



Universidade Federal Rural de Pernambuco  
Departamento de Estatística e Informática



## Aplicação de Uma Estratégia de Automação Contínua de Testes no Desenvolvimento de Software

Thaís Moura de Freitas

Recife

Janeiro de 2015

Thaís Moura de Freitas

# Aplicação de Uma Estratégia de Automação Contínua de Testes no Desenvolvimento de Software

Orientador: Teresa Maciel

Monografia apresentada ao Curso Bacharelado em Sistemas de Informação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Recife

Janeiro de 2015

Aos meus pais, Marcia e Abraão  
À minha orientadora, Teresa  
Aos meus amigos

## Agradecimentos

Aos meus pais, por me darem condições de estudar e de mostrar sempre a importância do conhecimento, tanto na vida profissional quanto na pessoal. Sempre acreditando em mim e mostrando que eu poderia conseguir tudo aquilo que eu almejasse, me deixando ciente que todo esforço investido sempre é recompensado e que eu não devo desistir nunca de correr atrás dos meus sonhos.

Aos meus orientadores Teresa Maciel e Giordano Cabral, que guiaram esse trabalho e conseguiram alinhar minhas motivações pessoais com as requeridas computacionalmente para um trabalho de conclusão de curso. Em especial a Maria Conceição (Professora do Curso de Sistemas de informações - UFRPE) que foi minha orientadora de projeto de iniciação científica e da monitoria durante a graduação e que me deu várias oportunidades para construção do conhecimento. A todo conhecimento que a Universidade Federal Rural de Pernambuco e seus professores me proporcionaram.

Aos meus amigos, que representam uma grande parte do meu incentivo, confiança e por acreditar que sou capaz de conquistar e ultrapassar barreiras. A Rebeka Maranhão (M.a em Ciências Biológicas - UFPE), Julia Mariana (Graduanda em Administração - UPE) por conseguirem me acalmar e dar conselhos especiais, por terem me encorajado com essa fase difícil e por estarem sempre ao meu lado. A tantos outros amigos que também participaram, apoiaram com conversas e com paciência para que toda essa jornada acabasse. A Johann Gomes (Bacharel em Sistemas de Informações na UFRPE) por ter sido um grande companheiro durante toda jornada na graduação e por partilhar as dificuldades, tornando esses desafios mais fáceis. A Luan Reis (Graduando em Sistemas de Informações na UFRPE) companheiro de projeto, por ser um grande amigo e dividir os momentos mais difíceis da nossa trajetória dentro da graduação e em especial no começo da nossa vida profissional, dividindo momentos difíceis e especiais. Agradeço especialmente, a Marília de Andrade (Médica Veterinária - UFRPE) por ter sido uma grande e especial companheira nesses momentos difíceis. A ThoughtWorks empresa que trabalho atualmente, aos amigos que fiz lá e ao meu time atual, por ter me fornecido todo suporte necessário para que conseguisse conciliar e finalizar este trabalho de conclusão de curso.

# Resumo

Devido à necessidade de softwares com alta qualidade e pouco tempo para atividades de validação e verificação, técnicas de teste de software foram desenvolvidas com a finalidade de identificar e corrigir erros pertinentes ao sistema, dando suporte ao desenvolvimento e integrando todo o ciclo de vida da aplicação.

Dentre as técnicas criadas, uma delas é a automação de testes, que em meio à correria do desenvolvimento de software se torna uma poderosa estratégia para minimizar o esforço com atividades de testes, principalmente os de regressão, que são: diminuição de tempo gasto em atividades de testes, reuso de testes, maior cobertura dos testes aplicados entre outros.

Com o objetivo de encontrar falhas o mais cedo possível, este trabalho apresenta uma estratégia de automação de testes durante todo o ciclo de vida do desenvolvimento de uma aplicação. Participando de todas as fases do desenvolvimento, realizando atividades em conjunto com o desenvolvedor, para cobrir todos os níveis possíveis de implementação do sistema. Os resultados apurados destacam o impacto de tal estratégia na realização dos testes, viabiliza a aplicação de testes automatizados durante o desenvolvimento e ressalta a importância que a qualidade final pode melhorar com este tipo de estratégia.

**Palavras-chave:** teste de software, automação de testes, teste de regressão.

# Abstract

Due to the necessity of high quality softwares and not enough time for activities like validation and verification, were developed software testing techniques in order to identify and correct relevant errors from the system, supporting the development and also integrating the entire application life cycle.

Amongst the techniques created, one of them is the automation of tests, that in way the running it software development it becomes a powerful strategy to minimize the effort with testing activities, especially regression testing.

In order to find as soon as possible any flaw, this work represents a test automation strategy throughout the life cycle of the development of an application. Participating in all stages of the development, carrying through activities jointly with the collaborator to cover all the possible levels of system implementation.

The results obtained highlight the impact of such a strategy in the accomplishment of the tests, make possible the application of tests automatized during the development and stand out the importance of the final quality of the product with this type of automation.

**Keywords:** Software testing, test automation, regression testing

# Sumário

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Apresentação . . . . .	1
1.2	Justificativas . . . . .	2
1.3	Objetivos . . . . .	6
1.3.1	Objetivo Geral . . . . .	6
1.3.2	Objetivo Específicos . . . . .	6
1.4	Organização do trabalho . . . . .	6
<b>2</b>	<b>Referencial Teórico</b>	<b>8</b>
2.1	Teste de Software . . . . .	8
2.2	Tipos de Teste de software . . . . .	9
2.2.1	Testes Manuais . . . . .	9
2.2.2	Testes Automatizados . . . . .	9
2.3	Níveis de testes de Software . . . . .	9
2.3.1	Testes Funcionais . . . . .	10
2.3.2	Testes Unitários . . . . .	10

2.3.3	Testes de Integração . . . . .	10
2.3.4	Teste de Sistema . . . . .	11
2.3.5	Teste de Regressão . . . . .	11
2.3.6	Testes de Aceitação . . . . .	11
2.3.7	Testes Não-Funcionais . . . . .	12
2.4	Automação de testes . . . . .	12
2.4.1	Vantagens da automação de testes . . . . .	13
2.4.2	Desafios da automação de testes . . . . .	14
2.4.3	Técnicas de automação de testes . . . . .	15
2.4.4	Ferramentas utilizadas na automação dos testes . . . . .	17
2.4.5	Características do BDD utilizadas neste trabalho . . . . .	18
2.4.6	Cucumber . . . . .	20
<b>3</b>	<b>Metodologia</b>	<b>22</b>
3.1	Definição do Problema . . . . .	23
3.2	Definição da Solução para aplicação da automação dos testes . . . . .	23
3.3	Referencial Teórico . . . . .	24
3.4	Seleção dos tipos de testes . . . . .	24
3.5	Aplicação de testes automatizados ao longo do desenvolvimento . . . . .	25
3.6	Análise do resultado dos testes automatizados . . . . .	25
3.7	Conclusão . . . . .	25
<b>4</b>	<b>Automação de testes no desenvolvimento de uma aplicação</b>	<b>26</b>



4.1	Contextualização da Aplicação . . . . .	26
4.2	Arquitetura da Aplicação . . . . .	27
4.3	Processo de automação dos testes . . . . .	28
4.3.1	Definição dos objetivos dos testes . . . . .	29
4.3.2	Implementação dos testes unitários e integração . . . . .	30
4.3.3	Implementando BDD com Cucumber . . . . .	35
4.4	Resultado da automação dos testes . . . . .	41
<b>5</b>	<b>Conclusão</b>	<b>45</b>
5.1	Conclusão do Trabalho . . . . .	45
5.2	Trabalhos futuros . . . . .	46

# Lista de Tabelas

4.1	Teste para validação do construtor da classe JAudioCommunicator . . . . .	31
4.2	Implementação dos testes unitários para classe JAudioCommunicator . . . . .	31
4.3	Implementação dos testes de integração para classe JAudioCommunicator . . . . .	32
4.4	Teste para validação do construtor da classe ACECommunicator . . . . .	32
4.5	Implementação dos testes unitários para classe ACECommunicator . . . . .	33
4.6	Implementação dos testes de integração para classe ACECommunicator . . . . .	34
4.7	Dados objetivos referentes ao desenvolvimento deste trabalho . . . . .	43
4.8	Quantidade e tipo de commits realizados dentro do projeto . . . . .	43
4.9	Porcentagem da cobertura dos testes que foram desenvolvidos . . . . .	44

# Lista de Figuras

1.1	Pirâmide de testes anti-padrão . . . . .	4
1.2	Pirâmide ideal de testes . . . . .	5
2.1	Modelo da estrutura de uma estória do usuário . . . . .	18
2.2	Exemplo de escrita de uma estória do usuário . . . . .	19
2.3	Exemplo de modelo da estrutura de um cenário . . . . .	19
2.4	Ilustração de um cenário no arquivo .feature . . . . .	21
4.1	Arquitetura utilizada no sistema desenvolvido . . . . .	28
4.2	Representação da integração do desenvolvimento e automação dos testes . . . . .	29
4.3	Arquitetura dos pacotes de testes automatizados de unidade e integração, dentro do pacote de testes Communicator. . . . .	30
4.4	Implementação de teste que valida o arquivo featureVectorFile. . . . .	35
4.5	Implementação de teste que valida a classificação de um áudio como sirene. . . . .	35
4.6	Protótipo da interface do sistema . . . . .	36
4.7	Estória do usuário desenvolvida pelo sistema . . . . .	36
4.8	Arquitetura dos pacotes responsáveis pela implementação dos testes de BDD com Cucumber. . . . .	37

4.9	Cenários de testes automatizados utilizando Cucumber . . . . .	38
4.10	Implementação dos steps de um cenário de teste com Cucumber . . . . .	39
4.11	Resposta da execução dos cenários de testes que foram implementados com Cucumber . . . . .	40
4.12	Resposta da execução de um cenário de testes que possui um step com falha	41
4.13	Quantidade de cenários de testes e de steps pertencentes à feature executada no Cucumber . . . . .	41
4.14	Pirâmide de testes dividida nas faces de negócio e tecnologia. . . . .	42
4.15	Pirâmide com as quantidades de testes desenvolvidos neste trabalho . . . . .	44

# Capítulo 1

## Introdução

### 1.1 Apresentação

No desenvolvimento de software muitos métodos ágeis, como Lean, Scrum e XP é recomendado que todas as pessoas de um projeto (programadores, gerentes, equipes de homologação e até mesmo os clientes) trabalhem controlando a qualidade do produto, todos os dias e a todo momento, pois, há evidências que prevenir defeitos é mais fácil e barato do que identificar e corrigir [27]. Vale ressaltar ainda, que os métodos ágeis não se opõem a quaisquer revisões adicionais que sejam feitas para aumentar a qualidade [27].

Uma importante etapa do ciclo de vida do desenvolvimento de software que compõe as metodologias ágeis é a fase que envolve os testes, pois a qualidade dos testes aplicados impacta diretamente no funcionamento estável das aplicações. Essa fase pode ser demorada e desgastante quando é executada de forma repetitiva, principalmente se falhas forem encontradas através dos testes exploratórios ou ah-doc, o que levará o testador possivelmente a investir tempo na investigação e entendimento das reais causas do problema [38].

Uma das principais razões para automatizar testes é a diminuição do tempo gasto nos testes manuais [22]. Além de aumentar a eficiência de etapas repetitivas para reprodução de funcionalidades do sistema, especialmente em testes de regressão, onde os testes são executados iterativamente e de maneira incremental após mudanças feitas no software [7].

## 1.2 Justificativas

A necessidade da entrega cada vez mais rápida de produtos de software faz com que o processo de teste necessite constantemente de rapidez e agilidade. Assim sendo, a automação de testes no ciclo de vida do desenvolvimento de software tem sido constantemente utilizada para suprir essa carência. É importante ressaltar que existe diferença entre testes e automação de testes, o primeiro termo se refere ao ato de testar, já automação é utilizar um software para imitar a interação do ser humano com a aplicação a ser testada [23].

Um fator que deve ser levado em consideração é o tempo de execução dos testes automatizados, pois ocorre em proporções bem menores em relação ao tempo executado por um processo de teste manual, onde nem sempre o time terá tempo hábil para aplicar todos os testes planejados. Isso não exclui a possibilidade do processo de testes ser híbrido, isto é, contemplar testes manuais e automatizados. Automação de testes pode ser uma poderosa forma de testes não funcionais, por exemplo na execução de testes de carga e regressão [28].

