${ESTAGIO}"
5.      Então confiro que o cadastro foi realizado com sucesso
```

Fonte: da autora, 2019

Analisando o *template*, na primeira linha o "Cadastrar lote" é o nome do *template*, é a partir dele que o *template* será referenciado quando necessário.

Na segunda linha temos a lista de argumentos "[Arguments] ${NOME_LOTE} ${ESTAGIO}", é a partir deles que as informações que serão cadastradas são passadas para o *template*.

Na terceira, quarta e quinta linha estão:

1. A *keyword* que criará a sessão, ou seja, estabelecerá a conexão com a API (Dado que sou um funcionário da granja);
2. A *keyword* que executará a requisição (Quando cadastro o lote "${NOME_LOTE}" em "${ESTÁGIO}");
3. A *keyword* que fará conferência da resposta (Então confiro que o cadastro foi realizado com sucesso).

Então no momento em que a execução de um *template* é necessária, apenas a referência ao *template* e os dados que serão enviados para a requisição precisam ser configurados. Na Figura 10 é possível ver como fica a chamada para o *template*.

Figura 9 - Chamada para o *template* "Cadastrar lote"

```
1. Cadastro de lote para teste de ficha da matriz
2.      [Template]   Cadastrar lote
3.      #NOME_LOTE   #ESTAGIO
4.      Lote ABC     Creche
5.      Lote DEF     Terminação
```

Fonte: da autora, 2019

A primeira linha “Cadastro de lote para teste de ficha da matriz” é uma descrição sobre o que será executado.

Para especificar qual *template* será usado, a configuração [Template] seguido do *template* que será executado deve ser configurado. Sendo assim, a segunda linha “[Template] Cadastrar lote” define que o *template* que será usado é o “Cadastrar lote”.

Na terceira linha uma descrição dos argumentos que deverão ser passados para o *template*, essa linha tem função informativa apenas.

Na quarta e quinta linha os dados que serão passados como argumento são informados, cada linha se refere a uma nova chamada ao *template*, e cada coluna se refere a um argumento, esses argumentos devem ficar na mesma ordem que estão no *template*.

Toda essa solução era funcional para cadastros simples, com poucos argumentos, no entanto quando se tratavam de requisições com muitos argumentos, essa solução não era a ideal, pois a *keyword* responsável pela requisição ficava muito extensa e difícil de manter.

Para resolver esse problema a dinâmica de como os dados eram montados foi mudada, cada informação precisava ser adicionada individualmente a uma estrutura que seria então convertida para requisição. A estrutura escolhida foi a de um dicionário de dados.

Como as informações enviadas são semelhantes, cada chave passou a ter sua própria *keyword*, onde cada uma teria os seus próprios tratamentos de dados antes de ser adicionado ao dicionário.

Dessa forma, a disposição dos *templates* passou a ser da seguinte forma, como apresentado na figura:

Figura 10 - *Template* para "Cadastrar macho" adicionando individualmente os itens da requisição

```
1. Cadastrar macho
2.      [Arguments]        ${DIAS_NASCIMENTO}        ${DIAS_EVENTO}
   ${IDENTIFICACAO}     ${RACA}     ${TIPO_ANIMAL}     ${FUNCIONARIO}
   ${PRECO}      ${PESO}      ${IDENTIFICACAO_SECUNDARIA}[Template]
   Cadastrar lote
3.       Dado que quero executar uma operação via API
4.       Quando seleciono a raça "${RACA}"
5.       E seleciono a data do evento há "${DIAS_EVENTO}" dias
6.       E seleciono a data de nascimento "${DIAS_NASCIMENTO}"
7.       E seleciono o identificador primário "${IDENTIFICACAO}"
8.       E seleciono o tipo do animal "${TIPO_ANIMAL}"
9.       E seleciono a entidade "${FUNCIONARIO}"
10.            E seleciono o valor da venda "${PRECO}"
11.            E seleciono o peso do animal "${PESO}"
12.      E      seleciono      o      identificador      secundário
   "${IDENTIFICACAO_SECUNDARIA}"
13.            E cadastro um macho via API
14.          Então conferir o response de sucesso ao cadastrar uma
   compra de macho
```

Fonte: da autora, 2019

A primeira e a segunda linha são o nome do *template* e a lista de argumentos, respectivamente.

A terceira linha cria o dicionário que conterá os dados que serão enviados para a requisição.

