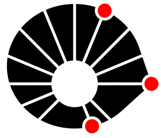

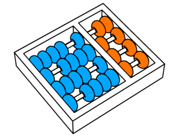

# Geração automática de testes combinando *Model Based Testing* e *Behavior Driven Development*

Relatório Técnico - IC-PFG-22-44

Projeto Final de Graduação

2022 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS

INSTITUTO DE COMPUTAÇÃO

The contents of this report are the sole responsibility of the authors.

O conteúdo deste relatório é de única responsabilidade dos autores.

# Geração automática de testes combinando *Model Based Testing* e *Behavior Driven Development*

Pedro Mota [*]　　Eliane Martins [†]

**Resumo**

Verificar o comportamento de sistema usando testes de software é um processo indispensável para o controle de qualidade. O *Behavior Driven Development* (BDD) é uma prática de desenvolvimento de software que gera testes como forma de documentação do comportamento do sistema, porém os testes são gerados manualmente. Já o *Model Based Testins* (MBT) é um método que permite gerar testes de maneira automática a partir de um modelo do comportamento do sistema. Este projeto se propõe a desenvolver um método que combine o uso de MBT e BDD para geração automática de testes, e que funcione em projetos de desenvolvimento incremental. Como sistema sob teste foi usado a funcionalidade de login com email e senha do Firebase. O método aplica o MBT para desenvolver o modelo de comportamento do sistema, e usa a ferramenta *Skyfire* para a partir do modelo de comportamento gerar testes seguindo a notação usada no BDD. O modelo de comportamento do sistema foi representado por um modelo de estados. Foram feitas duas iterações do método proposto, adicionando-se novos requisitos a cada iteração, a fim de avaliar o esforço associado. Ao final da execução sobre a funcionalidade testada, o método proposto foi capaz de gerar testes a partir de um modelo de estado, documentando o funcionamento do sistema de forma automática e incremental.

## 1 Introdução

### 1.1 Contexto

O tema desta tese é a combinação da prática de desenvolvimento *Behavior Driven Development* (BDD) com a técnica de geração de testes Model Based Testing (MBT). BDD é uma prática de desenvolvimento que procura representar o comportamento da aplicação por meio de cenários baseados em histórias de usuário. Estes cenários podem ser traduzidos em testes de aceitação de forma semiautomática, e auxiliar o time na verificação do comportamento. MBT é uma técnica que usa um modelo do comportamento do software para gerar casos de teste.

[*]Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

[†]Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

### 1.2 Motivação

Geração e execução de testes são etapas cruciais para garantir a qualidade de artefatos de software. Testes de software são capazes de identificar falhas no código, e aproximar o comportamento real do comportamento esperado. Porém, projetos de software estão sob constante mudança. É comum que requisitos novos sejam adicionados ao longo da vida de um projeto, ou que requisitos antigos sejam modificados. Desta forma, a rastreabilidade entre requisitos e testes se torna um desafio.

Nesse contexto, a prática do Behavior Driven Development (BDD) [1] se torna atrativa por usar uma estratégia de desenvolvimento guiada por Histórias de Usuário [2]. Este formato documenta a evolução de requisitos ao longo do tempo e, com o uso de ferramentas de apoio, permite que se gere cenários executáveis para testar os requisitos, além de expressá-los em uma linguagem não técnica e mais próxima de clientes de outras partes interessadas.

Porém, o BDD apresenta limitações à medida que a capacidade das Histórias de Usuário de expressarem os requisitos com clareza e completude, depende da experiência de quem as escreve. E como os cenários executáveis precisam ser escritos manualmente, o acúmulo de cenários e frequentes mudanças nos requisitos, dificulta sua execução e manutenção.

### 1.3 Objetivos

Este projeto final de graduação tem como objetivo desenvolver uma técnica de geração de testes que combine o uso de BDD com MBT. Com o uso do MBT espera-se obter a geração de testes de forma automática, facilidade de manutenção, cobertura mais completa do comportamento dos serviços testados em comparação a testes gerados manualmente e gerar diferentes conjuntos de teste de acordo com critérios de cobertura do modelo. E com o uso do BDD espera-se ter a especificação dos casos de teste em uma linguagem não técnica, entendível por pessoas que não sejam programadores ou testadores, mas que estejam envolvidas no projeto de alguma maneira

### 1.4 Metodologia

Para a realização deste projeto foi usada a metodologia de pesquisa em ação. Seu objetivo é a geração de conhecimento dentro da área da pesquisa, e é caracterizada por ser um método colaborativo entre orientador e aluno, onde ambos participam ativamente em um processo cíclico dividido em quatro etapas: diagnóstico, planejamento, execução e avaliação [6]. As atividades realizadas em cada etapa são descritas a seguir.

1. **Diagnóstico**: definir o tema e questões a serem respondidas com o projeto

2. **Planejamento**: definir as ações que serão realizadas a partir do contexto, propósito e problema principal idetificados no diagnóstico

3. **Execução**: executar as ações definidas durante a etapa de planejamento

4. **Avaliação**: examinar os resultados obtidos

# 2 Fundamentação teórica

## 2.1 *Behavior Driven Development* (BDD)

*Behavior Driven Development* (BDD) é uma prática de desenvolvimento de software, cujo objetivo é expressar os requisitos do sistema de forma clara para todas as partes interessadas. O ciclo do BDD é dividido em quatro etapas: discussão dos requisitos, formulação de exemplos, geração de testes automatizados e validação. Com o uso de Gherkin, notação adotada pelo BDD para expressar o comportamento de testes, é possível traduzir os exemplos formulados em casos de teste [2]. Em Gherkin, um caso de teste é formado pelas palavras chaves *Given*, *When*, *And* e *Then*. A Figura 1 mostra um caso e teste escrito em Gherkin. Palavras chaves definem um passo com uma função específica. Um passo *Given* define as condições iniciais do caso de teste; *When* indica uma ação ou evento e *Then* indica uma condição, ou resultado esperado. Ferramentas de apoio permitem implementar o código que deve se executado em cada passo, obtendo-se assim testes de aceitação executáveis. Por especificar os testes usando uma linguagem independente de linguagens de programação, o BDD auxilia no alinhamento da equipe dentro do projeto, na produção de código com menos defeitos e também na produção de testes como uma forma de documentar o comportamento esperado do sistema [1].

```
Feature: Learn Gherkin

    Scenario: Gherkin example

    Given a initial condition
    When action A is triggered
    Then condition B is expected to be true
```