A automação dos testes tem como intuito a maximização da cobertura dos testes dentro do tempo disponível, para a validação e construção do software, aumentando a confiabilidade e a qualidade [4]. Considerando que o esforço em atividades de testes em projetos podem ser responsável por até 50% do esforço total de desenvolvimento, automatizar o processo de testes é importante na redução e melhoria da eficácia dos testes realizados [21]. Os benefícios de tal abordagem em comparação com os testes manuais seriam: baixo custo de execução dos testes, possibilidade de replicar as sequências de teste velhas em novas versões de software (não gastando tempo com testes de regressão) e a possibilidade de realizar testes de estresse por um longo tempo de duração [4].

Ainda querendo responder a pergunta, porque automação de testes? Segundo James Crisp [8] esses seriam os principais motivos:

- Realização dos testes de regressão mais rápido para que os sistemas/aplicações possam continuar mudando ao longo do tempo, sem uma longa fase de testes no final de cada ciclo de desenvolvimento.
- Encontrar defeitos e problemas rapidamente, especialmente quando existe testes que

podem ser executadas em máquinas de desenvolvedores, e como parte dos serviços que realizão o trabalho de complicação em um servidor de CI.

- Certificação de que pontos de integração externos estão trabalhando da forma esperada.
- Assegurar que o usuário pode interagir com o sistema como desejado.
- Auxilia no debugging, escrita e desenho do código fonte.
- Ajuda a especificar o desempenho da aplicação.

No geral, com o uso de testes automatizados estamos aumentando a velocidade de entrega do projeto construído e com maior garantia de qualidade [8]. O que ameniza o esforço gasto pelas empresas no processo com testes manuais e aumenta a cobertura de testes que não seriam realizados, por falta de tempo e esforço.

Muitas organizações costumam cair na mesma armadilha da aplicação da pirâmide de testes invertidas ou anti-padrão, como pode ser visto na Figura 1.1 [33].

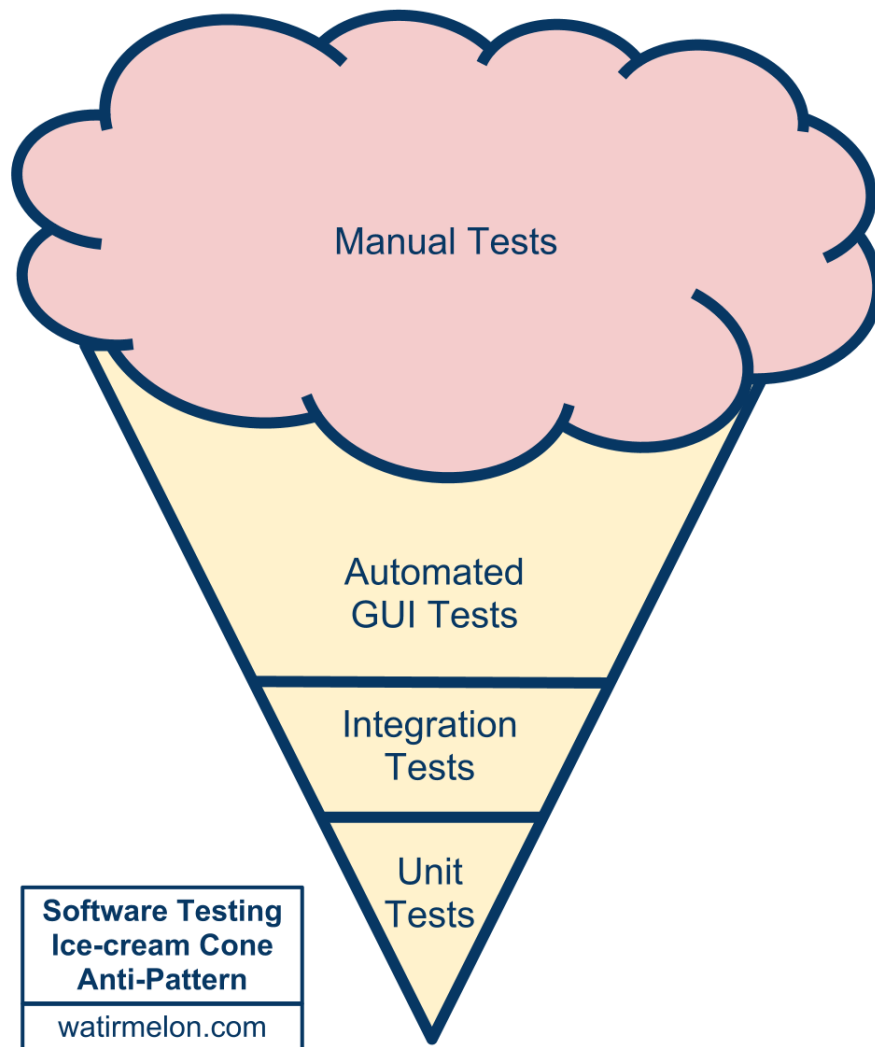


Figura 1.1: Pirâmide de testes anti-padrão

Na Figura 1.1, esta representado visualmente a quantidade de diferentes tipos de testes que podem ser aplicados pelo time no decorrer do desenvolvimento do software, onde é visto que boa parte dos testes são realizados manualmente e um quantidade pequena é realizada a nível unitário. O problema que esse tipo de estratégia de testes pode gerar é que boa parte dos testes ainda estão sendo realizados manualmente e um número muito pequeno de testes de unidade e de integração estão sendo implementado, o que faz dessa automação falha. Visto que uma boa estratégia de automação de testes tem como objetivo maximizar a cobertura de testes de unidade e integração afim de validar diminuir testes manuais e em



camadas superiores, dificultando a validação das funcionalidades e encontrar a real causa de falhas a nível de unidade e integração.

Uma parte importante da estratégia de testes, é saber o foco de cada tipo diferente de testes e fazer com que os diferentes tipos de testes trabalhem juntos [8]. Por exemplo, realizar testes de unidade, com alguns testes de integração e um pequeno número de testes de aceitação. Com essa mistura é possível cobrir caminhos alternativos de testes no código, alcançar e ultrapassar barreiras de testes mais rápido aplicando os testes de unidade [8]. Na Figura 1.2, é mostrada a pirâmide ideal para aplicação de teste automatizado, em que o foco não está apenas nos testes manuais e sim na base da pirâmide [9].

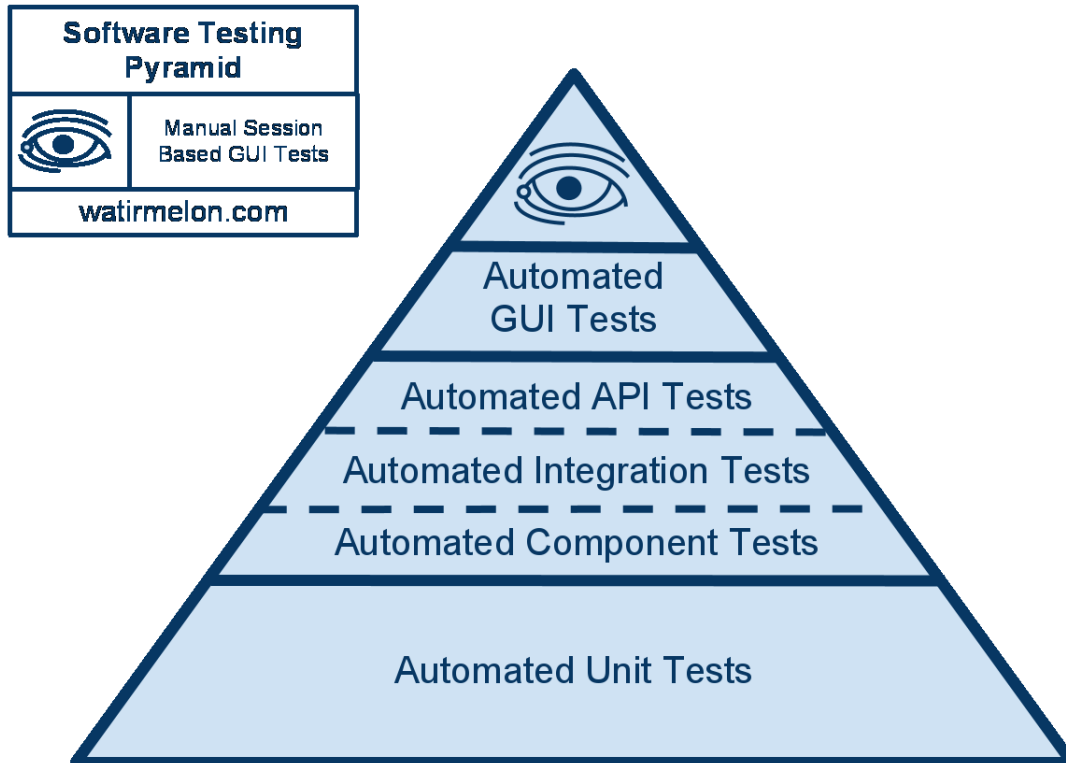


Figura 1.2: Pirâmide ideal de testes

Sendo assim, é de extrema importancia a aplicação de tipos de estratégia de testes como esta, para o desenvolvimento de software, garantindo que o que está sendo implementado seja conforme o esperado e de forma mais rápida e automática, envolvendo todo o time na qualidade final e em todas as fases do processo.

## 1.3 Objetivos

### 1.3.1 Objetivo Geral

O objetivo geral deste trabalho é a implantação e cobertura de testes automatizados durante o desenvolvimento de uma aplicação que é desenvolvida com tecnologia de classificação de sinais de áudio para dar suporte a pessoas com deficiência auditiva ou surda.

### 1.3.2 Objetivo Específicos

Dentre os objetivos específicos deste trabalho, temos:

- Gerar conhecimento em torno de automação de testes, pesquisando e selecionando as melhores técnicas para esta proposta;
- Gerar conhecimento necessário para aplicar automação dos testes da ferramenta jMIR utilizada para construção da aplicação;
- Investigar e selecionar as melhores técnicas e ferramentas aplicáveis na automação dos teste na aplicação escolhida;
- Aplicar automação de testes em todo ciclo de vida no desenvolvimento de uma aplicação;
- Aplicar automação de testes para validar a integração da aplicação com a tecnologia jMIR, mais especificamente o uso do jAudio e ACE que são os componentes principais;
- Realizar o levantamento de quantos testes foram automatizados em cada nível da pirâmide ideal de testes durante o desenvolvimento da aplicação;

## 1.4 Organização do trabalho

Este trabalho é composto por mais quatro capítulos. No capítulo 2 é feita uma revisão bibliográfica sobre os principais conceitos que norteiam este trabalho. No capítulo 3, são

---

apresentados os procedimentos metodológicos utilizados para atingir os objetivos desse trabalho. O capítulo 4 apresenta e contextualiza a aplicação que foi desenvolvida juntamente com este trabalho, o processo de testes realizado e os resultados da automação de testes durante o desenvolvimento. Por fim, o capítulo 5 mostra o impacto da automação e são feitas as considerações finais.

# Capítulo 2

## Referencial Teórico

É apresentado nesse capítulo um conjunto de definições relevantes para o tema dessa proposta de trabalho. Foi realizada uma pesquisa sistemática para selecionar os principais tópicos que apoiam o entendimento deste trabalho. Os detalhes estão descritos nas próximas seções.

### 2.1 Teste de Software

Teste de software é o processo que realiza a avaliação do sistema ou de seus componentes com o objetivo de desvendar o comportamento para garantir que os seus requisitos estão de acordo com o esperado ou não. Em outras palavras teste é execução do sistema, a fim de identificar eventuais lacunas, erros ou requisitos que não foram implementados de acordo com as necessidades requeridas [20]. Teste pode ser definido como o processo de análise do software para detectar as diferenças entre condições existentes e necessárias e avaliar as características do software [5].

Um bom teste é aquele que tem alta probabilidade de encontrar erros que ainda não foram expostos e um teste bem sucedido é aquele que revela um erro ainda não-descoberto. O teste pode ser manual, automatizado, ou ainda a combinação de ambos [31]. A redução de custo, tempo e retrabalho são proporcionais ao quão cedo o processo de testes for iniciado [20].

## 2.2 Tipos de Teste de software

Testes de software podem ser divididos em duas categorias, testes manuais e automatizados.