Da quarta até a décima segunda linha o *template* trata e adiciona cada parâmetro passado a requisição. Algumas das operações de tratamento que são feitas nessas *keywords* são:

- Os dias que são passados para o *template* como um número inteiro e são transformados em datas.
- A raça do animal é passada como texto e seu identificador é adicionado ao dicionário após recuperá-lo através de uma pesquisa na API, o mesmo acontece com o Funcionário.

Na décima terceira linha “cadastro um macho via API” realiza as seguintes operações:

- Cria uma sessão HTTP[3] com a API, ou seja, estabelece uma conexão com a API.
- Monta a URI[4] para envio da requisição para o endpoint correto.
- Envia a requisição.
- Salva a resposta.
- E apaga a sessão, ou seja, desfaz a sessão.
- Verifica se o código de status é o código de status correto.

Na décima quarta linha os valores retornados são validados e comparados com os que foram enviados.

Todas as alterações feitas foram versionadas no Bitbucket e foram integradas a execução diária dos testes. Após aplicadas essas alterações foram percebidas diferenças na execução dos testes. Para demonstrar essas diferenças serão mostrados a seguir os resultados dessas alterações no grupo de testes do estágio Maternidade.

O primeiro resultado positivo identificado foi que os falsos negativos que ocorriam durante o cadastro dos pré-requisitos não ocorreram mais, pois as requisições para cadastro dos pré-requisitos passaram a ser enviadas diretamente para a API, sem nenhum tipo de mediador.

Além disso, as alterações reduziram significativamente o tempo de cadastro dos pré-requisitos. O cadastro de quatrocentos e setenta e cinco (475) eventos demorava cerca de duas horas, cinquenta e seis minutos e vinte e seis segundos (02:56:26), conforme apresentado no Apêndice A. Após as alterações, todos esses cadastros passaram a demorar cerca de doze minutos e quinze segundos (00:12:15), segundo os dados tabela dos no Apêndice B.

4.5 TECNOLOGIAS APLICADAS NA SOLUÇÃO

Para implementar a solução foi usado o *framework Robot*, que teve a função de automatizar toda a solução. O *framework* é desenvolvido em Python, portanto sua instalação foi um pré-requisito para o uso do *framework*.

---

[3] Sessão HTTP: Segundo MDN (2019) em protocolos cliente-servidor as fases de uma sessão HTTP são compostas por uma conexão TCP estabelecida, um envio de requisição para o servidor pelo cliente, o envio da resposta e do código de status após o processamento da requisição pelo servidor. Conexões HTTP/1.1 podem ser usadas inúmeras vezes, pois a conexão não é fechada automaticamente após a execução de todas as fases.

[4] URI: De acordo com MDN (2019b) uma URI (Uniform Resource Identifier) é um texto que ser refere a um recurso. Os tipos mais comuns de URI são as URLs (Uniform Resource Locator) que identifica o recurso dando sua localização na Internet.

Além disso, combinadas ao *framework* foram usadas as bibliotecas RequestLibrary e Collections. Ao usar RequestsLibrary foi possível criar as conexões com a API, assim como enviar as requisições HTTP/1.1. Já a biblioteca Collections permitiu o manuseio e a montagem do dicionário com os dados a serem enviados para a requisição.

O editor Atom foi usado para a implementação do projeto, que foi versionado usando o sistema de controle de versão Git. O projeto, as alterações e seu histórico foram hospedados no serviço Bitbucket.

A ferramenta Jenkins foi usada para a criação de uma tarefa que executasse a solução diariamente, todos os dias durante a madrugada. Os erros encontrados durante essa execução foram reportados através da ferramenta Jira através da criação de chamados que entram na fila de atividades da equipe de desenvolvimento.

5 CONCLUSÕES E TRABALHOS FUTUROS

O presente capítulo trata das conclusões finais deste trabalho em relação a solução apresentada para a melhoria da criação das massas de dados com base nos problemas e cenários descritos, assim como sugestões de trabalhos futuros.

5.1 CONCLUSÕES

O objetivo deste trabalho foi apresentar um estudo de caso com a utilização de técnicas do BDD para execução de testes via API. Para atingir a meta proposta foi necessário entender a estrutura, o formato e os problemas presentes nos testes existentes na empresa Agriness, organização do estudo de caso.

