Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Uma Abordagem para Classificação e Sugestão de Atualizações em Máquinas de Estado a partir de Mudanças no Código-fonte

Matheus de Oliveira Barbosa

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Metodologia e Técnicas da Computação

Nome do Orientador
Franklin de Souza Ramalho

Campina Grande, Paraíba, Brasil
©Matheus de Oliveira Barbosa, 07/03/2019

B238a Barbosa, Matheus de Oliveira.

Uma abordagem para classificação e sugestão de atualizações em máquinas de estado a partir de mudanças no código-fonte / Matheus de Oliveira Barbosa. – Campina Grande, 2019.

93 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2019.

"Orientação: Prof. Dr. Franklin de Souza Ramalho".

Referências.

1. Engenharia de software. 2. Máquina de Estado. 3. Atualização. 4. Código-fonte. 5. Mudança. 6. Mudança em Código-fonte. I. Ramalho, Franklin de Souza. II. Título.

CDU 004.41(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA MARIA ANTONIA DE SOUSA CRB 15/398

"UMA ABORDAGEM PARA CLASSIFICA¤űO E SUGESTűO DE ATUALIZA¤ǚES EM MΞQUINAS DE ESTADO A PARTIR DE MUDAN¤AS NO CΦDIGO-FONTE"

MATHEUS DE OLIVEIRA BARBOSA

DISSERTA¤űO APROVADA EM 07/03/2019

FRANKLIN DE SOUZA RAMALHO, Dr., UFCG
Orientador

JOűO ARTHUR BRUNET MONTEIRO, Dr., UFCG
Examinador

UIRA KULESZA, Dr., UFRN
Examinador

CAMPINA GRANDE - PB

Resumo

Durante o processo de desenvolvimento de software, são elaborados diversos tipos de artefatos que representam níveis de abstrações e visões diferentes. Dentre estes artefatos, as Máquinas de Estado Comportamentais (ME) UML, são um dos modelos mais utilizados para representação do comportamento dinâmico do software. Devido às constantes mudanças que ocorrem no código-fonte do software, o comportamento ou características do sistema podem sofrer alterações, e com isso, as MEs podem necessitar, também, de atualizações, que demandam tempo e esforço. Neste trabalho, propomos uma abordagem para classificar e sugerir alterações nas MEs com base nas mudanças em código-fonte, tendo foco nos elementos de Estado e Transição. Essa abordagem é composta por uma taxonomia para alterações em MEs, um mapeamento entre as alterações em MEs e mudanças de código-fonte, um algoritmo capaz de classificar e sugerir essas alterações em MEs, e uma ferramenta que implementa o algoritmo proposto para utilização real. Foi realizado um estudo com projetos reais, onde foram avaliadas a precisão e cobertura da abordagem proposta, que alcançou uma taxa de precisão de 65,60% e uma cobertura de 50,80%. Desse modo, pode-se obter as alterações nas MEs e assim aplicá-las, necessitando de uma validação das sugestões, possibilitando uma redução de esforço nesse tipo de atividade.

Abstract

During the software development process, various types of artifacts are created that represent levels of abstractions and different views. Among these artifacts, the Behavioral State Machines (SM) UML, are one of the most used models to represent the dynamic behavior of the software. Due to the constant changes that occur in the source code of the software, the behavior or characteristics of the system may change, and with this, the SMs require updates and that demand time and effort. In this work, we propose an approach to classify and suggest changes in SMs based on changes in source code, focusing on the elements of State and Transition. This approach is composed of a taxonomy for changes in SMs, a mapping between changes in SMs and source code changes, an algorithm capable of classifying and suggesting these changes in SMs, and a tool that implements the algorithm proposed for actual use. A study with real projects was carried out, where the accuracy and coverage of the proposed approach were evaluated, achieving a precision rate of 65.60% and coverage of 50.80%. In this way, it is possible to obtain the changes in the SMs and thus to apply them, requiring a validation of the suggestions, allowing a reduction of effort in this type of activity.

Agradecimentos

Agradecer primeiramente à Deus, pois com toda certeza Ele atendeu todos os meus pedidos durante essa fase da minha vida. Manteve-me firme, com saúde até o final, além de diversos livramentos, bênçãos e vários outros motivos que serei eternamente grato por tudo que Ele tem feito por mim e por todos à minha volta.

Agradecer à minha família por ter me apoiado em todos os momentos e principalmente por acreditarem e não perderem a fé em mim. Aprendemos a lidar com a distância e com a ausência em datas especiais, pois foram necessários algumas ausências para conseguir concluir esta etapa da minha vida. Obrigado de todo coração por tudo o que fizeram e fazem por mim.

Agradecer à minha namorada Marcela, por ter estado ao meu lado durante essa fase, por ter tido compreensão e principalmente companheirismo, dando palavras de apoio para continuar nessa caminhada, foram essenciais para que eu continuasse avançando até aqui. Hoje espero poder compartilhar não só esta conquista com ela, como também, todas as outras que poderão vir e que possamos também compartilhar uma vida harmoniosa juntos.

Agradecer aos meus amigos Alysson, Indy, Melquisedec, Mirna, Thaciana que fizeram e fazem parte da família "Sala 105", pois todos eles contribuíram de alguma forma durante essa fase. Agradecer, também, à Jaziel por compartilhar conhecimento, auxiliando assim na solução de problemas em ferramentas que ambos utilizaram.

Agradecer à todos os demais amigos que sempre acreditarem e mandaram mensagens de força e entusiasmo nessa caminhada, mantendo-me sempre motivado.

Agradecer em especial ao meu orientador Franklin, por sua paciência e atenciosidade, pelos conselhos e orientações, pois sem isso, jamais teria conseguido concluir esta pesquisa. Agradecer por ter feito um excelente papel de orientador, nos guiando por caminhos corretos para que pudéssemos alcançar este resultado.

Agradecer ao CNPq e a Capes pelo apoio financeiro na produção deste trabalho, foi de imensurável ajuda.

Conteúdo

1 Introdução **1**

1.1 O Problema 2

1.2 Objetivos 5

1.3 Hipóteses 6

1.4 Escopo 6

1.5 Contribuições 7

1.5.1 Relevância das Contribuições 7

1.6 Organização do Documento 8

2 Fundamentação Teórica **9**

2.1 UML e Máquinas de Estado Comportamentais 9

2.2 Mudanças em Código Fonte 12

2.3 Recuperação de Informação 16

3 Trabalhos Relacionados **19**

4 Abordagem Proposta **23**

4.1 Classificação das Mudanças em Código-fonte 23

4.2 Taxonomia das Alterações em MEs 25

4.3 Algoritmo para Classificação de Alterações em MEs 29

5 Estudo Exploratório **35**

5.1 Mudanças no Código-fonte que causam Alterações nas MEs 35

5.2 Seleção das Unidades Experimentais 37

5.3 Tratamento dos Dados 38

5.4 Treinamento e Execução 40
5.5 Avaliação do Perfil do Participante 43
5.5.1 Avaliação das Dificuldades dos Participantes 45
5.6 Análise dos Resultados 51
5.7 Ameaças à Validade 59
5.7.1 Ameaças de Constructo 59
5.7.2 Ameaças Externas 59
5.7.3 Ameaças Internas 60
5.7.4 Ameaças de Conclusão 61
5.8 Discussão 61

6 Experimento - Avaliação do Algoritmo proposto 66
6.1 Design Experimental 68
6.2 Execução 69
6.3 Análise dos Resultados 70
6.4 Ameaças à Validade 79
6.4.1 Ameaças Internas 79
6.4.2 Ameaças Externas 80
6.4.3 Ameaças de Conclusão 80
6.4.4 Ameaças Constructo 81
6.5 Discussão 81

7 Conclusões 84
7.1 Contribuições Alcançadas 85
7.2 Limitações do algoritmo 85
7.3 Trabalhos Futuros 87

A Mapeamento - Tipos de mudanças no código-fonte que podem causar alterações em Máquinas de Estado 93

Lista de Figuras

1.1 ME antes das mudanças no código-fonte. 3
1.2 ME após as mudanças no código-fonte. 3
2.1 Exemplo de elementos presentes em uma ME (Produzido pelo Autor). . . . 11
2.2 Exemplo de uma ME (Produzido pelo Autor). 12
4.1 Visão geral em Componentes da técnica proposta. (Produzido pelo Autor) . 24
4.2 Exemplo de uma versão anterior da ME (Produzido pelo Autor). 28
4.3 Exemplo de uma versão atualizada da ME (Produzido pelo Autor). 28
4.4 Visão geral da técnica proposta (Produzido pelo Autor). 30
4.5 Exemplo da utilização do Lucene (Produzido pelo Autor). 32
4.6 Metamodelo da saída gerada pela técnica. 34
5.1 Diagrama de Atividades - Atividades executadas para realização do estudo. (Produzido pelo Autor) 42
5.2 Anos de experiência com programação na Linguagem JAVA dos participantes. (Produzido pelo Autor) 44
5.3 Quantidade de participantes que possuem empregos/cargos. (Produzido pelo Autor) 45
5.4 Nível de conhecimento dos participantes a respeito de MEs. (Produzido pelo Autor) 45
5.5 Dificuldade para identificar alterações nas MEs analisando as mudanças no código-fonte. (Produzido pelo Autor) 47
5.6 Relação entre Experiência com MEs e a dificuldade para identificar alterações nas MEs. (Produzido pelo Autor) 47

5.7 Dificuldade avaliada pelos participantes para manter as MEs de um projeto atualizadas. (Produzido pelo Autor) 48
5.8 Dificuldade para identificar quais mudanças no código-fonte causam alterações nas MEs. (Produzido pelo Autor) 48
5.9 Dificuldade para identificar quais mudanças no código-fonte podem causar alterações nas ME, após o aumento do número de linhas do código-fonte (Produzido pelo Autor) 49
5.10 Quão provável é a utilização de um mapeamento entre as mudanças nas MEs e código-fonte para identificar eventuais alterações nas MEs. (Produzido pelo Autor) 50
5.11 Versão desatualizada de uma Máquina de Estado do projeto DESystem. . . 53
5.12 Versão atualizada de uma Máquina de Estado do projeto DESystem. 53

6.1 Médias Gerais de Precisão e Cobertura do Algoritmo Proposto (Produzido pelo Autor). 78

Lista de Tabelas

2.1 Classificações das mudanças em códifo-fonte da *ChangeDistiller* adotadas em nossa técnica. 15

2.2 Exemplo do valores obtidos por meio da *ChangeDistiller* utilizando os códigos-fonte 2.1 e 2.2. 16

2.3 Exemplo do cálculo da métrica TF.IDF. (Produzido pelo Autor) 17

4.1 Classificação das alterações em MEs para o elemento de Estado e Estado Composto. 26

4.2 Classificação das alterações em MEs para o elemento de Transição. 27

5.1 Quantidade de mudanças no código-fonte por filtragem. Qtd = Quantidade; Mud = Mudanças em Código-fonte; Filt = Filtragem. 38

5.2 Correlação com o nível de conhecimento entre MEs e demais variáveis. . . 51

5.3 Respostas dos Participantes em relação às mudanças que causaram alterações nas MEs. 64

5.4 Mapeamento Inicial entre Mudanças no Código-fonte e Alterações em MEs. 65

6.1 Design Experimental Aninhado 69

6.2 Resultados do Algoritmo proposto para o projeto Smarthome. 71

6.3 Resultados do Algoritmo proposto para o projeto DesignPattern. 72

6.4 Resultados do Algoritmo proposto para o projeto DESystem. 74

6.5 Médias do Algoritmo proposto para o projeto DESystem. 76

6.6 Médias gerais de Precisão e Cobertura alcançadas pelo Algoritmo proposto. 77

Lista de Códigos Fonte

1.1 Versão anterior da classe ThingStatus presente no projeto Smarthome. . . . 4

1.2 Versão pós mudanças da classe ThingStatus presente no projeto Smarthome. 4

2.1 Exemplo versão 1 do Código 13

2.2 Exemplo versão 2 do Código 13

4.1 Overview do Algoritmo Proposto 30

4.2 Implementação do método de classificação de alterações em ME 33

5.1 Versão desatualizada de uma classe presente no projeto DESystem. 53

5.2 Versão atualizada de uma classe presente no projeto DESystem. 53

5.3 Linha de código-fonte inserida na nova versão em uma classe do projeto DESystem. 55

5.4 Trecho de código-fonte inserida na nova versão em uma classe do projeto Design Pattern Indeepth. 55

5.5 Inserção de chamada de método presente no projeto DESystem (1). 56

5.6 Inserção de chamada de método presente no projeto DESystem (2). 56

5.7 Versão anterior de Enumeration presente no projeto Smarthome. 57

5.8 Versão nova de Enumeration presente no projeto Smarthome. 57

6.1 Trecho de código-fonte retirado do projeto Smarthome 71

6.2 Trecho de código-fonte retirado do projeto DesignPattern 72

6.3 trecho de código retirado do projeto Smarthome 82

6.4 Condição IF retirada do projeto Smarthome 82

Capítulo 1

Introdução

Processo de desenvolvimento de sistema é um conjunto de atividades relacionadas que tem como resultado final o sistema [38]. Dentre estas atividades, está a atividade de especificação, a qual é responsável pela elaboração de documentos que representam o sistema como um todo, em diversos níveis de abstrações e visões para compreensão de todos os *stakeholders*. Dentre estes documentos que são elaborados, estão presentes os modelos UML que são bastante utilizados para representação comportamental e estrutural do sistema [3]. Com estes modelos UML, é possível especificar, visualizar e documentar aspectos estruturais e comportamentais do sistema [3].

Dentre os modelos UML, é possível destacar a Máquina de Estado Comportamental (**ME**), a qual é responsável por mostrar possíveis estados em que um objeto possa se encontrar durante a execução do sistema [31]. ME é um dos modelos mais utilizados para representação do comportamento dinâmico do sistema [10], devido a sua simplicidade para desenvolvimento e facilidade de uso. A ME pode ser composta por diversos elementos, tais como estados, transições, estados composto, submáquinas e outros [3].

Em um processo de desenvolvimento de software, conforme o software vai sendo implementado, é comum que o código-fonte vá sofrendo mudanças para se adequar as necessidades do cliente[38]. Essas mudanças podem alterar o comportamento do sistema, e com isso, as MEs que representem determinadas características podem necessitar de alterações para estarem em conformidade com o comportamento atual do software. Identificar quais mudanças no código-fonte podem causar alterações nas MEs, e assim, atualizar as MEs, pode ser considerado um trabalho oneroso.

Após ser realizado um levantamento de estudos relacionados, que abordam o processo de geração de MEs ou detecção de atualizações nas mesmas, foi identificada uma lacuna nesses temas. Muitos trabalhos, abordam a geração de MEs por meio da inferência de arquivos de logs [41] [26] [29]. Foram encontrados, também, estudos que buscavam propor uma geração de casos de testes, a partir de MEs, para identificar se o comportamento do software era o esperado [11] [5] [32]. Entretanto, não foram encontrados trabalhos que abordassem a temática de atualizar as MEs de um sistema já atualizado. Portanto, essa lacuna encontrada nos trabalhos, serviu como guia para o problema abordado neste estudo.

Para avaliar o algoritmo proposto e construir a taxonomia proposta, foram realizados um estudo exploratório, Capítulo 5 e um experimento, Capítulo 6. Inicialmente, foi realizado um estudo para obter mais informações para a taxonomia, como também validar as alterações com base nas respostas dos participantes. Com isso, foi possível elaborar a taxonomia de alterações, e aumentar a contribuição deste estudo. Em seguida, foi realizado um experimento com o algoritmo, para avaliar a eficiência no que diz respeito à precisão, que é a taxa de quantas alterações detectadas estavam corretas, e cobertura, que é a taxa de alterações que realmente ocorreram nas MEs e foram detectadas pelo algoritmo. O algoritmo alcançou uma taxa de precisão média de 65,60%, e na cobertura o algoritmo obteve uma média de 50,80%.

1.1 O Problema

Durante o processo de desenvolvimento de sistema, o código-fonte está em constante mudança, devido ao surgimento ou alterações de requisitos, processo de refatoração e correções de *bugs* [39] [17]. Com isto, as MEs que representam determinados objetos que sofreram alguma mudança, podem ser impactadas e assim necessitarem de atualizações, para que possam garantir que estão representando corretamente os objetos.

Identificar os elementos das MEs que precisarão ser atualizados para manter a sincronização entre código-fonte e ME não é uma atividade trivial, pois, é necessário verificar quais foram as mudanças no código-fonte e quais tiveram impacto nas MEs. Além disso, muitas destas MEs não estão documentadas em sua completude, deixando de conter algumas informações que dificultam o processo de identificar corretamente as atualizações necessárias nas MEs, para estarem em conformidade, corretamente e completamente, com o

código-fonte.

Para exemplificar o problema abordado, foi exposto uma ME, na Figura 1.1, de um sistema real chamado, Smarthome [1]. Este possui uma ME que sofre alteração após mudanças que foram feitas no código-fonte. Na Figura 1.1, encontra-se a versão anterior às mudanças que foram realizadas no código-fonte, enquanto na Figura 1.2 a versão atualizada da ME, para manter a conformidade com o código-fonte, ver Códigos-fonte 1.1 e 1.2. Embora este projeto possua as versões atualizadas das MEs, o processo de atualização pode ser uma atividade custosa.

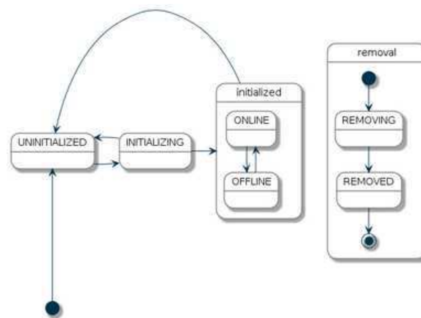


Figura 1.1: ME antes das mudanças no código-fonte.

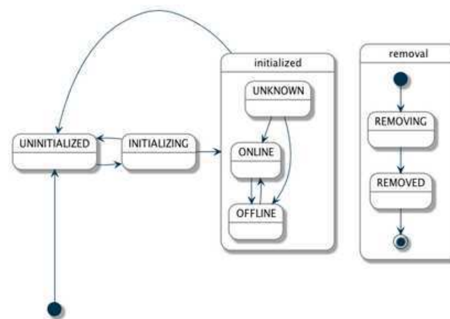


Figura 1.2: ME após as mudanças no código-fonte.

[1] https://github.com/eclipse/smarthome

Código Fonte 1.1: Versão anterior da classe ThingStatus presente no projeto Smarthome.

```
1 public enum ThingStatus {
2 UNINITIALIZED(0),
3 INITIALIZING(1),
4 ONLINE(2),
5 OFFLINE(3),
6 REMOVING(4),
7 REMOVED(5);}...
```

Código Fonte 1.2: Versão pós mudanças da classe ThingStatus presente no projeto Smarthome.

```
1 public enum ThingStatus {
2 UNINITIALIZED,
3 INITIALIZING,
4 UNKNOWN,
5 ONLINE,
6 OFFLINE,
7 REMOVING,
8 REMOVED;}
```

A ME exposta nas Figuras 1.1 e 1.2, representa um objeto do tipo ***Thing***, exibindo os possíveis estados que ele pode alcançar. A ME inicia com o estado de *UNINITIALIZED*, ocorrendo a transição para o estado de *INITIALIZING*, momento em que o objeto está sendo inicializado. No momento em que o objeto é inicializado, ele pode variar entre 3 estados *ONLINE*, *OFFLINE* ou *UNKNOWN*, conforme Figura 1.2, presente no estado composto nomeado de *initialized*. E por último, o processo de remoção do objeto inicia com o estado de *REMOVING*, e ao ser removido apresenta o estado de *REMOVED*, e assim encerrando seu ciclo.

Na Figura 1.1, é possível notar que quando um objeto do tipo *Thing* encontra-se no Estado Composto "*initialized*", ele pode variar entre dois Estados que são *ONLINE* e *OFFLINE*. Ao compararmos com o código-fonte 1.1 com a Figura 1.1, pode-se notar que todos o Estados estão presentes em ambos. Entretanto, ao ser inserido um novo valor de *Enumeration* na classe do código-fonte 1.2, é possível notar que houve, também, o acréscimo de um novo Estado na Figura 1.2. Embora seja um exemplo com uma simples mudança no código-fonte, pode-se ter uma visão que há a possibilidade de mudanças no código-fonte causarem alterações em MEs, para que estas reflitam o comportamento apresentado pelo código-fonte após as mudanças.

1.2 Objetivos

Buscando uma solução para o problema estamos propondo esta abordagem, sendo ela composta por quatro elementos produzidos na realização deste trabalho. Foi elaborado uma taxonomia de alterações em MEs com foco nos elementos de Estado, Transições e seus elementos internos, sendo esta utilizada para produzir um mapeamento entre quais tipos de mudanças no código-fonte podem causar alterações nas MEs e quais seriam estas alterações. Com esse mapeamento gerado, foi desenvolvido um algoritmo capaz de classificar alterações em MEs com base nas mudanças de código-fonte, seguindo as definições do mapeamento. Por último, foi desenvolvida uma ferramenta que implementa o algoritmo proposto, obtendo como saída uma lista de sugestões, com base na classificação do algoritmo, das alterações que devem ser aplicadas às MEs.

Portanto, pode-se definir que o objetivo geral deste trabalho é propor uma abordagem que possa ser capaz de identificar e sugerir alterações necessárias nas MEs, com base nas mudanças que ocorreram no código-fonte, preservando as características do design do software.

Os objetivos específicos desta pesquisa são:

- Contribuir com um estudo exploratório que tem como intuito identificar quais tipos de mudanças no código-fonte podem causar alterações em MEs, podendo servir como base para futuros estudos;
- Propor uma taxonomia de alterações que possam ser realizadas em MEs, respeitando o escopo deste trabalho e almejando auxiliar os usuários a saberem o que deve ser alterado na ME;
- Propor um algoritmo capaz de classificar e sugerir alterações em MEs, por meio de mudanças em código-fonte.
- Desenvolver uma ferramenta que implemente o algoritmo proposto, para utilização em ambientes reais;
- Realizar experimento para avaliar a eficiência do algoritmo no que diz respeito à sua precisão e cobertura.

1.3 Hipóteses

Com base nos objetivos descritos na Seção 1.2, foram levantadas hipóteses com o intuito de serem validadas ou refutadas, por meio dos estudos realizados neste trabalho. A seguir estão descritos as hipóteses levantadas para este trabalho:

- $H_1.0$ As mudanças em código-fonte não podem causar alterações nas ME;
- $H_1.1$ As mudanças no código-fonte podem causar alterações nas ME;
- $H_2.0$ Não há possibilidade de um mapeamento entre os tipos de mudanças em código-fonte e as alterações em MEs;
- $H_2.1$ É possível criar um mapeamento entre os tipos de mudanças no código-fonte e as alterações em MEs;
- $H_3.0$ O algoritmo proposto consegue classificar e sugerir alterações em ME com uma taxa de precisão inferior ou igual a 50%;
- $H_3.1$ O algoritmo proposto é capaz de classificar e sugerir alterações em ME com uma taxa de precisão superior a 50%.
- $H_4.0$ O algoritmo proposto consegue classificar e sugerir alterações em ME com uma taxa de cobertura inferior ou igual a 50%;
- $H_4.1$ O algoritmo proposto é capaz de classificar e sugerir alterações em ME com uma taxa de cobertura superior a 50%.

1.4 Escopo

A perspectiva deste trabalho está em identificar atualizações necessárias nas Máquinas de Estado Comportamental UML (ME), em particular, nos elementos estado e transição, que são uns dos principais. Para isso, usaremos as mudanças que são classificadas por meio da ferramenta *ChangeDistiller* [13] que eventualmente ocorrem no código-fonte, no contexto

da linguagem JAVA[2], durante as atividades de implementação ou evolução do sistema. Utilizando, também, técnicas de Recuperação de Informação para identificar quais são as classes que as MEs estão representando no código-fonte do sistema.

1.5 Contribuições

Este trabalho possibilitou a realização de dois estudos e o desenvolvimento de uma abordagem composta por 4 itens que, consequentemente, trouxeram as seguintes contribuições:

- Um estudo para identificar quais mudanças no código-fonte podem causar alterações em MEs;
- Taxonomia de alterações em MEs com foco nos elementos Estado e Transição;
- Mapeamento entre alterações em MEs e tipos de mudanças no código-fonte;
- Algoritmo proposto para classificação de alterações em MEs;
- Um estudo para avaliar a eficiência do algoritmo proposto, no que se diz respeito a suas taxas de Precisão e Cobertura;
- Ferramenta para utilização do algoritmo proposto, tendo como entrada o XML da ME a ser analisada e duas versões do código-fonte para classificação das mudanças no código-fonte, e como saída um documento, do tipo PDF ou XML, similar a um relatório.

1.5.1 Relevância das Contribuições

Com a aplicação da técnica proposta durante o processo de desenvolvimento de um software, espera-se alcançar uma redução no esforço aplicado para manter as MEs em conformidade com o código-fonte. Consequentemente, com essa redução de esforço, leva-se o foco para o processo de implementação, buscando reduzir os gastos com os processos de atualizações das MEs. Além disso, mantendo as MEs atualizadas, espera-se que torne a compreensão

[2]https://www.oracle.com/br/java/

comportamental do sistema mais claro, pois, a ME é um dos modelos mais utilizados para representar os comportamentos de um sistema [10].

Propondo uma taxonomia para alterações em MEs, espera-se contribuir com pesquisas relacionadas de modo que possam utilizá-la em seus estudos, fornecendo dados baseados em estudos já realizados. Além de possuir unidades experimentais passíveis de replicações, pois todos os seus códigos-fonte e suas documentações são disponíveis publicamente, por meio do Github.

Com a disponibilização do algoritmo em pseudocódigo e linguagem JAVA, permite-se sua análise em busca de novos aprimoramentos tanto em sua estrutura como decisões para classificação, podendo trazer outras contribuições para a área. Pode-se também, dar frutos a outros algoritmos, tendo um algoritmo base para utilizar em estudos comparativos avaliando diferentes métricas e cenários, para identificar qual melhor escolha de algoritmo para determinada situação.

1.6 Organização do Documento

Este documento está divido em capítulos, de tal maneira que, capítulo 2 descreve a fundamentação teórica necessária para entendimento do problema. O capítulo 3 contém trabalhos relacionados com a linha de pesquisa. O capítulo 4 descreve sobre a técnica que está sendo proposta neste trabalho. Os capítulos 5 e 6 descrevem os estudos que foram realizados neste trabalho, sendo o capítulo 5 um estudo exploratório para identificar quais tipos de mudanças podem causar alterações em MEs, e o capítulo 6 a avaliação da eficiência do algoritmo proposto, no que diz respeito à precisão e cobertura. O capítulo 7 estão descritas as conclusões, trabalhos futuros e as limitações da abordagem, almejando a melhora e continuidade deste trabalho.

Capítulo 2

Fundamentação Teórica

Neste capítulo são abordados conceitos básicos para compreensão do que se é proposto neste trabalho. Há conhecimentos a respeito de diagramas UML, e particularmente em ME, pois é sobre ela que o trabalho foca, tipos de mudanças no código-fonte e recuperação da informação.

2.1 UML e Máquinas de Estado Comportamentais

Havendo uma certa necessidade de documentar e modelar os sistemas para contribuir no entendimento e desenvolvimento de sistemas [42], foi criado uma linguagem para modelagem, Unified Modeling Language (**UML**). UML é uma linguagem padronizada utilizada para modelar os sistemas que seguem os conceitos do Paradigma Orientado a Objetos (**POO**) [20] [42], seja em uma visão estrutural ou comportamental do sistema. UML pode ser utilizada desde sistemas mais simples, como também aos mais complexo, pois podem aprimorar a comunicação entre diferentes domínios de conhecimento do sistema [20].