### 2.2.1 Testes Manuais

Este tipo de teste é feito pela execução manual do software, ou seja, sem o uso de qualquer ferramenta automatizada ou qualquer script. O testador assume o papel do usuário final para realizar os testes e identificar qualquer comportamento que não seja esperado ou revelar defeitos. Costumam usar planos de teste (um plano que deve ser seguido pelo testador para realização das atividades de testes), casos de teste (situações que validam as funcionalidades ou requisitos do software) e/ou cenários para garantir a integridade dos testes. Também pode ser incluso dentro deste tipo os testes exploratórios, no qual o testador irá explorar o software com o intuito de encontrar erros[20].

### 2.2.2 Testes Automatizados

Testes automatizados podem ser definidos como automação de atividades de teste de software, incluindo o desenvolvimento e execução de scripts de testes, verificação de requisitos, como também a utilização de ferramentas de teste automatizadas [30].

Entre as razões para o uso da automação dos testes de software podemos citar, por exemplo, que executar testes manuais é mais demorado, o uso da automação de testes aumenta a eficiência do processo de testes que seguido pelo time, ainda é valido levantar que temos os testes de regressão, onde os casos de testes são executados de forma iterativa, depois de alterações no software [30].

## 2.3 Níveis de testes de Software

Em Pressman [31], aponta que a realização de testes apenas quando o sistema está construído é uma abordagem ineficaz. Segundo o autor, a estratégia de testes de software de

possuir uma abordagem incremental, começando com os testes de unidade, seguindo os testes de integração, culminando com os testes do sistema final e ainda acoplando testes que se enquadrem em testes de aceitação.

### **2.3.1 Testes Funcionais**

Este tipo de teste é baseado nas especificações do software que será testado, a aplicação será testada através do fornecimento de entradas e em seguida, os resultados serão examinados para garantir a conformidade com os requisitos da funcionalidade [20]. Basicamente testar se os componentes e o sistema estão feitos como era esperado, por exemplo, uma atividade ou uma função específica do código está coerente com o esperado. Neste nível de teste perguntas como "O usuário poderá fazer isso?" ou "Esta função em particular funciona?" podem ser validadas tipicamente através de especificações de requisitos ou funcional [37].

### **2.3.2 Testes Unitários**

É a menor parte que pode ser testada do código que compõe o software, são de granulação fina, comportamento extremamente rápido, por exemplo, métodos de uma classe, uma classe ou classes que podem estar relacionadas. Não verificam o comportamento de unidades integradas com outros serviços ou dependências, garantem que sua função como unidade estão funcionando. Seu objetivo é isolar parte do programa e mostrar que partes individuais estão corretas em termos de requisitos e funcionalidades [8]. Existe um limite para cenários e dados que podem ser aplicados ao nível de testes de unidade, quando é nítido esse esgotamento é necessário o mistura com outras unidades do código e diferentes tipos de testes [20].

### **2.3.3 Testes de Integração**

Neste caso, é realizada a combinação de partes da aplicação que serão agrupadas para determinar se funcionamento desse conjunto está trabalhando corretamente [20]. Alguns defeitos não podem ser encontrados a nível unitário, mas são revelados através da integração com núcleos específicos [36].

### 2.3.4 Teste de Sistema

Após a realização dos testes de integração, uma vez que todos os componentes foram integrados, os testes do sistema como um todo são agora efetuados para garantir que a aplicação atende aos padrões de qualidade desejado [20]. A aplicação é testada minuciosamente para validar especificações funcionais e técnicas, é construído um ambiente o mais próximo possível do ambiente de produção, para que os testes sejam executados em um ambiente que corresponda ao ambiente no qual o sistema será implantado. O teste do sistema nos retornará a validação dos requisitos de negócio e da arquitetura como todo da aplicação.

### 2.3.5 Teste de Regressão

Significa testar uma aplicação após o seu código fonte ter sido modificado para validar se ainda continua funcionando devidamente. Consiste em reexecutar casos de testes existentes e verificar se alterações no código não interferiram em funcionalidades que estavam trabalhando corretamente, se foram inseridos novos erros ou causa problemas em erros que já foram reparados [37]. A fim de identificar defeitos introduzidos por novas funcionalidades ou correção de defeitos.

### 2.3.6 Testes de Aceitação

Nível mais alto que trata da aplicação como uma caixa preta, sem dúvidas considerado um dos mais importantes entre os outros, este tipo de teste é feito na grande parte pelo time de garantia da qualidade [8]. Onde todo time verificará se o produto preenche as especificações solicitadas e satisfaz os requisitos dos clientes. O time responsável terá um conjunto de cenários previamente escritos e que serão usados no momento da validação. Durante a fase de validação outros cenários podem surgir para aumentar a cobertura dos testes. Testes de aceitação não são apenas para identificação de erros de interfaces, mas também para encontrar divergências que poderão resultar em quebra ou erros graves do sistema [20]. Para a execução de desse tipo de testes é necessário definir critérios de aceitação a partir dos requisitos do software, estabelecendo como o teste será conduzido e a partir desses critérios,

avaliar se o produto satisfaz aos requisitos [34].

### 2.3.7 Testes Não-Funcionais

Requisitos não funcionais são declarações que define as qualidades globais ou atributos a serem atendidos pelo sistema resultante [18]. Estes requisitos, ao contrário dos requisitos funcionais, não expressão nenhuma função a ser realizada pelo software, e sim comportamentos e restrições que este software deve satisfazer [11]. Os testes desses requisitos são normalmente executados com ajuda de ferramentas especializadas, com grande planejamento, avaliação arquitetural, aplicando técnicas avançadas [20].

Alguns dos considerados mais importantes tipos de validação não funcionais são [20]:

- Teste de Performance
- Teste de Carga
- Teste de Stress
- Teste de Segurança
- Teste de Usabilidade
- Teste de Portabilidade

## 2.4 Automação de testes

Ainda que a atividade seja complexa, o teste de software nem sempre é realizado de forma sistemática devido a diversos fatores dos projetos, como: tempo e recursos limitados, qualificação do time e dos envolvidos, complexidade e rapidez da evolução dos sistemas. Sendo assim, a automação de testes é uma importante medida para melhorar a eficiência dessas atividades [6].

Automação de testes é uso de um software para controlar a execução das atividades de teste de um sistema, a comparação dos resultados esperados com os resultados reais, a configuração



das pré-condições de teste e outras funções de controle e relatório de teste [15]. Atualmente essa prática vem sendo difundida em processos ágeis e até mesmo em organizações que ainda utilizam metodologias tradicionais como Rational Unified Process [19].

Segundo Fewster e Graham [13] afirmam que a automação de testes de software pode reduzir de maneira significativa e impactante o esforço necessário para execução dos testes. Assim como Molinari[25], afirma que testes automatizados aumenta a produtividade e alcança em tempo menor os objetivos do teste. Apesar do grande poder que os testes automatizados apresentam, algumas empresas ainda resistem na aplicação de estratégias de automação de testes, sobretudo, por conta do esforço necessário para iniciar o processo e por demandar mudança na cultura da organização [19]. Mais rápido que os testes manuais e menos suscetível a erros, máquinas possuem probabilidade de erro bem menor. Aprimorar e aplicar testes automatizados tem se tornado algo crucial nas empresas atualmente, segundo pesquisa realizada em 2008 pelo Forrester Research Inc [2].

A seguir serão citadas e comentadas algumas vantagens e dificuldades na implantação de testes automatizados.

### 2.4.1 Vantagens da automação de testes

De acordo com Fewster e Graham [13], os principais benefícios da automação de testes de software são:

- Torna a execução dos testes de regressão mais fácil, tendo em vista a reutilização de casos de testes automatizados de versões antigas em versões seguintes aplicando o esforço manual mínimo em testes manuais.
- Permite executar um maior número de testes em menor tempo, aumentando a frequência dos testes e trazendo mais confiança ao sistema;
- Execução de testes que seriam inviáveis se realizados manualmente, como a verificação de cálculos extremamente complexos ou testes de carga com um número alto de acessos;
- Aprimorar a utilização dos recursos, permitindo que os testadores reduzam o esforço

em atividades repetitivas, conseguindo focar esforços em atividades de planejamento dos testes e na execução de cenários cuja automação é inviável;

- Aumentar a consistência dos resultados dos testes entre diferentes plataformas, já que as entradas e condições esperadas são as mesmas;
- Diminuição do “Time to Market”, tendo em vista a redução do tempo do projeto dedicado aos testes;
- Aumentar a credibilidade e confiança na qualidade do sistema como um todo, um bom volume de testes automatizados bem planejados e executados com êxito podem trazer para o time um sentimento de confiança na qualidade do produto final.

Com base nessas vantagens é possível perceber o grande potencial da automação dos testes na melhoria das atividades que envolvem testes de software. Alguns desses benefícios podem ser evidenciados através da aplicação que será utilizada como caso de estudo desta proposta de trabalho.

### 2.4.2 Desafios da automação de testes

Em conformidade com Fewster e Graham [13] as principais barreiras e desafios na implantação da automação de testes de software em uma organização são:

- Expectativas irreais: Automação de testes não é a solução de todos os problemas e não deve ser vista como tal, deve ser encarada de forma realista;
- Prática de testes pobre: Caso o processo de testes adotado pela organização seja ineficaz, com testes mal planejados e documentação inconsistente, é necessário melhorar e corrigir o processo antes da implantação da automação.
- Manutenção dos testes: alguns softwares sofrem muitas modificações, o esforço de manutenção dos testes automatizados pode ser maior que o esforço de reexecuta-los manualmente;

- Problemas técnicos: ferramentas de automação de testes também podem sofrer problemas, comprometendo a estratégia de automação. É necessário que haja um responsável pela automação e tenha conhecimentos técnicos detalhados das ferramentas, para que o sua utilização seja efetiva.
- Questões organizacionais: a automação de testes precisa ser implementada na cultura da organização, com alocação de tempo para a escolha de ferramentas e formação, visando entender e selecionar as práticas que melhor se adequam a organização;

Diante dessas dificuldades, a implantação de testes automatizados é uma atividade complexa. Deve, dessa forma, ser bem analisada sobre sua necessidade e como deve ser adaptada ao processo atual da organização e time.

### 2.4.3 Técnicas de automação de testes

As principais técnicas de automação de teste, presentes na literatura são: record and playback, scripts, data-driven e keyword-drivers. Essas técnicas serão expostas a seguir [6].

- Record and playback

Essa técnica se baseia na gravação das ações do usuário durante a interação com a interface gráfica do sistema, convertendo essas ações em scripts de testes que podem ser reproduzidos várias vezes for necessário. Para cada caso de teste é gravado um script de teste completo que inclui os dados de teste (dados de entrada e resultados esperados), o procedimento de teste (passo a passo que representa a lógica de execução) e as ações de teste sobre a aplicação.

Mesmo sendo considerada uma técnica simples e prática, sendo uma abordagem para testes executados poucas vezes, apresenta algumas desvantagens para uma grande quantidade de testes automatizados, com alto custo e dificuldade de manutenção, baixa taxa de reutilização, curto tempo de vida e alta sensibilidade a mudanças no software a ser testado e no ambiente de teste.

- Programação de scripts

É uma extensão da técnica record and playback, utilizando programação e seus recursos é possível alterar o código gerado através dos scripts resultantes da gravação do Record and Playback, permitindo alcançar uma maior quantidade de verificações de resultados esperados.

Com o uso desta técnica é possível obter uma maior taxa de reutilização dos scripts de testes, maior tempo de vida, melhor manutenção e maior robustez dos scripts de teste. A aplicação dessa técnica produz uma grande quantidade de scripts de teste, visto que para cada caso de teste deve ser programado um script de teste, o qual também inclui os dados de teste e o procedimento [6].

- Data driven

Consiste na extração, dos scripts de testes, os dados de entrada e armazená-los em arquivos separadamente, deixando com que os scripts possuam apenas a lógica de execução dos testes. Os dados de entrada são obtidos de um arquivo separado de acordo com cada caso de teste desenvolvido. Como principal vantagem, essa abordagem possui baixo esforço de manutenção ao adicionar, remover ou modificar um caso de teste [39].

- Keyword-driven

Técnica orientada a palavras chave, consistem na extração da lógica dos scripts de teste, que passam a ter apenas ações específicas de teste sobre a aplicação, que serão identificadas por palavras-chave. Essas ações podem ser comparadas a funções de um programa que são acionadas por palavras chave, durante a execução do caso de teste. Os passos executados pelo script são armazenados em um arquivo separado, na forma de um conjunto de palavras-chave e parâmetros de teste no próprio código, obtendo-os diretamente dos arquivos de procedimento de teste.