O resultado final do trabalho envolveu a criação de uma estrutura para montagem dinâmica de requisições a serem enviadas via API, essa estrutura foi organizada na forma de uma biblioteca de *templates* a ser consumido na configuração e montagem da massa de dados de cada teste. A ideia foi tornar todo o código envolvido no processo o mais reutilizável possível e que a configuração dessas massas de dados fosse o mais simples possível.

A primeira fase no desenvolvimento da solução foi o estudo dos fundamentos do BDD, pois todos os testes existentes no projeto estudo de caso são desenvolvidos usando o *template* "Dado que, Quando, Então", que é uma das principais características do BDD. E um dos desafios no desenvolvimento dessa solução era preservar essa estrutura.

Após entender esses fundamentos foi preciso identificar e entender os problemas na estratégia que estava em prática no projeto. Para isso foi estudado o funcionamento do *framework Robot* e sua dinâmica com o *software* em questão. Assim sendo, foi possível verificar a viabilidade de aplicação de uma solução criando e enviando requisições para as APIs que se comunicavam com a plataforma em questão.

Todo a solução foi desenvolvida com o apoio de ferramentas que foram apresentadas neste trabalho, sendo elas:

- ***Robot framework*:** toda a automação da solução foi criada usando este *framework*.
- **Biblioteca Collections:** as requisições são montadas usando as estruturas de dados dessa biblioteca.
- **Git:** todo o controle de versão do projeto é feito usando Git.

- **Python:** O *framework Robot* é desenvolvido nessa linguagem e, portanto, foi necessário toda a configuração e preparação da versão correta da linguagem para o melhor funcionamento da solução.
- **Bitbucket:** Todo o projeto e suas versões estão hospedados nesse serviço.
- **Jenkins:** A execução periódica dos testes é configurada usando Jenkins.
- **Atom:** O editor usado para a criação da solução foi o Atom.
- **Biblioteca RequestsLibrary:** Todas as requisições usadas na solução, que foram requisições do tipo Post, Get e Delete são enviadas usando esta biblioteca.
- **Jira:** Todos os erros encontrados foram reportados usando esta ferramenta.

Toda a solução passou por evoluções, cada evolução passou por um processo de criar o código, testar, refatorar e evoluir.

A primeira versão da solução era uma estrutura totalmente estática, pois os valores eram adicionados diretamente no código fonte dificultando a manutenção e o reuso.

A segunda versão da solução usava *templates*, onde os valores eram passados por parâmetro e as variáveis eram adicionadas diretamente no código fonte. Ainda uma solução estática, mas possibilitando certo nível de reuso.

Então todo o *template* foi reestruturado para adicionar cada item individualmente na requisição e assim poder ser usado por outros *templates*. Nesse momento passou a ser possível o reuso de cada elemento do *template* por outros.

Por fim toda a requisição passou a ser montada dentro de um dicionário de dados de forma dinâmica, adicionando cada elemento da requisição de forma individual e independente do contexto.

Todo esse processo adicionou manutenibilidade ao código, e permitiu que o tempo gasto criando código de testes passasse a ser insignificante comparado com as outras etapas de criação e elaboração dos pré-requisitos dos testes. E esse foi o maior ganho que a solução trouxe para projeto.

O segundo maior ganho foi que todos os problemas que as ferramentas enfrentavam ao interagir com a interface foram eliminados, ou seja, removemos todos os falsos negativos que foram identificados nessa etapa dos testes.

O terceiro maior ganho foi a redução considerável do tempo de execução dessa etapa, um conjunto de testes cuja execução demorava cerca de três horas, hoje a execução total dos mesmos eventos ocorre em cerca de doze minutos.

O quarto maior ganho foi que os testes cujos pré-requisitos passaram a ser montados usando a biblioteca ganharam independência.

Essa independência, no entanto, trouxe um problema para a cobertura dos testes, ou seja, as execuções que antes passavam por várias telas do sistema e que validavam várias operações foram removidas do fluxo. Sendo assim foi necessária a criação de vários novos testes para validar essas operações, o que aumentou o esforço planejado para garantir a cobertura dos testes na tela.

Outro problema encontrado foi a necessidade de reestruturar a organização de toda a biblioteca dentro do projeto de testes, esse, no entanto, não foi o foco desta solução.