Modelagem é uma parte essencial para projetos de grande porte, como também de pequeno e médio porte. A utilização de modelos é importante para assegurar que as necessidades finais do cliente serão realmente atendidas [3] [38]. Além disso, auxiliam na compreensão do que deve ser implementado, nos comportamentos esperados e também na elaboração de testes [42].

Modelos UML, são modelos bastante utilizados para representar de maneira comportamental e estrutural o sistema o qual estão sendo elaborado [3] [35] [4]. Normalmente,

estes modelos são gerados durante a atividade de especificação do sistema, com o intuito de auxiliar a equipe de desenvolvimento à implementar as funcionalidades de maneira correta e completa, como também validar com o cliente se satisfaz suas necessidades. Dentre os modelos UML, que representam uma visão comportamental do sistema, está contido o de **M**áquina de **E**stado Comportamental (**ME**), o qual foi selecionado e utilizado nesta pesquisa.

Nós optamos selecionar a ME por ser considerada um dos modelos mais utilizados para representação comportamental do sistema [10] [25], devido à facilidade de uso, e sua simplicidade muitos desenvolvedores optam por utilizá-la. Além disso, sistemas reais, tais como: Databus, sistema produzido pela LinkedIn[1], para captura de dados de mudança distribuídos independente de origem; Smarthome[2], um framework produzido pela Eclipse, onde pode ser utilizado em produtos de casas inteligentes; têm utilizado a ME em suas documentações, podendo ser localizadas nos próprios repositórios.

Inicialmente, a teoria da Máquina de Estado Finita é um conceito matemático que busca criar modelos para aproximação de eventos físicos e abstratos, de maneira que possa auxiliar na compreensão desses eventos. Quando partimos para a engenharia de sistema, ME são utilizadas como modelos para representar o comportamento do sistema [23]. Além disso, também são usadas para mostrar os possíveis estados que um determinado objeto pode alcançar durante seu ciclo de vida [4].

ME é composta por elementos tais como: estado; transição; estado composto; estado inicial; estado final [3]. A Figura 2.1 expõe estes e mais alguns como por exemplo "*Ações do Estado*", que são comportamentos que o objeto pode ter ao atingir aquele estado. Os estados são características ou configurações que um objeto pode alcançar durante seu ciclo de vida.

A Figura 2.1 exibe inicialmente, da direita para a esquerda, o elemento definido como estado, o qual é responsável por representar características ou configurações que um determinado objeto se encontra em um ciclo de vida. Normalmente, são representados por um retângulo com bordas arredondadas. No estados, também estão contidas informações complementares a respeito de possíveis ações que possam ser executadas quando determinado estado for alcançado. Ao final está presente o estado composto, que é um estado composto

[1]https://github.com/linkedin/databus

[2]https://github.com/eclipse/smarthome

por outros estados, como uma forma de abstração dos estados que estão sendo englobados. Mais ao centro da Figura 2.1, os elementos representados em formas de círculos, são pseudo-estado inicial e final, são responsáveis por indicar o início e o fim do ciclo de vida do objeto que está sendo modelado, respectivamente.

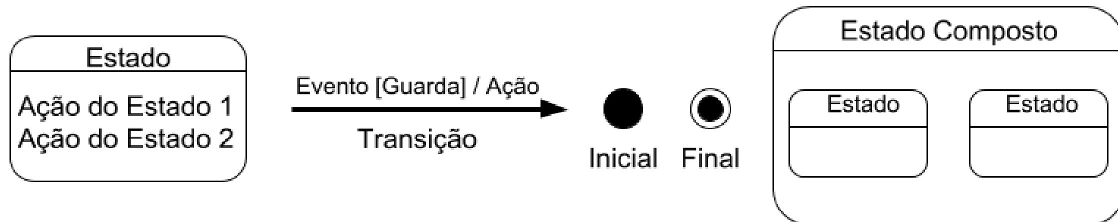


Figura 2.1: Exemplo de elementos presentes em uma ME (Produzido pelo Autor).

Ainda na Figura 2.1, estão expostos os elementos de transição, que são responsáveis por indicar o fluxo dos estados que o objeto alcançará em seu ciclo de vida [3], representada por uma seta a qual indica o sentido que seguirá. Uma transição pode ser opcionalmente constituída por evento, guarda e ação, os quais indicam os passos necessários para que ocorra a mudança de um estado X para um estado Y. O evento que, também pode ser interpretado como um sinal, ao ser disparado é iniciado o processo para mudança de Estado, onde em seguida é verificado se atende as condições da Guarda. Quando as condições forem atendidas será executada a Ação, para que em seguida ocorra a transição de um estado para outro.

Para ilustrarmos a utilização de uma ME, na Figura 2.2, nós produzimos um modelo exemplo de ME que seria responsável por demonstrar os estados do sistema, exibindo seu fluxo entre os estados desde sua inicialização até sua finalização. Este sistema é um exemplo e pode-se definir que tem como finalidade realizar o controle de tarefas a serem executadas. Ele seleciona as tarefas quando estiver no estado "ocioso", que após iniciar a execução da tarefa assume o estado de "executando", e ao final quando não houver mais nenhuma tarefas, é dado início ao processo de finalizando passando pelos estados de "finalizando" e "finalizado", encerrando assim seu ciclo de vida.

Suponhamos que o sistema será iniciado de tal maneira que é necessário iniciar todos os recursos necessários para sua utilização, podendo definir que encontra-se no Estado de "*INICIALIZANDO*". Nesse mesmo estado, está presente um comportamento de entrada definido como "*Entry : iniciar()*", o qual é executado para iniciar todos os recursos necessários para a

execução correta do sistema. Após ser inicializado o sistema pode assumir dois Estados que são "*EXECUTANDO*" ou "*OCIOSO*".

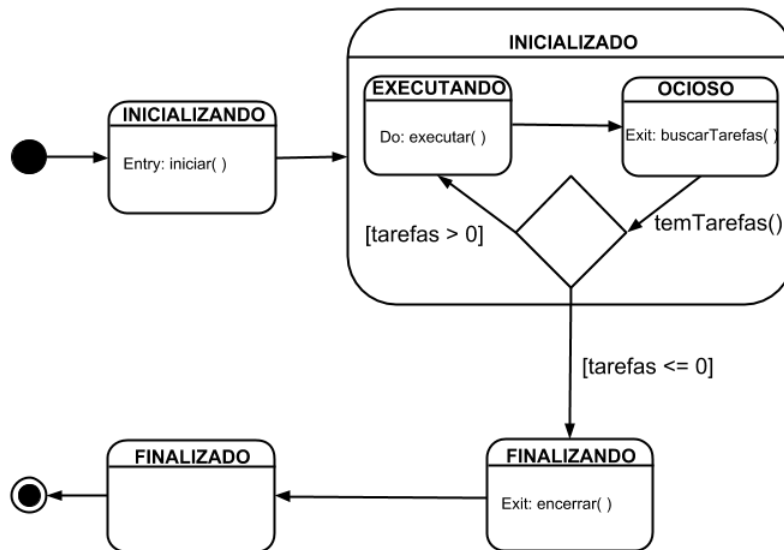


Figura 2.2: Exemplo de uma ME (Produzido pelo Autor).

O estado "*EXECUTANDO*" indica que umaa tarefa está sendo executada, conforme é exposto por meio da ação "*Do : executar()*". Enquanto o estado "*OCIOSO*" representa que o sistema não está executando nenhuma tarefa, mas antes de ocorrer a transição, ele executaria a ação de saída para buscar tarefas "*Exit : buscarTarefas()*", onde buscaria se há tarefas e caso haja retornaria para o estado de "*EXECUTANDO*". Ambos estados "*EXECUTANDO*" e "*OCIOSO*" podem ser abstraídos em um único estado que seria o estado composto "*INICIALIZADO*".

Quando não houver mais tarefas a serem executadas o fluxo seguirá para o estado de "*FINALIZANDO*", devido a condição de guarda "*[tarefas ¡= 0]*" ser satisfeita. No estado "*FINALIZANDO*", para que possa ocorrer a transição para o estado "*FINALIZADO*", tem que aguardar a ação de saída "*Exit : encerrar()*" ser finalizada, para que então seja encerrado o ciclo de vida do sistema.

2.2 Mudanças em Código Fonte

No contexto de nosso estudo, mudanças no código-fonte são alterações realizadas no código-fonte que mudam a estrutura entre versões ($V \Rightarrow V'$), podendo causar alterações no com-

portamento do sistema. Mudanças acontecem constantemente no código-fonte principalmente em sua fase de implementação e evolução [38] [13]. Normalmente, uma das formas de visualizações dessas mudanças é por meio de sistemas de controle de versões (CSV), no qual ficam armazenadas as diferenças entre as versões do código-fonte e rastreiam tais informações por meio de anotações informando se a linha foi adicionada ("*added*") ou foi removida ("*deleted*") [13].

Com o intuito de exemplificar o que são mudanças pode-se observar, nos Códigos-Fonte 2.1 que é a versão anterior, e 2.2 a versão atualizada, as mudanças que foram aplicadas, sendo essas destacadas em verde a nova versão e em vermelho a versão anterior. Dessa maneira, é possível visualizar e comparar para identificarmos as mudanças que ocorreram entre ambas versões. No código-fonte 2.1, a variável "*message*" foi removida na nova versão, além disso, a atribuição que antes era utilizava a variável "*message*" foi alterada passando a atribuir o valor para o parâmetro "*msg*", e por último o valor inserido na chamada no método "*show()*" também foi alterado, deixando de ser "*message*" e passando a ser "*msg*";

Código Fonte 2.1: Exemplo versão 1 do Código

```
1 void sendMessage(String msg) {
2     String message = msg;
3     message = message +
4         " :: " +
5         this.time();
6     show(message);
7 }
```

Código Fonte 2.2: Exemplo versão 2 do Código

```
1 void sendMessage(String msg) {
2     msg = msg +
3         " :: " +
4         this.time();
5     show(msg);
6 
7 }
```

Mudanças no código-fonte podem ser classificadas de diversas formas em diferentes níveis de abstração ou granularidade, de acordo com o que deseja ser analisado. Uma das definições a respeito de mudanças é o refatoramento, que é realizar uma mudança no sistema de tal modo que não altere o comportamento externo, ou seja a funcionalidade, mas que melhora a estrutura interna do código [15]. Entretanto, refatoramentos não são explorados, pois no contexto deste trabalho, optamos por focar mudanças de baixa granularidade, ou seja, a nível de linha ou elemento (atribuição, variável, chamada de método, entre outros) do código-fonte.

Além do refatoramento, há uma outra forma de definir as mudanças que acontecem no código-fonte que são definindo-as em mudanças de baixa granularidade. Fluri [14] criou uma classificação para esse tipo de mudança, sendo essa selecionada para utilização na abordagem proposta, pois com uma classificação mais detalhada, esperou-se ter melhores resultados com sua aplicação. Mudanças de baixa granularidade, diferentemente do refatoramento, são destinadas a um único elemento alterado em uma determinada classe. Para automatizar essa classificação, Fluri et al. [13] criaram uma ferramenta capaz de classificar as mudanças que ocorreram em diferentes versões do código-fonte, chamada de *ChangeDistiller*.

A *ChangeDistiller* recebe como entrada dois arquivos JAVA os quais são utilizados na geração de suas *Abstract Syntax Tree* (ASTs) para que em seguida possam ser realizadas as comparações entre ambas com o intuito de identificar quais mudanças no código-fonte ocorreram. A ferramenta compara as estruturas *ASTs* que foram geradas dos dois arquivos, verificando se pertencem a mesma classe e quais são as diferenças entre as *ASTs*, utilizando de cálculos de similaridade. Ao final, é criado uma lista de mudanças, a qual é utilizada para realização da classificação de acordo com cada tipo e elemento que sofreu a modificação.

Na Tabela 2.1 estão expostas, somente, as classificações que foram adotadas na abordagem, pois para o contexto são as que estão mais relacionadas a este estudo e que poderiam causar alterações em MEs. As demais mudanças, tais como inserções ou remoções de comentários ou javadocs, não foram incluídas na abordagem porque ao nosso entendimento não alterariam o comportamento ou elementos presentes na ME, ou não encontramos cenários reais onde causaram alterações, dessa forma nós decidimos por não incluí-las na técnica proposta.

Para exemplificar melhor como seria o resultado da utilização dessa classificação gerada pela *ChangeDistiller*, vamos assumir que a ferramenta recebeu como entrada os códigos-fonte 2.1 e 2.2 onde são a versão anterior e a versão com modificações, respectivamente. Após todos os processos para a classificação a ferramenta informa que houve quatro mudanças na estrutura do código, ver Tabela 2.2.

A primeira linha da Tabela 2.2 indica que houve uma inserção de uma chamada de método e a segunda que houve uma remoção, também, de uma chamada de método, onde podemos observar que são as mesmas declarações alterando somente o parâmetro da chamada. Entretanto, são consideradas como duas mudanças pela ferramenta, pois a ferramenta

não consegue identificar que era o mesmo elemento calculando sua singularidade. Em seguida, conforme a ordem da tabela, temos como saída a remoção de uma variável, pois a mesma não se encontra mais no corpo daquele método. Por último, temos a atualização de uma assinatura ou atribuição de valor, que anteriormente existia a variável "*message*" e foi substituída pelo parâmetro de entrada do método "*msg*";

Changed Entity	Change Type
Assignment	Statement Delete
Assignment	Statement Insert
Assignment	Statement Update
Method Invocation	Statement Delete
Method Invocation	Statement Insert
Method Invocation	Statement Update
IF Statement	Statement Delete
IF Statement	Statement Insert
IF Statement	Condition Expression Change
Else Statement	Alternative Part Insert
Else Statement	Alternative Part Delete
Method	Additional Functionality
Method	Removed Functionality
Method	Method Renaming
Return Statement	Statement Delete
Return Statement	Statement Insert
Return Statement	Statement Update
Field	Remove Value Enum
Field	Add Value Enum

Tabela 2.1: Classificações das mudanças em códifo-fonte da *ChangeDistiller* adotadas em nossa técnica.

Nós optamos por selecionar a *ChangeDistiller* por atender nossas necessidades quanto à identificação de mudanças que ocorreram entre uma versão e outra do sistema. Diferente-

mente de uma simples ferramente de Diff a qual normalmente exibe somente as linhas que sofreram modificações, mantendo o histórico de alterações [36], ela fornece uma classificação a qual foi útil para elaborar a abordagem proposta mantendo o foco na abordagem proposta.

Versão Anterior	Versão Nova	Elemento Mudado	Tipo Mudança
show(message);	show(msg);	METHOD INVOCATION	STATEMENT INSERT
show(message);	show(msg);	METHOD INVOCATION	STATEMENT DELETE
String message = msg;	–	VARIABLE DECLARATION STATEMENT	STATEMENT DELETE
message = message +"::"+ this.time();	msg = msg+ "::"+this.time();	ASSIGNMENT	STATEMENT UPDATE

Tabela 2.2: Exemplo do valores obtidos por meio da *ChangeDistiller* utilizando os códigos-fonte 2.1 e 2.2.

2.3 Recuperação de Informação

Recuperação de informação é o processo para recuperar informação de natureza não estruturada que satisfaça alguma necessidade de informação dentro de grandes coleções armazenadas [22] [30] [6]. Existem três processos básicos que um sistema de recuperação deve possuir: representação de documentos, representação da consulta e a comparação de ambas [19]. Dentre as métricas existentes para elencar os itens mais relevantes está a $TF \times IDF$ [6]. Onde TF é o valor de repetições de um termo T em um documento, enquanto IDF é o inverso da frequência de um termo T nos documentos [30], e com o resultado pode-se gerar um ranqueamento dos documentos mais relevantes ou relacionados a consulta.

A Tabela **??** exemplifica como é realizado o cálculo da métrica $TF \times IDF$. Dado que os termos Termo1, Termo2 e Termo3 repetem-se 8, 5 e 7 vezes em um documento, respectivamente, podemos ter o valor de TF, variando entre 0 e 1, como uma forma de suavizar o valor. Logo após, tem-se o cálculo do valor IDF, no qual assumimos que há um número total de 1000 documentos indexados na base de dados, em que os termos Termo1, Termo2 e

Termo3 aparecem em 235, 128 e 324 documentos diferentes, respectivamente. Por último, é realizado o cálculo da métrica $TF \times IDF$, multiplicando o resultado TF com o resultado do IDF.

	Doc		TF	IDF		TF.IDF
Termo1	8		8/8	$log_2(1000/235)$ = 2.09		2.09
Termo2	5		5/8	$log_2(1000/128)$ = 2.96		1,85
Termo3	7		7/8	$log_2(1000/324)$ = 1.62		1,41

Tabela 2.3: Exemplo do cálculo da métrica TF.IDF. (Produzido pelo Autor)

A utilização de Recuperação da Informação se deu devido à necessidade de ter uma técnica que contribuísse para a identificação de similaridades ou relações entre código-fonte e MEs, para que dessa forma pudéssemos obter melhores resultados. Além disso, pesquisadores já utilizam métodos de Recuperação da Informação para fazer o rastreamento entre documentos e código [43] [18]. Entretanto, estamos cientes de que problemas durante esse processo possam vir a surgir devido as MEs serem modelos muitas vezes gerados manualmente e com isso havendo a presença de linguagem natural na definição dos elementos, criando uma certa diferença entre como está presente no código-fonte e na ME.

Durante o processo de recuperação, todo documento necessita passar por um preprocessamento para que haja um tratamento adequado às palavras existentes, de modo que isso facilite os algoritmos que irão realizar a busca [30]. Um desses passos é o processo de remoção de *Stop Words* que são expressões ou palavras que podem interferir no processo de busca, na língua portuguesa seriam palavras tais como: “de”, “como”, “com”, “um”, “a”. Isso se deve à possibilidade de recuperar documentos irrelevantes aos termos presentes na busca, como também alterar os resultados da métrica utilizada para ranqueamento. Por exemplo, caso usássemos como entrada os termos T={ “Como”, “construir”, “uma”, “casa”}, todos os documentos que tivessem as palavras “uma” ou “como” seriam retornados, mesmo que não tivessem relevância alguma para o que está sendo buscado.

Após o pré-processamento dos documentos que serão utilizados para buscar as informações desejadas, é montado a consulta composta de palavras chaves que serão pesquisadas nos documentos. Essa consulta é formada por termos que o usuário deseja pesquisar em meio aos documentos que foram indexados ao sistema de Recuperação de Informação. Com base na consulta utilizada, será realizado o ranqueamento dos documentos que têm relação com os termos, com base nas suas relevâncias, seguindo a métrica $TF \times IDF$.

Para utilizarmos Recuperação de Informação em nosso estudo, optamos por selecionar a ferramenta *Lucene* da Apache, a qual foi desenvolvida em Java e é conhecida no mercado [2]. *Lucene* implementa os processos citados anteriormente a respeito do pre-processamento dos dados, como também a busca por meio de consultas informadas pelo o usuário e retorna como saída, os documentos com sua pontuação ou ranking. Além disso, a *Lucene* possui tipos de consultas que podem auxiliar na hora da busca pelos termos presentes na consulta, porém, para selecionar o tipo, faz-se necessário entender o funcionamento dos tipos que ela disponibiliza.

Capítulo 3

Trabalhos Relacionados

Na literatura são apresentadas diversas áreas nas quais as Máquinas de Estado Comportamentais (MEs) podem ser utilizadas, tais como geração de código na linguagem C por meio de modelos [12]; programação orientada por aspectos [45]; geração de testes por meio de modelos [4], dentre outros. No entanto, o contexto da presente pesquisa de Mestrado é para identificar e classificar as atualizações necessárias nas MEs, após mudanças no código-fonte. A seguir estão expostos trabalhos que possuem alguma similaridade ou relação com o tema desta pesquisa de mestrado.

A técnica proposta por Doungsa-ard et al. [11] JuiceGen, possibilita a geração de testes JUnit por meio da ME. Os testes gerados são sequências de ações que alteram os estados da máquina de estado. De acordo com os autores, utilizando Algoritmo Genético, eles conseguem ter uma cobertura acima de 95% das transições, por meio dos eventos, que são mapeados como métodos chamados nos teste.

Ainda na linha de geração de testes por meio dos modelos UML, há o trabalho de Alves et al. [5], o qual produz casos de testes por meio de transformações de modelos UML. A ferramenta produzida com esse trabalho, denominada MoBIT, consiste em realizar transformações de modelos especificadas em *Atlas Transformation Language* (ATL)[1]. Ao ter como entrada a ME, ele aplica transformações de ATL para que em seguida, possa ter como resultado um caso de teste baseado no comportamento do componente selecionado.

O trabalho desenvolvido por Paradkar [32] realiza a geração de uma ME para que possam ser criados casos de testes para o sistema. Seguindo um caminho um pouco diferente dos demais trabalhos citados anteriormente, Paradkar inseriu mutações no modelo com o intuito

de verificar se os testes gerados seriam capazes de identificar e eliminar os mutantes que foram inseridos.

Briand et al. [9], analisaram os relatórios de experimentos de testes, investigando a relação custo-eficácia de quatro critérios sobre cobertura de testes com ME: Todas as transições, todos pares de transições, todos os caminhos nas árvores de transições e, quando relevante, predicado completo. Para isso, eles utilizaram de mutações para poderem testar a abordagem. Com isso, conseguiram ter uma boa cobertura dos testes e detectar todas as falhas que foram inseridas no sistema. Christian Schwarzl e Bernhard Peischl [37], reforçam a ideia do elemento de condição não ser tão abordado na literatura. Ele sugere a geração de casos de testes com base nas MEs para descrever o comportamento de unidades eletrônicas de controle. Essa geração é controlada por uma abordagem sistemática e probabilística aplicada a um exemplo industrial.

Na pesquisa elaborada por Bedők et al. [7], eles utilizam do modelo matemático de redes de Petri para geração de máquinas de estado. Devido à semelhança de estados e transições, os autores utilizaram deste modelo para aplicar no âmbito da orientação a objetos (OO). Para isso, desenvolveram uma linguagem de marcação para auxiliar na geração da ME, a qual é necessária definir os eventos da rede previamente, para que possa identificar corretamente os eventos que irão criar as transições e detectar o comportamento da rede.

Walkinshaw et al. [41], desenvolveram uma ferramenta de engenharia reversa a qual produz uma ME como saída, com base em um arquivo de log da execução que é dado como entrada na técnica. A ME é gerada sem conhecimento do código-fonte do sistema, pois é por meio de um arquivo de log que ao ser interpretado pela técnica, a qual tem um padrão exposto no trabalho, é capaz de inferir os estados que o objeto selecionado pode alcançar. Esse método é utilizado principalmente para compreensão comportamental do sistema. Eles realizaram o estudo com base em um arquivo de log gerado por um sistema de controle de bombas hidráulicas. Entretanto, como os próprios autores afirmam, esse tipo de abordagem tem como foco prover uma compreensão comportamental do sistema para casos onde não se tenha acesso ao código-fonte.

Outro trabalho realizado por Walkinshaw e Hall [40], também no âmbito de inferência de ME, gera modelos de ME por meio de uma técnica baseada em Programação Genética [24]. Com essa nova técnica, eles não dependem mais de um algoritmo específico para inferir as

MEs. Para isso, assume-se que cada elemento contido nos rastro da execução (*trace*) é uma transição, e cada variável seguinte faz parte de um "conjunto de testes", que são utilizadas para inferir a ME por meio de Programação Genética e os resultados são acrescidos em uma lista de transições da ME gerada. Assim como sua outra pesquisa, a utilização desse tipo de abordagem tenta sanar problemas oriundos de falhas detectadas por testes, como também uma compreensão comportamental do sistema.

Seguindo o caminho de geração de MEs para análise e compreensão comportamental do sistema, há o algoritmo Gk-Tail o qual foi criado por Lorenzoli et al. [26]. O algoritmo Gk-Tail após receber como entrada o *trace* do sistema, interpreta a sequência de métodos que foram invocados, criando o fluxo de dados, em seguida realiza comparações para remover estados que possuem as mesmas transições, por meio de um *merge*, e ao final tem como saída as MEs. Porém, os próprios pesquisadores fizeram críticas quanto ao desempenho da técnica em outro trabalho [29], reportando que houve casos que demorou algumas horas para completar a execução, tornando-se inviável sua utilização para sistemas de larga escala e em rápida evolução. Mariani et al. [29], realizaram melhorias no algoritmo Gk-Tail conseguindo obter uma melhora significativa no desempenho e denominando o novo algoritmo de Gk-Tail+.

Um outro trabalho que também aborda a geração de MEs por meio de arquivos de log é um estudo produzido por Mariani e Pastore [28]. Nessa pesquisa, eles elaboraram uma técnica capaz de inferir MEs por meio de arquivos de logs gerados por testes, com o intuito de identificar falhas que se apresentavam no sistema. Após ter o modelo da ME gerado, eles comparavam com o arquivo do log na busca de encontrar anomalias ou diferenças entre o fluxo da ME e o log. Dessa forma, buscam reduzir os gastos com o tempo utilizado para identificar os erros nos sistemas, onde ao invés de analisarem os logs manualmente, a técnica elaborada os mostra as anomalias detectadas entre o modelo gerado e o arquivo de log inserido.

Ainda no âmbito de geração de ME, pesquisadores têm utilizados técnicas de Inteligência Artificial (IA [16] para obtenção dos modelos gerados por meio de sistemas *Black-Box*, tendo como entrada logs compostos por pares de entrada/saída. Há também estudo que utilizam técnicas de geração em busca de avaliar questões de segurança por meio das MEs utilizadas, como é o caso da pesquisada desenvolvida por Yoo e Shon [44].

Rástočný e Mlynčár [33] desenvolveram um trabalho que tem como objetivo manter os diagramas de Sequências atualizados, após mudanças no código-fonte. Eles buscam manter essa sincronização entre modelo e código-fonte de tal maneira que preserve as características e níveis de abstração definidos com base no Diagrama de Sequência utilizado. Por meio de análises estáticas tanto do código-fonte quanto do modelo, eles conseguiram demonstrar que conseguem prover uma automação no processo de atualização desse modelo na documentação do software. Eles desenvolveram uma pesquisa bastante relacionada com este trabalho de mestrado, entretanto, eles abordaram o problema de manter Diagrama de Sequência atualizado e em conformidade com o código-fonte.

Madari et al. [27] desenvolveram uma pesquisa que aborda a sincronização de um metamodelo de Sistema Visual de Modelagem e Transformação (VTMS) e código-fonte por meio de transformações. Nesse trabalho, eles elaboraram um processo incremental para a sincronização, onde por meio de metamodelos eles geram o código-fonte, e após isso realizam transformações para criação de um novo metamodelo. Entretanto, esse processo não é simultâneo, pois tanto o metamodelo quanto o código são alterados.

Segundo Walkinshaw e Hall [40], os modelos de inferência não determinísticos tem consequências quanto à falha na captura do fluxo dos dados, mas podem ser úteis para geração de testes. Em aspectos gerais, os trabalhos buscam inferir com maior corretude as MEs para conseguir obter uma compreensão e visão comportamental em relação a alguma funcionalidade do sistema. Dessa forma, detectamos uma falta de trabalhos que tenham como objetivo manter os modelos ME de um sistema sempre atualizados após a realização de mudanças no código-fonte.

Mesmo havendo diversos trabalhos que abordassem a utilização de MEs ou que até mesmo fizessem a geração de MEs, seja por engenharia reversa, técnicas de IA ou outro método, não foram encontradas técnicas similares. Em meio aos trabalhos encontrados, não foram identificados trabalhos que abordassem a atualização de MEs com base nas mudanças nos código-fonte, classificando ou sugerindo alterações a serem aplicadas nas MEs.

Capítulo 4

Abordagem Proposta

Este capítulo descreve as informações e conceitos necessários para compreender a abordagem que esta sendo proposta neste trabalho, contemplando os itens contidos ne abordagem. O capítulo está dividido em seções de tal maneira que cada uma trate de um tema específico. A seção 4.1 descreve sobre a classificação de mudanças que podem ocorrer no código-fonte e uma visão geral da técnica. A seção 4.2, aborda os conceitos sobre a taxonomia de alterações em MEs, que também está sendo proposta por este trabalho. E a seção 4.3, que contém as informações a respeito do algoritmo que está sendo proposto.