Na implementação de automação dos testes desta proposta, foram levantados as possíveis técnicas demonstradas aqui para serem aplicadas. Por ser um protótipo desktop que usa frameworks externos, praticamente todos os testes desenvolvidos foram baseados na técnica de Programação de Scripts. Para isso, foi possível incorporar outros recursos como a verificação automática de resultados com o jUnit [17], em casos de testes de unidade e integração,

o Cucumber [10] para alguns cenários demonstrativos aplicando BDD (Behaviour Driven Development) e testes de aceitação. Essas ferramentas serão explicadas na próxima seção. Técnicas como Driven e Keyword-Driven foram descartadas para essa aplicação devido aos casos de testes desenvolvidos não possuírem uma grande quantidade de dados de entrada que justificasse a adoção dessas técnicas.

#### 2.4.4 Ferramentas utilizadas na automação dos testes

Existe há discursão, desde o surgimento da área de testes de software, sobre a utilização de ferramentas que facilitem o trabalho dos profissionais de teste, tarefas como o planejamento de casos de testes, execução de testes, abertura de defeitos entre outros [35]. Objetivando dar suporte a estratégia de automação de testes apresentada neste trabalho, serão apresentadas a seguir as ferramentas que foram utilizadas.

- jUnit

Framework open source (pacote de software livre) que facilita o desenvolvimento e a execução dos testes unitários em Java. O jUnit permite que os testes possam ser executados sequencialmente ou de forma modularizada, dessa forma, os sistemas podem ser testados em partes ou de uma única vez [17]. Em uma visão geral, o jUnit tem o funcionamento com base na verificação das saídas dos métodos de uma classe, analisando se eles apresentaram os resultados esperados, mostrando de forma visual o resultado da execução dos testes.

- Desenvolvimento orientado por comportamento (Behaviour Driven Development - BDD)

Concentrado na definição de especificações de granulação fina do comportamento direto do sistema, de forma que eles possam ser automatizado. BDD (Behaviour Driven Development) tem como principal objetivo a obtenção das especificações executáveis de um sistema [35]. Os testes em BDD são escritos de forma clara e facilmente compreensível, fornecendo uma linguagem ubíqua específica que ajuda todos os envolvidos a captarem o contexto dos testes e participarem na escrita das especificações do sistema.

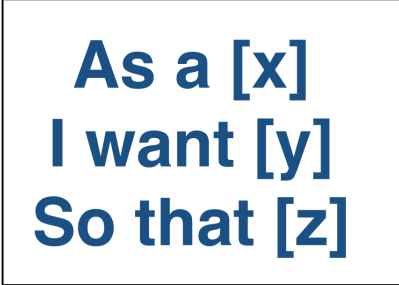
Existem vários kits de ferramentas para o desenvolvimento e aplicação das técnicas de BDD, como JBehave [16], Cucumber [10] e RSpec [32].

### 2.4.5 Características do BDD utilizadas neste trabalho

Com base em toda gama de atividades que constitui o desenvolvimento de software, incluindo obtenção de requisitos, análise, projeto, testes e implementação. Segundo a revisão da literatura [35] existem seis principais características fundamentais que constituem o BDD, para esta proposta de automação de testes e para o uso do framework Cucumber, serão explicados as características que foram usadas para implantação neste trabalho.

- Descrição de texto simples com estória de usuário e modelos de cenários

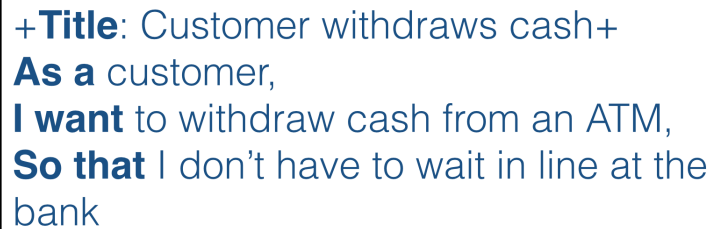
Utilizando textos simples para descrição de características, estórias de usuários e cenários, não possuem um formato aleatório. Um modelo pré-definido é usado para especificá-los. Esse modelo é definido através de uma linguagem simples ubíqua que BDD fornece. Normalmente estórias de usuário são especificadas usando o modelo da Figura 2.1 [26]:



**As a [x]  
I want [y]  
So that [z]**

Figura 2.1: Modelo da estrutura de uma estória do usuário

Onde Y é o comportamento ou funcionalidade esperada, Z é o benefício ou valor desta característica e X é o papel ou pessoa beneficiada com essa funcionalidade. Para ilustrar um exemplo de como a estória do usuário pode ser escrita, vide na Figura 2.2:

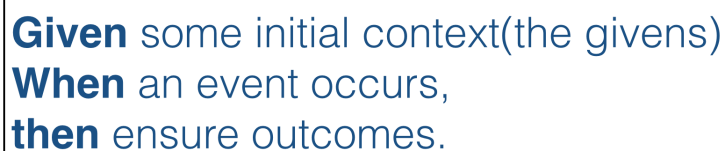
A rectangular box with a thin black border containing text in blue. The text is a user story written in the 'I want' format.

**+Title:** Customer withdraws cash+  
**As a** customer,  
**I want** to withdraw cash from an ATM,  
**So that** I don't have to wait in line at the  
bank

Figura 2.2: Exemplo de escrita de uma estória do usuário

O título da estória descreve a atividade que é realizada por um usuário em um determinado papel. A funcionalidade desejada fornecida pelo usuário, específica a característica que o sistema deve possuir, com a implementação desta funcionalidade o usuário obtém um benefício. Utilizando este modelo fica clara a necessidade de implementação de tal característica e o seu impacto para o usuário. Os desenvolvedores sabem como o sistema deve se comportar e com quem devem falar sobre tal requisito. Além disso, os usuários entram em questionamento sobre a real necessidade de tal característica, uma vez que é perguntado qual ganho ele terá com essa implementação.

Um cenário descreve a forma como o sistema deve se comportar quando esta em um estado específico e um evento acontece, para que a funcionalidade ou característica requisitada esteja funcionando de acordo com o esperado, como ilustrado na Figura 2.3. O resultado do cenário altera o estado do sistema ou produz uma saída para o usuário. Nós usamos o termo ação em vez de saída do sistema [35] porque, uma ação representa qualquer comportamento de reação do sistema.

A rectangular box with a thin black border containing text in blue. The text is a scenario model written in the 'Given-When-then' format.

**Given** some initial context(the givens)  
**When** an event occurs,  
**then** ensure outcomes.

Figura 2.3: Exemplo de modelo da estrutura de um cenário

- Automação de testes de aceitação com regras de mapeamento

Testes de aceitação com BDD é uma especificação do comportamento do sistema,

especificação executável que verifica as interações (conduta) dos objetos em vez de só avaliar seus estados [26]. Um cenário é composto de várias etapas, a etapa é uma abstração que representa um dos elementos do cenário que são: contexto, eventos e ações. Um exemplo do significado deles pode ser: em um caso particular de uma estória do usuário ou contexto C, quando um evento X acontece, a resposta do sistema deve ser Z. Cada passo desse é mapeado para um método de teste da aplicação. Para que um cenário passe (seja validado), é necessário que todas as suas etapas também passem, estas etapas são normalmente chamadas de “steps”. A principal ideia no uso do BDD é o foco em prevenir falhas de comunicação, o que significa que todos no time estão se comunicando de maneira mais frequente, eficaz e com base em exemplos reais, não se baseando em abstrações ou em requisitos imperativos.

Para a implementação da automação dos testes deste trabalho, foram escritos alguns cenários para demonstrar como pode ser utilizado na prática de desenvolvimento e testes com BDD (Behaviour Driven Development) usando a ferramenta Cucumber [10].

### 2.4.6 Cucumber

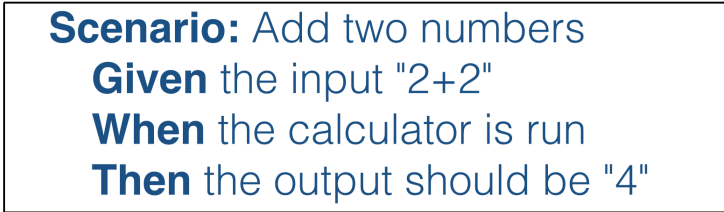
Existem diversas ferramentas que possibilita a aplicação do BDD, mas para este trabalho vamos nos ater a uma delas, o Cucumber. Na utilização do Cucumber, as funcionalidades do sistema são escritas em arquivos de texto e em uma linguagem de domínio específico chamada Gherkin [14], muito parecida com a linguagem natural, mas que contém algumas palavras chaves [10]. Para entender a estrutura básica que o Cucumber define para as estórias do usuário, a seguir são listados alguns conceitos.

- Para o Cucumber, todas as estórias do usuário referentes a uma funcionalidade do sistema estarão agrupadas em um arquivo com a extensão “.feature”;
- No início de cada arquivo existe um resumo da funcionalidade com um formato bem simples: um título, qual o problema a ser resolvido, qual ator trabalha nesta história e qual o resultado desejado;



- Logo depois são definidos os cenários, que é estória propriamente dita ou critérios de aceitação que validam a estória, cada arquivo tem pelo menos um cenário;
- Cada história ou cenário é composto por uma descrição ou título, uma ou mais pré-condições, uma ou mais ações e uma ou mais verificações.

Essa é a estrutura básica de um arquivo “.feature” do Cucumber, esta definição pode ser genérica e gerar dúvidas, então para maior entendimento vamos especificar melhor como é essa estrutura e seu funcionamento. Os testes utilizando BDD são compostos, basicamente, por arquivos que especificam as funcionalidades “.feature” como mencionado e por arquivos de definição de passos “steps”. Os arquivos com as funcionalidades são compostos por cenários, que exemplificam uma ou mais regras de negócio de como o sistema deve se comportar.



```
Scenario: Add two numbers  
Given the input "2+2"  
When the calculator is run  
Then the output should be "4"
```

Figura 2.4: Ilustração de um cenário no arquivo .feature

A fim de aplicar alguns cenários na automação dos testes desenvolvidos neste trabalho, foi utilizado o Cucumber-JVM [33] que implementa o Cucumber na linguagem de programação Java.

# Capítulo 3

## Metodologia

Tendo em vista que o teste de software é uma das principais atividades da Engenharia de Software e que demanda um grande esforço em projetos de desenvolvimento, esse trabalho de conclusão de curso parte do seguinte problema: como implantar uma solução de testes automatizados em um projeto de software, de forma que a cobertura dos testes esteja presente em todos os níveis do sistema, durante todo o ciclo de vida do desenvolvimento? E quais podem ser os impactos dessa estratégia?

Para responder esta pergunta, o presente trabalho teve como foco realizar o desenvolvimento dos testes automatizados da aplicação juntamente com o desenvolvimento do sistema. Trabalhando na qualidade do código gerado desde da concepção da ideia e tendo conhecimento de toda a arquitetura a fim de entender, selecionar e implantar a automação dos testes que foram necessários durante todo o ciclo de vida do desenvolvimento.

Estão fora do escopo deste trabalho o entendimento e análise profunda do framework de classificação de áudio jMir, como também a inteligência na classificação de áudio por trás dos algoritmos que é utilizado.

Como parte desta proposta foram efetuadas várias atividades, a seguir serão apresentados os passos para execução da realização deste trabalho.

## 3.1 Definição do Problema

Com base em conhecimento prévio na área de testes em times ágeis, foi observado todo o esforço e importância nas atividades envolvendo qualidade de software. Lidando no dia-a-dia com impasses de falta de tempo para realizar testes manuais, a não cobertura de testes durante o desenvolvimento, a exclusão da qualidade do software durante o ciclo de vida e muitos outros pontos foram inputs para a motivação da pesquisa e implantação de automação dos testes durante o desenvolvimento e o impacto que isso poderia gerar no resultado final do produto.

## 3.2 Definição da Solução para aplicação da automação dos testes

- Pesquisa sobre tecnologias de áudio

Para o desenvolvimento de uma aplicação que envolvesse tecnologias que tratam e possibilitam a manipulação de áudio, uma pesquisa sobre áudio monitoramento foi realizada, para entender melhor como essa tecnologia seria integrada com a aplicação. Com base nesta pesquisa, a proposta é automatizar testes de um sistema que utilizam um pacote de software que realiza a classificação de sinais sonoros.