Elenco a seguir os conhecimentos obtidos durante o desenvolvimento desta solução:

- Aprofundamento do conhecimento em automação de testes via API: foi obtida proficiência em testes via API, permitindo dessa forma aplicar soluções similares, caso necessário, em outras ferramentas de automação de testes.
- Aprofundamento do conhecimento do *framework Robot* e em suas bibliotecas: houve um estudo e entendimento das bibliotecas para *Robot Framework* disponíveis hoje compatíveis com a versão do Python usado no projeto, dessa forma foi adquirido domínio do *framework* e suas bibliotecas.
- Entendimento das limitações do projeto: durante a elaboração desta solução, outras possibilidades foram exploradas, no entanto a estrutura do *software* tornou-se uma limitação para a aplicação das mesmas, dessa forma foi necessário, antes de descartar essas possibilidades, entendê-las e identificá-las. Essas limitações não foram tratadas no presente projeto.
- Aprofundamento no conhecimento do processo de gestão da suinocultura: o processo de gestão da suinocultura é um processo complexo e cheio de regras, durante a elaboração e desenvolvimento da solução ocorreu um aprofundamento do conhecimento desse universo que é a suinocultura.

5.2 TRABALHOS FUTUROS

Como trabalhos futuros pode se elencar alguns dos descritos a seguir:

- Utilização de modelos canônicos: adequar um projeto de testes, de forma que a massa de dados de testes seja disponibilizada em um banco de dados canônico, onde os dados do teste são montados a cada uso conforme descrito em 2.5.3.
- Execução de scripts SQL no banco de dados: adequar um projeto de testes de forma que a massa de dados seja disponibilizada através da execução de scripts SQL no início de cada teste.
- Base de dados contida em um container Docker[5]: adequar um projeto de testes de forma que a massa de dados seja disponibilizada em um container Docker em uma base de dados similar à de produção, conforme descrito em 2.5.3.
- Padrões de projetos: adequar um projeto de testes de forma que toda essa estrutura de criação de massa de dados em tempo de execução seja estruturada usando padrões de projetos voltados para testes de *software*.
- Fazer um comparativo de horas/trabalho dos profissionais alocados no processo antes da implementação da solução e depois dela, através de um quadro comparativo. Seguindo a hipótese de redução da carga horária de trabalho para a execução do processo mencionado.

[5] Container Docker: Segundo Docker (2019) um container é um pacote de *software* em forma de unidades padronizadas para desenvolvimento, envio e implantação que contém código e todas as sua dependências para que a aplicação possa executar rapidamente e de forma confiável de um ambiente para outro.

REFERÊNCIAS

AGRINESS. **Agriness: Quem somos.** Disponível em: https://www.agriness.com/pt/quem-somos/. Acesso em: 15 out. 2019.

AGRINESS. **Innovation Factory.** Documentação interna da Empresa. Documento de acesso restrito. jun. 2019. Acesso em: 15 out. 2019b.

AGRINESS. **Agriness S4:** Inovação e tecnologia para gestão da produção em tempo real. Disponível em: https://www.agriness.com/pt/s4/. Acesso em: 15 out. 2019c.

ASSOCIAÇÃO BRASILEIRA DE CRIADORES DE SUÍNOS. **Produção de suínos: teoria e prática.** 1. ed. Brasília, DF. Associação Brasileira de Criadores de Suínos; Coordenação Técnica da Integrall Soluções em Produção Animal. 2014.

ATLASSIAN. **Produtos para equipes, de startups a organizações consolidadas.** Disponível em: https://www.atlassian.com/br/software. Acesso em: 7 out. 2019.

BECK, Kent.et al. **Manifesto for Agile Software Development**. Disponível em: http://agilemanifesto.org/. Publicado em: 2001. Acesso em: 03 mar. 2017.

BERNARDO, Paulo Cheque; KON, Fabio. A importância dos Testes Automatizado. **Engenharia de Software Magazine**, v. 1, n. 3, p. 54-57, 2008. Disponível em: http://www.devmedia.com.br/artigo-engenharia-de-software-3-a-importancia-dos-testes-automatizados/9532. Acesso em: 14 mai. 2012.

CAETANO, Cristiano. Melhores Práticas na Automação de Testes. **Engenharia de Software Magazine**, v.1, ed. 5. Disponível em: http://www.devmedia.com.br/artigo-engenharia-de-software-5-melhores-praticas-na-automacao-de-testes/10249. Acesso em: 15 mai. 2012b.