4.1 Classificação das Mudanças em Código-fonte

A abordagem proposta por esta pesquisa tem o intuito de classificar e sugerir atualizações nas MEs, a partir das mudanças que ocorreram no código-fonte. Para isso, inicialmente, foi realizado um estudo para identificar quais tipos de mudanças no código-fonte poderiam causar alterações nas MEs e quais seriam essas alterações, ver Capítulo 5.1. Por meio desse estudo, foi possível elaborarmos uma taxonomia para classificação de alterações em MEs, bem como um algoritmo para automatizar a classificação das alterações na ME com base nas mudanças do código-fonte, e sugerir essas alterações ao usuário.

Para compreender como é composto esta abordagem, a Figura 4.1, expõe na visão de componentes, como os elementos estão conectados. Inicialmente, os componentes "Extrator de Mudanças em Código-fonte" e "Extrator de Elementos em ME" recebem as versões antiga e nova do Código-fonte do projeto, e a ME a ser analisada, respectivamente. Com

essas informações, eles vão obter as mudanças que ocorreram no código-fonte e os elementos presentes na ME, utilizando de uma biblioteca para leitura do XML, e da ferramenta *ChangeDistiller*.

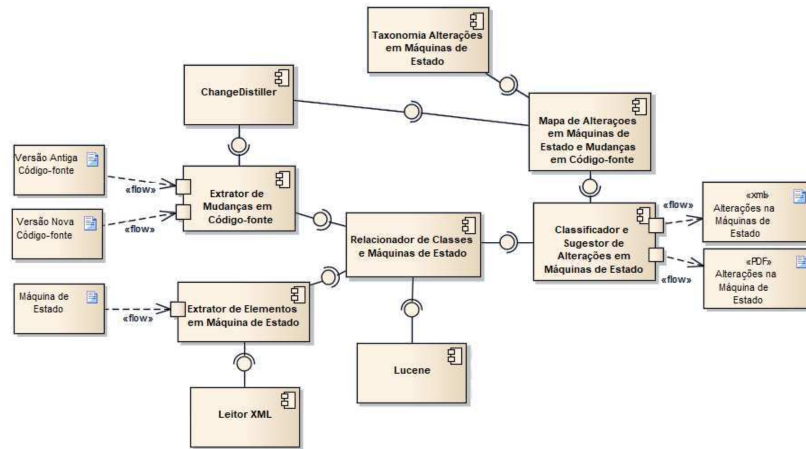


Figura 4.1: Visão geral em Componentes da técnica proposta. (Produzido pelo Autor)

Após isso, a técnica tenta prover um relacionamento entre as classes que tiveram mudanças e os elementos presentes na ME, utilizando a ferramenta Lucene[1], para que em seguida possa iniciar o processo de classificação e sugestão. Ao final, com as classes que contêm mudanças e estão relacionadas as MEs, o componente "Classificador e Sugestor de Alterações em ME" realiza o processo de classificação. Para isso, ele utiliza do Mapeamento gerado no Estudo 5.1, que é composto pela taxonomia de Alterações em ME e a classificação das mudanças presentes na *ChangeDistiller*, para emitir como saída as alterações sugeridas na ME, podendo variar entre o formato XML ou PDF.

Conforme pode ser observado, foi necessário ter uma ferramenta que provesse a classificação das mudanças em código-fonte, porque isso auxiliaria no processo de classificação de alterações nas MEs, logo, foi realizada uma busca por trabalhos e pesquisas voltadas para classificação de mudanças em código-fonte. Com isso, foi encontrada a ferramenta chamada *ChangeDistiller* [13], que classifica mudanças de baixa granularidade realizadas entre versões do código-fonte e atende aos requisitos necessários para nossa técnica.

A classificação da *ChangeDistiller* é realizada por meio da comparação de dois arquivos JAVA, onde, inicialmente é realizada uma comparação dos nomes das classes. Em seguida, é criado uma *Abstract Syntax Tree* (AST), a qual por meio dos nós e folhas é realizado

[1]http://lucene.apache.org/

todo o processo de classificação das mudanças que foram realizadas no código-fonte. Com essa classificação, é possível obter como saída os tipos de mudanças no código-fonte e seus respectivos elementos que sofreram mudanças. E então, em posse desses resultados, nós os utilizamos na nossa técnica para identificar e sugerir possíveis alterações nas MEs, para mais detalhes sobre a classificação e procedimentos (ver Seção 2.2).

Utilizando da classificação disponibilizada pela *ChangeDistiller*, realizamos um estudo 5.1 com o objetivo de identificar quais tipos de alterações em MEs podem ocorrer com base nas mudanças de código-fonte, ver Tabela 2.1. Com esse estudo, foi possível detectar quais mudanças no código-fonte podem causar alterações nas MEs. Além disso, foi identificado que era possível realizar um mapeamento entre as mudanças no código-fonte a as alterações a serem realizadas nas MEs.

Desse modo, foi detectado uma necessidade de uma padronização ou classificação para as alterações a serem realizadas em ME, e por não ter encontrado taxonomia a respeito de alterações em ME, optamos por propor uma. Com base nos elementos de ME que fazem parte do escopo deste trabalho, estado e transição, foi elaborada e proposta a taxonomia de alterações em MEs. Além disso, essa taxonomia proposta, foi utilizada para o desenvolvimento do algoritmo e do mapeamento que também compõem esta abordagem proposta.

4.2 Taxonomia das Alterações em MEs

Em busca de pesquisas e trabalhos que propusessem algum tipo de taxonomia a respeito de alterações em ME, com o intuito de aplicarmos em nossa técnica, não encontramos nenhum tipo de taxonomia que abordassem essa temática. Propomos então uma taxonomia que classifica alterações em MEs, inicialmente, com foco nos elementos de estado e transição, pois pode-se afirmar que são os principais elementos que compõe uma ME, mas há a possibilidade de expansão para os demais elementos.

Estado	
Tipo de Alteração	**Descrição**
Create a new State	Quando um novo Estado é criado e adicionado à Máquina de Estado
Remove a State	Quando um Estado é removido da Máquina de Estado
Rename a State	Quando um Estado é renomeado na Máquina de Estado
Create a new EntryAction	Quando uma nova EntryAction é criada e adicionada no Estado
Remove EntryAction	Quando uma EntryAction em um Estado é removida
Rename EntryAction	Quando uma EntryAction em um Estado é renomeada
Create a new ExitAction	Quando uma nova ExitAction é criada e adicionada à um Estado
Remove ExitAction	Quando uma ExitAction em um Estado é removida
Rename ExitAction	Quando uma ExitAction em um Estado é renomeada
Create a new DoAction	Quando uma nova DoAction é criada e adicionada à um Estado
Remove DoAction	Quando uma DoAction em um Estado é removida
Rename DoAction	Quando uma DoAction em um Estado é renomeada
Alter Composite State's body	Quando qualquer alteração acontece dentro de um Estado Composto
Create a new Composite State	Quando um novo Estado Composto é criado e adicionado à Máquina de Estado
Remove Composite State	Quando um Estado Composto é removido da Máquina de Estado

Tabela 4.1: Classificação das alterações em MEs para o elemento de Estado e Estado Composto.

Transition	
Type of Change	**Description**
New Transition	Quando uma nova Transição é criada e adicionada à Máquina de Estado
Remove Transition	Quando uma Transição é removida da Máquina de Estado
New Event	Quando um Evento é criado e adicionado à Transição
Remove Event	Quando um Evento é removido da Transição
Rename Event	Quando um Evento na Transição é renomeado
New Action	Quando uma Ação é criada e adicionada à Transição
Remove Action	Quando uma Ação é removida da Transição
Rename Action	Quando uma Ação na Transição é renomeada
New Guard	Quando uma Guarda é criada e adicionada à Transição
Remove Guard	Quando uma Guarda é removida da Transição
Update Guard	Quando a Guarda é atualizada na Transição

Tabela 4.2: Classificação das alterações em MEs para o elemento de Transição.

Considerando os elementos contidos na ME, tendo como foco os elementos de Estado e Transição, o próprio pesquisador fez um levantamento de cenários de alterações em MEs. Além disso, também foi feito uma revisão sobre a sintaxe e semântica dos elementos contidos em MEs. Com base nessas informações, identificamos as alterações que podem ser realizadas na ME. Desse modo, pode-se obter cenários mais representativos de alterações que ocorrem em MEs sendo validados com os resultados obtidos do Estudo Exploratório, ver Capítulo 5. Portanto, após esses procedimentos, foi elaborada a taxonomia proposta, tendo como objetivo auxiliar os usuários e outros trabalhos que possam utilizá-la, sobre que tipo de alteração deve ser feito na ME.

Nas Tabelas 4.1 e 4.2, estão descritos os tipos de alterações que podem ser aplicadas às MEs de um projeto real, com o foco nos elementos de Transição e Estado. A taxonomia proposta, abrange inclusões, remoções e também atualizações que possam ser aplicadas à ME. Além disso, também possuem descrições para auxiliar o entendimento do que deverá ser feito para realizar determinada alteração. Dessa forma, espera-se que ao utilizar a taxonomia

proposta, esteja contido informações a respeito do que deverá ser replicado na ME.

Para exemplificar a classificação da taxonomia que está sendo proposta, analisaremos as Figuras 4.2 e 4.3. As MEs expostas nessas figuras, estão modelando o que seria um sistema responsável por controlar a "luz" ou lâmpada, de um quarto. Inicialmente apresenta-se com somente dois estados "Apagada" e "Acesa", o que em nosso cotidiano é o mais comum. Entretanto vamos supor, que um novo nível de luminosidade tenha sido acrescentado a esse sistema, o qual definiríamos como "Média". Esse novo nível de luminosidade seria um novo Estado que poderia ser alcançado no momento que a luz estaria apagada (Estado "Apagada"), e ao receber algum evento diferente iria assumir o novo Estado "Média".

Para informar os procedimentos ou alterações que necessitariam ser feitas na ME, ver Figura 4.2, utilizaremos como base a taxonomia que está sendo proposta. Quando comparamos as duas versões da mesma ME, podemos notar que novos elementos foram acrescidos em sua estrutura, nós podemos classificar que houve um *"Create a new State"*, o qual é definido pelo no "Média". Em seguida, para que esse novo Estado seja alcançado, é necessário que haja também novas transições que levem o fluxo até esse novo Estado, como também que a partir dele, possa seguir seu fluxo até o seu final. Para atender essas necessidades, a taxonomia permite que classifiquemos que haja a adição de novas Transições por meio do tipo *"New Transition"*.

Figura 4.2: Exemplo de uma versão anterior da ME (Produzido pelo Autor).

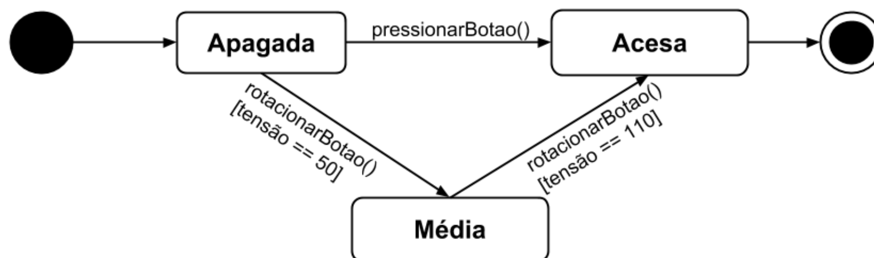


Figura 4.3: Exemplo de uma versão atualizada da ME (Produzido pelo Autor).

Com base nas Figuras 4.2 e 4.3, ainda é possível destacar outros tipos de alterações que foram aplicadas, tais como "New Event" e "New Guard". Para que o estado "Média" fosse

um estado alcançável, foram adicionadas as transições, mas além disso, podemos detectar outros elementos presentes que são os eventos e guardas, os quais definem quais eventos são aguardados para as transições serem acionadas, e as condições que devem ser atendidas para alterar os estados.

A criação da taxonomia, foi necessária para que pudéssemos desenvolver nossa técnica, pois é por meio dela que informamos ao usuário, no caso o desenvolvedor, o que deverá ser feito na MEs para que a mesma fique em conformidade com a nova versão do código-fonte. Além disso, também utilizamos-a para realizar o mapeamento entre mudanças no código-fonte e alterações nas MEs (ver Seção 5.1), o qual deu origem ao nosso algoritmo e ferramenta[2] para classificação de alterações em MEs por meio de mudanças no código-fonte.

4.3 Algoritmo para Classificação de Alterações em MEs

O algoritmo para classificação de alterações em MEs é o *core* de toda a técnica, pois o mesmo contempla o objetivo principal, que é a classificação e sugestões das alterações em MEs. Nessa parte da abordagem, o algoritmo recebe as classes JAVA que estão relacionadas à ME com suas respectivas mudanças, e com base no mapeamento gerado (ver Seção 5.1), ele realiza as comparações para classificar e logo após sugerir as alterações ao usuário.

Inicialmente, o algoritmo precisa receber como entrada a ME a ser analisada, em formato XML, para que possam ser extraídos seus elementos. Também são dados como entrada os arquivos contendo a versão anterior e a versão recente do código-fonte Java, para que possam ser classificadas as mudanças no código-fonte, por meio da *ChangeDistiller*. Havendo mudanças classificadas pela *ChangeDistiller*, os arquivos que tiverem mudanças são separados para que em seguida seja feita uma relação entre os elementos da ME e os arquivos que tiveram mudanças. Na presença dos arquivos com mudanças no código-fonte e que estão relacionadas com a ME, é realizado o processo de classificação das alterações a serem aplicadas na ME, levando a emitir como saída estas alterações em um formato XML ou PDF, esse último similar a um relatório.

A Figura 4.4, apresenta uma visão geral de todas as etapas realizadas pela abordagem proposta, desde o que recebe como entrada, até as suas possíveis saídas. Com o intuito de

[2]https://github.com/MatheusOliveiraBarbosa/changeFSM

agregar mais uma contribuição a este trabalho, foi desenvolvido o código-fonte 4.1, possibilitando a compreensão de como encontra-se modularizado o algoritmo. Desse modo, outras pessoas podem analisar o algoritmo e contribuir na melhoria da abordagem proposta refinando-o com novos conhecimentos.

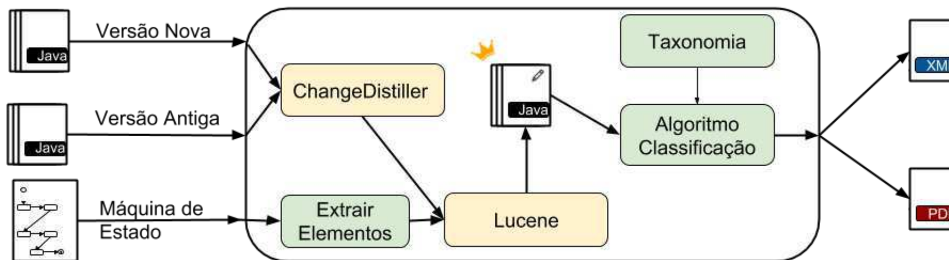


Figura 4.4: Visão geral da técnica proposta (Produzido pelo Autor).

Código Fonte 4.1: Overview do Algoritmo Proposto

```
1  Input:
2      Old Version Code as OldProjet
3      New Version Code as NewProject
4      StateMachine as SM
5  Output:
6      SuggestionUpdatesME (PDF or XML)
7  Begin classifyUpdateOnSM(OldProject, NewProject, SM):
8      for each element e in SM, do:
9          if(e.getType() == state), do:
10             allSMElements.add(e);
11         if(e.getType() == transition), do:
12             allSMElements.add(e);
13     Array[SourceCodeChange] classifiedCodeChanges =
14         ChangeDistiller.getClassifiedCodeChanges(OldProject, NewProject);
15     Set [ClassAndChanges] classesAndChanges =
16         getClassesAndChanges(classifiedCodeChanges);
17     Set [CandidateCodeClass] candidateCodeClasses =
18         mappedClassesWithStateMachine(classesAndChanges, allSMElements);
19     Set [SMUpdate] SMUpdates =
20         classifySMUpdates(allSMElements, candidateCodeClasses);
21     return generateSuggestUpdatesOnSMReport(SMUpdates);
22 End.
```

O código-fonte 4.1 expõe de uma maneira geral quais são os procedimentos realizados para obter a saída desejada com as sugestões de alterações a serem realizadas nas MEs. Inicialmente, é feita a extração dos elementos presentes no XMI (linhas 8-12), parte essencial para dar continuidade aos passos seguintes, pois é nela, em que são obtidos os elementos presentes nas MEs que serão utilizados a seguir. Ao ser feita a inserção do arquivo XMI referente à ME, o algoritmo procura pelas palavras chaves e seus respectivos valores os quais estão presentes nos arquivos, para que em seguida possam ser transformados em objetos JAVA. Dessa forma, o algoritmo obtém as informações necessárias, tais como nome, eventos, guardas, entre outras, para que sejam utilizadas nos procedimentos que são realizados nos procedimentos seguintes.

Em seguida, com a ajuda da ferramenta *ChangeDistiller* é realizada a classificação das mudanças presentes no código-fonte (linhas 13-14). As linhas 13-14, expõem de uma forma abstrata como estão sendo obtidas as mudanças no código-fonte entre uma versão e outra dos projetos, pois é necessário inserir como entrada ambas versões de um arquivo JAVA por vez. Isso ocorre devido a *ChangeDistiller* realizar comparações de arquivos, ao invés de projetos (diretórios), estando esse detalhe implementado na técnica. Conforme as mudanças do código-fonte vão sendo classificadas, elas vão sendo adicionadas em uma lista de *SourceCodeChange*, para que em seguida sejam verificadas quais foram as classes que tiveram mudanças, e então é criada uma lista com classes JAVA e suas respectivas mudanças (linhas 15-16).

As linhas 15-16 são responsáveis por transformar a lista passada como parâmetro no método *getClassesAndChanges()* em uma lista de elementos do tipo *ClassAndChanges*. Esse objeto *ClassAndChanges* é composto pelo arquivo que contém as mudanças, e suas respectivas mudanças que foram classificadas pela *ChangeDistiller*. Desse modo, as informações contidas nos arquivos poderão sem indexadas na ferramenta Lucene, para realizar um processo de Recuperação de Informação.

Em seguida, após obter as classes e suas respectivas mudanças em uma lista, o algoritmo utiliza da ferramenta *Lucene*, indexando em forma de documento todo o conteúdo das classes, para procurar pelos termos que forem inseridos na consulta. Na seleção dos termos que são utilizados na consulta, são selecionados os nomes dos Estados e suas respectivas ações, como também os termos presentes nas Transições.

Para exemplificarmos a aplicação da Lucene em nosso contexto, a Figura 4.5 expõe isso de uma maneira abstrata. Primeiramente, a Lucene indexa os arquivos dos códigos-fonte do sistema, para que possam ser realizados as consultas pelos termos definidos. Os termos presentes na consulta são extraídos da ME inserida como entrada, extraindo os valores contidos em seus estados e transições, tais como nomes, eventos, ação, guardas e ações de estado. Após os termos serem selecionados e inseridos na consulta, são obtidos os arquivos indexados que tiveram melhor ranqueamento, por meio do cálculo de TF-IDF, indicando quais são os arquivos JAVA que têm maior relação com a ME a ser analisada.

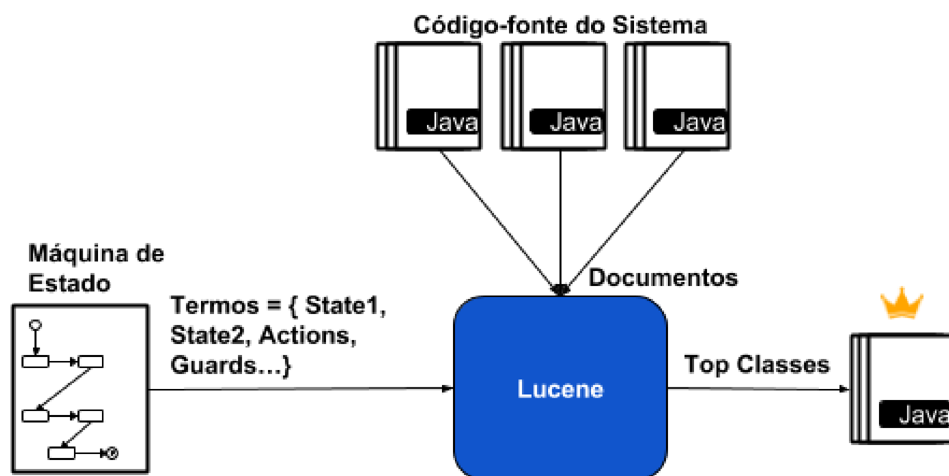


Figura 4.5: Exemplo da utilização do Lucene (Produzido pelo Autor).

Feito isso, com base nas classes que tiveram um melhor ranqueamento, é gerado uma nova lista com as classes que podem estar relacionadas à ME. Em seguida, (linhas 19-20), utilizando das classes que possivelmente estão relacionadas à ME, o algoritmo realiza o processo de classificação das alterações em MEs, com base nas mudanças contidas em cada classe que foi relacionadas.

O pseudocódigo 4.2, expõe como é realizado o processo de classificação das alterações em uma ME. Após receber como parâmetros a lista dos elementos da ME e as classes que estão relacionadas à ME (linha 1), o algoritmo percorre todas as mudanças do código-fonte de cada classe (linha 3). Em seguida, é verificado o tipo de elemento que sofreu a mudança no código-fonte (linha 8), seguindo o mapeamento gerado no estudo 5.1, para identificar se o mesmo pode causar a alteração na ME. Caso o elemento possa causar alteração na ME, é analisado há presença de algum termo da ME em seu corpo, havendo a presença do termo, é

realizada a classificação com base no elemento que foi alterado (linhas 9-11), e em seguida adicionado à lista que contém todas as alterações em ME (linha 13).

Por último, a linha 21 corresponde à saída da técnica a qual tem a possibilidade de emitir duas saídas, sendo elas em formato XML ou PDF. Uma das possibilidades é a geração de um arquivo PDF com formato de relatório para que possa auxiliar o desenvolvedor a identificar e realizar as alterações sugeridas de acordo com as mudanças que ocorreram no código-fonte. A outra, é a geração de um arquivo XML, que poderá ser utilizado por alguma outra técnica que faça a interpretação dos valores presentes.

Esse arquivo XML segue um meta-modelo, Figura 4.6, criado pelo autor da pesquisa com o intuito de auxiliar na compreensão e identificação dos elementos presentes na saída. O meta-modelo contempla os elementos que estão contidos no XML, e seus respectivos atributos. Além disso, expõe os tipos de mudanças no código-fonte aplicados na abordagem proposta, como também os tipos de alterações da taxonomia proposta, explicando como é composto o elemento de atualização da ME, denominado *updateSM*.

Código Fonte 4.2: Implementação do método de classificação de alterações em ME

```
1  classifySMUpdates( CandidateCodeClass ccc, allSMElements elementsSM) {
2      Set[SMUpdate]         SMUpdates;
3      for each change in ccc.getChanges(), do:
4          SMUpdate smUpdate;
5          smUpdate.change = change;
6          smUpdate.SM = elementsSM.getNameSM();
7          smUpdate.class = change.getClass();
8          switch (change.getElement() ){
9                  case(IF), do:
10                         smUpdate.kind = classifyUpdateByIFChanges
11                                        (change, elementsSM);
12                         if(smUpdate.kind != 'Nothing'), do:
13                             SMupdates.add( smUpdate ) ;
14               ...
15     return SMUpdates
```

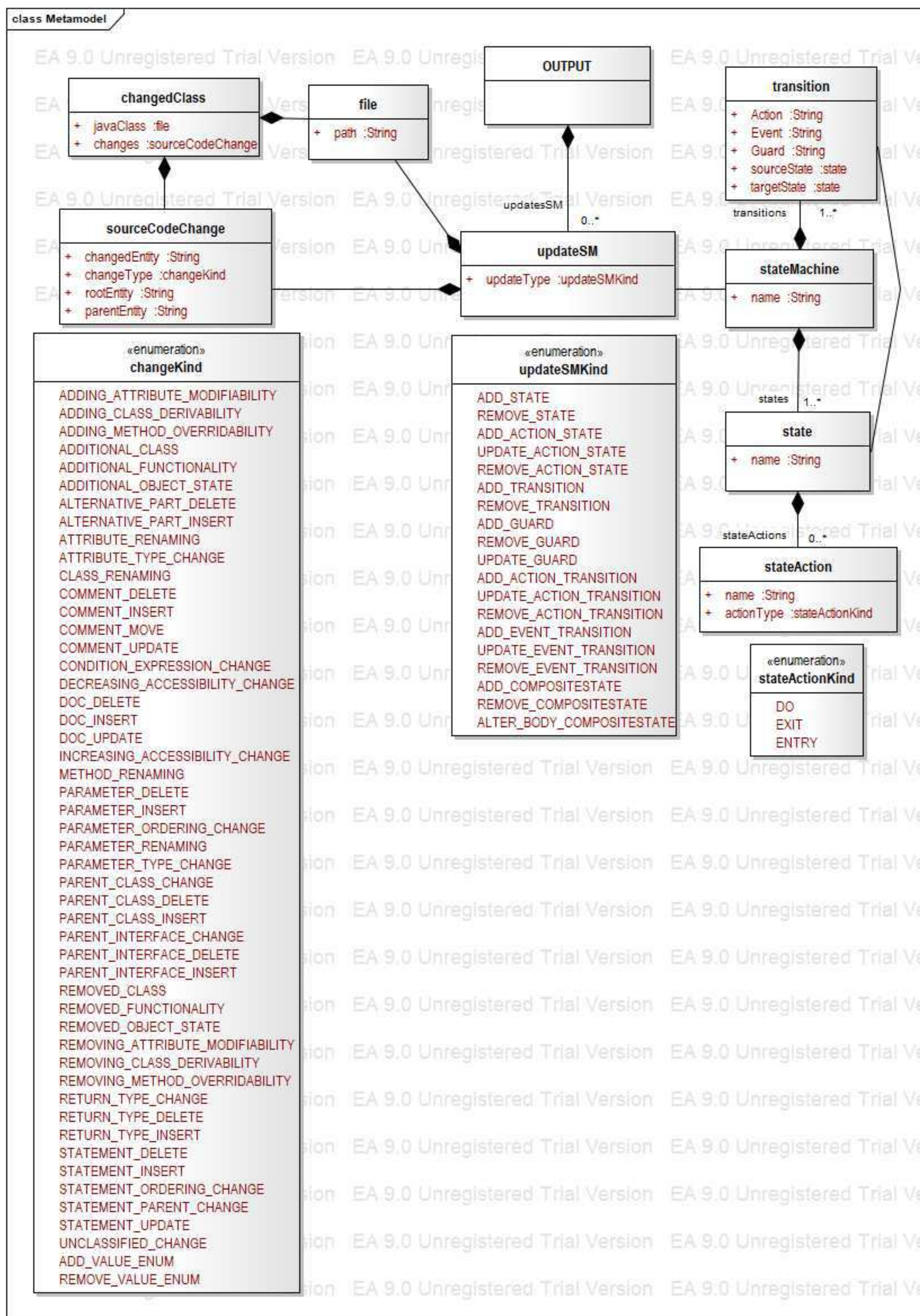


Figura 4.6: Metamodelo da saída gerada pela técnica.

Capítulo 5

Estudo Exploratório

Neste capítulo, estão descritas as informações a respeito do estudo exploratório realizado neste trabalho, cujo o intuito foi identificar quais os tipos de mudanças no código-fonte que poderiam causar alterações em MEs e quais seriam essas alterações. Para isso, foi necessário a participação de voluntários que dispuseram-se a contribuir com o estudo, como também a utilização de sistemas com código-aberto para avaliação e validação do estudo.