- Identificação da solução de um problema

Ao encontrar ferramentas e apoio de tecnologias que consigam prover a classificação de sinais de áudio, a realização de um Brainstorm foi feita para coletar ideias que serviram de base para a definição de um problema e criação de uma aplicação.

*Brainstorm* (ou *Brainstorming*) é um processo de geração de ideias que explora a capacidade criativa das pessoas, por meio da produção em massa de ideias em um curto espaço de tempo, para posterior avaliação. Foram escolhidos alguns temas para facilitar o encaminhamento de problemas envolvendo a classificação de áudio, tais como: saúde, educação, segurança, música, trânsito, redes sociais (comunicação). A fim de melhorar a identificação das áreas mencionadas, foi atribuída uma cor de cartão a cada área. Cada rodada durava 3 minutos, para que os participantes gerassem ideias

correspondentes à área selecionada, ao fim deste tempo, a área em destaque era trocada e se repetia os mesmo passos. Depois da inserção de ideias, foram lidas todas as sugestões e as ideias foram melhor discutidas e explicadas aos participantes. Dando início à próxima fase, que foi a que os participantes votavam nas três melhores ideias geradas. Ao final da votação foi escolhido as três ideais mais votadas, o grupo então precisou pensar quais problemas cada ideia poderia resolver.

- **Concepção da aplicação/solução**

Com base na ideia gerada através do Brainstorm mencionado, foi concebida a estória do usuário principal para a definição do escopo da aplicação.

### **3.3 Referencial Teórico**

Definida a aplicação em que esta proposta seria aplicada, foi realizada a revisão da literatura com intuito de sintetizar os principais conceitos, definições e técnicas. Os assuntos que serviram de tema para este estudo teórico foram: Testes de Software, Tipos de Testes de Software, Níveis de Testes de Software e Automação de Testes.

### **3.4 Seleção dos tipos de testes**

Com o escopo da aplicação definida, com o conhecimento necessário sobre quais tecnologias de classificação de áudio utilizar e com o embasamento teórico como base, foi possível realizar o reconhecimento de quais tipos de testes poderiam ser aplicados, quais tecnologias utilizar para cada tipo de testes selecionado.

### **3.5 Aplicação de testes automatizados ao longo do desenvolvimento**

Como parte da proposta deste trabalho, testes eram automatizado com base no código que era desenvolvido ao mesmo tempo, gerando validação do que era produzido, reparação de falhas e divergências da implementação. O desenvolvimento do sistema mencionado, foi desenvolvido em conjunto com Luan Reis, também aluna do curso de Sistemas de Informação da Universidade Federal Rural de Pernambuco.

### **3.6 Análise do resultado dos testes automatizados**

Com base nos resultados da aplicação da automação dos testes durante o desenvolvimento do começo ao fim do sistema desenvolvido, foi possível levantar vários pontos como benefícios e reconhecer o impacto de tal estratégia de testes. Por fim, tornou-se concreto garantir que o sistema desenvolvido tinha todos os níveis de testes cobertos pela automação implementada.

### **3.7 Conclusão**

Nesta fase é realizada uma reflexão sobre a implantação dos testes automatizados como parte do processo de desenvolvimento e de como isso pode trazer benefícios a qualidade final do produto. Onde trabalhos futuros podem ser gerados através da aplicação e melhoria de estratégias como esta.

# Capítulo 4

## Automação de testes no desenvolvimento de uma aplicação

### 4.1 Contextualização da Aplicação

Para realização da proposta deste trabalho, um ideia de protótipo foi gerada a partir de um *Brainstorm*, a fim de criar uma aplicação para dispositivos móveis que fosse capaz de ajudar pessoas com deficiência auditiva ou surda, utilizando a tecnologia de classificação de sinais de áudio.

Segundo o censo de 2010 do IBGE, 45.623.910 de pessoas, que equivalia a 23,9% da população brasileira, declarou apresentar algum tipo de deficiência física. Sendo 9.722.163 de deficientes auditivos [29]. Um fator agravante desta realidade é a intensidade de sons que as pessoas estão submetidas ao decorrer do dia-a-dia. Alguns dos sons a seguir são exemplos que contribuem para a perda da audição: sons emitidos durante o tráfego de automóveis, obras em andamento, shows, boates e músicas em volume exagerado nos fones de ouvido [12]. Segundo especialistas, o limite saudável à exposição aos sons é de 80 decibéis (dB). Uma exposição diária acima desse limite pode causar danos irreversíveis para o indivíduo [24].

Para que este protótipo de sistema fosse desenvolvido, foi necessário alguns requisitos:

- Possuir uma base de áudios que serviriam para treinar o classificador, de forma que ele

aprenda a classificar um som, se baseando em sons já existentes e conhecidos. Caso o som tenha sido gravado em um ambiente com pouquíssimo ruídos, o classificador terá maior chance de classificar corretamente o áudio, logo ter um som de qualidade pode interferir no resultado final. Caso o som não possua o mínimo de qualidade as chances da classificação ser errada são altas.

- Utilização de uma ferramenta de extração e de classificação de áudio, para este trabalho a ferramenta escolhida foi o jMIR. O jMIR é uma suíte de software de código aberto implementado em Java para uso na Recuperação de Informação de Musical (*Musical Information Retrieve* – Recuperação de Informação Musical). Oferecendo suporte na manipulação de arquivos de áudio, como também para arquivos de formato simbólico. Possui ainda pacotes de softwares para: extração de características de áudio, aplicação de algoritmos de aprendizado de máquina, mineração de metadados e análise de metadados [3].

Com base nestes recursos foi possível que o desenvolvimento ocorresse e a estruturação da automação dos testes acompanhasse simultaneamente esse processo.

## 4.2 Arquitetura da Aplicação

Como componentes principais da estrutura do projeto foi utilizado o jAudio, ferramenta responsável pela extração de características do áudio. Este pacote pertencente ao jMIR, onde é utilizado para extração de recursos de arquivos de áudio. Essas características podem ser usadas na recuperação das informações da música (*Musical Information Retrieve* – Recuperação de Informação Musical). Para realizar a classificação dos áudios foi usado a ferramenta também pertencente ao jMIR, *Autonomous Classification Engine*.

O ACE (*Autonomous Classification Engine*) é um pacote de software de meta aprendizado para a seleção, otimização e aplicação do algoritmo de máquina de aprendizado em música. ACE é projetado para aumentar a taxa de sucesso na classificação, facilitar a aplicação da máquina de aprendizado para todos os níveis de usuários e fornecer a experimentação dele com novos algoritmos [1]. Aplicação desenvolvida está estruturada da conforme é mostrado

Figura 4.1.

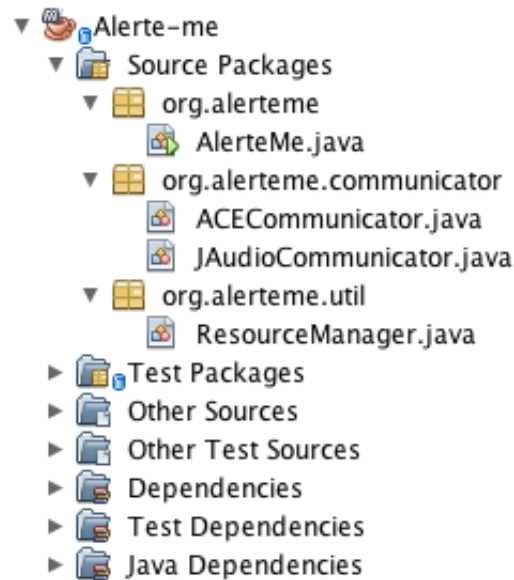


Figura 4.1: Arquitetura utilizada no sistema desenvolvido

Com base na Figura 4.1 podemos ver que o projeto conta com um pacote chamado **communicator**, que por sua vez estão presentes as classes `ACECommunicator` e `JAudioCommunicator`, responsáveis por executarem suas respectivas ferramentas e se comunicarem com o `AlertMe`. A classe `AlertMe` é responsável por chamar todas as classes do pacote `communicator` e por executá-las. O projeto possui um pacote chamado `Test`, neste pacote estão pacotes e classes com a tarefa de automatizar todos os testes desenvolvidos no `AlertMe`.

### 4.3 Processo de automação dos testes

O processo de automação de testes para o sistema desenvolvido foi organizado com o intuito de unir atividades de desenvolvimento e a qualidade do software gerado em tempo de desenvolvimento e não apenas como uma atividade final de validação. Para que isso fosse possível, as atividades de automação foram conduzidas juntamente com o desenvolvedor, o trabalho foi realizado iterativamente com base no que era criado para o sistema, conforme é mostrado no diagrama da Figura 4.2. Todo momento durante o desenvolvimento a implementação e codificação do sistema era feita em par, para que os testes também fizessem



parte da construção da aplicação.

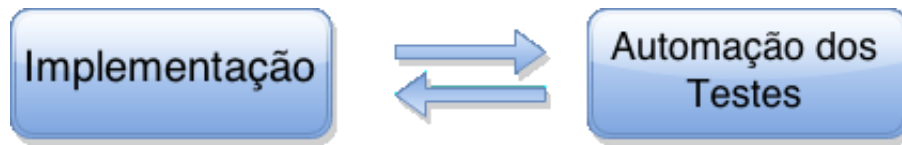


Figura 4.2: Representação da integração do desenvolvimento e automação dos testes

Foi possível observar, que a utilização dessa integração entre desenvolvimento e testes automatizados representou grande impacto na implementação e preocupação na qualidade do código produzido. Toda nova implementação realizada era validada através da automação de novos testes e a reutilização de testes que já tinham sido desenvolvidos, garantindo que essa nova versão ou funcionalidade não quebraria o sistema.

### 4.3.1 Definição dos objetivos dos testes

Inicialmente, foi conduzida uma análise das ferramentas que iriam compor a arquitetura do projeto: jAudio e ACE. Como elas iriam se integrar dentro da aplicação. Essa análise foi feita a partir de discussões com o desenvolvedor para entender como esses pacotes de softwares funcionavam e de como iriam ser utilizados. Essas conversas serviram muitas vezes como análise da própria arquitetura, antes da implementação e do levantamento dos testes que iriam ser automatizados. Foi possível também identificar que os testes de unidade referentes ao jAudio e ACE não fariam parte do escopo deste trabalho, por serem frameworks já testados e que o principal desafio para automação proposta, seria então desenvolver testes automatizados que validassem a integração entres essas ferramentas e o sistema protótipo proposto. Logo, os objetivos da automação dos testes dessa aplicação foram a integração das ferramentas (jAudio e ACE), o funcionamento delas separadamente, mas não testar os algoritmos utilizados por ambas e sim o que era esperado como resposta da execução dessa integração.

### 4.3.2 Implementação dos testes unitários e integração

Depois da análise feita para identificação dos objetivos da automação e implementação do sistema, foram desenvolvidas duas classes: ACECommunicator e JAudioCommunicator. Estas por sua vez, são as responsáveis por executar as ferramentas ACE e jAudio dentro da aplicação. A seguir, serão listados os tipos de automação de testes desenvolvidos para cobertura de níveis unitários e de integração. As classes responsáveis por este nível de testes automatizados encontram-se dentro do pacote communicator que pertence ao pacote de testes do projeto.

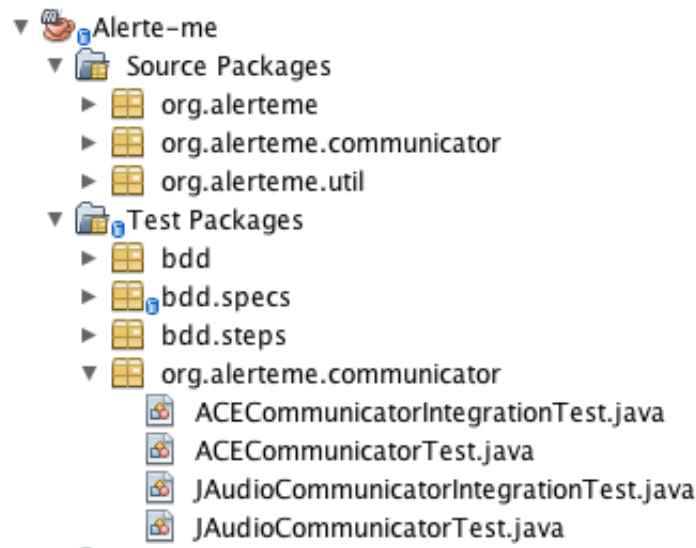


Figura 4.3: Arquitetura dos pacotes de testes automatizados de unidade e integração, dentro do pacote de testes Communicator.