CRISPIN, Lisa; GREGORY, Janet. **Agile testing: a practical guide for testers and agile teams.** 1. ed. Boston, MA. Addison Wesley. 2009.

DOCKER. **What is a Container?.** Disponível em: https://www.docker.com/resources/what-container. Acesso em: 9 nov. 2019.

JENKINS. **Jenkins User Documentation.** Disponível em: https://jenkins.io/doc/. Acesso em: 8 out. 2019.

GIT. **Primeiros passos - Sobre Controle de Versão.** Disponível em: https://git-scm.com/book/pt-br/v1/Primeiros-passos-Sobre-Controle-de-Vers%C3%A3o. Acesso em: 7 out. 2019.

GITHUB. **Why Atom.** Disponível em: https://flight-manual.atom.io/getting-started/sections/why-atom/. Acesso em: 8 out. 2019.

GREGORY, Janet; CRISPIN, Lisa. **More agile testing: learning journeys for the whole team.** 1. ed. Addison Wesley. 2015.

IBM. **Test case and test suite overview.** Disponível em: https://www.ibm.com/support/knowledgecenter/SSYMRC_6.0.1/com.ibm.rational.test.qm.doc/topics/c_testcase_overview.html. Acesso em: 9 nov. 2019.

MASUDA, Cristiana Yukie. **Processo de automação de testes de software com ferramentas *open source*: um estudo de caso com integração contínua.** Trabalho de conclusão de curso em Sistemas de Informação, Universidade do Sul de Santa Catarina, Palhoça, SC, 2009.

MCCONNELL, Steve. **Code Complete:** Um guia prático para a construção de software. 2. ed. Porto Alegre. Bookman. 2005.

MDN. **A typical HTTP session.** 23 mar. 2019. Disponível em: https://developer.mozilla.org/en-US/docs/Web/HTTP/Session. Acesso em: 9 nov. 2019.

MDN. **MDN Web Docs Glossary:** Definitions os Web-related terms. 5 set. 2019. Disponível em: https://developer.mozilla.org/en-US/docs/Glossary. Acesso em: 9 nov. 2019b.

MEDEIROS, Manoel Pimentel. **Extreme Programming – Conceitos e Práticas.** 2006. Disponível em: https://www.devmedia.com.br/extreme-programming-conceitos-e-praticas/1498. Acesso em: 5 nov. 2019.

MESZAROS, Gerard. **xUnit Test Patterns: Refactoring Test Code.** Addison-Wesley Signature Series. 2007.

MICHAELIS. **A mais completa linha de dicionários do Brasil.** Disponível em: http://michaelis.uol.com.br/. Acesso em: 9 nov. 2019.

MOLINARI, Leonardo. **Inovação e automação de testes de software**. São Paulo. Érica, 2010.

MOLINARI, Leonardo. **Testes de software:** Produzindo sistemas melhores e mais confiáveis. São Paulo. Érica, 2003.

NOGUEIRA, Elias. **Automação de Teste – Planejamento, Design e Desenvolvimento dos Testes – parte 3.** 26 abr. 2010. Disponível em: http://www.eliasnogueira.com/automacao-de-teste-planejamento-design-e-desenvolvimento-dos-testes-parte-5/. Acesso em: 15 mai. 2012.

NORTH, Dan. **Introducing BDD.** Mar. 2006. Disponível em: https://dannorth.net/introducing-bdd/. Acesso em: 24 set. 2019.

PAULA FILHO, Wilson de Pádua. **Engenharia de software: fundamentos, métodos e padrões.** LTC, 2012.

PARREIRAS, F. S.; OLIVEIRA, G. S. **Análise comparativa de processos de desenvolvimento de software sob a luz da gestão do conhecimento:** um estudo de caso de empresas mineiras. In: SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE, 3, 2004, Brasília. **Anais...** , 2004. Disponível em: http://www.fernando.parreiras.nom.br/publicacoes/WGC_Parreiras04.pdf. Acesso em: 02 out. 2011.

PEZZÈ, Mauro; YOUNG, Michal. **Teste e análise de software: processos, princípios e técnicas.**Bookman, 2008.

PRESSMAN, Roger S. **Engenharia de Software.** 3.ed. São Paulo. Ed. McGrawHill, 1995.

PRESSMAN, Roger S. **Engenharia de Software.** 6.ed. São Paulo. Ed. McGrawHill, 2006.