5.1 Mudanças no Código-fonte que causam Alterações nas MEs

Sabe-se que as MEs podem representar as características, comportamentos e configurações que o sistema possa ter [3]. Logo, durante o processo de implementação de um sistema é comum que haja modificações no código-fonte, sejam elas por meio de adições de novas funcionalidades[38], ou alteração do comportamento de alguma funcionalidade[15]. Havendo essas mudanças no código-fonte, busca-se identificar se tais mudanças podem causar alterações nas MEs, e quais seriam essas alterações.

Com o objetivo de identificar quais tipos de mudanças no código-fonte podem causar alterações nas MEs, decidimos realizar esse estudo exploratório. Logo, para que pudéssemos alcançar esse objetivo, foi feito levantamento de duas questões de pesquisa a serem respondidas com base nos resultados, as quais são estas:

- **QP1** - Um tipo de mudança no código-fonte, pode causar mudanças em diferentes

elementos nas Máquinas de Estados Comportamentais?

H_0 - Nenhum tipo de mudança pode causar mudanças em diferentes elementos da Máquina de Estado Comportamental;

H_1 - Um mesmo tipo de mudança pode causar mudança, somente, em um único elemento na Máquina de Estado Comportamental;

H_2 - Um mesmo tipo de mudança pode causar diferentes mudanças em mais de um elemento da Máquina de Estado Comportamental.

- **QP2** - É possível fazer um mapeamento entre as mudanças que ocorrem no código-fonte e as mudanças que ocorrem nas Máquinas de Estado Comportamentais?

  H_0 - Não é possível fazer um mapeamento entre as mudanças no código-fonte e as mudanças nas Máquinas de Estado Comportamentais;

  H_1 - É possível fazer um mapeamento entre as mudanças no código-fonte e as mudanças nas Máquinas de Estado Comportamentais.

Para realização deste estudo, foi utilizado um repositório de projetos com código-aberto chamado de Github[1], devido a ser um dos mais populares repositórios de código-aberto e conter vários projetos disponíveis, facilitando o acesso aos dados utilizados. No Github foram selecionados quatro projetos, sendo o último excluído, seguindo os critérios de inclusão e exclusão explicitados na seção5.2, que são os seguintes projetos: (i) Smarthome[2], um framework desenvolvido pela Eclipse para produtos domésticos inteligentes; (ii) Design Pattern Indeepth[3], um projeto cujo objetivo é ensinar e estimular o conhecimento juntamente com o cérebro do desenvolvedor a utilizar padrões de projetos em suas aplicações, por meio dos exemplos presentes no código-fonte; (iii) DESystem[4], um projeto cuja a finalidade é simular as ações presentes em um elevador, envolvendo motores, portas, sensores, lâmpadas e demais elementos presentes em um elevador; e (iv) ALeTrainSystem[5], um sistema autónomo para trens de LEGO Mindstorm, sendo esse removido seguindo o critério de exclusão.

[1]https://www.github.com

[2]https://github.com/eclipse/smarthome

[3]https://github.com/taoning2014/design-pattern-indeepth

[4]https://github.com/Liso/DESystem

[5]https://github.com/henrihs/ALeTrainSystem

Dentre os 4 projetos que foram selecionados, os 3 últimos listados, foram extraídos de uma base de dados com mais de 90 mil links para modelos ou diagramas, disponibilizada por um trabalho realizado por Gregorio Robles [34], enquanto, o projeto do Smarthome, foi localizado por meio do motor de busca do próprio Github.

5.2 Seleção das Unidades Experimentais

Para o processo de inclusão e exclusão dos projetos, foram escolhidos critérios para seleção das unidades experimentais mais adequadas para a realização do experimento. Os critérios de inclusão foram: (i) O código-fonte do sistema ser Java; (ii) Haver MEs disponíveis; (iii) Haver controle de versão nas MEs, ou seja, haver alterações nas MEs durante o processo de implementação, sendo esse o critério principal; (iv) As MEs estarem relacionadas ao código-fonte; E (v) Ser código aberto.

Por meio de um procedimento manual para analisar as relações entre código-fonte e as MEs (ver Seção 5.3), foi definido o critério de exclusão: não ser possível, para o condutor deste experimento, identificar quais partes do código-fonte estão sendo modeladas pelas MEs. Isso foi definido após a análise do código-fonte, não ser possível identificar quais partes do código-fonte foram modeladas pelas MEs. Portanto, reduzindo o risco de identificar equivocadamente quais mudanças no código-fonte podem ter causado as alterações nas MEs.

Uma vez definidos os projetos que seriam utilizados no estudo, foram realizadas duas análises para identificação e classificação das mudanças, isso aconteceu para tentar evitar que mudanças no código-fonte não fossem detectadas. Dentre as duas análises, uma ocorreu de maneira automatizada, por meio da ferramenta ChangeDistiller [13], e a outra, de maneira semi-automática com utilização das ferramentas DiffNow[6] e Github[7], por meio de suas ferramentas de diff.

Para realização destas análises, foram selecionadas duas versões dos projetos por meio dos históricos de commits do Github. Essa seleção se deu por meio de uma análise manual buscando entre versões mais antigas do código-fonte, onde havia sido feitas alterações tanto nas MEs quanto no código-fonte, de modo que houvesse alguma relação entre a mudança no

[6]https://www.diffnow.com/

[7]https://github.com/

código-fonte e a alteração presente na ME.

5.3 Tratamento dos Dados

Após selecionar as duas versões do código-fonte para cada projeto, nas quais teriam mudanças no código-fonte e alterações nas MEs, foram realizadas duas análises para identificar, classificar e contar as mudanças presentes de uma versão para outra. Sendo uma automática, e a outra análise, realizada pelo pesquisador deste experimento, que também é o condutor.

Primeiramente, foi feita uma análise automática, por meio da ferramenta ChangeDistiller, para obter um número aproximado de quantas mudanças foram realizadas e uma visualização das classificações das mudanças no código-fonte que ocorreram. Na Tabela 5.1, está contido o número de mudanças que foi obtido de cada projeto inicialmente, e depois pelos processos de filtragens realizados pelo pesquisador.

Nome Projeto	Qtd Mud. Inicial	Qtd Mud. 1ª Filt.	Qtd Mud. 2ª Filt.
DesignPattern Indeepth	22	22	6
DESystem	1030	912	75
Smarthome	2659	2331	2

Tabela 5.1: Quantidade de mudanças no código-fonte por filtragem. Qtd = Quantidade; Mud = Mudanças em Código-fonte; Filt = Filtragem.

Com essa quantidade de mudanças presentes em todas os projetos (ver Tabela 5.1, coluna "Qtf Mud. Inicial"), isso tornaria o estudo inviável devido à participação de voluntários. Portanto, foi realizada uma primeira filtragem nos tipos de mudanças almejando reduzir essa quantidade. Inicialmente, foram removidas as mudanças relacionadas aos comentários e javadocs presentes no código-fonte, pois não deveriam afetar o comportamento do sistema, entretanto, ainda permaneceu um número elevado para o estudo (ver Tabela 5.1, coluna "Qtf Mud. 1º Filt.").

Com esse número elevado de mudanças mesmo após a primeira filtragem, foi realizada uma segunda filtragem, essa de maneira semi-automática, dando prosseguimento à análise

manual para identificação das possíveis mudanças que estariam relacionadas às MEs. Essa segunda filtragem se deu com a ajuda da ferramenta, DiffNow[8], na qual após a identificar as classes que estavam sendo modeladas pelas MEs, o pesquisador identificou possíveis mudanças que poderiam causar alterações nas MEs. Isso foi possível, pois estava sendo utilizadas ambas versões das MEs para realizar uma associação entre o que está sendo modificado entre uma versão e outra do código-fonte.

Tendo as duas versões das MEs disponíveis, o pesquisador, deste estudo, teve a viabilidade para comparar o que havia sido alterado na ME e o que havia sido alterado no código-fonte, com o intuito de detectar relação entre código-fonte e ME. Em alguns dos casos, foi possível realizar uma associação direta, pois havia comentários indicativos no código-fonte referente às MEs. Entretanto, houve outros casos em que foi necessário compreender o código-fonte, para ter maior segurança no momento de afirmar que determinada mudança no código-fonte pode ter causado a alteração na ME. Para mitigar afirmações equivocadas de mudanças que poderiam ter causado alterações nas MEs, foram desconsideradas as que causaram indecisões no momento de seleção, do pesquisador.

Após as etapas de filtragem, o pesquisador selecionou as mudanças que poderiam causar as alterações nas MEs, resultando em um total de 83 mudanças no código-fonte de todos os projetos, ver última coluna na Tabela 5.1, e formulando assim nosso oráculo para verificação das respostas. Esse oráculo foi gerado, com base nas diferenças entre as versões das MEs e os código-fontes, buscando mitigar erros nesse processo. Além disso, foi possível reduzir o número de mudanças, possibilitando a realização do estudo, visando uma distribuição igualitária do número de mudanças no código-fonte, para todos os participantes, evitando um número exorbitante de mudanças no código-fonte a serem analisadas pelos participantes.

Vale ressaltar que embora o pesquisador tenha realizado essa filtragem para selecionar as mudanças que poderiam causar alterações nas MEs, foi necessário a solicitação de voluntários para verificarem se determinada mudança pode ter causado ou não alteração. Portanto, foram convidados participantes para analisarem as mudanças selecionadas pelo pesquisador. Os participantes tinham que obedecer a dois requisitos que são: ter conhecimento sobre MEs e ter experiência de desenvolvimento com a linguagem JAVA. Para serem chamados a participar deste estudo, foram enviados e-mails à listas de universidades, e mensagens

[8]https://www.diffnow.com/

em canais de comunicação em laboratórios de pesquisa e desenvolvimento. Os participantes tinham como finalidade informar se determinada mudança no código-fonte causaria alteração na ME, o que deveria ser alterado na ME e o porquê ele afirma isso.

Portanto, as mudanças no código-fonte, as quais foram selecionadas pelo pesquisador, foram divididas aleatoriamente para os participantes. Foram dividas aleatoriamente três mudanças, de uma mesma classe JAVA, para cada participante, havendo ressalva para um caso específico. Esse caso ocorreu quando a classe Java teve menos que três mudanças selecionadas, desso modo o participante ficou alocado com duas classes, respeitando o limite máximo de três mudanças por participante.

Essa divisão de no máximo três mudanças a serem analisadas por participante, ocorreu após a realização de um estudo piloto. Por meio desse estudo piloto, foi possível constatar, por meio de *feedbacks*, que três mudanças para serem analisadas demandou aproximadamente 30 minutos, e caso houvesse uma quantidade maior de mudanças a serem analisadas, poderia tornar a atividade cansativa, passível de abandono e de respostas onde o objetivo seria responder sem realizar os que se era pedido.

5.4 Treinamento e Execução

Inicialmente, foi enviado um convite para obtenção de participantes para o estudo. Esse convite, continha informações sobre as atividades que seriam realizadas e indicavam que o participante deveria ter conhecimento prévio sobre MEs, para que houvesse certo nível de conhecimento sobre o tema. Ao final do prazo de resposta, foi obtido um total de 27 participantes que se voluntariaram, podendo ser divididos em alunos de graduação, pós-graduação e profissionais do mercado de trabalho.

Para realização do estudo, foi necessário um treinamento individual com cada participante, onde foram revisados conceitos sobre MEs para que fossem supridas quaisquer dúvidas que pudessem haver sobre o assunto. Foram revistos conceitos básicos, tais como os principais elementos que compõem uma ME: Estado, DoAction, EntryAction, ExitAction, Transição, Evento/Sinal, Guarda, Ação e etc. Além disso, foi demonstrado como seria a utilização da ferramenta para visualização das mudanças, no caso a DiffNow, expondo os seus elementos, de maneira que pudessem ser utilizados pelos participantes durante a

execução.

Ainda durante o treinamento, foi explicado sobre os dois formulários que os participantes iriam responder durante a realização do estudo. Um desses formulários, foi utilizado como guia para informar quais seriam as mudanças a serem analisadas e também para coletar informações sobre as possíveis alterações nas MEs. O segundo formulário foi utilizado como forma de coletar informações sobre os perfis dos participantes, nível de conhecimento sobre MEs, grau de dificuldade encontradas nas atividades, importância de manter as MEs atualizadas, entre outras informações.

Para finalização da etapa de treinamento, foi explanado uma visão geral sobre o projeto responsável pelo código-fonte a ser analisado, para que então o foco fosse direcionado para a classe onde teria as mudanças. Logo em seguida, explicado o que a classe era responsável por fazer dentro daquele projeto, sem entrar em detalhes da implementação. Por último, foi exposta a ME que seria analisada junto às mudanças no código-fonte, com o intuito de esclarecer as dúvidas sobre os comportamentos que a ME modelava. Todo esse processo do treinamento, levou em média 30 minutos de duração, para que desta forma, todos os participantes tivessem o mesmo tempo de treinamento.

Finalizada a etapa de treinamento, foi disponibilizado o material para que então fosse realizado o estudo, onde os participantes receberam um documento, cada um contendo suas particularidades a respeito das mudanças e MEs que seriam analisadas e contendo os links para os dois formulários citados anteriormente. Além desses, também haviam links para materiais que continham revisões sobre MEs[9] e a ferramenta que estava sendo utilizada[10], e links para as duas versões do código-fonte, para caso, o participante sentisse interesse de analisar o código-fonte separadamente.

Com todos os materiais entregues aos participantes, foi dado um prazo de até sete dias para cada um responder aos formulários. Após o prazo foram coletadas as respostas dos participantes que concluíram e responderam aos formulários, sendo 20 que responderam e 7 que abandonaram. Houve também pedidos de prorrogações por parte dos participantes, pois eles teriam atividades fora do experimento, alegando que não tiveram tempo, dentro do prazo de sete dias, para se dedicarem à execução do estudo, sendo atendido e dado mais alguns dias

[9] `https://bit.ly/2TyyUF0`
[10] `https://bit.ly/2SXnhCl`

para responderem.

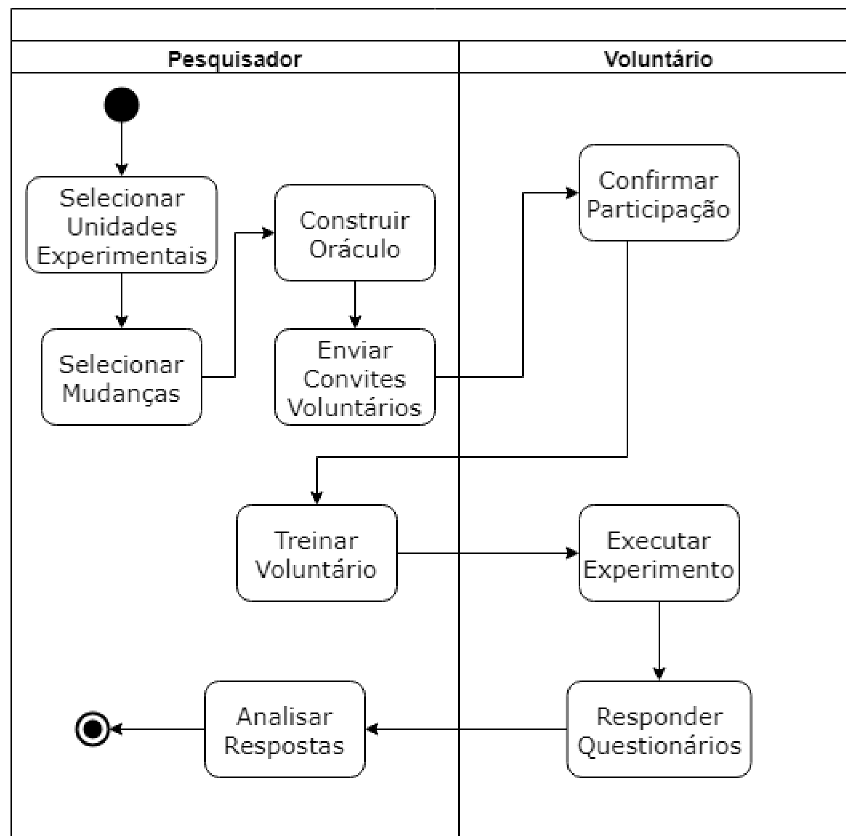


Figura 5.1: Diagrama de Atividades - Atividades executadas para realização do estudo. (Produzido pelo Autor)

Com o intuito de demonstrar todo o processo seguido para realização deste estudo, a Figura 5.1 expõe o fluxo entre as atividades que foram executadas neste estudo. Desse modo, pode-se observar a interação entre Pesquisador e Voluntários, separando por raias as atividades que cada um foi responsável por desempenhar para finalização do estudo.

Inicialmente foram selecionadas as unidades experimentais obedecendo aos critérios de inclusão e exclusão previamente definidos. Com as unidades experimentais definidas, foi realizado o processo de filtragem e por fim seleção das mudanças que poderiam causar alterações nas MEs, tendo como resultado a construção do oráculo utilizado neste estudo. A construção do oráculo ocorreu seguindo as alterações presentes em versões diferentes de suas respectivas MEs, juntamente com as mudanças que ocorreram também entre as versões do código-fonte. Para isso, o pesquisador teve o cuidado de selecionar mudanças que estavam mais aptas a causar alterações na MEs, dessa forma mitigando uma ameaça, mas criando

outra devido à complexidade do estudo.

Com o oráculo construído e passível de utilização para verificação das respostas dos voluntários, foram enviados convites para e-mails e listas de grupos para captação dos voluntários para o estudo. Após os convites serem entregues, os voluntários tinham que responder ao convite, informando ou não a vontade de participar. Logo, após obtenção das respostas do voluntários, o pesquisador deu continuidade ao processo, iniciando a etapa de Treinamento dos Voluntários. Nessa etapa, foram apresentados conceitos a respeito de ME e os tipos de mudanças no código-fonte, e o que deveria ser feito para execução do estudo. Em seguida, após treinamento, os voluntários executaram as atividades e ao final responderam aos questionários de onde foram coletadas as respostas e em seguida analisado os resultados.

Como última atividade, foi realizado a tarefa de análise das respostas, buscando validar a taxonomia que está sendo proposta neste trabalho, como também a produção do mapeamento entre mudanças no código-fonte e alterações em ME. Nessa atividade, por meio das respostas dos voluntários, foi possível a validação da taxonomia proposta e a elaboração do mapeamento, também, proposto.

Como já havia sido elaborada a taxonomia, foi necessária a realização de um estudo para que pudesse ser validada, se as alterações contidas, realmente ocorriam. Por meio deste estudo, foi possível analisar nas respostas dos voluntários, quais as alterações sugeridas faziam parte da taxonomia, possibilitando assim sua validação. Além disso, após serem analisadas as respostas, também foi possível propor um mapeamento entre alterações em ME e mudanças no código-fonte, agregando mais uma contribuição ao estudo.

5.5 Avaliação do Perfil do Participante

Para realizar o experimento, foram convidados participantes de modo que pudessem analisar as mudanças no código-fonte e informar quais teriam causado alterações nas MEs. Sendo assim, foi tomada a decisão de coletar informações sobre os perfis dos participantes, de maneira que pudesse ser avaliado o tempo de experiência e o nível de conhecimento em relação à linguagem de programação JAVA e à MEs.

Ao total, 20 participantes responderam aos questionários que foram entregues, e os outros sete participantes não o fizeram, pois desistiram da realizá-los alegando estarem sempre

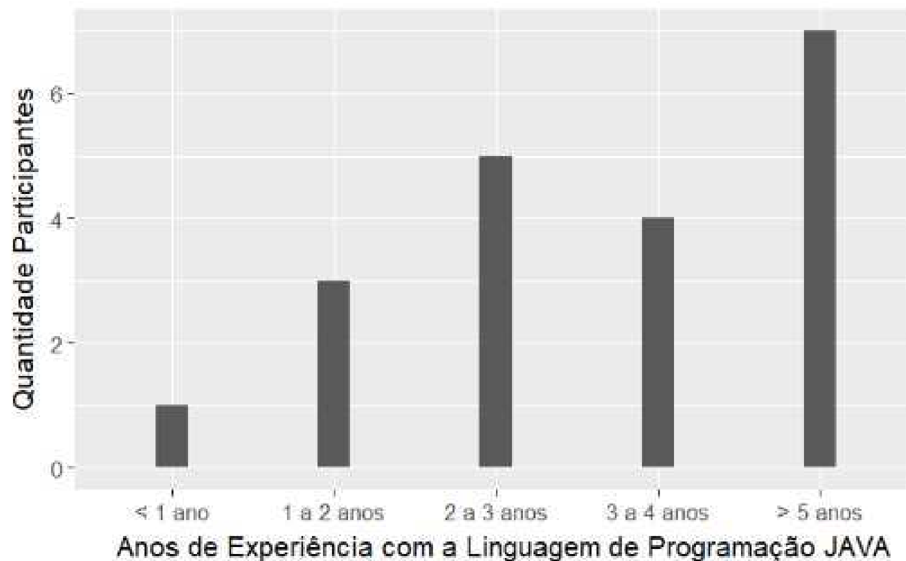


Figura 5.2: Anos de experiência com programação na Linguagem JAVA dos participantes. (Produzido pelo Autor)

muito ocupados e não poderem responder sem empecilhos. Inicialmente foi feito uma análise do tempo de experiência com a linguagem de programação JAVA, para que pudesse ser avaliado o nível de conhecimento em relação à linguagem de todos os participantes, ver Figura 5.2.

Um total de 16 participantes alegaram ter 2 anos ou mais de experiência com a linguagem JAVA, o que pode indicar que tenham certa familiaridade na compreensão dos códigos-fontes escritos nessa linguagem. Dentre os 20 participantes que concluíram o estudo, apenas 5 possuíam algum cargo em empresas de desenvolvimento, podendo contribuir com informações mais próximas a cenários de desenvolvimento de sistemas reais, ver Figura 5.3.

Conforme foi citado na seção "Treinamento e Execução", os participantes precisavam ter um conhecimento prévio sobre MEs, para tentarmos mitigar a incompreensão dos elementos presentes e o fluxo que as MEs seguem. Tendo isto em mente, foram coletadas informações a respeito do nível de conhecimento dos participantes em relação aos conceitos de MEs. A Figura 5.4 expõe que 15 dos participantes declararam que tinham nível de conhecimento médio ou superior, enquanto apenas 5 declararam ter nível de conhecimento baixo ou muito baixo, a respeito de ME. Entretanto, todos receberam um treinamento para realização do experimento, onde em alguns casos, o participante fazia perguntas para sanar suas dúvidas a

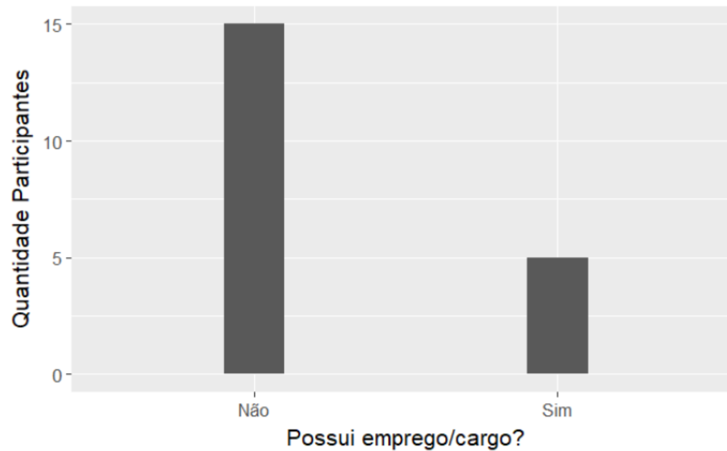


Figura 5.3: Quantidade de participantes que possuem empregos/cargos. (Produzido pelo Autor)

respeito dos conceitos sobre ME. Assim, todos estariam hábeis para realizar o experimento.

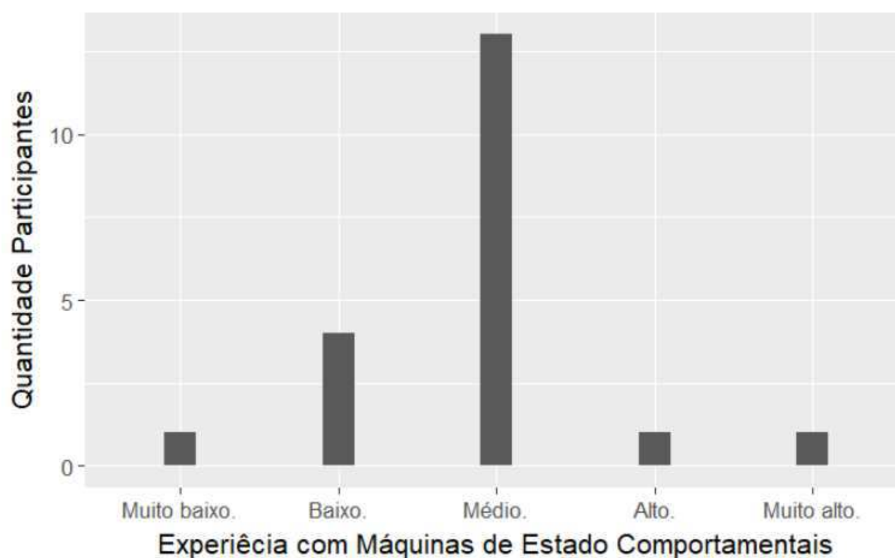


Figura 5.4: Nível de conhecimento dos participantes a respeito de MEs. (Produzido pelo Autor)

5.5.1 Avaliação das Dificuldades dos Participantes

Um dos objetivos deste trabalho de mestrado, é a produção de uma ferramenta que possa automatizar o processo de atualização das MEs, por meio da classificação e sugestão de alterações nas MEs causadas pelas mudanças no código-fonte. Foi questionado aos parti-

cipantes, o quão difícil eles avaliavam a atividade para identificar alterações nas MEs com base nas mudanças no código-fonte. Na Figura 5.5, é possível visualizar que apenas dois participantes consideraram uma tarefa fácil, enquanto os demais consideraram uma tarefa de dificuldade média ou difícil. Além do mais, as respostam podem indicar a necessidade de um meio para auxiliar os desenvolvedores nesse processo de atualização das MEs.

Após, visualizar que dois participantes consideraram a atividade “Fácil”, foi tomada a decisão de verificar se foram os mesmos que informaram ter conhecimento “Muito alto” e “Alto” sobre MEs. Na Figura 5.6, é exposto que o participante que alegou ter conhecimento “Muito alto” sobre MEs, foi um dos que responderam que a atividade de identificar alterações nas MEs analisando as mudanças no código-fonte teria um nível de dificuldade “Fácil”, enquanto o outro respondeu como uma dificuldade “Média”. Essa variação, pode ter sido decorrente das experiências individuais dos participantes, como também dos cenários que obtiveram na realização do estudo. No entanto, é possível notar que a dificuldade para identificar alterações nas MEs tende a ser considerada “Média”.

No intuito de avaliar qual a dificuldade para manter as MEs atualizadas, pois pode ser uma atividade que se repita algumas vezes durante a implementação do software, foi questionado aos participantes, como eles avaliariam este tipo de atividade. Na Figura 5.7, é possível ver que todos os participantes consideraram que a atividade para manter as MEs de um projeto atualizadas teria uma dificuldade média ou superior. Esse tipo de informação, reforça a ideia de que há uma necessidade de uma ferramenta que possa auxiliar os desenvolvedores nesse tipo de atividade, com o intuito de automatizar este tipo de tarefa.

Com a intenção de identificar o nível de dificuldade que está presente no processo de atualização das MEs, durante o desenvolvimento de um sistema, buscamos saber dos participantes, qual o nível de dificuldade que eles consideravam a respeito de identificar quais mudanças no código-fonte causariam alterações nas MEs. Na Figura 5.8, é exposto que somente um participante considerou a atividade com dificuldade “Fácil”, enquanto o restante optou por “Média”, “Difícil” e somente uma por “Muito Difícil”, o que nos dá indícios de que não seja uma atividade simples de ser executada.