- JAudioCommunicator

Essa classe possui a responsabilidade de extração de características do áudio e como resposta, a criação de um arquivo **output.xml** que irá conter todas as informações necessárias deste áudio para serem usadas na classificação. Ela contém um método chamado **extractFeatures** que recebe como parâmetro: arquivo de áudio a ser classificado, arquivo **settings.xml** que possui as configurações que o algoritmo de extração do jAudio necessita e nome do arquivo de output que será preenchido com as características que a extração realizará. Para que a extração das características e o consequente uso eficaz do jAudio funcione este método precisa validar todos estes argumentos.

Nestas tabelas, são apresentados os testes desenvolvidos para validação como unidade do `jAudio` e estão presentes na classe `JAudioCommunicatorTeste`.

<b>Teste Unitário - Construtor da classe JAudioCommunicator</b>	
<b>Argumento</b>	<b>Método de Teste</b>
Arquivo .jar do <code>jAudio</code>	<code>shouldThrowErrorIfJAudioJarFileIsNull</code>

Tabela 4.1: Teste para validação do construtor da classe `JAudioCommunicator`

O teste da Tabela 4.2, garante que caso o sistema não encontre o arquivo ".jar" do `jAudio`, a aplicação retornará uma mensagem de erro.

<b>Teste Unitário - Método extractFeatures</b>	
<b>Argumento</b>	<b>Método de Teste</b>
Arquivo de áudio	<code>shouldThrowErrorIfAudioFileIsNull</code>
Arquivo de configuração ( <code>settings.xml</code> )	<code>shouldThrowErrorIfSettingsFileIsNull</code>
Nome do Arquivo de saída ( <code>output</code> )	<code>shouldThrowErrorIfOutputFileIsNull</code>

Tabela 4.2: Implementação dos testes unitários para classe `JAudioCommunicator`

Esses testes da Tabela 4.2 validam que o arquivo de áudio, arquivo de configuração e o nome do arquivo de saída não são nulos. Todos os testes garantem que caso um destes arquivos não sejam encontrados ou estejam nulos a aplicação retornará uma mensagem facilitando a vida do usuário para entender qual o erro aconteceu e onde aconteceu.

Como resultado da extração das características do áudio, temos a criação de dois arquivos de output, o `outputFeatureVector` e o `outputFeatureKey`. Alguns testes de integração para validação da criação destes arquivos de output foram desenvolvidos, como é mostrado na Tabela 4.3

<b>Testes de Integração - JAudioCommunicator</b>	
<b>Método</b>	<b>Descrição</b>
shouldCreateOutputFiles	Acerta que os dois arquivos de saída foram criados com sucesso.
shouldNotCreateOutputFilesWhenTheAudioFileHasInvalidPath	Faz a validação que os arquivos não devem ser criados porque o áudio passado possui um caminho inválido.
shouldNotCreateOutputFilesWhenTheJAudioJarFileHasInvalidPath	Faz a validação que não foram criados arquivos de output, porque o arquivo .jar do jAudio possui um caminho inválido.

Tabela 4.3: Implementação dos testes de integração para classe JAudioCommunicator

Estes testes estão em uma classe chamada JAudioCommunicatorIntegrationTest, pois necessitam de arquivos que servem como entrada para realizar o teste e validar a integração do jAudio com a aplicação.

- ACECommunicator

Esta classe é responsável em executar o ACE e integra-lo com o sistema, é esta classe que possui um método chamado classify, responsável pela realização de fato da classificação do arquivo de áudio que foi passado para o jAudio. Para que a realização da classificação ocorra, este método recebe alguns parâmetros. Os parâmetros são: taxonomyFile, featureKeyFile, featureVectorFile, trainedMachineFile. Na tabela 4.4 a seguir mostra uma tabela com os métodos que cobrem a validação destes parâmetros e do arquivo .jar que é passado como argumento do construtor da classe.

<b>Teste Unitário - Construtor da classe ACECommunicator</b>	
<b>Argumento</b>	<b>Teste</b>
Arquivo .jar do ACE	shouldThrowErrorIfAceJarFileIsNull

Tabela 4.4: Teste para validação do construtor da classe ACECommunicator

Este método de teste garante que caso a classe ACECommunicator não encontre o “.jar” será lançada uma exceção para notificar que o arquivo não foi encontrado.

A seguir são apresentados os testes unitários desenvolvidos para validar o ACECommunicator:

<b>Testes Unitários – Método Classify</b>	
<b>Método</b>	<b>Descrição</b>
shouldThrowErrorIfTaxonomyFileIsNull	É esperado deste método que caso não seja passado um taxonomyFile como argumento, válido seja lançada uma exceção.
shouldThrowErrorIfFeatureKeyFileIsNull	É esperado deste método que caso não seja passado um featureKeyFile como argumento válido, seja lançada uma exceção.
shouldThrowErrorIfFeatureVectorFileIsNull	É esperado deste método que caso não seja passado um featureVectorFile válido seja lançada uma exceção.
shouldThrowErrorIfTrainedMachineFileIsNull	É esperado deste método que caso não seja passado um trainedMachineFile como argumento válido, seja lançada uma exceção.

Tabela 4.5: Implementação dos testes unitários para classe ACECommunicator

Assim como o JAudioCommunicator, esses testes garantem que caso um arquivos não sejam encontrados ou estejam nulos a aplicação lançará uma mensagem de erro (exceções) facilitando a vida do usuário e de todo time, no entendimento de qual erro aconteceu e onde aconteceu.

Como resultado da classificação, o ACECommunicator retornará uma lista contendo, o caminho do arquivo de áudio passado e com a resposta da classificação. Para validação desta lista foram criados vários métodos que testam a integração do ACE com a aplicação, testando cenários, por exemplo, com entradas de arquivos válidos e inválidos. Segue a Tabela 4.6 dos métodos de testes de integração desenvolvidos para o ACECommunicator:

Testes de Integração - ACECommunicator	
Método	Descrição
shouldReturnErrorMessageWhenFeatureVectorFileHasErrors	Este método recebe um arquivo de output.xml inválido como parâmetro no classify, é esperado que seja lançada uma mensagem de erro e a classificação não aconteça.
shouldClassifySireneFeatureVectorFileAsSirene	Este método recebe um áudio de sirene como parâmetro no classify, é esperado que seja realizada a classificação com sucesso e seja retornada a seguinte mensagem: "CLASSIFICATION: Sirene"
shouldClassifyBuzinaFeatureVectorFileAudioAsBuzina	Este método recebe um áudio de buzina como parâmetro no classify, é esperado que seja realizada a classificação com sucesso e seja retornada a seguinte mensagem: "CLASSIFICATION: Buzina"
shouldReturnErrorMessageWhenTaxonomyFileIsInvalid	Este método recebe um arquivo de taxonomyFile.xml inválido como um dos parâmetros no classify, é esperado que seja lançada uma mensagem de erro e a classificação não aconteça.
shouldReturnErrorMessageWhenFeatureKeyFileIsInvalid	Este método recebe um arquivo de featureKeyFile.xml inválido como um dos parâmetros no classify, é esperado que seja lançada uma mensagem de erro e a classificação não aconteça.
shouldReturnErrorMessageWhenFeatureVectorFileIsInvalid	Este método recebe um arquivo de featureVectorFile.xml inválido como um dos parâmetros no classify, é esperado que seja lançada uma mensagem de erro e a classificação não aconteça.
shouldReturnErrorMessageWhenTrainedMachineFileIsInvalid	Este método recebe um arquivo de trainedMachineFile.xml inválido como um dos parâmetros no classify, é esperado que seja lançada uma mensagem de erro e a classificação não aconteça.

Tabela 4.6: Implementação dos testes de integração para classe ACECommunicator

Os testes de integração da aplicação estão em uma classe chamada ACECommunicatorIntegrationTest, dentro do mesmo pacote estão vários arquivos que foram utilizados como entradas dos cenários de testes.

Um dos métodos, responsável por testar o **classify**, recebe como parâmetro um arquivo **invalidOutputFV.xml** (featureVectorFile) inválido para executar o método **classify** e depois tentou realizar a classificação, mas é esperado que seja gerada uma mensagem de erro para que o time e/ou usuário entenda porque a classificação não foi realizada com sucesso e qual foi o motivo do insucesso.

```
@Test
public void shouldReturnErrorMessageWhenFeatureVectorFileHasErrors() throws IOException, InterruptedException {
    File featureVectorFileError = resourceManager.getFile("invalid_output_FV.xml");

    List<String> response = aceCommunicator.classify(
        taxonomyFile,
        featureKeyFile,
        featureVectorFileError,
        trainedMachineFile
    );

    String errorMessage = "ERROR: The " + featureVectorFileError.getAbsolutePath() + " file is not a valid XML file.";
    assertThat(response.get(0), is(errorMessage));
    assertThat(response.get(1), is(""));
}
}
```

Figura 4.4: Implementação de teste que valida o arquivo featureVectorFile.

Outro método implementado, utiliza um arquivo xml vindo de um áudio de buzina válido, que foi previamente extraídas suas características com o método extractFeatures pertencente ao JAudioCommunicator, é passado como parâmetro para que o método classify realize a classificação como esperado mostrado a seguir.

```
@Test
public void shouldClassifySireneFeatureVectorFileAsSirene() throws IOException, InterruptedException {
    File sireneFeatureVectorFile = resourceManager.getFile("sirene_FV.xml");

    List<String> response = aceCommunicator.classify(
        taxonomyFile,
        featureKeyFile,
        sireneFeatureVectorFile,
        trainedMachineFile
    );

    assertThat(response.get(3), containsString("CLASSIFICATION: Sirene"));
}
}
```

Figura 4.5: Implementação de teste que valida a classificação de um áudio como sirene.

### 4.3.3 Implementando BDD com Cucumber

O sistema inicial resultante do desenvolvimento do projeto que este trabalho automatizou os testes não possui interface, mas a ideia protótipo desta interface para smartphone é que seja uma tela simples, com apenas um botão para inserir um áudio e um botão para realizar a classificação, como na ilustração da Figura 4.2 seguir, que será desenvolvida em trabalhos futuros.

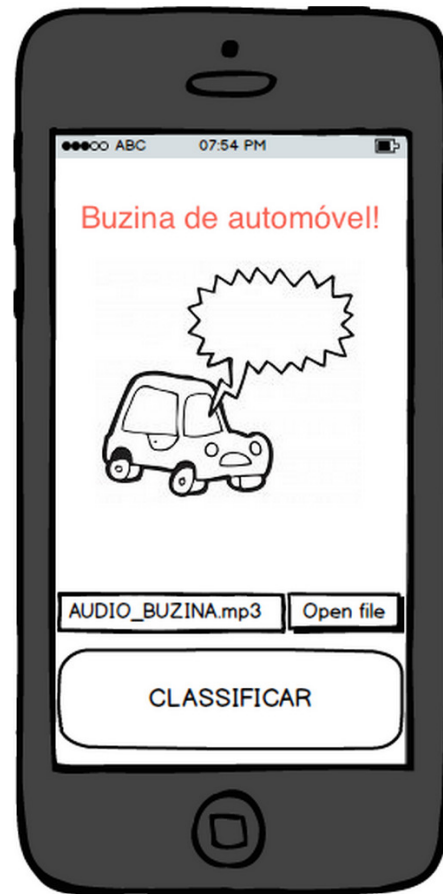


Figura 4.6: Protótipo da interface do sistema

No início da ideação deste projeto uma estória do usuário geral foi criada com intuito de validar a ideia gerada resultante do *Brainstorm*, como mostrado na Figura 4.13

**As** user  
**Want** to choose an audio file  
**To** be classified

Figura 4.7: Estória do usuário desenvolvida pelo sistema



Levando em consideração que o BDD nos permite a análise do comportamento que é esperado do software através da escrita de histórias do usuário com uma linguagem simples e que todos os envolvidos podem entender. Fornecendo cenários para validação do comportamento final, os cenários devem ser criados com base na história para que validem diversos comportamentos dos critérios de aceitação para o cliente ou usuário. Para que a implantação dos testes utilizando BDD com Cucumber fosse possível, dentro do projeto foi criada a arquitetura que o Cucumber entende de pacotes.