Figura 1: Caso de teste escrito em Gherkin

## 2.2 *Model Based Testing* (MBT)

Model Based Testing é uma técnica que permite gerar testes automatizados a partir de um modelo do sistema [5]. O modelo é uma abstração das funcionalidades do sistema, e deve ser desenvolvido a partir dos requisitos levantados. Ferramentas de apoio permitem percorrer o modelo e gerar casos de testes de maneira automatizada. O uso de MBT facilita a geração e implementação dos testes [3]. O modelo desenvolvido torna o comportamento do sistema entendível para equipes não técnicas, fazendo com que defeitos possam ser identificadas rapidamente. E como os testes são gerados automaticamente, não é preciso reescrever os testes quando os requisitos mudam. O modelo é atualizado, e os testes são gerados novamente.

### 2.3 Testes baseados em modelos de estado

Um modelo de estado é uma das notações utilizadas para representar comportamento de um sistema. Em um modelo de estados o comportamento é expresso usando estados e transições. Um estado representa o contexto do sistema no momento. Em modelos de estado finitos há um estado inicial e um final, que indicam onde o modelo inicia e onde ele para. Transições levam o sistema de um estado a outro dependendo do valor de entrada que acionou a transição. O MBT pode representar o sistema sob teste por um modelo de estado, e usar ferramentas de apoio para gerar os casos de teste. É possível gerar conjuntos de teste diferentes dependendo do critério de cobertura escolhido, como percorrer todos os estados ou todas as transições. Expressar o comportamento por meio de modelos de estado facilita a compreensão do funcionamento do sistema, pois comunica forma visual todas as situações possíveis de contexto.

## 3 Diagnóstico

Diante da importância de testes no desenvolvimento de software, BDD se torna uma estratégia atrativa. O uso do BDD gera uma documentação dos requisitos por meio de casos de teste executáveis, usando uma linguagem acessível. Porém, a estratégia é limitada por não suportar geração automática dos casos de teste, o que dificulta a geração e manutenção da suite de testes a longo prazo.

A estratégia MBT propõe-se a resolver os desafios de geração automática e manutenção de casos de teste. Como os testes são gerados de forma automática a partir do modelo, a escrita de testes se torna mais fácil. Ferramentas de análise do modelo auxiliam o desenvolvedor na geração de testes. E por fim, como o modelo é uma abstração do comportamento do software, adaptá-lo às mudanças dos requisitos também se torna uma tarefa mais fácil.

O método proposto neste projeto busca combinar o uso de BDD e MBT, para gerar casos de teste em Gherkin automaticamente, a partir de um modelo de estados que descreva o comportamento do sistema sob teste.

## 4 Planejamento

### 4.1 Método proposto

O método proposto neste projeto combina técnicas de BDD com MBT, e é composto pelos passos listados abaixo. Os passos são ilustrados pela Figura 2, e serão detalhados na seção 5.

1. **Histórias de Usuário**: Escrever os requisitos na forma de histórias de usuário.

2. **Modelagem**: A partir das Histórias de Usuário, construir um modelo de estado que expresse o comportamento esperado.

3. **Validação do modelo**: Validar se o modelo desenvolvido está correto, e se é possível utilizá-lo.

4. **Geração de Cenários**: A partir do modelo validado, usar ferramentas de apoio a MBT para gerar casos de teste de forma automática.

5. **Completar Cenários**: Transformar os cenários gerados em instâncias reutilizáveis e adicionar parâmetros de entrada e saída esperada.

6. **Implementar testes de aceitação**: Implementar os testes de aceitação, traduzindo os passos descritos no cenário de teste em ações do sistema testado.

7. **Execução dos testes de aceitação**: Executar os casos de teste implementados.

8. **Validação dos testes de aceitação**: Observar o resultado, sucesso ou falha, da execução dos testes.

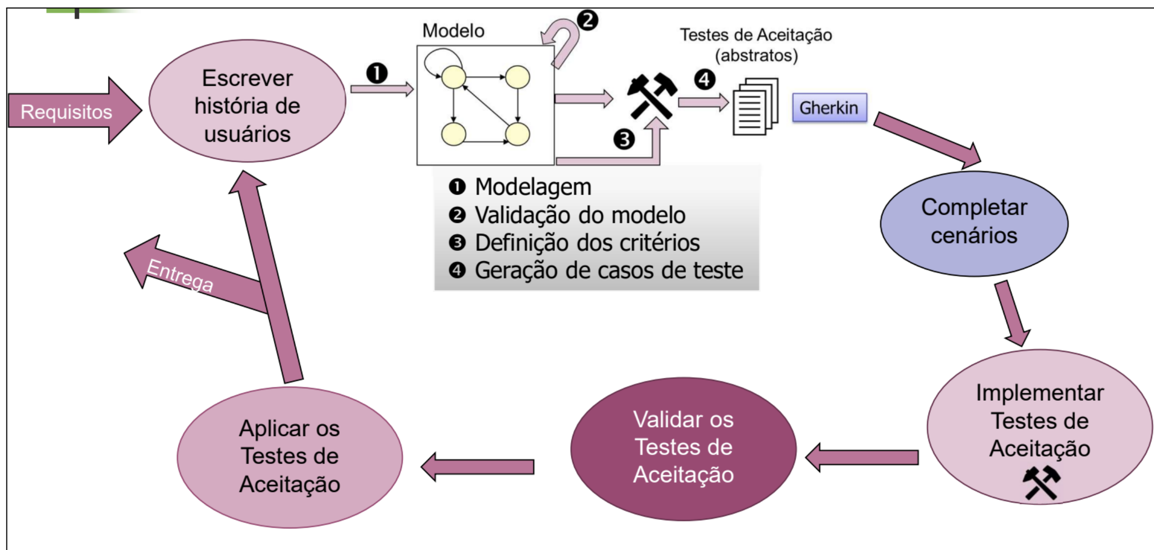


Figura 2: Método proposto para geração de testes

## 4.2 Ferramentas utilizadas

## 4.3 Papyrus

Para desenvolver os modelos do projeto, usou-se a *Papyrus.* A *Papyrus* faz parte do projeto *Model Development Tools* do Eclipse, e oferece ferramentas de modelagem e validação para modelos do tipo *Eclipse Modeling Framework* (EMF).