PRESSMAN, Roger S. **Engenharia de Software: Uma abordagem profissional.** 7. ed. Porto Alegre. AMGH Editora Ltda, 2011.

PYTHON SOFTWARE FOUNDATION. **Python.** Disponível em: https://www.python.org/about/. Acesso em: 7 out. 2019.

RIOS, Emerson; MOREIRA, Trayahú. **Teste de Software.** 2. ed. Rio de Janeiro. Alta Books, 2006.

RIOS, Emerson et al. **Base de conhecimento em Teste de Software.** 2. ed. Rio de Janeiro. Martins, 2007.

ROBOT FRAMEWORK FOUNDATION. **Robot Framework.** Disponível em: https://robotframework.org/#introduction. Acesso em: 2 out. 2019.

SILVA, Edna Lúcia da; MENEZES, Estera Muszkat. **Metodologia da Pesquisa e Elaboração de Dissertação.** 4. ed. Florianópolis: UFSC, 2005.

SMARTBEAR. **API Endpoints - What Are They? Why Do They Matter?.** Disponível em: https://smartbear.com/learn/performance-monitoring/api-endpoints/. Acesso em: 9 nov. 2019.

SOARES, Michel dos Santos. **Comparação entre Metodologias Ágeis e Tradicionais para o Desenvolvimento de Software.** Disponível em: wiki.dcc.ufba.br/pub/Aside/ProjetoBitecIsaac/Met._Ageis.pdf. Acesso em: 02 out. 2011.

SOMMERVILLE, Ian. **Software Engineering.**5 ed. Inglaterra. Addison Wesley, 1997.

SOMMERVILLE, Ian. **Engenharia de Software.**6 ed. São Paulo. Addison Wesley, 2003.

SOMMERVILLE, Ian. **Engenharia de Software.**9 ed. São Paulo.Person Prentice Hall, 2011.

APÊNDICES

APÊNDICE A – Tempo de execução da geração da massa de dados no estágio Maternidade via interface

Tipo de Lançamento	Quantidade de dados cadastrados	Tempo de execução do *setup*
Checklist Parto	166	01:02:09
Checklist Desmame	276	01:42:29
Checklist Morte de Leitão	33	00:11:48
Total	475	02:56:26

APÊNDICE B – Tempo de execução da geração da massa de dados no estágio Maternidade usando *templates* de negócio via API

Tipo de Lançamento	Quantidade de dados cadastrados	Tempo de execução do *setup*
Checklist Parto	166	00:02:59
Checklist Desmame	276	00:09:00
Checklist Morte de Leitão	33	00:00:16
Total	475	00:12:15

APÊNDICE C – Declaração de ciência e concordância das instituições envolvidas no trabalho de conclusão de curso

UNIVERSIDADE DO SUL DE SANTA CATARINA
Ciência da Computação
Av. José Acácio Moreira, 787 – Bairro Dehon –
CEP 88704-900 – Tubarão - SC

DECLARAÇÃO DE CIÊNCIA E CONCORDÂNCIA DAS INSTITUIÇÕES ENVOLVIDAS NO TRABALHO DE CONCLUSÃO DE CURSO

Local e data: Florianópolis – 19 de setembro de 2019

O representante legal da instituição AGRINESS SISTEMAS E TECNOLOGIAS DE INFORMAÇÃO S.A, objeto do estudo de caso da estudante Débora da Silva, executora do **Trabalho de Conclusão do Curso Ciência da Computação** que tem por tema Behavior Driven Development: Um estudo de caso, declara estar cientes e de acordo com o desenvolvimento desse trabalho.

O mesmo também autoriza [X], não autoriza [ ] o uso do nome da instituição no referido Trabalho de Conclusão de Curso.

Ass. do(s) aluno(s) executores do Trabalho de Conclusão de Curso (UNISUL)

UNISUL - Universidade do Sul de Santa Catarina
Congregação do Curso Ciência da Computação

Ass. e carimbo da Professora Orientadora do Curso
Prof. Maria Inês Castineira
Coordenadora
Por Delegação do Reitor através
Portaria GR nº 1211 de 08/06/2011

Ass. e carimbo do responsável da instituição envolvida no estudo
04.394.546/0001-25
AGRINESS Sistemas e Tecnologias de Informação S.A.

Adaptado do Comitê de Ética em Pesquisa – UNISUL (CEP UNISUL)