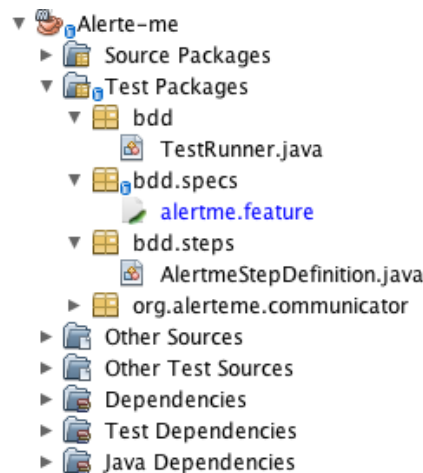


Figura 4.8: Arquitetura dos pacotes responsáveis pela implementação dos testes de BDD com Cucumber.

Dentro do pacote `bdd`, temos outros dois pacotes o **specs** e os **steps**. O pacote `specs` possui os arquivos **.features** e o **steps** possui os arquivos que implementam os passos que os cenários descrevem. Ainda na raiz do pacote BDD possuímos a classe java chamada `TestRunner.java` que é responsável por executar os testes Cucumber.

Foram desenvolvidos três cenários para validar a história desta aplicação. Como teste de classificação foram usados dois áudios um de buzina e outro de sirene, que serviram para confirmar que a aplicação está realizando a classificação da forma correta. Outro cenário foi a validação de que o sistema reconhece um arquivo passado por parâmetro caso esse seja inválido. O formato de como estes cenários foram escritos pode ser visto na Figura 4.15.

```
@RunWith
┌ Feature: Audio Classification
│   As user
│   Want to choose an audio file
│   To be classified
└─┘

┌ Scenario: Classification of the audio sample as Buzina
│   Given I have a audio file sample "buzina.wav"
│   When the application extract the features
│   And the classification function is called
│   Then I should see the classification as "Buzina"
└─┘

┌ Scenario: Classification of the audio sample as Buzina
│   Given I have a audio file sample "sirene.wav"
│   When the application extract the features
│   And the classification function is called
│   Then I should see the classification as "Sirene"
└─┘

┌ Scenario: Classification of the invalid audio sample
│   Given I have a audio file sample "invalid_audio.wav"
│   When the application extract the features
│   And the classification function is called
│   Then I should see the classification as "ERROR"
└─┘
```

Figura 4.9: Cenários de testes automatizados utilizando Cucumber

Para que o Cucumber entenda estes cenários, os steps (passos) que eles possuem devem ser implementados e chamam os métodos responsáveis para que a pré-condição exista e seja validada. Na Figura 4.16 é possível ver a implementação de um dos cenários que foram escritos como visto na Figura 4.15.

```
@Given("^I have a audio file sample \"([^\"]*)\"$")
public void I_have_a_audio_file_sample_audioName(String audioName) throws Throwable {
    audioFile = chooseAudioFile(audioName);
}

@When("^the application extract the features$")
public void the_application_extract_the_features() throws Throwable {
    jAudioCommunicator.extractFeatures(
        audioFile,
        settingsFile,
        outputFile);
}

@And("^the classification function is called$")
public void the_classification_function_is_called() throws Throwable {
    response = aceCommunicator.classify(
        taxonomyFile,
        featureKeyFile,
        featureVectorFile,
        trainedMachineFile
    );
}

@Then("^I should see the classification as \"([^\"]*)\"$")
public void I_should_see_the_classification_as_audioFile(String audioType) throws Throwable {
    assertThat(response.get(classificationMessagePosition), containsString(audioType));
}
```

Figura 4.10: Implementação dos steps de um cenário de teste com  
Cucumber

Na Figura 4.17 é mostrado o resultado ao executarmos os cenários de testes, onde são validados e não são encontrados erros:

```
Running bdd.TestRunner
@RunWith
Feature: Audio Classification
  As user
  Want to choose an audio file
  To be classified

  Scenario: Classification of the audio sample as Buzina
    Given I have a audio file sample "buzina.wav"
    When the application extract the features
    And the classification function is called
    Then I should see the classification as "Buzina"
  (String)

  Scenario: Classification of the audio sample as Buzina
    Given I have a audio file sample "sirene.wav"
    When the application extract the features
    And the classification function is called
    Then I should see the classification as "Sirene"
  (String)

  Scenario: Classification of the invalid audio sample
    Given I have a audio file sample "invalid_audio.wav"
    When the application extract the features
    And the classification function is called
    Then I should see the classification as "ERROR"
  (String)

3 Scenarios (3 passed)
12 Steps (12 passed)
0m3.002s
```

Figura 4.11: Resposta da execução dos cenários de testes que foram implementados com Cucumber

Esse resultado é gerado através da execução sem erros de todos os cenários que foram escritos para testarem os testes de aceitação da aplicação.

A Figura 4.18 a seguir é a demonstração de quando um dos cenários não passa ou acontece algum erro na implementação:

```
Scenario: Classification of the audio sample as Buzina # alertme.feature:7
  Given I have a audio file sample "buzina.wav" # AlertmeStepDefinition.I_have_a_audio_file_sample_audioName(C
  When the application extract the features # AlertmeStepDefinition.the_application_extract_the_featuresC
  And the classification function is called # AlertmeStepDefinition.the_classification_function_is_called
  Then I should see the classification as "Sirene" # AlertmeStepDefinition.I_should_see_the_classification_as_aud
(String)
  java.lang.AssertionError:
  Expected: a string containing "Sirene"
  but: was " CLASSIFICATION: Buzina"
  at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20)
  at org.junit.Assert.assertThat(Assert.java:956)
  at org.junit.Assert.assertThat(Assert.java:923)
  at bdd.steps.AlertmeStepDefinition.I_should_see_the_classification_as_audioFile(AlertmeStepDefinition.java:62)
  at *.Then I should see the classification as "Sirene"(alertme.feature:11)
```

Figura 4.12: Resposta da execução de um cenário de testes que possui um step com falha

Este cenário falhou porque era esperado que o resultado da classificação fosse "Sirene" mas a classificação retornou "Buzina". Também como output da execução dos testes automatizados com Cucumber podemos ver o tempo que a execução durou, a quantidade de cenários e testes que falharam ou passaram, com visto na Figura 4.19.

```
3 Scenarios (1 failed, 2 passed)
12 Steps (1 failed, 11 passed)
0m3.163s
```

Figura 4.13: Quantidade de cenários de testes e de steps pertencentes à feature executada no Cucumber

## 4.4 Resultado da automação dos testes

Conforme dito previamente este trabalho teve o objetivo de selecionar, implementar e analisar automação de testes durante o desenvolvimento de uma aplicação, para que todo os níveis possíveis da pirâmide de testes [9] fossem contemplados com testes automatizados. Como observado na análise dos objetivos da automação, boa parte dos testes foram realizados para validar a integração dos pacotes de software ACE e jAudio com o sistema. O que justifica tomarmos como base para este trabalho a pirâmide, conforme a Figura 4.20.

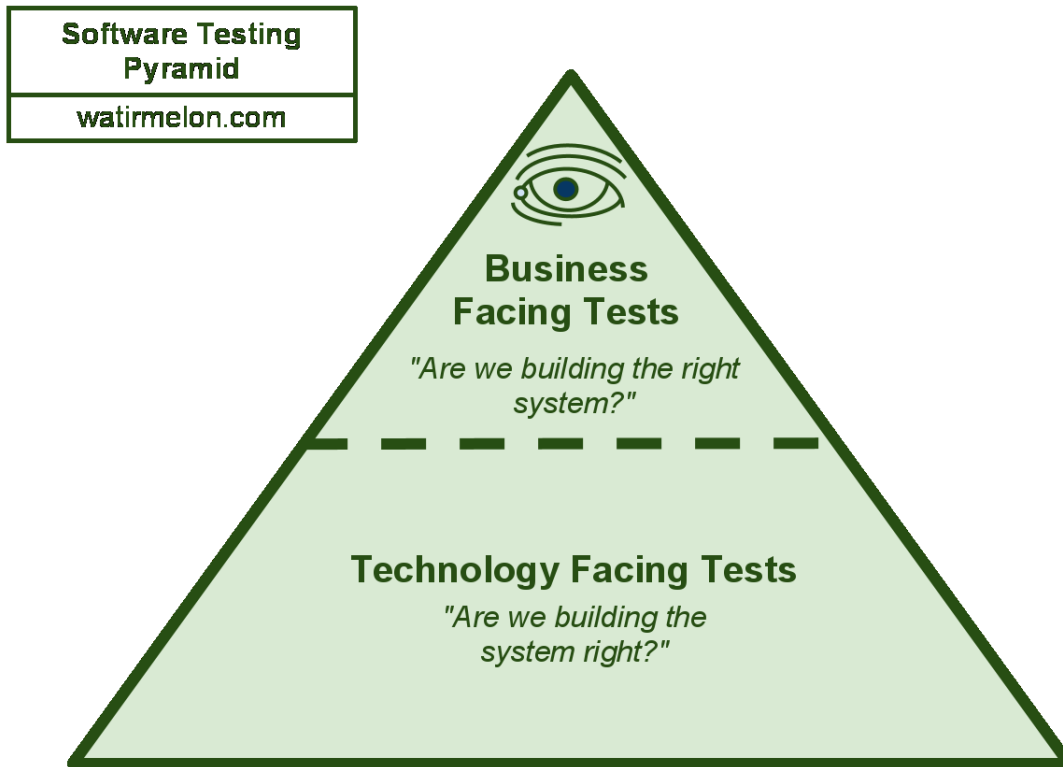


Figura 4.14: Pirâmide de testes dividida nas faces de negócio e tecnologia.

Levando em consideração a análise dos objetivos da automação, é visto que esta é a estratégia para aplicação dos testes que melhor se aplica aos resultados que foram coletados ao final deste trabalho.

Para validação da pergunta no topo da pirâmide: “Are we building the right system?” (estamos construindo o sistema certo?), foi realizada a validação com base nos testes de aceitação utilizando BDD com a ferramenta Cucumber, onde foram desenvolvidos 3 cenários e 12 steps (passos). Apenas os três cenários foram suficiente para validar a funcionalidade principal do sistema, que era a classificação de áudio. Este tipo de validação nos ajuda a encontrar divergências entre os principais objetivos do sistema e o seu resultado final esperado, garantindo a verificação do comportamento da aplicação.

Conforme apresentado na figura, todos os testes de unidade e de integração se encontram na base da pirâmide, respondendo a pergunta: “Are we building the system right?” (Estamos construindo o sistema de forma correta?). A aplicação dos testes automatizados, realizados

neste trabalho, resultou em um total de 37 testes, onde 34 destes foram de integração mais unidade e 3 cenários (12 steps) de testes que validam aceitação. É relevante ressaltar que as ferramentas utilizadas ACE e jAudio já possuem uma gama de testes que não fazem parte do escopo desta proposta, mas que ressaltam a importância dos testes que foram realizados serem de integração. Com isso é factível assumir que o desenvolvimento foi coberto por atividades de testes automatizados desde a sua concepção até o final do desenvolvimento. Com praticamente todas as classes e integrações testadas automaticamente ao realizar alguma mudança no projeto e ao gerar um novo build.

A Tabela 4.7 mostra os dados objetivos relevantes para o desenvolvimento deste trabalho.

<b>Problema</b>	Classificação de Áudio
<b>Equipe</b>	1 DEV e 1 QA (os dois com perfil graduando)
<b>Tecnologia DEV</b>	Java, jMIR(ACE+jAudio)
<b>Tecnologia Testes</b>	jUnit, Cucumber
<b>Tempo de desenvolvimento</b>	4 meses

Tabela 4.7: Dados objetivos referentes ao desenvolvimento deste trabalho

Para realizar o trabalho juntamente com o desenvolvedor foi utilizado o git para controlar a versão e manter o código do sistema sempre atualizado. Depois do desenvolvimento do sistema, foi feito um levantamento número de commits e quais teriam sido seu papel no projeto, o resultado disto pode ser visto na Tabela 4.8.

<b>commits</b>	Quantidade
<b>total do projeto</b>	64
<b>refactoring (melhoria e implementação de testes)</b>	31

Tabela 4.8: Quantidade e tipo de commits realizados dentro do projeto

Com base no número de linhas das classes que pertencem ao sistema foi feita uma porcentagem de testes que estavam sendo cobertos. Foi levantado o número de linhas de uma classe, o número total de testes desenvolvidos e quantos testes validavam estas linhas ou classe, o resultado destes dados pode ser visto na Tabela 4.9.