## 4.4 Cucumber

Para implementar os casos de teste utilizou-se Cucumber, um framework de suporte a BDD [7]. O Cucumber permite que se associe código executável aos passos de um cenário de teste escrito em Gherkin por meio de *Step Definitions* [8]. Essa funcionalidade permite definir

uma implementação em código do que deve ser executado ao chamar um passo escrito em Gherkin, produzindo casos de teste concretos.

### 4.5 *Skyfire*

Para geração de Cenários de teste em Gherkin foi utilizada a ferramenta Skyfire [9]. Seu objetivo é automatizar a geração de cenários de teste BDD escritos em Gherkin a partir de modelos de máquinas de estado. O modelo de entrada deve ser um modelo de estados UML baseados no Eclipse Modeling Framework (EMF), e conter, no mínimo, um estado inicial e um estado final. Este modelo é convertido em um grafo. Esse grafo é usado para gerar testes abstratos que, por fim, são convertidos em Cenários de teste escritos em Gherkin.

O Cenário de teste em Gherkin é gerado mapeando as transições que partem do estado inicial, e que começam com o prefixo *initialize*, em um passo *Given*. Invariantes de estado são mapeadas para passos *Then*, onde o nome da invariante é o nome do passo. Aqui vale ressaltar que a Skyfire identifica apenas invariantes de estado. Condições de guardas não são identificadas, mas é possível obter o mesmo resultado usando estruturas como o pseudo estado *choice*. As demais transições são mapeadas para passos *When* ou *And*.

A Skyfire oferece quatro critérios de cobertura para geração de cenários. A especificação de cada critério é detalhada abaixo.

1. **Node Coverage:** percorre por todos os estados do modelo

2. **Edge Coverge:** percorre todas as transições do modelo

3. **Edge-pair Coverage:** percorre todos os caminhos alcançáveis de no máximo duas transições

4. **Prime-path Coverage:** percorre todos os caminhos primos do modelo, que são caminhos simples que não são subcaminhos de nenhum outro caminho simples. Um caminho simples é aquele em que nenhum vértice é repetido, exceto o primeiro e o último.

## 5 Execução do método

### 5.1 Aplicação utilizada

Como System Under Test (SUT) deste projeto, foi usada a API Rest do serviço de login com email e senha, do Firebase. O Firebase é uma ferramenta de Backend as a Service (BaaS), desenvolvida e mantida pela Google [11]. Dentre os serviços oferecidos, está o de autenticação por email e senha de usuários já cadastrados no sistema. A Tabela 1 abaixo mostra os parâmetros que devem ser informados no corpo do request HTTPS para fazer autenticação por meio da interface REST disponibilizada. Para que se possa fazer uso do serviço, também é necessário informar na url do request a API Key fornecida pelo Firebase ao criar um projeto na plataforma. A resposta da API conta com várias informações sobre o resultado do request. Porém, para este projeto levou-se em consideração apenas o código e status de erro. A Tabela 2 abaixo descreve os códigos e status de erro observados.

| Parâmetro | Tipo | Descrição |
|---|---|---|
| email | string | Email da conta do usuário |
| password | string | Senha da conta do usuário |
| returnSecureToken | boolean | Indica se o retorno deve informar um ID e token de atualização. Deve sempre ser verdadeiro |

Tabela 1: Campos do corpo do request HTTPS do serviço de login

| Código | Mensagem | Descrição |
|---|---|---|
| 200 | | Autenticação feita com sucesso |
| 400 | EMAIL_NOT_FOUND | O email não foi encontrado na base de usuários |
| 400 | INVALID_EMAIL | O email não respeita as regras de formação |
| 400 | MISSING_EMAIL | Request feito com parâmetro de email vazio |
| 400 | INVALID_PASSWORD | A senha não corresponde a senha do usuário |
| 400 | MISSING_PASSWORD | Request feito com parâmetro de senha vazio |
| 400 | API key not valid.Please pass a valid API key | API Key inválida |
| 403 | The request is missing a valid API key | Request feito com API Key vazia |

Tabela 2: Possíveis mensagens e códigos de erro

## 5.2 Primeira iteração do método

### 5.2.1 Histórias de Usuário

Os requisitos da funcionalidade de login com email e senha foram escritos na forma de Histórias de Usuário. Utilizou-se o formato "Como uma persona, eu realizo uma ação, para obter um resultado". Este formato permite documentar os requisitos de forma não técnica, além de comunicar de forma clara a ação executada e o resultado esperado. Nesta primeira iteração do método, foi levantada uma versão limitada da funcionalidade de login com email e senha. Como o parâmetro *returnSecureToken*, mostrado na Tabela 1, é opcional, foram considerados apenas os parâmetros de email e senha. A Tabela 3 mostra a história final da primeira iteração.

| Como | Ação | Objetivo |
|---|---|---|
| Um usuário já registrado | Informo meu email e senha | Autenticar minha sessão |

Tabela 3: Requisitos de login em forma de Histórias de Usuário

### 5.2.2 Modelagem

O modelo de estado da Figura 3 expressa o comportamento descrito pela História de Usuário da Tabela 3. Neste modelo, a API é inicializada com valores vazios para email e senha, e o usuário realiza o login até que seja autenticado com sucesso. Aqui vale ressaltar algumas decisões de design que se repetirão para a segunda iteração do modelo. Primeiro, os nomes de eventos e dos estados refletem o domínio do problema. Esta decisão é importante para permitir que tanto o modelo quanto os cenários gerados a partir do modelo possam ser compreendidos por todas as partes interessadas. As condições de guarda seguem o mesmo princípio: dado que a Skyfire não lida com expressões lógicas nas transições, é dado um nome para a condição que representa o domínio do problema. Por exemplo, *InputValidCredentials* e *InputInvalidCredentials* , que diferenciam os casos de entradas com credenciais válidas e inválidas. Segundo, as invariantes de estado *LoginSuccess* e *LoginError* estão vinculadas aos estados *UserAuthenticated* e *UserNonAuthenticated*, respectivamente. Elas representam condições que devem ser cumpridas ao atingir esses estados. E por último, os símbolos de *warning* nas transições de estado *Login* são gerados automaticamente pela *Papyrus* quando duas transições têm o mesmo nome. Porém, manter a nomeação para transições que representam a mesma ação é importante para implementação dos testes.