Além de avaliar a dificuldade sobre essas atividades para manter uma ME atualizada, sabemos que durante o processo de desenvolvimento de um software, pode haver um aumento na quantidade de linhas, devido à implementação de novas funcionalidades. Com isso, foi

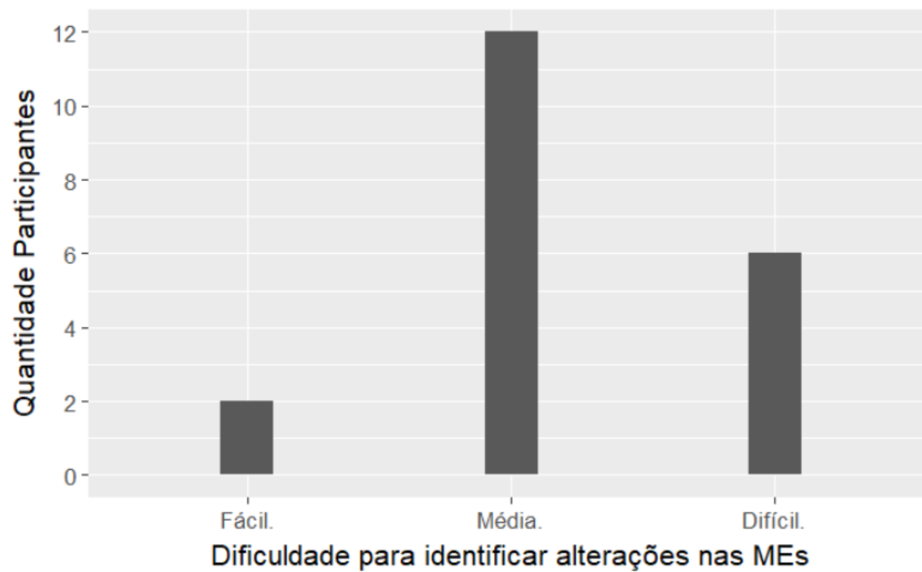


Figura 5.5: Dificuldade para identificar alterações nas MEs analisando as mudanças no código-fonte. (Produzido pelo Autor)

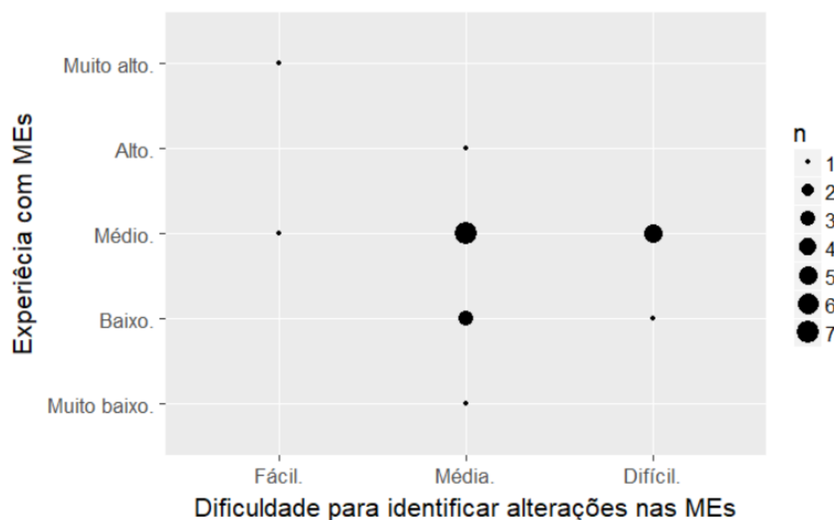


Figura 5.6: Relação entre Experiência com MEs e a dificuldade para identificar alterações nas MEs. (Produzido pelo Autor)

questionado aos participantes se eles consideram que o com o aumento na quantidade de linhas do código-fonte, a dificuldade para identificar as mudanças no código-fonte que causariam alterações nas MEs iria aumentar. Na Figura 5.9, é demonstrado que apenas duas pessoas responderam que aumentaria "Pouco" ou "Não aumentaria e não diminuiria", enquanto o restante respondeu que aumentaria "Muito" e "Muitíssimo", podendo indicar que

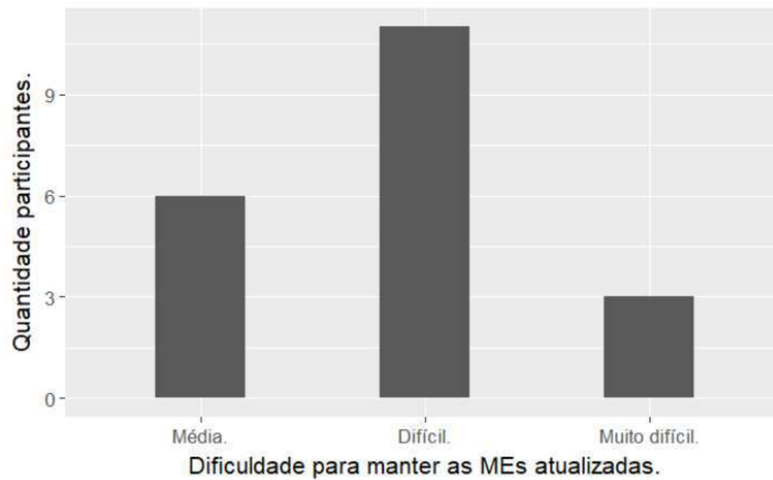


Figura 5.7: Dificuldade avaliada pelos participantes para manter as MEs de um projeto atualizadas. (Produzido pelo Autor)

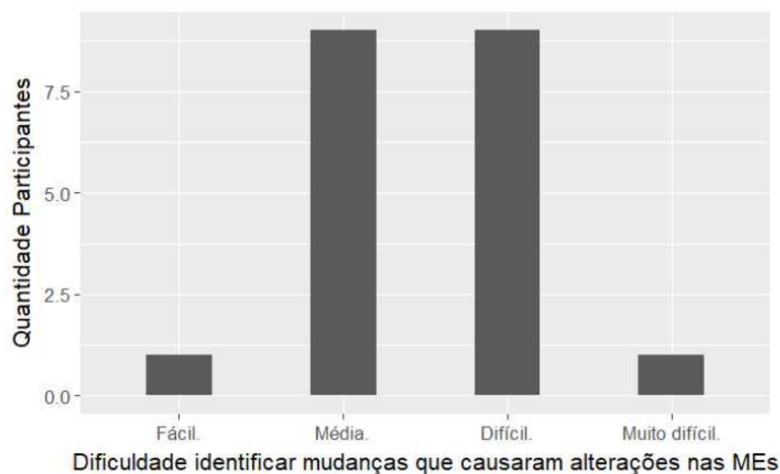


Figura 5.8: Dificuldade para identificar quais mudanças no código-fonte causam alterações nas MEs. (Produzido pelo Autor)

com o crescimento do código-fonte, realizar esse tipo de atividade tende a se tornar mais difícil.

Foi questionado aos participantes se o mapeamento entre mudanças no código-fonte e mudanças nas MEs poderiam auxiliar na identificação de possíveis alterações nas MEs. Na Figura 5.10, está indicado que quatro participantes afirmaram que a utilização desse tipo de estratégia seria "Pouco provável" ou "Não é provável e não é improvável", enquanto os outros 16 afirmaram que seriam "Provável" ou "Muito provável". Tendo esses resultados, obtivemos indícios que podemos sim realizar um mapeamento entre os tipos de mudanças

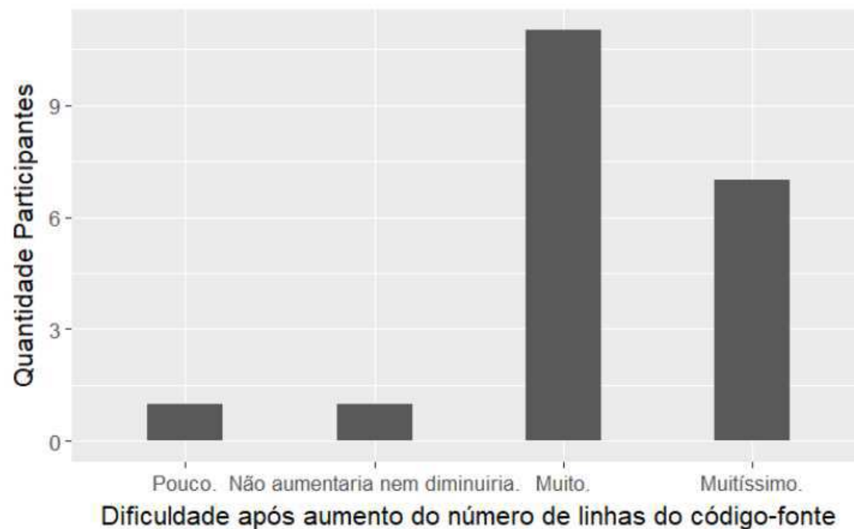


Figura 5.9: Dificuldade para identificar quais mudanças no código-fonte podem causar alterações nas ME, após o aumento do número de linhas do código-fonte (Produzido pelo Autor)

no código-fonte e as alterações que elas podem causar nas MEs. Além disso, que podemos utilizar esse mapeamento para auxiliar os desenvolvedores a manterem as MEs dos projetos atualizadas, na produção de uma ferramenta que venha a automatizar esse processo.

Com o intuito de analisar se a experiência com MEs tinha alguma correlação com os demais dados coletados, foi utilizado o teste de correlação de Pearson [8] para esta abordagem. A Tabela 5.2, mostra os resultados obtidos separados por cada variável que foi utilizada. Em todos os cenários, é possível considerar que os valores obtidos por meio do teste, são de correlação fraca ou mínima. Dessa forma, para o cenário no qual foi realizado o estudo, pode-se desconsiderar que há uma correlação entre quanto maior o conhecimento em MEs mais as atividades se tornam mais fáceis.

Também fizemos um levantamento sobre a importância de manter as MEs de um sistema atualizadas. Para compreender melhor a importância de manter as MEs atualizadas, perguntamos aos participantes o porquê eles consideravam importante que as MEs estivessem atualizadas. As respostas coincidiram em alguns aspectos, tais como: compreensão do comportamento do sistema, auxílio para novos membros compreenderem o comportamento do sistema e identificação do fluxo de atividades. Além dessas, também houve participantes que afirmaram que é possível utilizar as MEs atualizadas para detecção e correção de bugs,

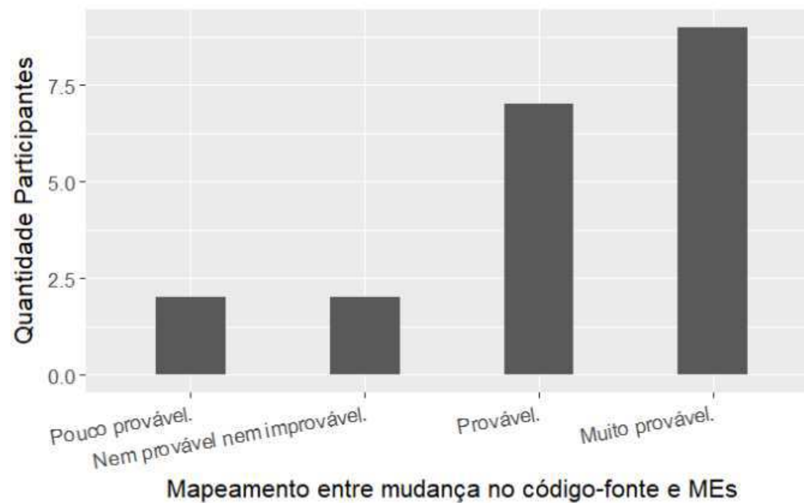


Figura 5.10: Quão provável é a utilização de um mapeamento entre as mudanças nas MEs e código-fonte para identificar eventuais alterações nas MEs. (Produzido pelo Autor)

como também para auxiliar na conversação com o cliente acerca de possíveis alterações no sistema. Abaixo estão transcritos algumas das respostas dos participantes.

> "A principal importância pra mim é a documentação, pois as vezes o código é complexo, e é necessário um auxílio visual para entender, além disso quando a rotatividade do projeto é alta e um novo desenvolvedor, é difícil que tenha noção de tudo ao ver somente o código. Outra vantagem de manter atualizada é para guiar discussões tanto no time de desenvolvimento como com os clientes."

> "É importante pois, caso esteja desatualizada, seu propósito não será atendido, não sendo possível compreender o sistema através de sua análise. Isto pode acarretar sérios problemas."

> "Com uma máquina de estado atualizada isso permite uma compreensão visual do comportamento de um objeto ou sistema, rastreamento das mudanças ocorridas, podendo facilitar o desenvolvimento e compreensão no geral do sistema, assim como ajuda no treinamento de novos desenvolvedores. Além do mais, pode ajudar a resolver bugs no sistema, uma vez que você pode rastrear as mudanças."

Correlações	
Dificuldade para identificar quais são as alterações nas MEs após mudanças no Código-fonte.	-0.252
Dificuldade para manter as MEs atualizadas.	-0.043
Dificuldade para identificar qual mudança no codigo-fonte causou alteração na ME.	-0.047
Dificuldade para identificar alterações nas MEs com o aumento do numero de linhas do código-fonte.	0.387
Probabilidade da utilização do mapeamento para auxílio.	-0.036

Tabela 5.2: Correlação com o nível de conhecimento entre MEs e demais variáveis.

5.6 Análise dos Resultados

Essa análise tem o intuito de responder às questões de pesquisas levantadas no início desta análise, e com os dados obtidos criar um mapeamento entre as mudanças no código-fonte e as alterações nas MEs. A primeira questão de pesquisa é a seguinte:

- **QP1** - Um tipo de mudança no código-fonte, pode causar mudanças em diferentes elementos nas Máquinas de Estados Comportamentais?

  H_0 - Nenhum tipo de mudança pode causar mudanças em diferentes elementos da Máquina de Estado Comportamental;

  H_1 - Um mesmo tipo de mudança pode causar mudança, somente, em um único elemento na Máquina de Estado Comportamental;

  H_2 - Um mesmo tipo de mudança pode causar diferentes mudanças em mais de um elemento da Máquina de Estado Comportamental.

Analisando as respostas dos participantes, um total de 18 entre 20 (90%) dos participantes afirmaram que ao menos uma das mudanças que foram analisadas causariam alterações nas MEs. Dentre esses 18 participantes, 10 participantes afirmaram que todas as três mudanças que receberam causaram alterações nas MEs, enquanto outros cinco afirmaram que apenas duas, e os outros três afirmaram que somente uma das três causaram alterações nas MEs.

Na Tabela 5.3, tem-se uma melhor visualização dos resultados obtidos pelos participantes do experimento, onde expõem as alterações nas MEs sugeridas por cada participante, com base nas mudanças que receberam.

Quando partirmos para a análise das respostas dos participantes em relação às quais mudanças no código-fonte causariam alterações nas MEs, vimos que 43 em 60 (71,66%) dos casos, os participantes afirmaram que haveria alguma alteração na ME analisada, enquanto 17 em 60 (28,33%) não causariam nenhuma alteração. O esperado era que para os 60 casos selecionados, todos fossem afirmados que causariam alguma alteração na ME, devido à seleção das mudanças terem sido passadas pelas filtragens. Entretanto, como o objetivo foi confirmar que mudanças no código-fonte podem causar alterações nas MEs, e quais tipos de alterações seriam realizadas, a presença de respostas negativas não têm impactos negativos, mas sim uma perda de informação que poderia beneficiar mais este estudo.

Para auxiliar a refutar a hipótese nula, realizamos também uma análise do código-fonte em ambas versões dos projetos, juntamente com as versões das MEs. Foi possível observar, que os novos estados inseridos na MEs, ver Figuras 5.11 e 5.12, também estão no código-fonte, ver Códigos-fonte 5.1 e 5.2. Além disso, os valores presentes no código-fonte refletiam as características que estavam sendo modeladas pelas MEs, isso podendo ser observado em outro cenários com outras MEs dos projetos. Todas essas análises, juntamente com as respostas dos participantes, reforçaram a ideia que mudanças no código-fonte podem sim causar alterações nas MEs.

Nas Figuras 5.11 e 5.12, estão as versões de uma das MEs do projeto DESystem, a qual é responsável por representar o objeto Dispatcher. No contexto do projeto, esse objeto é responsável por ser o despachante do elevador, possuindo estados conformes apresentados nos códigos-fonte 5.1 e 5.1, que fazem referência ao elevador se alcançou o andar desejado, ou não. Essas informações a respeito do projeto, e mais especificamente do objeto, foram obtidas por meio da leitura e interpretação do código-fonte, além do auxílio da ME. Podemos observar e comparar as versões das MEs e do código-fonte, para notar que os elementos que foram acrescidos no código-fonte estão presentes na ME e vice-versa.

Dessa forma, torna-se possível refutar a hipótese nula, de que mudanças no código-fonte não causam alterações nas MEs. Pois, conforme, como foi respondido pelos participantes, mostrado por meio do teste de proporção e analisando as mudanças no código-fonte junto

com as alterações nas MEs apresentadas nas Figuras 5.11 e 5.12, para o cenário utilizado no experimento, as mudanças no código-fonte podem sim causar alterações. Para que desta forma, as MEs representem corretamente o comportamento atual dos código-fontes.

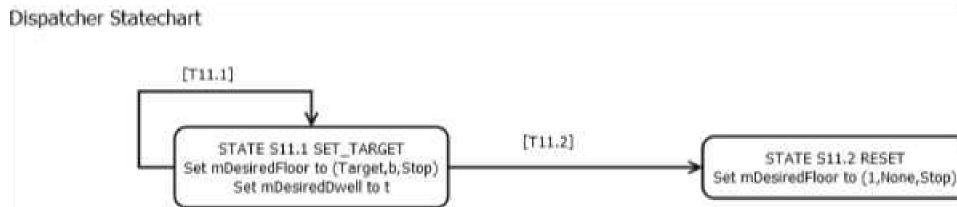


Figura 5.11: Versão desatualizada de uma Máquina de Estado do projeto DESystem.

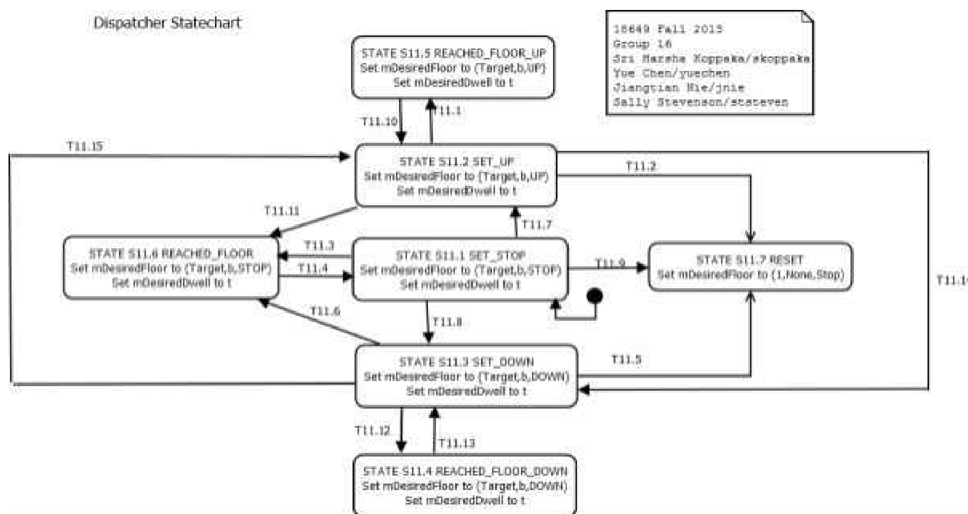


Figura 5.12: Versão atualizada de uma Máquina de Estado do projeto DESystem.

Código Fonte 5.1: Versão desatualizada de uma classe presente no projeto DESystem.

```
1 private enum State{
2         STATE_SET_TARGET,
3         STATE_RESET,
4 }
```

Código Fonte 5.2: Versão atualizada de uma classe presente no projeto DESystem.

```
1 private enum State{
2         STATE_UP,
3         STATE_DOWN,
4         STATE_REACHED_FLOOR_UP,
5         STATE_REACHED_FLOOR_DOWN,
6         STATE_REACHED_FLOOR,
7         STATE_RESET,
8 }
```

Como é possível visualizar na Tabela 5.3, houve cenários em que para uma mudança no código-fonte, os participantes sugeriram que fossem realizadas mais de uma alteração (Tabela 5.3, Participante P3). Houve 14 cenários de mudanças no código-fonte que causaram mais de um tipo de alteração na MEs a qual ele estava sendo representado. Com isso, vemos que há um total de 14 em 43 (32,55%) de mudanças que causam mais de um tipo de alteração na ME, quando olhamos somente para os cenários que causaram algum tipo de alteração a ME. Quando partimos para a visão de todos os cenários, vemos que o percentual cai, 14 em 60 (23,33%). Essas informações, podem refutar a hipótese nula, de que as mudanças no código-fonte não causariam alterações nas MEs.

Ao analisar as outras hipóteses, vemos que a segunda hipótese é válida, pois como podemos observar nos Códigos-fonte 5.1 e 5.2, as inserções de novos valores na classe de *Enumeration* indicam que novos estados deverão ser inseridos nas MEs, e nada mais que isso. Portanto, com esse tipo de mudança, podemos validar a segunda hipótese (H_1), pois nos demais cenários que essa mudança acontece, as alterações sugerida para as MEs são as mesmas.

Ao analisar a terceira hipótese (H_2), notamos que quando tratadas as mudanças isoladamente, sem levar em consideração as demais mudanças que ocorreram no código-fonte, podemos considerar que uma mudança pode causar alterações em mais de um elemento da ME. Como por exemplo, no Código-fonte 5.3, o participante informou que realizaria a adição de um novo Estado na ME juntamente com a adição de uma nova Transição, pois o Estado “STATE_REACHED_FLOOR_UP” não estava presente na ME e, consecutivamente, ele partiria de um outro Estado para ele. Então, ao analisarmos o código-fonte, percebemos que o mesmo Estado está presente no Código-fonte 5.2 em forma de Enum. Logo, para que o trecho do Código-fonte 5.3 pudesse ser compilado e executado, o valor do Enum já deveria existir, podendo dizer que o novo Estado já estaria inserido no momento que foi acrescido no Enum.

Outro cenário em que os participantes alegaram haver mais de uma alteração nas MEs foi com as mudanças realizadas nos *IF* ou em chamadas de métodos que aconteciam no corpo do *IF* (ver Código-fonte 5.4). Para alguns participantes, a inserção da chamada de método dentro do *IF*, mesmo sendo especificado que seria somente uma linha do código-fonte, fez com que eles afirmassem a necessidade de alterações nas MEs acrescentando uma Transição

e a condição do *IF* como respectiva Guarda. O mesmo tipo de resposta aconteceu quando a linha selecionada era a do *IF* e o cenário se repetia como no Código-fonte 5.4.

Código Fonte 5.3: Linha de código-fonte inserida na nova versão em uma classe do projeto DESystem.

```
1 newState = state.STATE_REACHED_FLOOR_UP;
```

Código Fonte 5.4: Trecho de código-fonte inserida na nova versão em uma classe do projeto Design Pattern Indeepth.

```
1 if(position > commitPointUp) {
2           newstate = State.STATE_SLOW_UP; }...
```

Entretanto, quando analisamos os cenários onde as mudanças estavam associadas às chamadas de método, ver Códigos-fonte 5.5 e 5.6, vimos que elas poderiam causar mudanças em tipos de elementos diferentes nas MEs, onde dependendo do contexto ele poderia ser considerado uma Transição ou algum tipo de Ação do Estado (*DoAction*, *ExitAction*, *EntryAction*). No código-fonte 5.5, é exposto um trecho de código do Projeto Design Pattern Indeepth, responsável por alterar o estado em que o objeto "gumballMachine" encontra-se, assumindo o estado de "WinnerState", linha 2. Enquanto isso, no código-fonte 5.6, a chamada de método na linha 2, é referente a um comportamento que o estado "State_Down" pode realizar, pois no código-fonte contém comentários confirmando essa afirmação.

Portanto, pode-se deduzir que ao realizar inserções de chamadas de métodos, elas podem indicar que será uma nova Transição ou uma nova Ação do Estado, sendo necessário analisar o contexto a qual se encontra a mudança que foi realizada no código-fonte, se realmente aquela mudança causará alguma alteração na ME e em qual elemento. Dessa forma, podemos considerar que um tipo de mudança no código-fonte pode causar alterações em mais de um elemento da ME, validando assim a hipótese H_2.

Código Fonte 5.5: Inserção de chamada de método presente no projeto DESystem (1).

```
1 if (random.nextInt(10) == 0 &&
      gumballMachine.getCount() >
      1) {
2      gumballMachine.setState(
          gumballMachine.
          getWinnerState());...
3 }...
```

Código Fonte 5.6: Inserção de chamada de método presente no projeto DESystem (2).

```
1 case STATE_DOWN:
2     mDesiredFloor.set(targetFloor
          , hallway,
          desiredDirection);
```

A última questão de pesquisa aborda se é possível fazer um mapeamento entre as mudanças no código-fonte e as alterações nas MEs.

- **QP2** - É possível fazer um mapeamento entre as mudanças que ocorrem no código-fonte e as mudanças que ocorrem nas Máquinas de Estado Comportamentais?

  H_0 - Não é possível fazer um mapeamento entre as mudanças no código-fonte e as mudanças nas Máquinas de Estado Comportamentais;

  H_1 - É possível fazer um mapeamento entre as mudanças no código-fonte e as mudanças nas Máquinas de Estado Comportamentais.

Após a realização do experimento foi detectado que quando certos tipos de mudanças no código-fonte aconteciam, as respostas eram similares ou até iguais. Por exemplo, nos Códigos-fonte 5.1, 5.2, 5.7 e 5.8 ocorreram a inserção de um novo valor no *Enum*, e para os demais casos semelhantes os participantes informaram que deveriam ser acrescentados novos Estados às suas respectivas MEs. Isso nos deu indícios de que poderíamos sim realizar um mapeamento entre as mudanças no código-fonte e as alterações nas MEs.

Código Fonte 5.7: Versão anterior de Enumeration presente no projeto Smarthome.

```
1 public enum ThingStatus {
2 UNINITIALIZED(0),
3 INITIALIZING(1),
4 ONLINE(2),
5 OFFLINE(3),
6 REMOVING(4),
7 REMOVED(5);}...
```

Código Fonte 5.8: Versão nova de Enumeration presente no projeto Smarthome.

```
1 public enum ThingStatus {
2 UNINITIALIZED,
3 INITIALIZING,
4 UNKNOWN,
5 ONLINE,
6 OFFLINE,
7 REMOVING,
8 REMOVED;}
```

Foi feito um mapeamento com base nas respostas obtidas pelos participantes do experimento, ver Tabela 5.4, considerando os cenários que mais se repetiam. Em seguida, analisando as respostas dos participantes, foram obtidas mais informações, sendo essas geradas ao analisar os tipos de mudanças e os elementos alterados nas MEs. Identificando que determinado elemento na ME era alterado, quando ocorria certo tipo de mudança no código-fonte, foi possível deduzir para outros cenários de alterações para este mesmo elemento da ME.

Supondo que inserção de valor no *Enum* JAVA, quando causa alteração na ME, sempre está relacionado ao elemento de Estado da ME, então pode-se generalizar para mudanças no código-fonte de remoção ou atualização no *Enum*, que podem causar alterações de Adição, Remoção ou Atualização do Estado na ME. Um outro exemplo, ao identificar que guardas das MEs podem ser associadas aos IFs do código-fonte, pode-se deduzir que mudanças de atualizações, inserções ou remoções de IFs, vão refletir da mesma forma nas guardas das MEs que estejam sendo representadas. Desse modo, foi possível aprimorar o mapeamento gerado inicialmente, obtendo novos valores para serem adicionados.