Classe	N de linhas	% Cobertura dos testes
AlertMe	48	67.54
JAudioCommunicator	35	27
ACECommunicator	54	40.54

Tabela 4.9: Porcentagem da cobertura dos testes que foram desenvolvidos

A Figura 4.21 a seguir mostra a representação da pirâmide resultante deste projeto.

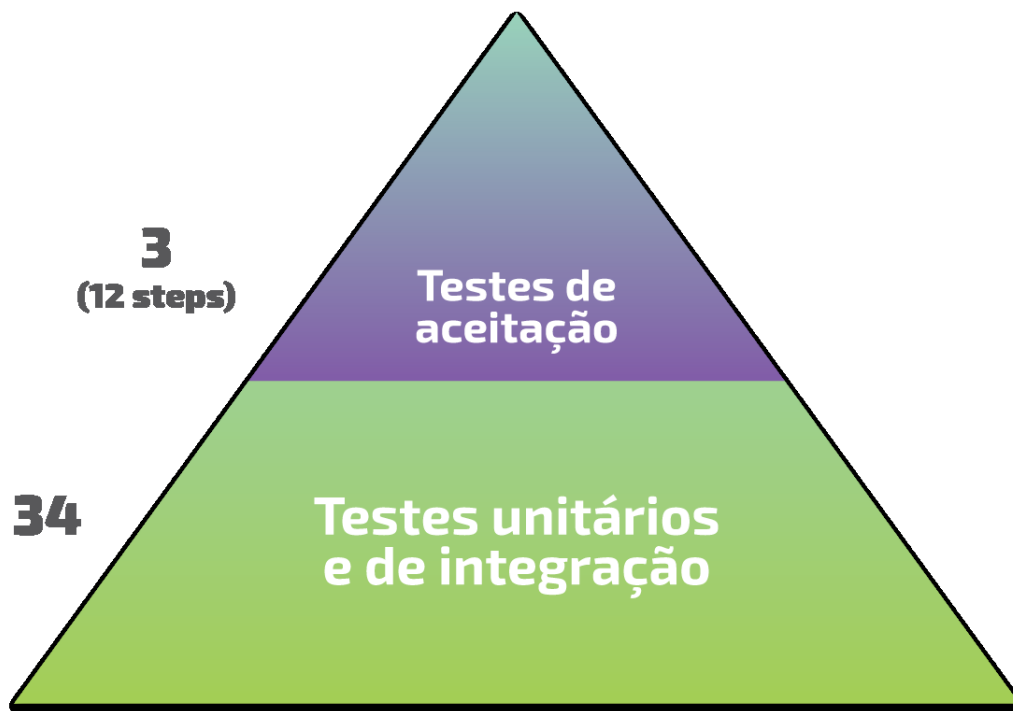


Figura 4.15: Pirâmide com as quantidades de testes desenvolvidos neste trabalho



# Capítulo 5

## Conclusão

### 5.1 Conclusão do Trabalho

Este trabalho teve como objetivo a implantação e cobertura de testes automatizados durante o desenvolvimento de uma aplicação que foi desenvolvida com tecnologia de classificação de sinais de áudio para dar suporte a pessoas com deficiência auditiva ou surdas.

Foi possível observar que a implementação da automação dos testes desde do começo do desenvolvimento e em conjunto com o desenvolvedor é uma estratégia possível para realizar o desenvolvimento de testes automatizados. Ainda em tempo de desenvolvimento das funcionalidades foi possível corrigir os erros que eram encontrados durante a implantação da automação dos testes.

Sendo assim, aplicação de testes automatizados em todos os níveis de testes, como a pirâmide ideal de testes indica, foi empregado através de técnicas e ferramentas que deram o suporte necessário desta estratégia de implementação. Um dos benefícios percebidos foram os ganhos de tempo na execução dos testes automatizados, visto que o tempo para desenvolvimento do sistema e dos testes automatizados era curto e não foram realizados praticamente testes manuais e a frequência com eles podem ser executados, uma vez que eles foram desenvolvidos, podem ser reproduzidos em vários ambientes, caso o sistema no futuro seja implantado em diferentes plataformas e ao longo de alterações para validar novas versões a um baixo custo.

Tal característica ressalta o potencial dos testes automatizados para testes de regressão.

Por fim, é perceptível que a implantação de testes automatizados não é um processo fácil, que demanda uma mudança, tanto de como o processo de testes são executados quanto na capacitação técnica dos responsáveis pela automatização dos testes do time, mas que sua utilização sistemática pode ser eficaz e trazer benefícios reais a um projeto de desenvolvimento de software.

## 5.2 Trabalhos futuros

Em conjunto com o desenvolvimento, chegou se a conclusão que é factível dar continuidade ao desenvolvimento e aprimoramento da aplicação juntamente com a automação dos testes. Realizando melhorias nos próximos passos do desenvolvimento e aprimorando as técnicas e aderindo novas tecnologias para validar o quanto antes a aplicação. Ainda também garantir testes a nível final com o usuário e de interface, coletando novos dados para continuar aprofundando estudos deste tipo de estratégia e aplicando diferentes tipos de testes automatizados.

# Referências Bibliográficas

- [1] Ace. Disponível em: <http://jmir.sourceforge.net/ACE.html>. Acessado em 30/12/2014.
- [2] Forrester research. Disponível em: <https://www.forrester.com/Market+Overview+2008+Automated+Testing+Software/fulltext/-/E-RES47064>. Acessado em 01/01/2015.
- [3] jmir. Disponível em: <http://jmir.sourceforge.net/overview.html>. Acessado em 30/12/2014.
- [4] T. Wissink; C. Amaro. Strategies for agile software testing automation: An industrial experience. In *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*, pages 265–266, September 2006.
- [5] IEEE Standards Board. Ieee guide for software verification and validation plans. Technical report, The Institute of Electrical and Electronics Engineers, Inc, December 1993.
- [6] FANTINATO M.; CUNHA A.; DIAS S.; MIZUNO S.; E CUNHA C. Um framework reutilizável para a automação de teste funcional de software. In *SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE*, 2004.
- [7] E. Collins and Lucena V. F. Dias-Neto. Strategies for agile software testing automation: An industrial experience. In *Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual*, pages 440–445, July 2012.

- 
- [8] James Crisp. Automated testing and the test pyramid, May 2011. Disponível em: <http://jamescrisp.org/2011/05/30/automated-testing-and-the-test-pyramid/>. Acessado em: 30/11/2014.
- [9] James Crisp. Yet another software testing pyramid, June 2011. Disponível em: <http://watirmelon.com/2011/06/10/yet-another-software-testing-pyramid/>. Acessado em: 30/11/2014.
- [10] cukes.info. Cucumber. Disponível em: <http://cukes.info/>. Acessado em 31/12/2014.
- [11] L. M CYSNEIROS. Integrando requisitos não funcionais ao processo de desenvolvimento de software. Master's thesis, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 1997.
- [12] Jornal Dia Dia. Barulho, inimigo da boa audicao, 2014. Disponível em: <http://www.jornaldiadia.com.br/news/noticia.php?Id=49258#.VKG078AAA/>. Acessado em 03/01/2015.
- [13] D FEWSTER, M e GRAHAM. *Software Test Automation - Effective use of test execution tools*. Addison-Wesley, first edition, 1999.
- [14] Gherkin. Gherkin. Disponível em: <http://cukes.info/gherkin.html>. Acessado em 02/01/2015.
- [15] R. Hooda. Automation of software testing: A foundation for the future. *INTERNATIONAL JOURNAL OF LATEST RESEARCH IN SCIENCE AND TECHNOLOGY*, 1:152–154, 2012.
- [16] JBehave.org. Jbehave. Disponível em: <http://jbehave.org/>. Acessado em 31/12/2014.
- [17] junit.org. Junit. Disponível em: <http://junit.org/>. Acessado em 31/12/2014.
- [18] A. M. KIRNER, T. G.; DAVIS. Nonfunctional requirements of real-time systems. *Advances in Computers*, 42:1–37, 1996.
- [19] T. Lima. Usando o silktest para automatizar testes: um relato de experiencia. In *6TH BRAZILIAN WORKSHOP ON SYSTEMATIC AND AUTOMATED SOFTWARE TESTING*, 2012.

- [20] Tutorials Point India Private Limited. Software testing-overview. Disponível em: [http://www.tutorialspoint.com/software\\_testing/software\\_testing\\_overview.htm](http://www.tutorialspoint.com/software_testing/software_testing_overview.htm). Acessado em 30/12/2014.
- [21] BUDNIK C. J.; CHAN W. K.; KAPFHAMMER G. M. Bridging the gap between the theory and practice of software test automation. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on (Volume:2)*, pages 445–446, May 2010.
- [22] J. C. Maldonado and A. Vincenzi. Aspectos teóricos e empíricos de testes de cobertura de software. Technical report, Instituto de Ciências Matemáticas e de Computação - ICMC-USP, June 1998.
- [23] Wanessa Mariana and Angelica Toffano. Ferramentas free para teste de software: um estudo comparativo. Technical report, Centro Universitario de Brasilia-Uniceub, August 2012.
- [24] Nuno Trigueiros-Cunha Marie Camilleri. Ruido: Atencao perigo ! proteccao. Disponível em: <http://www.cochlea.org/po/ruido>. Acessado em 03/01/2015.
- [25] Bruno Molinari. Implementacao de ambiente de testes automatizados para avaliacao de parametros de qos em redes de comunicacao de dados baseado em software livre. Technical report, UNIVERSIDADE SAO FRANCISCO-Engenharia de Computacao, 2002.
- [26] D. North. Introducing bdd. Disponível em: <http://dannorth.net/introducing-bdd/>. Acessado em 31/12/2014.
- [27] Fabio Kon Paulo Cheque. A importancia dos testes automatizados. *Engenharia de Software Magazine*, 72:54–57, 2008.
- [28] Bret Pettichord. Seven steps to test automation success. STAR West, San Jose, November 1999. Disponível em: [http://https://www.prismnet.com/~wazmo/papers/seven\\_steps](http://https://www.prismnet.com/~wazmo/papers/seven_steps). Acessado em: 30/11/2014.
- [29] O portal de informacao para pessoa com deficiencia. Resultados do censo 2010 feito pelo ibge sobre pessoas com deficiência. Disponível em: <http://www.deficientefisico.com/>

- resultados-do-censo-2010-feito-pelo-ibge-sobre-pessoas-com-deficiencia/. Acessado em 02/01/2015.
- [30] Elfriede Dustin; J. Rashka; and J. Paul. *The Art of Agile Development*. Addison-Wesley, first edition, 1999.
- [31] Pressman; R.S. *Software engineering: A practitioners approach*. Technical report, McGraw Hill, Inc, 2002.
- [32] rspec.info. Rspec. Disponível em: <http://rspec.info/>. Acessado em 31/12/2014.
- [33] Alister Scott. Introducing the software testing ice-cream cone (anti-pattern), January 2012. Disponível em: <http://watirmelon.com/2012/01/31/introducing-the-software-testing-ice-cream-cone/>. Acessado em: 25/11/2014.
- [34] RECIFE SOFTEX. Fundamentos do teste de software. Disponível em: [http://ava.nac.softex.br/pluginfile.php/602/mod\\_resource/content/2/Aula%202.pdf](http://ava.nac.softex.br/pluginfile.php/602/mod_resource/content/2/Aula%202.pdf). Acessado em 02/01/2015.
- [35] Xiaofeng Wang Solis C. An integration testing coverage tool for object-oriented software. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 383–387, Aug 2011.
- [36] Pachawan Augsornsri; Taratip Suwannasart. An integration testing coverage tool for object-oriented software. In *Information Science and Applications (ICISA), 2014 International Conference on*, pages 1–5, May 2014.
- [37] K.N. Leung Hareton; Lee White. A study of integration testing and software regression at the integration level. In *Software Maintenance, 1990, Proceedings., Conference on*, pages 290–301, November 1990.
- [38] Weinan Shanxi XiangFeng Meng. Analysis of software automation test protocol. In *International Conference on Electronic and Mechanical Engineering and Information Technology*, pages 4138 – 4141, August 2011.
- [39] K. Zambelich. Totally data-driven automated testing. Disponível em: [http://www.sqatest.com/w\\_paper1.html](http://www.sqatest.com/w_paper1.html). Acessado em 30/12/2014.