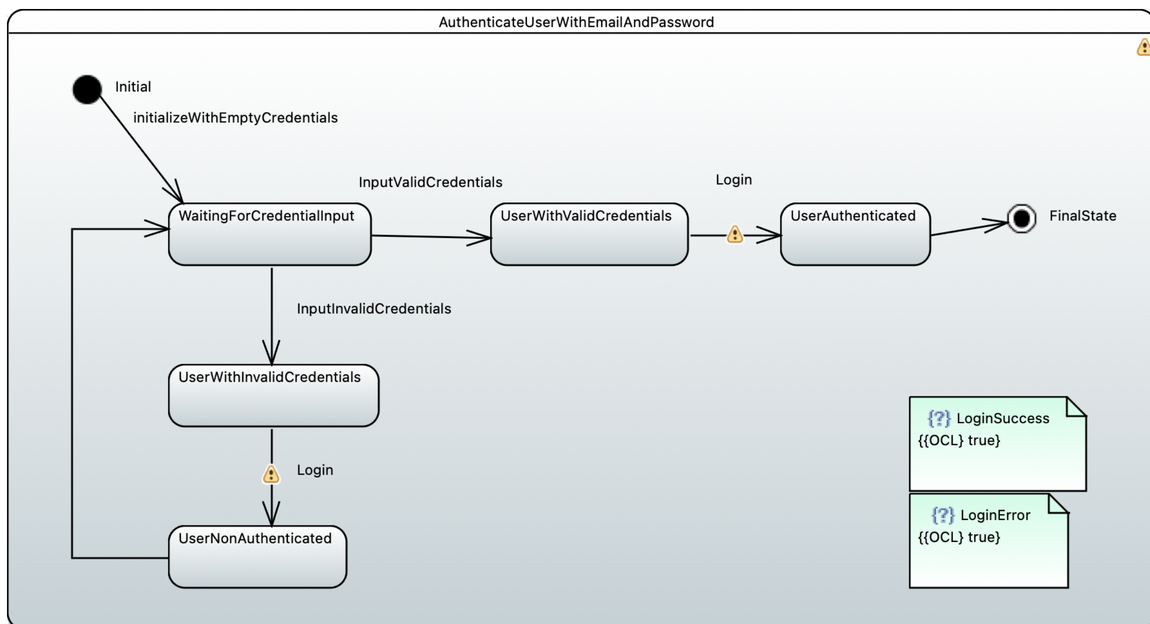


Figura 3: Modelo de estados para login com email senha

### 5.2.3 Validação

Para validar o modelo desenvolvido foi usada a funcionalidade de validação da *Papyrus*. A funcionalidade de validação identifica partes do modelo que violam regras de modelagem EMF. As violações e identificadas são marcadas no modelo para correção. No modelo da Figura 3 as transições de *Login* foram marcadas com indicador de "warning"por terem o

mesmo nome. Porém, a nomeação foi mantida por possibilitar reutilização da implementação dos casos de teste, como será mostrado na seção 5.2.6.

### 5.2.4 Geração de Cenários

A Skyfire oferece uma API em Java para geração de Cenários de teste em Gherkin a partir de modelos de estado. Em especial, o método estático *generateCucumberScenario* da classe *CucumberTestGenerator*. Este método possui quatro parâmetros: o caminho para o arquivo .uml do modelo de estado, o critério de cobertura escolhido, descrição do que está sendo testado e o caminho do arquivo onde serão salvos os cenários de teste.

Foram gerados casos de teste usando os quatro critérios de cobertura, a fim de comparar a diferença entre eles. A Figura 4 mostra um caso de teste gerado pelo critério *Edge Coverage*. Neste Cenário, o usuário faz uma tentativa de login com uma combinação de email e senha inválida e gera um erro. As credenciais são corrigidas, e o login é feito com sucesso. Note-se que os passos dos cenários são os nomes de transições e de condições. Isto reforça a necessidade de escolher nomes de acordo com o domínio do problema, para que os cenários sejam compreensíveis por todos. Os critérios *Node*, *Edge-Pair* e *Prime-path* geraram cenários semelhantes ao mostrado na Figura 4, variando no número de tentativas de login com credenciais inválidas. Esse número variou de zero a duas tentativas com erro. A Tabela 4 mostra o número de cenário gerados, o número mínimo e máximo de passos para cada critério de cobertura.

```
Feature: Login

Scenario: initializeWithEmptyCredentials InputInvalidCredentials Login  InputValidCredentials Login

Given initializeWithEmptyCredentials
When InputInvalidCredentials
And Login
Then LoginError
When InputValidCredentials
And Login
Then LoginSuccess
```

Figura 4: Cenário de teste em Gherking gerado pelo critério *Edge Coverage*

| Critério | N Cenários | N máximo de passos | N mínimo de passos |
|---|---|---|---|
| *Node Coverage* | 1 | 7 | 7 |
| *Edge Coverage* | 1 | 7 | 7 |
| *Edge-pair Coverage* | 3 | 4 | 10 |
| *Prime-path Coverage* | 2 | 4 | 10 |

Tabela 4: Dados sobre cenários gerados a partir do primeiro modelo

#### 5.2.5 Parametrização dos casos de teste

Nesta etapa os casos de teste em Gherkin gerados na etapa anterior foram convertidos em *Scenario Outline*. *Scenario Outlines* permitem definir os daos usados no diferentes cenários. Assim para cada cenário abstrato, várias instâncias serão criadas de acordo com o número de linhas da tabela.