Usando como base o mapeamento presente na Tabela 5.4, foram adicionadas informações relacionadas aos cenários onde ocorreram as alterações nas MEs, pois, foi solicitado aos participantes que eles justificassem o porquê de estarem realizando tais alterações nas MEs. Ao final, conseguimos elaborar um mapeamento que especifica quais elementos no código-fonte poderiam causar que alterações nas MEs, e o que seria realizado, além de possíveis justificativas para tais operações, como também exemplos para auxiliar o leitor a entender e caso deseje utilizá-la (ver apêndice A). O mapeamento gerado (ver apêndice A), ilustra

como estão dispostas as informações obtidas por meio do estudo, onde são apresentados o elemento no código-fonte que sofreu a mudança, em seguida o tipo de mudança realizada, qual o tipo de alteração que deverá ser realizada na ME, a justificativa comumente usada pelos participantes para realização de tal alteração e ao final um exemplo guia, para servir de base para os desenvolvedores.

Inicialmente, já havia sido elaborado um mapeamento entre mudanças no código-fonte e as alterações nas MEs, juntamente com suas respectivas justificativas e descrições, como por exemplo uma mudança de um *Enum* JAVA e a alteração no elemento estado da ME. Esse mapeamento, foi gerado com base em possíveis cenários de mudanças em código-fonte que poderiam causar alterações em MEs, e também os cenários que o pesquisador se deparou durante as buscas por projetos.

Para que pudéssemos validar esse mapeamento, precisaríamos que outros desenvolvedores também afirmassem que os mesmos tipos de mudanças no código-fonte poderiam causar alterações nos cenários que havíamos previsto. Com esse estudo, foi possível descartar algumas relações entre alterações em ME e mudanças no código-fonte, como por exemplo, que *Switch* no código-fonte seriam guardas na ME, como também, validar parte dos mapeamentos que já estavam presentes. Infelizmente, não foi identificado nenhum novo mapeamento com esse estudo, mas, somente validado parte dos mapeamentos, que ocorreram nos cenários do estudo. Entretanto, há a possibilidade de que venham surgir novos mapeamento com a adição de novos projetos ou estudos a respeito disso.

Portanto, vimos que é possível realizar mapeamentos entre os tipos de mudanças no código-fonte com as alterações em MEs, pois, quando determinados tipos de mudanças no código-fonte acontecem, os tipos de alterações nas MEs também eram semelhantes. Com isso, inicialmente, foram reforçados os indícios de que era possível, e que ao final do estudo, conseguimos produzir uma tabela com auxílio dos resultados. Desse modo, espera-se ter uma boa contribuição com um estudo para identificar quais tipos de mudanças no código-fonte podem causar alterações nas MEs, e um mapeamento entre essas mudanças no código-fonte e alterações em ME. Além disso, espera-se que esse mapeamento possa auxiliar os desenvolvedores no processo de atualização das MEs de seus projetos.

5.7 Ameaças à Validade

Como é comum em todo experimento haver a presença de fatores que possam ameaçar a validade do experimento, esta seção trata das ameaças identificadas e como foram mitigadas.

5.7.1 Ameaças de Constructo

Inicialmente, foram feitas análises e estudos para tentar adquirir conhecimento a respeito de se mudanças poderiam realmente causam alterações nas MEs, e se haveria algum tipo de relação. Esses procedimentos foram realizados com o intuito de evitar que não tivessem conceitos ou definições confusas no momento de planejamento do experimento como um todo, mitigando o máximo possível que houvesse uma explicação pré-operacional adequada ao que estava sendo estudado.

Tentando alcançar todos os cenários possíveis, foi necessário a seleção de diferentes unidades experimentais que abordassem cenários distintos da utilização de ME. Desse modo, tentando obter o máximo de informação dos cenários possíveis, com base nas unidades experimentais que foram selecionadas. Além disso, por meio da análise e dedução das respostas, foi viável expandir os resultados obtidos além dos presentes nas respostas.

Para tentar mitigar ameaças em relação aos participantes tentarem adivinhar as expectativas do pesquisador, toda a parte de respostas dos participantes aconteceu sem a presença do pesquisador. Essa tomada de decisão foi definida, pois, os participantes poderiam querer extrair informações a respeito das mudanças a serem analisadas, devido a presença do pesquisador. Além disso, o pesquisador tomou a decisão de não responder possíveis dúvidas sobre acertos ou erros, que pudessem inserir um viés com base na resposta dada pelo pesquisador. Entretanto, é ciente que mesmo tendo precauções os participantes podem tentar identificar o que se espera como resposta, e isso gerar ameaças as conclusões alcançadas com o estudo.

5.7.2 Ameaças Externas

Inicialmente, houve a presença de uma ameaça para detecção de mudanças no código-fonte, que seria a utilização da ferramenta *ChangeDistiller*, pois, houve cenários onde a mesma não foi possível de identificar todas as mudanças que aconteceram na classe Java. Desta forma, o

número obtido de mudanças pode ter sido um pouco inferior, do que realmente deveria, nas primeiras extrações (ver Tabela 5.1).

Além disso, a não identificação de algumas mudanças pela *ChangeDistiller*, poderia comprometer na hora da seleção das mudanças no código-fonte que causaram alterações nas MEs. Para mitigar este tipo de ameaça, foi realizada a análise e filtragem manual, como foi citada na Subseção 5.3. Dessa forma, foi possível capturar mudanças que tivessem maior chances de causarem mudanças nas MEs. Para classificar os tipos de mudanças no código-fonte, foi utilizada a classificação fornecida pela ferramenta *ChangeDistiller*, tendo assim, uma padronização nas classificações.

Entretanto, essa análise manual da seleção das mudanças a serem utilizadas no estudo, pode apresentar uma ameaça, onde há a possibilidade do pesquisador, mesmo que inconsciente, selecione mudanças desejadas ou esperadas ao estudo. Além disso, como a construção do oráculo foi realizada pelo pesquisador após essa análise manual, há a possibilidade de que informações novas tenham sido perdidas, ou que os resultados obtidos, não agreguem tantos novos dados.

5.7.3 Ameaças Internas

Uma ameaça que poderia influenciar nos resultados do experimento, seria a questão da maturação, devido ao tempo para realização do estudo. Pois, caso houvesse uma grande quantidade de mudanças no código-fonte para cada participante analisar, isso poderia cansar os participantes fazendo com que os mesmos não respondessem corretamente de maneira a finalizar o quanto antes as atividades. Tendo isso em mente, foi tomada a decisão de que cada participantes analisariam somente 3 mudanças, conforme citado na Seção de "Treinamento e Execução", com o intuito de mitigar o máximo possível a maturação, mas que pudesse ser extraído o máximo possível de informação de cada participante.

Com a presença de participantes, para realização do experimento, era necessário que eles tivessem conhecimento prévio sobre ME, para que desta forma, pudesse ser mitigada a ameaça relacionada à seleção dos participantes. Além dessa condição para seleção, foi realizado um treinamento sendo revisado os conceitos sobre MEs, para que pudessem ser reforçados os conceitos sobre MEs (ver Subseção 5.4) e utilização de demais ferramentas presentes no experimento.

Como estavam sendo utilizadas outras ferramentas para a realização do experimento, isto poderia interferir nos resultados, pois, não compreender a utilização das ferramentas poderia ser um problema. A fim de mitigar este tipo de ameaça, durante a fase de treinamento, foram explicadas as ferramentas que foram utilizadas como também os formulários, códigos-fonte e MEs que foram utilizadas, tentando dessa forma, suprir todas as dúvidas que viessem a surgir durante a fase de execução do experimento.

5.7.4 Ameaças de Conclusão

As unidades experimentais selecionadas, para realização do experimento, podem apresentar uma ameaça, pois somente a Smarthome é um sistema real, impossibilitando a generalização para demais cenários. Entretanto, são códigos abertos que foram desenvolvidos para simulações de aplicações reais. Além disso, após serem realizadas pesquisas nos motores de busca, e na base de dados fornecida pelo Gregorio Robles [13], os projetos selecionados para este estudo, foram os únicos que atenderam aos critérios de inclusão e exclusão definidos.

5.8 Discussão

Conforme visto na Subseção 5.6 , conseguimos obter respostas para todas as questões de pesquisas que foram levantadas para este experimento. Como também foi visto que a utilização do mapeamento pode vir a ser útil no processo de manter as MEs atualizadas. Então, nesta seção iremos discutir sobre as informações obtidas por meio deste experimento.

Inicialmente iremos responder às questões de pesquisas que foram levantadas, e discorrer sobre elas e demais informações que foram obtidas.

- **QP1** - Um tipo de mudança no código-fonte, pode causar mudanças em diferentes elementos nas Máquinas de Estados Comportamentais?

Como podemos observar, mudanças no código-fonte podem sim causar mudanças nas MEs, pois, as MEs, comumente, são utilizadas para representação comportamental do sistema que está sendo implementado. Logo, caso ocorram mudanças no código-fonte que alterem o comportamento, seja o acréscimo ou remoção de uma ação, para que a ME permaneça

modelando corretamente o comportamento, poderá ser necessário algumas alterações na ME. Com isso, podemos concluir que nos cenários onde as MEs estão representando o comportamento do sistema, e uma mudança no código-fonte venha alterar o comportamento do sistema, esta mudança poderá causar alterações nas MEs.

Existem tipos de mudanças que não causam alteração na ME. Além dessas, há também as questões de condições ou justificativas para realizar uma alteração na ME, ou seja, é necessário que a ME esteja modelando o comportamento daquela parte do código-fonte, seja essa parte um método, uma atribuição, declaração de condições, entre outros. Com isso, podemos identificar alguns dos fatores que são necessários para que uma mudança no código-fonte cause alteração na ME, fatores estes como o tipo da mudança no código-fonte e também se determinado trecho de código-fonte refere-se a algum comportamento relacionado ao que está sendo modelado pela ME.

Conforme o que foi visto na Subseção 5.6, um tipo de mudança no código-fonte pode sim causar alteração em mais de um elemento na ME. Isso se deu por conta das inserções de chamadas de métodos, que em um determinado cenário era considerada como uma Ação do Estado, enquanto no outro foi considerado como uma Transição. Assim, podemos afirmar que há sim, tipos de mudanças no código-fonte que pode ocasionar alterações em diferentes elementos na ME.

Também foi possível visualizar que há mudanças no código-fonte que ocasionam alterações nas ME somente em um elemento. Um exemplo desses casos, são as inserções de novos valores nos Enumerations em que seus valores sempre estavam representando os Estados que determinado objeto poderia alcançar. Desse modo, afirmando que a outra hipótese levantada, também era válida, onde um tipo de mudança no código-fonte poderia causar alteração em um único elemento da ME.

- **QP2** - É possível fazer um mapeamento entre as mudanças que ocorrem no código-fonte e as mudanças que ocorrem nas Máquinas de Estado Comportamentais?

Conforme foi exposto na Subseção 5.6, notamos que quando ocorria certos tipos de mudanças no código-fonte, as alterações que eram realizadas nas MEs eram semelhantes ou iguais. Com isso, vimos que em determinados cenários, poderíamos utilizar as mudanças para sugerir possíveis alterações nas MEs. Portanto, seguindo esse raciocínio, elaboramos

o mapeamento entre mudanças no código-fonte e alterações nas MEs, buscando assim auxiliar os desenvolvedores a manter as MEs atualizadas, ou seja, em conformidade com o comportamento atual do sistema.

Este mapeamento gerado, foi utilizado para o desenvolvimento da ferramenta, que terá como finalidade identificar e sugerir possíveis alterações nas MEs, por meio das mudanças no código-fonte, levando em consideração os tipos de mudanças no código-fonte que foram realizadas. A ferramenta tem como objetivo prover suporte automatizado aos desenvolvedores para que possam ter auxílio nas atividade de manter as MEs dos projetos sempre atualizadas.

Participante	Mudança N1	Mudança N2	Mudança N3
P1	AddState	AddState	AddState + AddTransition
P2	AddState	AddTransition	AddState
P3	AddState + AddAction + AddTransition	AddState + AddAction + AddTransition	AddState + AddAction + AddTransition
P4	AddTransition	AddGuard	AddGuard
P5	AddState	AddTransition + Add Guard	AddTransition + Add Guard
P6	AddState	AddState + AddTransition + Alter CompositeState	——
P7	AddTransition + Add Guard	AddTransition + Add Guard	UpdateTransition
P8	AddState	AddState + AddTransition	AddGuard
P9	AddState	AddState	AddState
P10	Remove DoAction	Remove DoAction	Remove DoAction
P11	Remove DoAction	Remove DoAction	Não
P12	AddState	AddTransition	Não
P13	Não	AddTransition	AddState + AddTransition
P14	Não	AddState + AddAction	AddState
P15	AddTransition	Não	AddState + AddTransition
P16	Não	AddState + AddTransition	Não
P17	Não	Não	AddState
P18	AddState	Não	Não
P19	Não	Não	Não
P20	Não	Não	Não

Tabela 5.3: Respostas dos Participantes em relação às mudanças que causaram alterações nas MEs.

Tipo de Mudança Código-fonte	Alteração em ME
Remoção de Chamada de Método	Remoção de DoAction
Adição de Valor no Enum	Adição de State
Remoção de Declaração de Objeto	Remoção de DoAction
Atualização do Valor de Return	Alteração do corpo do Composite State + Adição de Transition
Adição de Switch Case	Adição de State
Adição de Chamada de Método	Adição de State + Adição de Transition + Adição de DoAction
Atribuição de Valor	Adição de State + Adição de Transition + Adição de Actions no State
Adição de IF	Adição de Guard + Adição de Transition + Adição de State
Atualização de Condições no IF	Atualização de Guard

Tabela 5.4: Mapeamento Inicial entre Mudanças no Código-fonte e Alterações em MEs.

Capítulo 6

Experimento - Avaliação do Algoritmo proposto

Com base nos resultados do estudo exploratório (ver Capítulo 5), conseguimos criar um mapeamento entre as mudanças do código-fonte e quais alterações poderiam causar nas MEs. Dando prosseguimento, com o intuito de automatizar e tornar esse mapeamento utilizável, criamos um algoritmo e o implementamos em uma nova ferramenta. Entretanto, é necessário a realização de um estudo para avaliarmos a eficácia do algoritmo.

Para avaliarmos a eficácia do algoritmo, no que diz respeito a sua Precisão e Cobertura para classificar as alterações necessárias nas MEs, tendo como base as mudanças no código-fonte, foi realizado um experimento. Logo, com o objetivo deste experimento definido, foi levantada a seguinte questão de pesquisa: "**QP1 -** Qual a taxa de precisão e cobertura ao classificar as alterações necessárias nas MEs após ocorrências de mudanças no código-fonte?".

- $H_{0.1}$ - A taxa de precisão do algoritmo para sugestão de alterações em ME é inferior a 50%.
- $H_{0.2}$ - A taxa de cobertura do algoritmo para sugestão de alterações em ME é inferior a 50%.
- H_1 - A taxa de precisão do algoritmo para sugestão de alterações em ME é de 50%.
- H_2 - A taxa de cobertura do algoritmo para sugestão de alterações em ME é de 50%.

- H_3 - A taxa de precisão do algoritmo para sugestão de alterações em ME é superior a 50%.
- H_4 - A taxa de cobertura do algoritmo para sugestão de alterações em ME é superior a 50%.

Devido a não terem sido encontrados trabalhos que abordassem o mesmo propósito, os valores definidos na avaliação de precisão e cobertura foram especificados seguindo a lógica de um algoritmo randômico para “sim” e “não”. Seguindo essa definição, foi determinado que os valores de precisão e cobertura teriam de alcançar uma taxa média superior a 50%, para que o algoritmo tivesse resultados melhores que um simples código randômico de “sim” ou “não”. Portanto, com base nos valores de cobertura e precisão decididos, as hipóteses nulas são as inferiores a essa taxa, e as hipóteses a serem validadas, são iguais ou superiores a taxa média de 50% para precisão e cobertura.

Após definido o objetivo e a questão de pesquisa deste experimento, foram selecionadas as unidades experimentais as quais seriam utilizadas para avaliar o algoritmo. Para isso, foram selecionadas as mesmas unidades experimentais do estudo exploratório (ver Seção 5.2): DESystem[1], DesignPattern[2], Smarthome[3]; Todos os três projetos sendo disponibilizados por meio do Github[4]. Sendo essas, selecionadas, pois já contemplavam os critérios de exclusão e inclusão definidos para este experimento, que são:

- Exclusão:
  1. Não ser possível, para o pesquisador, identificar quais partes do código-fonte estão sendo modeladas pelas MEs.
- Inclusão:
  1. O código-fonte do sistema ser Java;
  2. Haver MEs disponíveis;

[1]https://github.com/Liso/DESystem
[2]https://github.com/taoning2014/design-pattern-indeepth/
[3]https://github.com/eclipse/smarthome
[4]https://github.com/

3. Haver controle de versão nas MEs, ou seja, haver alterações nas MEs durante o processo de implementação;
4. As MEs estarem relacionadas ao código-fonte;
5. Ser código aberto.

Definidas as unidades experimentais que foram utilizadas neste experimento, foi planejado como os dados seriam trabalhados para serem usados na execução do experimento. Portanto, foi definido o design de experimento aninhado, de modo em que as MEs ficariam alocadas com seus respectivos projetos, para evitar resultados falsos que pudessem alterar as conclusões.

6.1 Design Experimental

Para que pudesse ser executado este experimento, foi necessário um planejamento e definição do design experimental mais adequado ao experimento. Desse modo, definimos quais seriam as variáveis dependentes e independentes, como foram configurados os valores para a execução e o que é esperado como saída para ser realizada a análise, validação e avaliação do algoritmo proposto.

No que diz respeito às variáveis independentes para este experimento, foram definidas as seguintes: (i) Versão anterior da ME, a qual foi usada como entrada para identificar possíveis alterações na mesma; (ii) Versão da ME mais atualizada, a qual foi utilizada como oráculo, para validarmos a saída do algoritmo; (iii) Versão Antiga do código-fonte, que foi utilizada para obter as mudanças que ocorreram no código-fonte em comparação com a nova versão; (iv) Versão nova do código-fonte, que foi utilizada para obter as mudanças que ocorreram no código-fonte em comparação com a versão antiga; e (v) Número de classes retornadas pela IR, determinando quantas classes devem ser retornadas pela Lucene[5], ferramenta que nos disponibiliza implementações de técnicas de IR, para realizar um procedimento interno do algoritmo que define quantas classes serão analisadas. Enquanto isso, a variável dependente é definida como: Alterações nas Máquinas de Estado sugeridas pelo algoritmo; podendo variar entre verdadeira ou falsa de acordo com a nova versão da ME (oráculo).

[5]http://lucene.apache.org/

Para isso, conforme foi citado anteriormente, o design experimental selecionado para este experimento foi o Design Aninhado. Na Tabela 6.1, é exposto como ocorreu a divisão dos projetos com suas respectivas MEs, onde é notável que o projeto denominado PJ3 possui um número maior de MEs em relação aos outros. Portanto, as MEs tiveram de ficar alocadas ao projeto as quais pertencem, com o intuito de evitar variações indesejadas e reduzir os gastos sobre execução do experimento.

Para avaliar os resultados obtidos pelo algoritmo, foram selecionadas as métricas de **Precisão** e **Cobertura**, embora sejam usualmente utilizadas em Recuperação da Informação [18], adequaram-se ao nosso propósito. No contexto desse experimento, pode-se definir que precisão é a quantidade de alterações das ME que o algoritmo detectou corretamente, em comparação com as versões atualizadas das MEs utilizadas como oráculo. Enquanto cobertura, é definida como a quantidade de alterações existente entre as versões das MEs que o algoritmo foi capaz de identificar corretamente.

Projetos	Máquinas de Estado
PJ1	PJ1.ME1
PJ2	PJ2.ME2
PJ3	PJ3.ME1, PJ3.ME2, PJ3.ME3, PJ3.ME4, PJ3.ME5, PJ3.ME6 e PJ3.ME7.

Tabela 6.1: Design Experimental Aninhado

6.2 Execução

A execução do experimento foi realizada seguindo os passos necessários para utilização do algoritmo, esse sendo executado de maneira automática, com um acréscimo do processo de validação, sendo esse um processo manual, onde é utilizada a nova versão da ME para poder verificar e então validar os resultados. No contexto deste experimento, foram avaliados os resultados obtidos para os seguintes números de classes retornadas: 1, 2, 3, 5 e 8, para avaliarmos a eficácia do algoritmo quando houvesse o aumento do número de classes. Esses números foram selecionados seguindo a sequência de Fibonacci[21]. Desta forma, podemos analisar as variações dos resultados de acordo com o número de classes retornadas.

Por último, ao obtermos as alterações sugeridas do algoritmo como saída, utilizamos o que seria a versão atualizada da ME para verificamos quais alterações estariam corretas. Dessa forma, podemos analisar quais foram as alterações sugeridas pelo algoritmo que estavam presentes na nova versão, como também as que não estavam e fazer observações a respeito do mesmo.

6.3 Análise dos Resultados

Após a execução do experimento, foi realizada a análise dos resultados obtidos, com o intuito de responder a questão de pesquisa levantada para este experimento.

- QP1 - Qual a taxa de precisão e cobertura ao classificar as alterações necessárias nas MEs após ocorrências de mudanças no código-fonte?

  $H_{0.1}$ - A taxa de precisão do algoritmo para sugestão de alterações em ME é inferior a 50%.

  $H_{0.2}$ - A taxa de cobertura do algoritmo para sugestão de alterações em ME é inferior a 50%.

  H_1 - A taxa de precisão do algoritmo para sugestão de alterações em ME é de 50%.

  H_2 - A taxa de cobertura do algoritmo para sugestão de alterações em ME é de 50%.

  H_3 - A taxa de precisão do algoritmo para sugestão de alterações em ME é superior a 50%.

  H_4 - A taxa de cobertura do algoritmo para sugestão de alterações em ME é superior a 50%.

Para isso, avaliamos os resultados obtidos separadamente por classes, projetos e quantidade de classes que seriam retornadas, conforme estão expostos nas Tabelas 6.2, 6.3 e 6.4, e a média geral na Tabela 6.6. Nas colunas "Nº Alt ME", estão contidas as informações a respeito da quantidade de alterações que foram realizadas nas MEs, seguindo a taxonomia proposta no capítulo 5 e a comparação entre as versões anteriores e novas das MEs. Nas

colunas “Nº Classes”, estão contidos os números de classes que devem ser retornadas. Nas colunas “Detectadas”, estão contidos os números de alterações nas MEs que foram detectadas pelo algoritmo.

Smarthome					
Nome ME	**Nº Alt ME**	**Nº Classes**	**Detectadas**	**Precisão**	**Cobertura**
StatusSmarthome	6	1	1	100%	16,67%
		2	8	12,50%	16,67%
		3	8	12,50%	16,67%
		5	14	7,14%	16,67%
		8	15	6,67%	16,67%

Tabela 6.2: Resultados do Algoritmo proposto para o projeto Smarthome.

Conforme exibido na Tabela 6.2, a qual expõe os resultados para o projeto Smarthome, é possível notar que a técnica obteve um baixo percentual de cobertura, e precisão, com ressalva para o cenário de somente uma classe retornada pela técnica de IR. Mesmo com uma baixa quantidade de alterações realizadas na ME, o que poderia melhorar o desempenho quanto a cobertura, o algoritmo não foi capaz de melhorar essa taxa mesmo elevando o número de classes retornadas para oito. Isso ocorreu devido as classes retornadas não conterem mudanças relevantes para o algoritmo.

Ao analisar o código-fonte das classes retornadas, foi visto que haviam termos iguais aos da ME, podendo ter prejudicado o processo de relação entre ME e código-fonte (ver Código-fonte 6.1), onde o termo “offline” o qual é considerado como estado na ME, está presente também no código-fonte. Esse tipo de ocorrência pode ter ocasionado a perda de classes com informações que seriam mais relevantes para nosso algoritmo, de tal maneira, resultando em uma Cobertura inferior a 50%, para este projeto.

Código Fonte 6.1: Trecho de código-fonte retirado do projeto Smarthome

```
1 logger.debug((((("Ignoring command " + channelUID.getId()) + " = ") +
    command) + " because device is offline."));
```

Além disso, notamos que com o aumento de classes retornadas, o percentual de Precisão do algoritmo, também regrediu extremamente, saindo de um extremo de 100% a atingir

6,67%. Isso se deu devido à forma a qual o algoritmo foi implementado, buscando termos referentes aos nomes dos Estados para que fossem classificados de alguma maneira. Ainda no Código-fonte 6.1, esse foi um dos cenários em que o algoritmo classificou como sendo uma nova Transição, devido à presença do termo "offline", mesmo termo usado no nome de um dos Estados. Além disso, as classes retornadas pela técnica de IR, também causaram equívocos na classificação do algoritmo, também ocasionado devido a sua implementação.

DesignPattern					
Nome ME	**Nº Alt ME**	**Nº Classes**	**Detectadas**	**Precisão**	**Cobertura**
GumballMachine	9	1	0	0%	0%
		2	2	100%	22,22%
		3	2	100%	22,22%
		5	2	100%	22,22%
		8	2	100%	22,22%

Tabela 6.3: Resultados do Algoritmo proposto para o projeto DesignPattern.

Conforme exposto na Tabela 6.3, para os cenários do projeto DesignPattern, as taxas de precisão mostraram-se ter excelentes resultados alcançando os 100%, mesmo com o aumento do número de classes retornadas. Isso foi ocasionado porque as classes que foram retornadas não continham mudanças no código-fonte que seguissem os requisitos para serem classificadas como alterações na ME, evitando equívocos na classificação do algoritmo. Entretanto, ao observar o primeiro cenário onde somente uma classe foi retornada, o algoritmo não foi capaz de identificar nenhuma alteração na ME.

Após ser verificado o código-fonte para identificar a razão, foi visto que havia uma mudança no código-fonte que deveria ser classificada pelo algoritmo como uma alteração, mas não ocorreu. Isso aconteceu, pois devido a uma limitação do algoritmo, ele não é capaz de identificar quando os estados da ME são as próprias classes. Desse modo, a mudança no código-fonte que estava presente, seria a adição de uma nova transição para um estado que também estaria sendo criado, ver Código-fonte 6.2.

Código Fonte 6.2: Trecho de código-fonte retirado do projeto DesignPattern

```
1 gumballMachine.setState(gumballMachine.getWinnerState());
```

Apesar do algoritmo ter alcançando uma taxa de 100% no critério de Precisão nesse projeto, quando analisados os resultados no que diz respeito à Cobertura, não foram obtidas taxas equivalentes à de Precisão, sendo inferiores a 50% para todos os cenários. Uma das razões foi a consequência de uma limitação do algoritmo já citada. Além disso, foi identificado que só informar que há uma nova Transição, pode não ser informação suficiente, vide que Transições podem conter outros elementos (Evento, Guarda e Ação [3]), então haveria uma necessidade de aprimorar o algoritmo para trazer mais informações a respeito de Transições.

Na Tabela 6.4, estão expostos os resultados obtidos no projeto DESystem, de tal maneira que é possível notar que há uma variação entre os resultados de acordo com a ME dada como entrada. Na primeira ME, "Carbuttoncontrol", o algoritmo obteve taxas de precisão e cobertura excelentes, obtendo 100% em ambas. Entretanto, quando o número de classes aumentou para cinco e oito, a taxa de Precisão diminuiu drasticamente, alcançando a marca de 4,88%. Um dos motivos para isso, foi a adição de valores a *enums* que não estavam sendo modelados pela ME. Isso acontece devido a uma limitação do algoritmo, de maneira que qualquer adição de um novo valor a um *ENUM* é classificado como um novo Estado, causando impacto nas classificações de novas Transições.

Como próximo caso a ser analisado, sendo a ME denominada de "Carpositionl", o algoritmo não foi capaz de detectar nenhuma alteração na ME, nem mesmo com o aumento do número de classes foi possível obter alguma alteração. Isso foi ocasionado, pois o elemento presente no código-fonte o qual a ME estava modelando, não possuía o mesmo nome na ME. Enquanto na ME a *DoAction* que foi removida era definida como *mCarPositionIndicator*, no código-fonte estava definido como *mCPI*, sendo as iniciais de cada termo. Entretanto, alteramos esse trecho do código-fonte, para que ficasse igual à ME, e foi observado que o algoritmo pode detectar tal alteração na ME. Dessa forma, foi detectada outra limitação do algoritmo, no que se diz respeito a abreviaturas e iniciais de nome sejam elas no código-fonte ou na ME.

No caso seguinte, trata-se da ME denominada "Hallbutton", onde o algoritmo também conseguiu obter resultados ótimos, alcançando os 100% para ambas as taxas, precisão e cobertura. O algoritmo manteve o excelente resultado até o número de três classes retornadas, entretanto quando houve o aumento para o número de cinco e oito, o algoritmo sofreu uma

DESystem					
Nome ME	**Nº Alt ME**	**Nº Classes**	**Detectadas**	**Precisão**	**Cobertura**
Carbuttoncontrol	2	1	2	100%	100%
		2	2	100%	100%
		3	2	100%	100%
		5	41	4,88%	100%
		8	41	4,88%	100%
Carpositionl	1	1	0	0%	0%
		2	0	0%	0%
		3	0	0%	0%
		5	0	0%	0%
		8	0	0%	0%
Hallbutton	2	1	2	100%	100%
		2	2	100%	100%
		3	2	100%	100%
		5	8	25%	100%
		8	41	4,88%	100%
LanternControl	2	1	2	100%	100%
		2	2	100%	100%
		3	2	100%	100%
		5	2	100%	100%
		8	43	4,65%	100%
Dispatcherl	30	1	31	58,06%	60%
		2	34	52,94%	60%
		3	37	48,65%	60%
		5	41	43,90%	60%
		8	68	26,47%	60%
Doorcontrol	12	1	6	50%	25%
		2	6	50%	25%
		3	6	50%	25%
		5	39	10,26%	25%
		8	39	10,26%	25%
DriveControl	18	1	8	75%	33,33%
		2	8	75%	33,33%
		3	14	42,86%	33,33%
		5	14	42,86%	33,33%
		8	51	11,76%	33,33%

Tabela 6.4: Resultados do Algoritmo proposto para o projeto DESystem.

queda em sua precisão. Conforme já foi citado anteriormente, isso ocorre, também, devido às classes possuírem mudanças no código-fonte, onde estariam presentes termos utilizados no estado, dessa maneira, causaram equívocos no algoritmo induzindo-o a classificar erroneamente as mudanças no código-fonte.

Na ME denominada "LanterControl", o algoritmo também obteve ótimos resultados com o valor de 100% em precisão e cobertura. Para essa ME, o algoritmo conseguiu manter as taxas inalteradas até com o número de cinco classes retornadas. Entretanto, quando o número de classes retornadas foi oito, a taxa de precisão caiu para o valor de 4,65%. Esse efeito de queda foi ocasionado, como as demais citadas anteriormente, devido ao modelo seguido na implementação, o qual considera quaisquer mudanças em *ENUM* como alterações em estado na ME, e transições ou guardas quando há a presença de algum termo ou nome de estado presente na mudança do código-fonte, nos elementos de: Chamada de método, Condições (IF) ou Atribuição de valores.

Na ME denomiada "Dispatcherl", o algoritmo obteve resultados razoáveis para precisão e cobertura. O algoritmo conseguiu obter uma taxa de 60% para cobertura, mesmo havendo um grande número de mudanças presentes na ME. Esse valor, manteve-se em todos os números de classes retornadas. Enquanto a taxa de cobertura manteve inalterada nas cinco variações de número de classes retornadas, a precisão apresentou uma leve queda em seu percentual. Somente nos casos com uma e duas classes retornadas, o algoritmo obteve uma precisão superior a 50%, e nos casos seguintes seguiram uma queda alcançando o percentual de 26,47%.

Conforme citado nos casos anteriores, isso deve-se por conta de que o algoritmo leva em consideração para a classificação termos presentes na ME que também estão presentes em mudanças no código-fonte. No caso da ME "Dispatcherl", também foi notado diversas classificações de "adição de guarda", causadas pelas adições de *IF* ao código-fonte, contendo em seu corpo referência a algum estado na ME, entretanto a ME não modela nenhuma condição de guarda.

Para a ME "Doorcontrol", foram obtidos resultados medianos para as taxas de precisão e cobertura. O algoritmo alcançou um percentual de 50% para precisão, com até três classes retornadas. Entretanto, quando houve o acréscimo para cinco e oito classes, esse percentual de precisão caiu para 10,26%, ocasionado, também, devido às limitações em que o algoritmo

se encontra. Já a taxa de cobertura manteve baixa, em 25%, devido às peculiaridades neste caso específico, como também à limitação da técnica para classificar novas ações de estado. Diferente dos casos citados anteriormente, o código-fonte modelado pela ME “Doorcontrol”, já possuía alguns detalhes da implementação que embora não estivessem modelados na ME, estavam presentes em ambas versões do código-fonte. Logo não seria possível detectar tais mudanças no código-fonte que causariam as alterações na ME, e portanto, reduzindo a cobertura obtida pelo algoritmo.

O último caso, na ME “DriveControl”, o algoritmo obteve nos cenários de uma e duas classes retornadas uma boa taxa de precisão, alcançando o percentual de 75%. No entanto, manteve-se em 75% somente nos dois primeiros cenários, para uma e duas classes retornadas. Quando aumentado o número de classes retornadas, houve uma queda na precisão, caindo inicialmente para 42,86%, para três e cinco classes retornadas e por último caindo para 11,76% para oito classes retornadas. Enquanto isso, o percentual da taxa de Cobertura ficou inferior a 50%, estando com um percentual estável de 33,33% em todas as variações dos números de classes retornadas.

Com a presença de diversas MEs no projeto DESystem, foram calculadas as médias de precisão e cobertura de todos os cenários contidos nesse projeto. Na Tabela 6.5, estão expostos os resultados obtidos com os cálculos das médias de precisão e cobertura. Podemos destacar que a taxa média de cobertura se manteve em 59,76%, em todos os números de classes retornadas, no entanto, a taxa de precisão foi de 69,01% a 8,99% a medida que o número de classes retornadas foi aumentando.

DESystem		
Nº Classes	**Precisão**	**Cobertura**
1	69,01%	59,76%
2	68,28%	59,76%
3	63,07%	59,76%
5	32,41%	59,76%
8	8,99%	59,76%

Tabela 6.5: Médias do Algoritmo proposto para o projeto DESystem.

Houve uma queda na precisão, pois, o algoritmo classifica adições de valores no *enum* como “adição de estado”, mesmo que esse não esteja sendo modelado pela ME. Isso ocorreu

na ME “Dispatcherl”, pois quando houve o acréscimo no número de classes, o algoritmo começou a classificar alterações erradas. Além disso, devido essa classificação equivocada de estado, o algoritmo classificou novas transições com bases no valores de *enum* que não condiziam com a ME. Enquanto isso, a queda da cobertura ocorreu, pois havia alterações na ME que já estavam presentes na versão anterior do código-fonte, ocorrido na ME “Doorcontrol”, logo seria incapaz do algoritmo detectar a alteração na ME sem haver a mudança no código-fonte. Outra limitação do algoritmo, incapacita a classificação de novas ações de estado, pois não houve uma linha de base para que o algoritmo siga na sua classificação.

Após finalizar as análises de cada caso separadamente, foi feito uma análise dos resultados de um modo geral, com o intuito de obter a resposta da questão de pesquisa levantada neste experimento. Para isso, na Tabela 6.6 e Figura 6.1, estão expostos os resultados obtidos por meio do cálculo da média. Portanto, com base nos resultados, é possível notar que para os cenários com até 3 classes retornadas, o algoritmo obteve uma precisão e cobertura média acima de 50%, com exceção do cenário de uma classe, onde o algoritmo obteve a cobertura inferior a 50,80%.

Médias Gerais		
Nº Classes	**Precisão**	**Cobertura**
1	64,78%	48,33%
2	65,60%	50,80%
3	61,56%	50,80%
5	37,12%	50,80%
8	18,84%	50,80%

Tabela 6.6: Médias gerais de Precisão e Cobertura alcançadas pelo Algoritmo proposto.

Foram analisadas as taxas de precisão e cobertura para cada projeto, buscando identificar em qual projeto o algoritmo obteve melhores resultados. Na Figura 6.2a, o projeto que obteve as melhores taxas de precisão, foi o Projeto DesignPattern, com a ME denominada ”GumbalMachine”. Isso ocorreu, pois o algoritmo conseguiu selecionar somente duas alterações que ocorreram na ME, mantendo esse resultado nos demais cenários, com o aumento de classes retornadas. Por outro lado, na Figura 6.2b, para a taxa de cobertura, o projeto no qual o algoritmo obteve melhores resultados, foi o DESystem. Foi observado que embora houvessem muitas alterações nas diferentes MEs do projeto, o algoritmo foi capaz de detectar

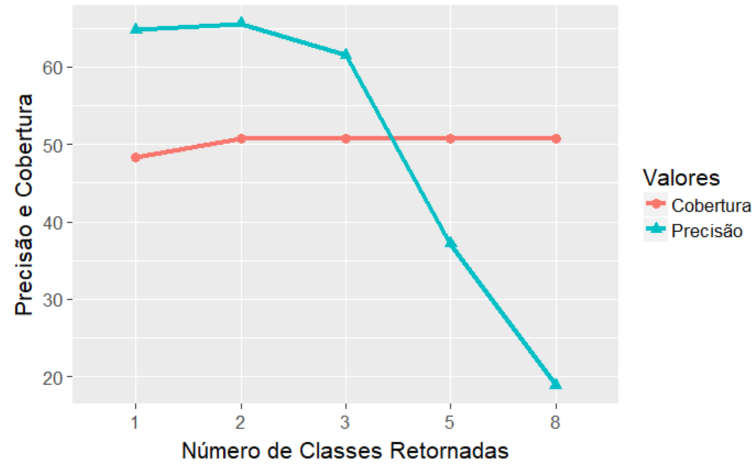
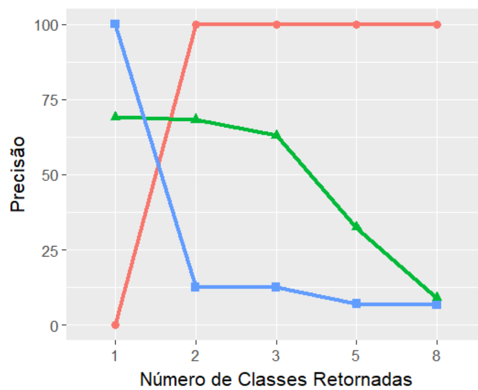
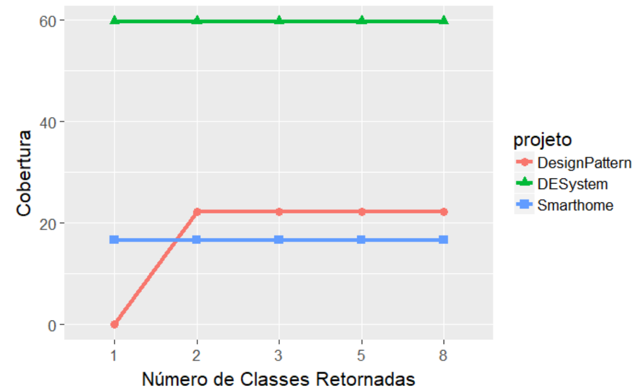


Figura 6.1: Médias Gerais de Precisão e Cobertura do Algoritmo Proposto (Produzido pelo Autor).



(a) Taxas de precisão média alcançadas pelo Algoritmo (Produzido pelo Autor).



(b) Taxas de cobertura média alcançadas pelo Algoritmo (Produzido pelo Autor).

aproximadamente 60% das alterações que ocorreram entre as versões das MEs.

Com base nas hipóteses levantadas para este experimento, é notável que para os cenários de duas e três classes retornadas, as hipóteses $H_0.1$ e $H_0.2$ foram refutadas, como também as hipóteses H_1 e H_2. Isso ocorreu, pois em ambos os cenários há valores superiores a 50% para as taxas de cobertura e precisão. Desse modo, é visto que para as unidades experimentais selecionadas, o algoritmo obteve resultados consideráveis, quando selecionado duas ou três classes JAVA. Entretanto, para os cenários com uma, cinco e oito classes retornadas, não são refutáveis as hipóteses $H_0.1$, $H_0.2$, H_1 e H_2.

Para o cenário com uma classe retornada, os resultados obtidos permitem que as hipóteses $H_0.1$, H_1, H_2 e H_4 sejam refutadas, pois, a taxa de Precisão alcançada foi de 64,78% e a

de Cobertura de 48,33%, não sendo possível refutar umas das hipóteses nulas $H_0.2$. Essa oscilação no cenário com uma classe retornada, foi ocasionada pela não identificação de nenhuma alteração na ME no caso da ME denominada "GumballMachine", ver Tabela 6.3, tornando-a o único caso com média de Cobertura inferior a 50%.

Para os cenários com cinco e oito classes retornadas, os valores obtidos em Precisão foram baixos em comparação aos demais, ficando em 37,12% e 18,84% de Precisão para cinco e oito classes, respectivamente. Com esses resultados, não foi possível refutar a hipótese nula $H_0.1$, somente a $H_0.2$ e H_2 as quais dizem respeito à taxa de Cobertura. Dessa forma, obtendo os piores resultados em comparação aos demais.

Portanto, pode-se definir que os melhores cenários para utilização do algoritmo é com até no máximo três classes retornadas, e o melhor cenário com duas. Desse modo, consegue obter os melhores resultados do algoritmo, visto que com duas classes ele obteve uma precisão média de 65,60% e a cobertura média de 50,80%. Respondendo a questão de pesquisa, para cada cenário há uma média de precisão e cobertura, conforme Tabela 6.6, onde os melhores cenários identificados foram para duas e três classes retornadas, obtendo precisão superior a 60% e cobertura superior a 50%.

6.4 Ameaças à Validade

Com o intuito de validar esse experimento, evitando que falhas no design, elaboração, execução ou documentação, possam interferir nos resultados e assim nas conclusões, foi necessário um levantamento das ameaças que possam conter. Portanto, identificar e tentar reduzir ao máximo as consequências dessas ameaças é um fator importante em qualquer experimento. As ameaças à validade deste experimento estão separadas e descritas abaixo.

6.4.1 Ameaças Internas

Uma das ameaças internas identificadas que poderiam afetar os resultados para este experimento, é a de instrumentação. Para mitigar esse tipo de ameaça, as MEs atualizadas, utilizadas como oráculos para validar os resultados do algoritmo, foram selecionadas com a mesma versão do código-fonte, além de passar por uma verificação do pesquisador para identificar que estão em conformidade com o código-fonte. Desse modo, impossibilitando suposições

de alterações que deveriam conter nas MEs, utilizando somente as que estavam presentes no processo de validar os resultados do algoritmo.

Almejando analisar ambientes diferentes, de modo que as unidades experimentais contemplem diferentes aspectos, foram selecionadas unidades experimentais que estão apresentadas em diferentes contextos. Além disso, buscando obter uma representatividade maior da população, as unidades experimentais selecionadas diferenciam entre sistemas reais, e outros menores.

6.4.2 Ameaças Externas

Para que possa ser generalizado para demais casos, faz-se necessário uma boa seleção das unidades experimentais que serão utilizadas em um experimento. Portanto, para este experimento, foram selecionadas unidades que pudessem de algum modo representar sistemas reais. Logo, foram selecionadas unidades de códigos-fonte abertos, preferencialmente, de empresas como no caso da unidade Smarthome. Dessa maneira, tendo projetos reais que podem representar as implementação de sistemas reais.

6.4.3 Ameaças de Conclusão

Tanto as MEs quanto os códigos-fonte, foram selecionados em versões iguais, diferenciando somente entre versão atualizada e desatualizada, tentando mitigar a possibilidade de que houvesse desigualdades entre versões das MEs e dos códigos-fonte. Para identificar as alterações nas ME, foi feito um processo de análise manual do próprio pesquisador para identificar os erros e acertos do algoritmo, levando em consideração somente as alterações que estavam presente entre as versões das MEs. Além disso, para que fossem obtidas as mudanças de código-fonte, foi utilizada uma ferramenta já testada e avaliada em estudos, no caso a *ChangeDistiller*, de modo que mantenha as mesmas classificações de mudanças para todas as unidades experimentais.

Obter o número de quantas alterações ocorreram entre versões da ME, pode causar equívocos nos resultados obtidos para precisão e cobertura. Buscando mitigar essa ameaça, o próprio pesquisador realizou uma contagem manual com base na taxonomia de alterações em ME, proposta no estudo exploratório (ver Seção 5), para manter-se um padrão em todas

as unidades experimentais. Desse modo, os resultados obtidos pelo algoritmo poderiam ser comparados corretamente com as alterações feitas nas MEs.

Foram utilizados sistemas reais, disponibilizados no Github, para que pudesse ter ambientes diferentes e reais sendo abordados no experimento. Desse modo, evita-se projetos com homogeneidade e permite uma visão de como a técnica se sairia em ambientes de sistemas reais, além de vislumbrar possíveis melhorias no algoritmo com base nas implementações presentes nos projetos.

6.4.4 Ameaças Constructo

Para mitigar os erros relacionados às explicações ou definições inadequadas, do que seriam alterações nas MEs e quais resultados estariam certos ou errados, estes conceitos foram definidos separadamente. Alterações nas MEs foram definidas em um experimento (ver Seção 5.1) com base em classificação feita por voluntários, enquanto os resultados que estariam certos ou errados, foram definidos com bases nas alterações que estavam presentes entre uma versão da ME e outra.

Com o intuito de evitar um viés mono-operação, foram analisados diferentes cenários e casos de alterações nas MEs, de tal modo que fossem analisados o maior número possível de alterações apresentadas nas ME. Entretanto, é ciente que não foi possível cobrir todos os cenários possíveis de alterações realizadas em MEs.

Para evitar uma avaliação de somente uma métrica ou alguma métrica que não fosse representativa o suficiente, com base em outros estudos foram selecionadas duas métricas comuns de avaliação, que se adequaram ao objetivo do experimento: Precisão e Cobertura. Por meio dessas métricas, pôde-se avaliar os resultados alcançados pelo algoritmo mantendo o objetivo do experimento.

6.5 Discussão

Com base nos resultados obtidos nesse experimento foi possível avaliar o algoritmo, e buscar melhorias e eliminações de limitações. Inicialmente, será respondido à questão de pesquisa levantada para esse experimento: "**QP1 -** Qual a taxa de precisão e cobertura ao classificar as alterações necessárias nas MEs após ocorrências de mudanças no código-fonte?".

Com os resultados obtidos nesse experimento (ver Tabela 6.6), pode-se identificar que os melhores cenários para utilização do algoritmo, são com até no máximo 3 classes retornadas. Uma das causas para quando o número de classes retornadas aumenta e a precisão desce, é devido à forma na qual o algoritmo está implementado. O algoritmo considera que a presença de termos referente aos estados em chamadas de métodos ou atribuições de valores são classificadas como novas transições, sem nenhum pré ou pós processamento (ver Código-fonte 6.3).

Código Fonte 6.3: trecho de código retirado do projeto Smarthome

```
1 logger.warn("BaseThingHandler.initialize()
2     will be removed soon, ThingStatus can be set
3     manually via updateStatus(ThingStatus.ONLINE)");
```

Além disso, a classificação de novas guardas é feita ao localizar termos que representem os estados dentro dos corpos das condições IF ou ELSE IF. Isso ocasionou uma queda na precisão do algoritmo, pois as MEs não continham condições de guardas em suas transições. Para exemplificar o cenário de adição de guarda, o Código-fonte 6.4 expõe a condição de IF que foi classificada como sendo uma guarda, entretanto não estava presente na ME atualizada.

Código Fonte 6.4: Condição IF retirada do projeto Smarthome

```
1 if (ip == null || StringUtils.isEmpty(pin) || port.intValue() == 0) {
2     updateStatus(ThingStatus.OFFLINE, ThingStatusDetail.
        CONFIGURATION_ERROR, "Configuration incomplete");
3 }
```

Outra melhoria que pode ser realizada e foi identificada durante a análise, foi no que se diz respeito às adições de novos estados às MEs. O algoritmo não faz nenhum tipo de processamento para avaliar se aquela adição de valor no *ENUM* será realmente um novo estado. Essa limitação do algoritmo, contribuiu para que a precisão também caísse quando novas classes que tivessem *Enumeration* sofressem alguma adição de valor ao *ENUM*.

A presença de alterações nas MEs que já estavam presentes no código-fonte mesmo na versão antiga do código-fonte, ocasionaram uma queda na cobertura do algoritmo, já que o mesmo baseia-se na presença de mudanças de código-fonte. Entretanto, devido a forma como o algoritmo proposto busca as alterações nas MEs em nosso contexto, esse tipo de

situação estaria fugindo do que estamos propondo, mas que não deixa de ser uma possível melhoria ao algoritmo.

Além disso, foi possível identificar possíveis melhorias para aplicar no algoritmo e assim realizar novos estudos comparativos para identificar se realmente houve melhoras na sua eficácia. Algumas dessas melhorias podem ser realizadas com um pós-processamento dos resultados, tentando identificar repetições nas transições, entretanto necessitaria de uma análise dinâmica para avaliar em qual estado o objeto se encontra antes de ocorrer a mudança de Estado. A adição de um pré-processamento para identificar elementos no código-fonte que realmente podem causar alterações, evitando valores contidos em Strings e que podem acabar confundindo o algoritmo.

Portanto, o algoritmo obteve resultados satisfatórios no que se diz respeito a sua eficiência, verificando as taxas de precisão e cobertura. Mesmo havendo casos em que ele não foi capaz de identificar alterações devido à forma como o elemento da ME estava representado no código-fonte ou simplesmente não foi capaz de detectar, a média dos resultados manteve-se superior a 50% para precisão e cobertura. É ciente que o algoritmo ainda é passível de melhorias, que podem trazer bons resultados e novos estudos para essa área.

Capítulo 7

Conclusões

Para prover uma solução que auxilie no processo de manter as MEs de um sistema atualizadas, este trabalho foi desenvolvido para classificar e sugerir alterações nas MEs e fornecê-las ao responsável por mantê-las atualizadas. Para que essa solução fosse alcançada, foram realizados um estudo e um experimento, para identificarmos as mudanças em código-fonte que poderiam causar alterações em ME, e avaliarmos a eficiência do algoritmo proposto, respectivamente.

Após o primeiro estudo, foi possível constatar que as mudanças no código-fonte podem causar alterações nas MEs, e além disso, podem ser mapeadas com as alterações em MEs. Com isso, elaboramos uma taxonomia para alterações em MEs, de modo a auxiliar o usuário na identificação do que deve ser alterado, como também no processo de classificação e sugestão das alterações em MEs.

Com base nos resultados obtidos a partir do primeiro estudo, foi elaborado um algoritmo para automatizar o processo de classificação e sugestão de alterações, tendo, também, como base a taxonomia proposta. Em seguida, esse algoritmo foi implementado em uma ferramenta para que pudesse ser executado em um ambiente real de desenvolvimento, buscando reduzir o esforço no processo de manter as MEs atualizadas.

Foi realizado um experimento para avaliar a eficiência da técnica, no que se diz respeito a suas taxas de precisão e cobertura. Este experimento, constatou que o algoritmo proposto conseguiu obter uma taxa média de precisão de 65,60% e uma cobertura de 50,80%. Analisando os resultados, foi detectado que em alguns casos, o algoritmo sugeriu alterações que poderiam estar corretas, pois poderiam se adequar a alguma restrição de guarda da ME,

entretanto, como não estas alterações nas estavam presentes na nova versão da ME, foi considerada como errada.

7.1 Contribuições Alcançadas

Com os resultados obtidos por meio dos estudos desenvolvidos neste trabalho, pode-se elencar as seguintes contribuições alcançadas:

1. um estudo que expôs tipos de mudanças no código-fonte que podem causar alterações em MEs, gerando uma taxonomia de alterações em ME, servindo como base para o mapeamento entre mudanças no código-fonte e alterações em ME;

2. um mapeamento entre mudanças de código-fonte e alterações em ME, gerado por meio do estudo exploratório, tendo o intuito de contribuir e auxiliar na elaboração do algoritmo proposto;

3. um algoritmo que tem como base esse mapeamento, capaz de sugerir e classificar alterações em MEs, tendo como entrada diferentes versões do código-fonte de um sistema e as MEs que deverão ser analisadas;

4. uma ferramenta, que implementa esse algoritmo proposto, disponível para utilização em ambientes reais.

7.2 Limitações do algoritmo

É ciente que o algoritmo proposto tem capacidade de melhorias para atingir resultados melhores no que diz respeito a suas taxas de precisão e cobertura, como também uma ampliação da taxonomia para demais elementos da ME que não foram contemplados. Desse modo, abrindo novos horizontes que possam ser explorados com base nos resultados obtidos por este trabalho e com outras pesquisas que possam incrementar as informações aqui alcançadas, podem ser listados as seguintes limitações:

1. O algoritmo ainda não sugere adição de novos comportamentos (Do, Entry e Exit) ao Estado. Isso ocorre por não ter sido identificado um padrão para classificar se deter-

minado método deve ser um comportamento específico ou não do Estado. Portanto, impossibilitando a classificação da adição de ações do estado na ME;

2. O algoritmo realiza a classificação de alterações em transição ou guarda baseadas somente na localização do termo referente ao estado presentes nos corpos de métodos, *IFs*, ou parâmetros de chamadas de métodos. Isso tem como consequência a limitação de sugerir só quando houver a presença do termo nos elementos citados, ou causar a sugestão de alterações erradas, pois o termo embora igual a ME pode não representar o estado na ME;

3. O algoritmo não realiza nenhum pós-processamento, para evitar a duplicação de sugestões a serem aplicadas nas MEs, tendo como consequência uma verificação a mais para confirmar se a sugestão é válida ou não pelo o usuário ao receber a saída do algoritmo;

4. A indexação dos documentos na ferramenta Lucene é feita sem nenhum pré-processamento, no qual poderia remover valores de Strings no processo de indexação, podendo reduzir o número de classes retornadas que não têm relação com a ME;

5. O algoritmo considera todas as adições de valores no Enum como adições de novos Estados, podendo reduzir a precisão do algoritmo, caso haja valores de Enum não condizentes com os Estados presentes na ME;

6. O algoritmo não sugere atualizações de estado, somente adição ou remoção, havendo uma necessidade de análise comparativa entre código-fonte e ME. Desse modo, pode-se sugerir duas alterações ao invés de somente uma que teria o mesmo efeito;

7. Devido ao escopo da pesquisa, o algoritmo só classifica alterações em estados, transições e estados composto, não contemplando outros elementos e com isso, podendo perder Cobertura quando apresentar elementos não contidos no escopo;

8. A ferramenta só recebe uma ME por classificação, tendo a necessidade de repetir a execução caso haja mais que uma ME a ser analisadas;

9. A ferramenta só recebe a ME em formato XML, e que os elementos de Estado e Transição estejam com as *tags state* e *transition*. Caso esteja presente em outro for-

mato, não será possível extrair os elementos, logo não haverá classificação ou sugestão de alterações.

7.3 Trabalhos Futuros

Conforme foi citado na seção anterior, neste trabalho há a possibilidade de incrementá-lo e aprimorá-lo para obter resultados melhores, e podemos destacar algumas possibilidades de trabalhos que podem dar continuidade a este:

- Ampliar a taxonomia de alterações em MEs incluindo novos elementos da ME que ainda não foram contemplados, como por exemplo *Choice* e *Submachine State*;
- Realizar um pós-processamento nas classificações sugeridas pelo algoritmo, a fim de aprimorar a precisão alcançada pelo algoritmo reduzindo duplicações nas sugestões na ME, ou evitar sugestões de alterações que já estão presentes na ME, ou alguma verificação para identificar se é realmente uma alteração na ME;
- Realizar um pré-processamento com um intuito de buscar melhorar os resultados obtidos pela técnica de IR, como também abstrair valores que possam confundir o algoritmo, como por exemplo valores em Strings contendo termos da MEs, podendo melhorar a precisão do algoritmo;
- Realizar um estudo em um ambiente real de desenvolvimento para avaliar a redução do esforço no processo de atualização das MEs, como também coletar informações que possam contribuir na avaliação do resultado gerado em formato PDF, pela ferramenta.