A Figura 5 mostra os casos de teste já parametrizados. Como o modelo é simples e os casos de teste gerados pelos critérios foram similares, não foi preciso parametrizar todos os casos gerados. Nos passos *InputInvalidCredentials* e *InputValidCredentials* foram adicionados os parâmetros *invalidEmail* , *invalidPassword*, *validEmail* e validPassword. Esses parâmetros receberão os valores de entrada para email e senha. No passo *LoginError* foram adicionados mais dois parâmetros: *errorCode* e *errorMessage*, que definem qual o caso de erro esperado. Finalmente, no passo *LoginSuccess* foi adicionado apenas um parâmetro: *successCode*. Como foi mostrado na Tabela 2, em caso de sucesso não há mensagem de status, portanto não foi preciso adicionar mais um parâmetro, como feito para o passo de *LoginError*.

```
Feature: Login with email and password

    Scenario Outline: Login registered user with email and password

        Given initializeWithEmptyCredentials
        When InputInvalidCredentials "<invalidEmail>" "<invalidPassword>"
        And Login
        Then LoginError "<errorCode>" "<errorMessage>"
        When InputValidCredentials "<validEmail>" "<validPassword>"
        And Login
        Then LoginSuccess "<successCode>"

        Examples:
            | invalidEmail   | invalidPassword | errorCode | errorMessage     | validEmail     | validPassword | successCode |
            | user@gmail     | 123456          | 400       | EMAIL_NOT_FOUND  | user@gmail.com | 123456        | 200         |
            | user           | 123456          | 400       | INVALID_EMAIL    | user@gmail.com | 123456        | 200         |
            | user@gmail.com |                 | 400       | MISSING_PASSWORD | user@gmail.com | 123456        | 200         |
            | user@gmail.com | 123             | 400       | INVALID_PASSWORD | user@gmail.com | 123456        | 200         |

    Scenario Outline:

        Given initializeWithEmptyCredentials
        When InputValidCredentials "<validEmail>" "<validPassword>"
        And Login
        Then LoginSuccess "<successCode>"

        Examples:
            | validEmail     | validPassword  | successCode |
            | user@gmail.com | 123456         | 200         |
```

Figura 5: Parametrização dos casos de teste para login com email e senha

#### 5.2.6 Implementação dos casos de teste

Nesta etapa foram implementados os casos de teste. Isso foi feito usando o framework Cucumber para JavaScript. A Figura 6 mostra o resultado final da implementação. No passo *WaitingForCredentialInput* são estabelecidos os valores inicias para os parâmetros de email e senha. Nos passos *InputValidCredentials* e *InputInvalidCredentials*, são salvas as credenciais válidas e inválidas fornecidas pelas tabelas da Figura 5. A chamada para o serviço de login do Firebase é feito no passo *Login*, com as credenciais salvas anteriormente. A chamada é feita usando a biblioteca de pedidos HTTPS para JavaScript: axios. Por fim, nos casos *LoginSuccess* e *LoginError* são feitas as comparações entre o retorno da API e os valores informados pela tabela de parâmetros.

```
const assert = require("assert");
const axios = require("axios");
const { Given, When, Then }  = require("cucumber");

const url = "https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?
key=AIzaSyBBoDJfWLCpgLQa821_6igi3GZUmtMWbzY"

Given('initializeWithEmptyCredentials', function () {
    this.email = undefined
    this.password = undefined
});

When('InputValidCredentials {string} {string}', function (validEmail, validPassword) {
     this.email = validEmail
     this.password = validPassword
});

When('InputInvalidCredentials {string} {string}', function (invalidEmail, invalidPassword) {
    this.email = invalidEmail
    this.password = invalidPassword
});

When('Login', async function () {
    try {
        this.response = await axios.post(url, {
            email: this.email,
            password: this.password,
            returnSecureToken: false,
        })
    } catch (error) {
        if (error.response) {
            this.error = error.response.data.error
        }
    }
});

Then('LoginSuccess {string}', function (successCode) {
    assert.strictEqual(`${this.response.status}`, successCode)
});

Then('LoginError {string} {string}', function (errorCode, errorMessage) {
    assert.strictEqual(`${this.error.code}`, errorCode)
    assert.strictEqual(`${this.error.message}`, errorMessage)
});
```

Figura 6: Parametrização dos casos de teste para login com email e senha

#### 5.2.7 Validação dos casos de teste

Nesta etapa foi verificado se era possível executar os testes. Primeiro foi verificado se todos os passos em Gherkin tinham um *Step Definition* associado. Depois procurou-se por erros da implementação do código. Os erros encontrados foram corrigidos, até que os testes pudessem ser executados sem problema.

#### 5.2.8 Execução dos casos de teste dos casos de teste

Foram executados os testes, e colhidos os resultados de sucesso ou falha. A execução foi feita usando o comando ”cucumber-js test”da CLI do Cucumber para JavaScript.

### 5.3 Segunda iteração do método

Nesta seção é apresentado como se dá a evolução do modelo para a introdução de novos requisitos.

### 5.3.1 Histórias de Usuário

Nesta etapa foi escrito o requisito de autenticação do serviço via API Key na forma de história de usuário. A Tabela 5 mostra o resultado final. A autenticação via API Key é necessária para que o sistema que está sendo desenvolvido possa usar os serviços do Firebase.

| **Como** | **Ação** | **Objetivo** |
|---|---|---|
| Um desenvolvedor do sistema | Informo a API Key do projeto | Para que eu possa utilizar os serviços ofereicos pelo Firebase |

Tabela 5: Requisitos de autenticação via API Key em forma de Histórias de Usuário

### 5.3.2 Modelagem

A Figura 7 abaixo mostra o modelo feito a partir das Histórias de Usuário da Tabela 5. Ele foi construído a partir do modelo mostrado pela Figura 3, aproveitando os estados, transições e invariantes de estado da funcionalidade de login. Para incluir o requisito de autenticação por API Key, foram adicionadas duas transições a partir do estado inicial do modelo: *initializedWithValidKey* e *initializedWithInvalidKey*. A primeira representa o caso do desenvolvedor informar uma API Key válida. Neste caso, a funcionalidade de login pode ser usada normalmente. A segunda representa o caso de uma API Key inválida e, desta forma, a funcionalidade de login está indisponível e API deve gerar um erro de chave, *KeyError*, ao tentar usá-la.

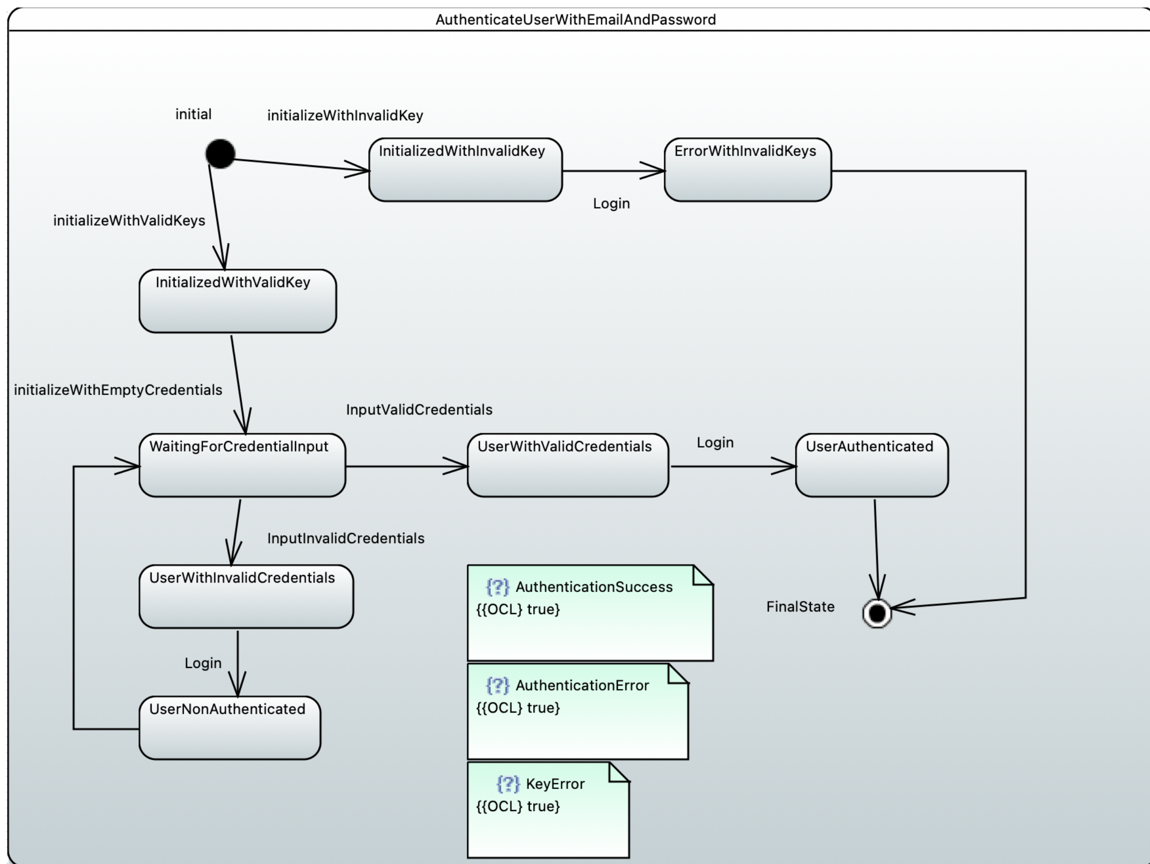


Figura 7: Modelo de estados com autenticação via API Key

### 5.3.3 Validação

A validação foi feita igual à seção 5.2.3. Para o modelo da Figura 7, além do aviso de nomeação repetida para a transição de *Login*, não foram encontradas outras violações.

### 5.3.4 Geração dos Cenários

Os Cenários para este modelo foram gerados da mesma forma que os cenários apresentados na seção 5.2.4. A Figura 8 mostra os casos de teste gerados a partir do critério *Edge Coverage*. A Tabela 11 mostra o número de cenário gerados, o número mínimo e máximo de passos para cada critério de cobertura.

```
Feature: Authenticate with API Key and Login

Scenario: initializeWithInvalidKey Login

Given initializeWithInvalidKey
When Login
Then KeyError


Scenario: initializeWithValidKeys initializeWithEmptyCredentials InputInvalidCredentials Login
InputValidCredentials Login

Given initializeWithValidKeys
Given initializeWithEmptyCredentials
When InputInvalidCredentials
And Login
Then LoginError
When InputValidCredentials
And Login
Then LoginSuccess
```

Figura 8: Caso de teste em Gherkin gerado com o critério *Edge Coverage*

| Critério | N Cenários | N máximo de passos | N mínimo de passos |
|---|---|---|---|
| *Node Coverage* | 2 | 3 | 8 |
| *Edge Coverage* | 2 | 3 | 8 |
| *Edge-pair Coverage* | 4 | 3 | 11 |
| *Prime-path Coverage* | 3 | 3 | 11 |

Tabela 6: Dados sobre cenários gerados a patir da segunda iteração do modelo

### 5.3.5 Parametrização dos Cenários

A Figura 9 mostra a parametrização feita para os casos de teste gerados a partir da segunda iteração do modelo. Assim como para o primeiro modelo, os critérios e cobertura da *Skyfire* geraram casos de teste semelhantes e não foi preciso parametrizar todos.

```
Feature: Authenticate with API Key and Login

Scenario Outline: initializeWithInvalidKey Login

    Given initializeWithInvalidKeys "<invalidKey>"
    When Login
    Then KeyError "<errorCode>" "<errorMessage>"

    Examples:
    | invalidKey | errorCode |                   errorMessage                  |
    | xxxxxxxxxx | 400       | API key not valid. Please pass a valid API key. |
    |            | 403       | The request is missing a valid API key.         |

Scenario Outline: initializeWithValidKeys initializeWithEmptyCredentials InputValidCredentials Login

    Given initializeWithValidKeys "<validKey>"
    Given initializeWithEmptyCredentials
    When InputValidCredentials "<validEmail>" "<validPassword>"
    And Login
    Then LoginSuccess "<successCode>"

    Examples:
    |               validKey                | validEmail     | validPassword | successCode |
    | AIzaSyBBoDJfWLCpgLQa821_6igi3GZUmtMWbzY | user@gmail.com | 123456        | 200         |

Scenario Outline: initializeWithValidKeys initializeWithEmptyCredentials InputInvalidCredentials Login  InputValidCredentials Login

    Given initializeWithValidKeys "<validKey>"
    Given initializeWithEmptyCredentials
    When InputInvalidCredentials "<invalidEmail>" "<invalidPassword>"
    And Login
    Then LoginError "<errorCode>" "<errorMessage>"
    When InputValidCredentials "<validEmail>" "<validPassword>"
    And Login
    Then LoginSuccess "<successCode>"

    Examples:
    |               validKey                | invalidEmail   | invalidPassword | errorCode | errorMessage     | validEmail     | validPassword | successCode |
    | AIzaSyBBoDJfWLCpgLQa821_6igi3GZUmtMWbzY | user@gmail     | 123456          | 400       | EMAIL_NOT_FOUND  | user@gmail.com | 123456        | 200         |
    | AIzaSyBBoDJfWLCpgLQa821_6igi3GZUmtMWbzY | user           | 123456          | 400       | INVALID_EMAIL    | user@gmail.com | 123456        | 200         |
    | AIzaSyBBoDJfWLCpgLQa821_6igi3GZUmtMWbzY | user@gmail.com |                 | 400       | MISSING_PASSWORD | user@gmail.com | 123456        | 200         |
    | AIzaSyBBoDJfWLCpgLQa821_6igi3GZUmtMWbzY | user@gmail.com | 123             | 400       | INVALID_PASSWORD | user@gmail.com | 123456        | 200         |
```