Bibliografia

[1] The atlas transformation language. Accessed: 2018-11-28.

[2] Lucene apache. Accessed: 2018-11-14.

[3] OMG object management group. Accessed: 2017-07-14.

[4] Manuj Aggarwal and Sangeeta Sabharwal. Test case generation from uml state machine diagram: A survey. In *Computer and Communication Technology (ICCCT), 2012 Third International Conference on*, pages 133–140. IEEE, 2012.

[5] Everton L. G. Alves, Patricia D. L. Machado, and Franklin Ramalho. Automatic generation of built-in contract test drivers. *Software & Systems Modeling*, 13(3):1141–1165, Jul 2014.

[6] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[7] David Bedok. Application of petri-nets in object-oriented environment. In *Computational Intelligence and Informatics (CINTI), 2016 IEEE 17th International Symposium on*, pages 000117–000122. IEEE, 2016.

[8] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. *Pearson Correlation Coefficient*, pages 1–4. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[9] Lionel C Briand, Yvan Labiche, and Yihong Wang. Using simulation to empirically investigate test coverage criteria based on statechart. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 86–95. IEEE, 2004.

[10] Eladio Domı, Beatriz Pérez, Ángel L Rubio, et al. A systematic review of code generation proposals from state machine specifications. *Information and Software Technology*, 54(10):1045–1066, 2012.

[11] Chartchai Doungsa-ard, Keshav Dahal, and Zeeshan Pervez. Juicegen: The junit test generation tool from the uml state machine diagram. In *Software, Knowledge, Information Management and Applications (SKIMA), 2014 8th International Conference on*, pages 1–8. IEEE, 2014.

[12] Rainer Findenig, Thomas Leitner, Michael Veiten, and Wolfgang Ecker. Fast and accurate uml state chart modeling using tlm+ control flow abstraction. In *High Level Design Validation and Test Workshop (HLDVT), 2010 IEEE International*, pages 97–102. IEEE, 2010.

[13] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, Nov 2007.

[14] Beat Fluri and Harald C Gall. Classifying change types for qualifying change couplings. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 35–45. IEEE, 2006.

[15] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[16] R. Groz, A. Simao, N. Bremond, and C. Oriat. Revisiting ai and testing methods to infer fsm models of black-box systems. In *2018 IEEE/ACM 13th International Workshop on Automation of Software Test (AST)*, pages 16–19, May 2018.

[17] Abdelwahab Hamou-Lhadj and Timothy C Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 42–55. IBM Press, 2004.

[18] Jane Huffman Hayes, Alex Dekhtyar, Senthil Karthikeyan Sundaram, E. Ashlee Holbrook, Sravanthi Vadlamudi, and Alain April. Requirements tracing on target (retro):

improving software maintenance through traceability recovery. *Innovations in Systems and Software Engineering*, 3(3):193–202, Sep 2007.

[19] Djoerd Hiemstra. A probabilistic justification for using tf× idf term weighting in information retrieval. *International Journal on Digital Libraries*, 3(2):131–139, 2000.

[20] G. Hong-jie, M. Fan-rong, and X. Zhan-guo. Research on uml-based modeling of workflow manage system. In *2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 835–839, Aug 2008.

[21] A. F. Horadam. A generalized fibonacci sequence. *The American Mathematical Monthly*, 68(5):455–459, 1961.

[22] Bernard J Jansen and Soo Young Rieh. The seventeen theoretical constructs of information searching and information retrieval. *Journal of the Association for Information Science and Technology*, 61(8):1517–1534, 2010.

[23] Amir A Khwaja and Joseph E Urban. A property based specification formalism classification. *Journal of Systems and Software*, 83(11):2344–2362, 2010.

[24] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[25] David Lo, Leonardo Mariani, and Mauro Santoro. Learning extended fsa from software: An empirical assessment. *Journal of Systems and Software*, 85(9):2063 – 2076, 2012. Selected papers from the 2011 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA 2011).

[26] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 501–510, May 2008.

[27] I. Madari, L. Lengyel, and G. Mezei. Incremental model synchronization by bidirectional model transformations. In *2008 IEEE International Conference on Computational Cybernetics*, pages 215–218, Nov 2008.

[28] L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, pages 117–126, Nov 2008.

[29] Leonardo Mariani, Mauro Pezze, and Mauro Santoro. Gk-tail+ an efficient approach to learn software models. *IEEE Transactions on Software Engineering*, 2016.

[30] A. Mishra and S. Vishwakarma. Analysis of tf-idf model and its variant for document retrieval. In *2015 International Conference on Computational Intelligence and Communication Networks (CICN)*, pages 772–776, Dec 2015.

[31] Iftikhar Azim Niaz, Jiro Tanaka, et al. Mapping uml statecharts to java code. In *IASTED Conf. on Software Engineering*, pages 111–116, 2004.

[32] A. Paradkar. Plannable test selection criteria for fsms extracted from operational specifications. In *15th International Symposium on Software Reliability Engineering*, pages 173–184, Nov 2004.

[33] Karol Rástočný and Andrej Mlynčár. Automated change propagation from source code to sequence diagrams. In A Min Tjoa, Ladjel Bellatreche, Stefan Biffl, Jan van Leeuwen, and Jiří Wiedermann, editors, *SOFSEM 2018: Theory and Practice of Computer Science*, pages 168–179, Cham, 2018. Springer International Publishing.

[34] Gregorio Robles, Truong Ho-Quang, Regina Hebig, Michel R. V. Chaudron, and Miguel Angel Fernandez. An extensive dataset of uml models in github. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 519–522, Piscataway, NJ, USA, 2017. IEEE Press.

[35] Atanas Rountev, Olga Volgin, and Miriam Reddoch. Static control-flow analysis for reverse engineering of uml sequence diagrams. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 96–102. ACM, 2005.

[36] M. Sasaki, S. Matsumoto, and S. Kusumoto. Integrating source code search into git client for effective retrieving of change history. In *2018 IEEE Workshop on Mining and Analyzing Interaction Histories (MAINT)*, pages 1–5, March 2018.

[37] Christian Schwarzl and Bernhard Peischl. Test sequence generation from communicating uml state charts: An industrial application of symbolic transition systems. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 122–131. IEEE, 2010.

[38] Yan Sommerville. *Software Engineering*. Addison-Wesley, 9a edição edition, 2011. ISBN-13: 978-0-13-703515-1.

[39] Koji Taniguchi, Takashi Ishio, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Extracting sequence diagram from execution trace of java program. In *Principles of Software Evolution, Eighth International Workshop on*, pages 148–151. IEEE, 2005.

[40] N. Walkinshaw and M. Hall. Inferring computational state machine models from program executions. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 122–132, Oct 2016.

[41] Neil Walkinshaw, Ramsay Taylor, and John Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, Jun 2016.

[42] S. M. White. Modeling a system of systems to analyze requirements. In *2009 3rd Annual IEEE Systems Conference*, pages 83–89, March 2009.

[43] Suresh Yadla, Jane Huffman Hayes, and Alex Dekhtyar. Tracing requirements to defect reports: an application of information retrieval techniques. *Innovations in Systems and Software Engineering*, 1(2):116–124, Sep 2005.

[44] H. Yoo and T. Shon. Inferring state machine using hybrid teacher. In *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 504–509, Dec 2016.

[45] Jingjun Zhang, Furong Li, and Yang Zhang. Aspect-oriented requirements modeling. In *Software Engineering Workshop, 2007. SEW 2007. 31st IEEE*, pages 35–40. IEEE, 2007.

Apêndice A

Mapeamento - Tipos de mudanças no código-fonte que podem causar alterações em Máquinas de Estado

Devido a tabela que expõe o mapeamento ter um tamanho considerável, ela também está disponível on-line, por meio do seguinte link: https://drive.google.com/open?id=12y8sO_ZG150bv3Uk4Sk6nE1wShf4bgKtvfpn-TaL6IQ. A tabela do mapeamento encontra-se alocado nas páginas seguintes.

Types of Changes in Source Code that Cause Changes in State Machines

Element in Source Code	Source Code Change Type	State Machine Change Type	Justification	Scenario Able	Scenario Unable	Example	

Method Call	Create a new Method Call	Create a new Transition	Whenever create a new method call in source code, which will indicate the state does switch used in the StateMachine, like a Transition, it's necessary update the State Machine with create a new Transition	Given a JAVA class that has a method where was created a new method call, which will indicate the state does switch, then, it's necessary update de StateMachine creating a new transition.	Whenever the created method call doesn't represent any Transition in the StateMachine, and hasn't any switch of the States.	system.setStatus(Status.OFF);
						...
						turnOn();
						...
						public void turnOn(){
						...
						system.setStatus(Status.ON);
						...
						}
							Turn On
							Off
							On

		Create a new Event	Whenever create a new method call and it is responsible for define the Event in a Transition.	Given a new method call, which will be used in the StateMachine like a Event. This new method trigger a event necessary to occur a Transition, so it's emerges a necessity off create a new Event in the Transition	Whenever the created method call doesn't represent any Event in the StateMachine, and hasn't any method call that represent Transition or Action.	startSystem();
						...
						loginSystem();
						...
						public void loginSystem(){
						...
						system.setStatus(Status.
						INITIALIZED);
						...}
							start system, **login System**
							user == userSaved && password == passwordSaved
							initialize
							Off
							On
							Initialized

		Create a new Action	Whenever create a new method call in source code which indicate that a Action will be call in the StateMachine.	Given a new method call which will be used in a Action into a StateMachine. This new method call is used after a event and a guard are satisfied, so it's emerges a necessity of create a new Action in the Transition.	Whenever the created method call doesn't represent any Action in the StateMachine.	public void startSystem(){
						...
						runServer();
						system.setStatus(Status.
						INITIALIZED);
						...
						}
							start system, login System
							user == userSaved && password == passwordSaved
							initialize, **run Server**
							Off
							On
							Initialized

		Create a new Entry Action	Whenever create a new method call in source code which indicate that a Entry Action will be call in the StateMachine.	Given a new method call which will be used in a Entry Action into a StateMachine. This new method call is called when a object assume a new State and execute the Entry Action, so it's emerges the necessity of create a new Entry Action in the State.	Whenever the created method call doesn't represent any Entry Action in the StateMachine, and doesn't happen any switch state after.	...
						player.setState(States.
						INITIALIZING);
						...
						try{
						startScene();
						player.setState(States.PLAYING);
						}
						...
							Idle
							Initializing
							Playing
							entry / start scene

		Create a new Exit Action	Whenever create a new method call in source code which indicate that a Exit Action will be call in the StateMachine.	Given a new method call which will be used in a Entry Action into a StateMachine. This new method is called when get out of a state to other, so it's emerges the necessity of create a new Exit Action.	Whenever the created method call doesn't represent any Exit Action in the StateMachine, and doesn't happen any switch state before.	...
						player.setState(States.
						INITIALIZING);
						...
						try{
						startScene();
						player.setState(States.PLAYING);
						}
						...
						quitGame();
						...
						finalize;
							Idle
							Initializing
							Playing
							entry / start scene
							do / counting time
							exit / quit game

		Create a new Do Action	Whenever create a new method call in source code which indicate that a Do Action will be call in the StateMachine.	Given a new method call which will be used like a Do Action. Then, when happens switch state, this method call can be used and executing only during the specific state.	Whenever the created method call doesn't represent any Do Action in the StateMachine, and this method call will doesn't called in the State.	...
						if (player.isPlaying()) {
						countingTime();
						}
						...
							Idle
							Initializing
							Playing
							entry / start scene
							do / counting time

		Rename a Transition	Whenever a method call is used for represent a Transition in the StateMachine and the same method call has change in its name.	Given a method call is used for represent a Transition (i.e turnOn()) in the StateMachine was updated name (i.e startSystem()), so it's necessary update the StateMachine with Rename a Transition.	Whenever the method call doesn't represent any Transition in the StateMachine, and hasn't any switch of the States.	system.setStatus(Status.OFF);
						...
						~~**turnOn();**~~
						startSystem();
						...
						public void ~~**turnOn**~~ **startSystem**()
						{
						...
						system.setStatus(Status.ON);
						...
						}
							~~Turn On~~
							Start System
							Off
							On

		Rename a Event	Whenever a method call is used for represent a Event in the StateMachine and the same method call has change in its name.	Given a method call is used for represent a Event (i.e loginSystem()) in the StateMachine was updated name (i.e loadSystem()), so it's necessary update the StateMachine with Rename a Event.	Whenever the method call doesn't represent any Event in the StateMachine.	... ~~**loginSystem();**~~ **loadSystem();** ... public void ~~**loginSystem**~~ **loadSystem**(){ ... system.setStatus(Status. INITIALIZED); ...}	start system, ~~login~~ **load** System user == userSaved && password == passwordSaved initialize, run Server Off On Initialized

		Rename a Action	Whenever a method call is used for represent a Action in the StateMachine and the same method call has change in its name.	Given a method call is used for represent a Action (i.e runServer()) in the StateMachine was updated name(i.e startServer()), so it's necessary update the StateMachine with Rename a Action.	Whenever the method call doesn't represent any Action in the StateMachine.	public void loadSystem(){
						...
						~~**runServer();**~~
						startServer();
						system.setStatus(Status.
						INITIALIZED);
						...
						}
							start system, load System
							user == userSaved && password == passwordSaved
							initialize, ~~run~~ **start** Server
							Off
							On
							Initialized

Types of Changes in Source Code that Cause Changes in State Machines

Element in Source Code	Source Code Change Type	State Machine Change Type	Justification	Scenario Able	Scenario Unable	Example

		Rename a Entry Action	Whenever a method call is used for represent a Entry Action in the StateMachine and the same method call has change in its name.	Given a method call is represented like a Entry Action (i.e startScene()) in the StateMachine was updated name (i.e startPhase()), so it's necessary update the StateMachine with Rename a Entry Action.	Whenever the method call doesn't represent any Entry Action in the StateMachine.	...
						player.setState(States.
						INITIALIZING);
						...
						try{
						~~startScene();~~
						startPhase();
						player.setState(States.PLAYING);
						}
						...

		Rename a Exit Action	Whenever a method call is used for represent a Exit Action in the StateMachine and the same method call has change in its name.	Given a method call is represented like a Exit Action (i.e quitGame()) in the StateMachine was updated name(i.e exitGame()), so it's necessary update the StateMachine with Rename a Exit Action.	Whenever the method call doesn't represent any Exit Action in the StateMachine, and doesn't happen any switch state before.	...
						player.setState(States.
						INITIALIZING);
						...
						try{
						startPhase();
						player.setState(States.PLAYING);
						}
						...
						~~quitGame();~~
						exitGame();
						...
						finalize;

		Rename a Do Action	Whenever a method call is used for represent a Do Action in the StateMachine and the same method call has change in its name.	Given a method call is represented like a Do Action (i.e countingTime()) in the StateMachine was updated name(i.e changeSkin()), so it's necessary update the StateMachine with Rename a Do Action.	Whenever the method call doesn't represent any Do Action in the StateMachine, and this method call doesn't called in the State.	...
						if (player.isPlaying()) {
						~~countingTime();~~
						changeSkin();
						}
						...

		Remove a Transition	Whenever the method call was used for represent a Transition in the StateMachine and is removed.	Given a method call which was represented a Transition in the StateMachine, and after it's removed of the code. Then, it's necessary update the StateMachine with Remove Transition.	Whenever the method call doesn't represent any Transition in the StateMachine, and hasn't any state change.	system.setStatus(Status.OFF);
						...
						~~startSystem()~~;
						...
						public void startSystem(){
						...
						system.setStatus(Status.ON);
						...
						}

		Remove a Event	Whenever the method call was used for represent a Event in the StateMachine and this method is removed.	Given a method call which was represented a Event in the StateMachine, and after it's removed of the code. Then, it's necessary update the StateMachine with Remove Event.	Whenever the method call doesn't represent any Event in the StateMachine, and it doesn't responsible for trigger a transition or action.	... ~~loadSystem();~~ ... public void loadSystem(){ ... system.setStatus(Status. INITIALIZED); ...}

		Remove a Action	Whenever the method call was used for represent a Action in the StateMachine and it is removed.	Given a method call which was represented a Action in the StateMachine, and after it is removed of the code. Then, it's necessary update the StateMachine with Remove Action.	Whenever the method call doesn't represent any Action in the StateMachine.	public void loadSystem(){
						...
						~~startServer();~~
						system.setStatus(Status.
						INITIALIZED);
						...
						}

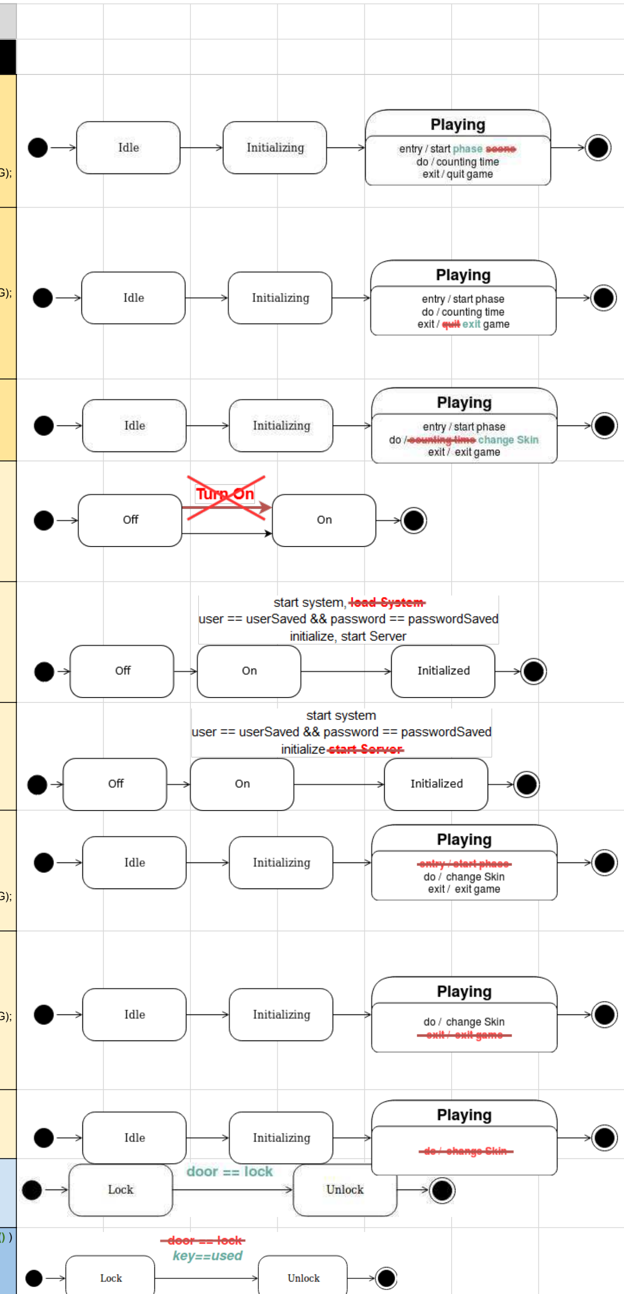
		Remove a Entry Action	Whenever the method call was used for represent a Entry Action in the StateMachine and is removed.	Given a method call which was represented a Entry Action in the StateMachine, and after it is removed of the code. Then, it's necessary update the StateMachine with Remove Entry Action.	Whenever the method call doesn't represent any Entry Action in the StateMachine, or it wasn't in some state.	...
						player.setState(States.
						INITIALIZING);
						...
						try{
						~~startPhase();~~
						player.setState(States.PLAYING);
						}
						...

		Remove a Exit Action	Whenever the method call was used for represent a Exit Action in the StateMachine and it is removed.	Given a method call which was represented a Exit Action in the StateMachine, and after it is removed of the code. Then, it's necessary update the StateMachine with Remove Exit Action.	Whenever the method call doesn't represent any Exit Action in the StateMachine, or it wasn't in some state.	...
						player.setState(States.
						INITIALIZING);
						...
						try{
						startPhase();
						player.setState(States.PLAYING);
						}
						...
						~~exitGame();~~
						...
						finalize;

		Remove a Do Action	Whenever the method call was used for represent a Do Action in the StateMachine and it is removed.	Given a method call which was represented a Do Action in the StateMachine, and after it is removed of the code. Then, it's necessary update the StateMachine with Remove a Do Action.	Whenever the method call doesn't represent any Do Action in the StateMachine, and this method doesn't called in the State.	...
						if (player.isPlaying()) {
						~~changeSkin();~~
						}
						...

	Create IF	Create a new guard	Whenever create a condition IF which will represent a Guard in a Transition, so it's necessary add a new guard in the Transition.	Given a new condition IF create in a method of the source code, which is used in StateMachine like a guard of a Transition, it's necessary update in the StateMachine.	Whenever the created IF doesn't represent a guard in the StateMachine, and it doesn't compose a method represented in the StateMachine.	... IF (door.isLock) { door. openDoor(); }...

			Whenever update a condition IF in the source code changing a expression, and this IF is used in the StateMachine like a Guard.	Given a condition IF in the source code, which is used in State Machine like a Guard into a Transition, was changed a expression, thus it's necessary update the StateMachine with alter guard.	Whenever the IF doesn't represent a guard in the StateMachine, and it doesn't compose a method represented in the StateMachine.	...IF (~~door.isLock~~ key.isUsed())
						{
						door.openDoor();
						}
						...



Types of Changes in Source Code that Cause Changes in State Machines

Element in Source Code	Source Code Change Type	State Machine Change Type	Justification	Scenario Able	Scenario Unable	Example
IF Condition	Update Condition IF	Update Guard and Expression	Whenever update a condition IF in the source code adding a new expression, and this IF is used in the StateMachine like a Guard.	Given a condition IF in the source code, which is used in State Machine like a Guard into a Transition, was added a new expression, thus it's necessary update the StateMachine with alter guard.	Whenever the IF doesn't represent a guard in the StateMachine, and it doesn't compose a method represented in the StateMachine.	...IF (door.isLock && key.isUsed()) { door.openDoor(); } ...
			Whenever update a condition IF in the source code removing a expression, and this IF is used in the StateMachine like a Guard.	Given a condition IF in the source code, which is used in State Machine like a Guard into a Transition, it was removed a expression, thus it's necessary update the StateMachine with alter guard.	Whenever the IF doesn't represent a guard in the StateMachine, and it doesn't compose a method represented in the StateMachine.	...IF (door.isLock ~~&& key.isUsed()~~) { door.openDoor(); } ...
	Remove IF	Remove a guard	Whenever remove a condition IF and this IF is used in the StateMachine like a Guard. Then, it's necessary update the StateMachine removing a guard.	Given a condition IF in the source code, which is used in State Machine like a Guard into a Transition, it was remove, then it's necessary update the StateMachine with remove guard.	Whenever the removed IF doesn't represent a guard in the StateMachine, and it doesn't compose a method represented in the StateMachine.	...~~IF (door.isLock && key.isUsed()) {~~ door.openDoor(); ~~}~~ ...
Scape (Return)	Create Scape Return Type	Alter body Composite State	Whenever a method is void but became a method with return and is created the scape (return) of the method, or it is created a new return of the method which do verification of the states that compose the Composite State. (PS: "Alter body Composite State" is a Cascade Change, is necessary analize other changes like "Create a new Transition" and "Remove Transition", etc...).	Given a method void, which became a method with return, or it's created a new return of the method, it has the states that compose a Composite State, it is necessary update the body of the Composite State with that states in return method.	Whenever the scpae (return) of the method hasn't any state or value that represent some state.	public boolean isInitialized( ){ ... **return state.ONLINE || state.UNKNOW || state.OFFLINE;** }
	Remove Scape Return Type	Alter body Composite State	Whenever remove a scape (return) of the method which did verification of the states that compose the Composite State. (PS: "Alter body Composite State" is a Cascade Change, is necessary analize other changes like "Create a new Transition" and "Remove Transition", etc...).	Given a scape (return) of the method, which has the states that compose a Composite State, this scape (return) of the method is removed, then it is necessary update the StateMachine altering body Composite State.	Whenever the scape (return) of the method hasn't any state or value that represent some state.	public boolean isInitialized( ){ ... ~~return state.ONLINE || state.UNKNOW || state.OFFLINE;~~ }
	Update Scape Return Type	Alter body Composite State	Whenever update a scape (return) of the method, which do verification of the states that compose the Composite State, either for add a new value/condition or remove a value/condition or update a value/condition. (PS: "Alter body Composite State" is a Cascade Change, is necessary analize other changes like "Create a new Transition" and "Remove Transition", etc...).	Given a scape (return) of the method which has the states that compose a Composite State, it's updated scape (return) of the method, then it is necessary update the StateMachine with alter body Composite State.	Whenever the scape (return) of the method hasn't any state or value that represent some state.	public boolean isInitialized( ){ ... return state.ONLINE ~~|| state.UNKNOW~~ || state.OFFLINE; }
Enum	Create a new Value	Create a new State	Whenever the new value in the Enum Java will be used like a State in a StateMachine, so it's necessary update the StateMachine creating a new State. (PS: Create a new State is a Cascade Change, is necessary analize other changes like "Create a new Transition").	Given an Enum JAVA which is used for represent some or all states in a StateMachine, after add a new value in Enum is necessary update the StateMachine, because the values in Enum are States in a StateMachine.	Whenever the Enum JAVA doesn't represent the states of the StateMachine.	Public Enum States{... **STOP**, IS_RUNNING, IDLE; ... }
	Update a Value	Rename State	Whenever the value in the Enum Java is used like a State in the StateMachine, after update value Enum it's necessary update the StateMachine with rename State.	Given a Value Enum JAVA which is used for represent some state in the StateMachine, if value change in the Enum is necessary update the StateMachine with Rename State, because the values in Enum are States in a StateMachine.	Whenever the Enum JAVA doesn't represent the states of the StateMachine.	Public Enum States{... ~~STOP~~ **FINISHED**, IS_RUNNING, IDLE; ... }
	Remove a Value	Remove a State	The value in a Enum Java was used like a State in a StateMachine, so after remove the value it's necessary realize a change removing the State in the StateMachine. (PS: "Remove a State" is a Cascade Change, is necessary analize other changes like "Remove a Transition" and "Remove Do Action" and "Remove Entry Action" and etc)	Given a Value Enum JAVA is used for represent some state that compose the StateMachine, when removed the value in the Enum JAVA it's necessary update the StateMachine with Remove State.	Whenever the Enum JAVA doesn't represent the states of the StateMachine.	Public Enum States{... ~~FINISHED~~, IS_RUNNING, IDLE, ON; ... }
Assignment	Create a new Assignment	Alter Target State	Whenever create a new assignment and it is responsible for define the Target State in a Transition.	Given a assignment is created for represent the Target State of a Transition. Then, it's necessary update the StateMachine with Alter Target State.	Whenever the created assignment doesn't represent a target State.	public void accelerate(){ ... **newState = Status.ON;** car.setStatus(newState); ...}
	Update a Assignment	Alter Target State	Whenever the assignment define the Target State in a Transition and happens change the Target State.	Given a assignment is used for represent the Target State in a Transition, and this assignment was another value of the state, so it's necessary update the StateMachine with Alter Target State.	When the assignment doesn't related change of state.	... ~~newState = Status.ON;~~ **newState = Status.INITIALIZED;** ... system.setStatus(newState); ...
	Remove a Assignment	Alter Target State	Whenever the assignment defined the Target State in a Transition and was removed.	Given a assignment was used to define and represent the Target State in a Transition, and this assignment was removed, so it's necessary update the StateMachine with Alter Target State, because in another part of code can have a new Target State.	Whenever the assignment doesn't be related state change, or doesn't link to Target State.	system.setStatus(newState); ... ~~newState = Status.On;~~ ... system.setStatus(newState); ...