Figura 9: Caso de teste em Gherkin gerado com o critério *Edge Coverage*

#### 5.3.6 Implementação dos casos de teste

A Figura 10 abaixo mostra a implementação dos casos de teste para o Scenario Outline da Figura 9. Foi preciso fazer algumas alterações em relação à implementação mostrada na Figura 6. Como o segundo modelo verifica se a API Key é válida, foi preciso inicializar os parâmetros da da url. Isso é feito nos passos *initializeWithvalidKey* e *initializeWithInvalidKey*, em vez de defini-la na constante *baseUrl*. Também foi preciso implementar o passo *KeyError*, e fazer a verificação do código de erro gerado. Os demais passos tiveram sua implementação aproveitada.

```
const assert = require("assert");
const axios = require("axios");
const { Given, When, Then }  = require("cucumber");

Given('initializeWithValidKeys {string}', function (validKey) {
    this.key = validKey
});

Given('initializeWithInvalidKeys {string}', function (invalidKey) {
    this.key = invalidKey
});

Given('initializeWithEmptyCredentials', function () {
    this.email = undefined
    this.password = undefined
});

When('InputValidCredentials {string} {string}', function (validEmail, validPassword) {
     this.email = validEmail
     this.password = validPassword
});

When('InputInvalidCredentials {string} {string}', function (invalidEmail, invalidPassword) {
    this.email = invalidEmail
    this.password = invalidPassword
});

When('Login', async function () {

    this.endpoint = `https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword`
    this.url = `${this.endpoint}?key=${this.key}`

    try {
        this.response = await axios.post(this.url, {
            email: this.email,
            password: this.password,
            returnSecureToken: false,
        })
    } catch (error) {
        if (error.response) {
            this.error = error.response.data.error
        }
    }
});

Then('LoginSuccess {string}', function (successCode) {
    assert.strictEqual(`${this.response.status}`, successCode)
});

Then('LoginError {string} {string}', function (errorCode, errorMessage) {
    assert.strictEqual(`${this.error.code}`, errorCode)
    assert.strictEqual(`${this.error.message}`, errorMessage)
});

Then('KeyError {string} {string}', function (errorCode, errorMessage) {
    assert.strictEqual(`${this.error.code}`, errorCode)
    assert.strictEqual(`${this.error.message}`, errorMessage)
});
```

Figura 10: Caso de teste em Gherkin gerado com o critério *Edge Coverage*

### 5.3.7 Validação dos casos de teste

Nesta etapa repetiu-se o que foi feito na seção 5.2.7 durante a validação dos casos de teste para primeira iteração do método.

#### 5.3.8 Execução dos casos de teste

Nesta etapa foram os executados os testes e colhidos os resultados de sucesso e falha. Como foi feito para primeira iteração do método.

## 6 Monitoração e Observação

Esta seção mostra o resultado dos testes executados durante a execução do método. As Tableas 7 e 8 mostram os dados sobre os passos e cenários executados durante a primeira iteração do método. As Tabelas 9 e 10 mostram os resultados da segunda iteração.

| **Cenários executados** | **Sucessos** | **Falhas** |
|---|---|---|
| 5 | 5 | 0 |

Tabela 7: Resultado dos Cenários executados durante a primeira iteração

| **Passos executados** | **Sucessos** | **Falhas** |
|---|---|---|
| 32 | 32 | 0 |

Tabela 8: Resultado dos passos executados durante a primeira iteração

| **Cenários executados** | **Sucessos** | **Falhas** |
|---|---|---|
| 7 | 7 | 0 |

Tabela 9: Resultado dos Cenários executados durante a segunda iteração

| **Passos executados** | **Sucessos** | **Falhas** |
|---|---|---|
| 7 | 7 | 0 |

Tabela 10: Resultado dos passos executados durante a segunda iteração

| Passos executados | Sucessos | Falhas |
|---|---|---|
| 43 | 43 | 0 |

Tabela 11: Dados sobre cenários gerados a patir da segunda iteração do modelo

# 7 Discussão

## 7.1 Sobre o uso MBT e BDD

A etapa de modelagem foi a etapa que mais demandou esforço durante este projeto. Foram necessárias várias versões de modelo até que se acumulasse o conhecimento necessário e se chegasse aos modelos das Figuras 3 e 7. Por ser uma estratégia de geração de testes pouco adotado na indústria, há pouco material de apoio disponível sobre boas práticas. A *Papyrus*, ferramenta utilizada para desenvolver o modelo, apresenta a mesma limitação. A documentação existente está desatualizada ou incompleta. O aprendizado sobre modelagem e ferramenta se deu por tentativa e erro, e comparações com o modelo UML dado como exemplo no repositório da *Skyfire*.

## 7.2 Sobre o uso da *Skyfire* para gerar casos de teste automaticamente

Foi possível gerar casos de teste em *Gherkin* usando a *Skyfire*. O artigo de referência da *Skyfire* [9] é detalhado e explica de forma clara seu funcionamento. Os critérios de cobertura oferecidos pela ferramenta, foram capazes de gerar casos de teste diversos que exploram bem o comportamento expresso pelo modelo. Isso foi visto, principalmente, para os casos de testes gerados a partir do modelo da Figura 7, usando o critério de cobertura *Prime-path*. Foram gerados quatro cenários em *Gherkin*, cada um explorando um caso de utilização diferente.

Foram identificadas três principais limitações para uso da *Skyfire*. Primeiro, a nomeação dos casos de teste não é amigável. Os cenários gerados automaticamente são identificados pelos passos que o compõem. O caso de teste da Figura 4 é identificado como "*Scenario: Login intializeWithEmptyCredentials InputInvalidCredentials Login InputValidCredentials Login*". Isso dificulta o entendimento de novos testadores, ou pessoas sem o conhecimento técnico do sistema. Segundo, a *Skyfire* ainda não e capaz de trabalhar com condições de guarda. É preciso usar transições de estado para cada caso, ou outros componentes de modelo de estado como *choice*. Por fim, por ser uma ferramenta que combina BDD e MBT seu uso exige que o testador tenha conhecimento e prática em ambas.

No entanto, a *Skyfire* mostrou-se uma ferramenta interessante para suporte a desenvolvimento usando BDD e MBT. As limitações encontradas durante este projeto podem ser contornadas, e seu uso permite que testadores invistam esforço em entender e modelar o comportamento do sistema ao invés de gerar casos de teste manualmente. Para estressar o comportamento da *Skyfire* seria interessante usá-la para gerar testes a partir de um modelo maior e mais complexo.

### 7.3 Sobre facilidade de desenvolvimento incremental

Quanto ao modelo, foi possível trabalhar de forma incremental sem dificuldades. O incremento feito no modelo mostrado pelo Figura 7, permitiu que grande parte do primeiro modelo, mostrado pela Figura 7, fosse reutilizada. Isso permitiu que os *Steps Definitions* mostrados pela Figura 6 também fossem reutilizados. Dado que os passos (Given, When e Then) correspondem a transições do modelo de estados, fica mais fácil reutilizar as definições dos passos: só é necessário criar implementações para passos novos. Como a tarefa de implementar os passos é manual, esta característica facilita as tarefas dos testadores.

## 8 Conclusão e Trabalhos Futuros

Este projeto se propôs a desenvolver uma técnica de geração de testes que combinasse o uso de BDD e MBT. Com o uso do MBT esperava-se gerar casos de teste de maneira automática a partir de um modelo de estado, e a vantagem de fazer alterações no modelo para cobrir novos requisitos ao invés de escrever novos testes manualmente. Com o uso do BDD esperava-se usar os casos de teste como uma documentação do comportamento do sistema.

O método proposto foi executado tendo a funcionalidade de login com email e senha do Firebase como sistema sob teste. Foram feitas duas iterações do método, adicionando-se um requisito novo da primeira para a segunda. O modelo de estados foi desenvolvido e validade usando a ferramenta *Papyrus*. Os casos de teste foram gerados em Gherkin de forma automática, usando a *Skyfire*. Por fim, os testes foram implementados, validados e executados usando o framework *Cucumber*.

O método mostrou-se capaz de gerar testes automaticamente a partir do modelo desenvolvido. A etapa de desenvolvimento do modelo demandou a maior parte dos esforços. Por MBT ser uma técnica nova e, até o momoento, com pouca adoção do mercado, há pouco material de apoio disponível sobre boas práticas e ferramentas. Porém, com o modelo desenvolvido foi possível gerar casos de teste em Gherkin sem dificuldades. Como a *Skyfire* gera os casos de teste em Gherkin, é mantida a vantagem do BDD de gerar testes como documentação do sistema.

Gerar os casos de teste a partir do modelo também facilitou o desenvolvimento incremental. Trabalhar com um modelo é uma vantagem, pois fazer mudanças em um alto nível de abstração demando menos esforço. O novo requisito foi introduzido no modelo e os testes foram gerados automaticamente. Com a experiência adquirida durante o projeto, a atualização do modelo não demandou tanto esforço quanto na primeira iteração. Foi possível reutilizar parte da implementação dos testes entre as iterações. As mundanças feitas no modelo não alteraram o nome nem o tipo de passo em Gherkin gerado pela *Skyfire*.

Como trabalho futuro sugere-se aplicar o método durante o desenvolvimento de um sistema. O Firebase, usada neste projeto como objeto de teste, já é uma aplicação em uso e por isso já passou por um extenso processo de testagem. Aplicar o método proposto nesse projeto durante o desenvolvimento de um sistema trará novos desafios quanto levantamento de requisitos, eficácia do método em desenvolvimento incremental e da capacidade dos testes gerados acharem falhas.

## Referências

[1] SMART, John Ferguson. **What is Behavior Driven Development? The executive summary:** What are the benefits of Behavior Driven Development. Disponível em: https://johnfergusonsmart.com/behaviour-driven-development-3-minute-rundown. Acesso em: 17 Dez 2022

[2] SMARTBEAR SOFTWARE. **Cucumber Open Docs:** Gherkin Reference. Disponível em: https://cucumber.io/docs/gherkin/reference. Acesso em: 17 Dez 2022

[3] UTTING, Mark; PRETSCHNER, Alexander; LEGEARD, Bruno. A taxonomy of model-based testing approaches. Software testing, verification and reliability, v. 22, n. 5, p. 297-312, 2012

[4] D. Chelimsky et al. "The RSpec book: Behaviour driven development with Rspec, Cucumber, and friends". Pragmatic Bookshelf, 2010

[5] ISTQB®, 2015. Foundation Level Certified Model-Based Tester Syllabus

[6] David Coghlan, D. Teresa Brannick, T. 2014. Doing Action Research in Your Own Organization. Sage Publications Ltd

[7] SMARTBEAR SOFTWARE. **Cucumber Open Docs:** What is Cucumber. Disponível em: https://cucumber.io/docs/guides/overview. Acesso em: 17 Dez 2022

[8] SMARTBEAR SOFTWARE. **Cucumber Open Docs:** Step Definition. Disponível em: https://cucumber.io/docs/cucumber/step-definitions/?lang=javascript. Acesso em: 17 Dez 2022

[9] LI, Nan; ESCALONA, Anthony; KAMAL, Tariq. **Skyfire: Model-Based Testing With Cucumber**.

[10] P. Ammann and J. Offutt, Introduction to Software Testing. Cambridge, UK: Cambridge University Press, 2008.

[11] GOOGLE. **Firebase:** Get started. Disponível em: https://firebase.google.com. Acesso em: 17 Dez 2022