UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CÂMPUS CORNÉLIO PROCÓPIO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

ROMULO DE ALMEIDA NEVES

AUTOMATIZAÇÃO DA GERAÇÃO DE CASOS DE TESTES: UM ESTUDO DE CASO NA EMPRESA EXACTUS SOFTWARE

DISSERTAÇÃO

CORNÉLIO PROCÓPIO

2019

ROMULO DE ALMEIDA NEVES

AUTOMATIZAÇÃO DA GERAÇÃO DE CASOS DE TESTES: UM ESTUDO DE CASO NA EMPRESA EXACTUS SOFTWARE

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Informática da Universidade Tecnológica Federal do Paraná – UTFPR como requisito parcial para a obtenção do título de "Mestre em Informática".

Orientador: Prof. Dr. Willian M. Watanabe

CORNÉLIO PROCÓPIO

2019

Dados Internacionais de Catalogação na Publicação

N518 Neves, Romulo de Almeida

Automatização da geração de casos de testes : um estudo de caso na empresa Exactus Software / Romulo de Almeida Neves – 2019.
70 f. : il. color.; 31 cm.

Orientador: Willian Massami Watanabe.
Dissertação (Mestrado) – Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Informática. Cornélio Procópio, 2019.
Bibliografia: p. 67-70.

1. Interfaces de usuário (Sistemas de computação). 2. Interação homem-máquina. 3. Software - Testes. 4. Informática – Dissertações. I. Watanabe, Willian Massami, orient. II. Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Informática. III. Título.

CDD (22. ed.) 004

Biblioteca da UTFPR - Câmpus Cornélio Procópio
Bibliotecário/Documentalista responsável:
Romeu Righetti de Araujo – CRB-9/1676

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Cornélio Procópio
Programa de Pós-Graduação em Informática

Título da Dissertação Nº 58:

"AUTOMATIZAÇÃO DA GERAÇÃO DE CASOS DE TESTES: UM ESTUDO DE CASO NA EMPRESA EXACTUS SOFTWARE".

por

Romulo de Almeida Neves

Orientador: **Prof. Dr. Willian Massami Watanabe**

Esta dissertação foi apresentada como requisito parcial à obtenção do grau de MESTRE EM INFORMÁTICA – Área de Concentração: Computação Aplicada, pelo Programa de Pós-Graduação em Informática – PPGI – da Universidade Tecnológica Federal do Paraná – UTFPR – Câmpus Cornélio Procópio, às 14h30 do dia 05 de junho de 2019. O trabalho foi _____ pela Banca Examinadora, composta pelos professores:

Prof. Dr. Willian Massami Watanabe
(Presidente – UTFPR-CP)

Prof. Dr. Cleber Gimenez Correa
(UTFPR-CP)

Prof. Dr. Vinícius Humberto Serapilha Durelli
(UFES)
Participação à distância via _____

Visto da coordenação:

Danilo Sipoli Sanches
Coordenador do Programa de Pós-Graduação em Informática
UTFPR Câmpus Cornélio Procópio

A Folha de Aprovação assinada encontra-se na Coordenação do Programa.

Dedico este trabalho à Deus e à minha família.

AGRADECIMENTOS

Primeiramente, gostaria de agradecer a Deus, que nos momentos difíceis me confortou, me amparou e me deu forças para concluir mais uma etapa da minha vida, pois “Aquele que pede, recebe; e o que busca, encontra; e ao que bate, abrir-se.” (Mateus 7, 7-8).

Ao Prof. Dr. Willian M. Watanabe por todo o apoio durante todo o projeto.

Agradeço a minha esposa Andressa da Silva Neves e a minha filha Maria Clara da Silva Neves por toda a compreensão e apoio em todos os momentos.

Agradeço a minha mãe Maria de Almeida Neves, que me auxilou e sempre estando presente em todos os momentos da minha vida.

A todos os meus amigos e familiares que direta ou indiretamente participaram de todo o processo de mais uma etapa concluída na minha vida.

”No mundo haveis de ter aflições. Coragem! Eu venci o mundo.”

João 16,33

RESUMO

NEVES, Romulo de Almeida. AUTOMATIZAÇÃO DA GERAÇÃO DE CASOS DE TESTES: UM ESTUDO DE CASO NA EMPRESA EXACTUS SOFTWARE. 70 f. Dissertação – Programa de Pós-graduação em Informática, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2019.

Contexto: *Model Driven Development* (MDD) eleva a importância de modelos dentro do ciclo de vida do software, incorporando-os como parte do produto final por meio de técnicas de modelagem e geração de código. Com isso, parte da complexidade do software fica escondida dentro de geradores de código. Além disso, os softwares são compostos de interfaces gráficas denominadas *User Interfaces* (UIs), que são responsáveis por realizar a interação do software com o usuário, oferecendo, assim, uma maior flexibilidade na execução de tarefas. Esta interface é formada por *widgets* que permitem a utilização do software tais como: botões, itens de menus e caixas de texto. **Problema:** Os testes são realizados de forma manual, com isso os custos associados à elaboração dos casos de testes são altos. **Objetivo:** O principal objetivo deste trabalho é verificar a eficiência das abordagens da geração de casos de testes a partir de modelos no processo de desenvolvimento na indústria. **Justificativa:** Garantir a qualidade do software, reduzir o tempo geral do ciclo de vida do software e os custos associados aos testes. **Método:** Desenvolver e elaborar técnicas de geração de casos de teste dentro de diferentes contextos e avaliar sua eficiência para processos de desenvolvimento de software. Foram conduzidos dois estudos de caso: um para geração de casos de teste utilizando *Model-Driven Testing* dentro de um processo baseado em *Model-Driven Development*; e um para geração de casos de teste utilizando componentes de interface em aplicações Web. **Resultados:** Cada uma das abordagens foi validada separadamente e os resultados indicam evidências que: (i) a geração automática de casos de teste do processo MDD da empresa Exactus Software melhora a eficiência da abordagem de teste na empresa Exactus Software; e (ii) a abordagem *Morpheus Web Testing* consegue alcançar uma maior cobertura de código em comparação com uma técnica do estado da arte, pois para todos os cenários o *Morpheus Web Testing* conseguiu alcançar um desempenho melhor na média. **Conclusão:** Em ambos estudos de caso os valores de cobertura atingidos representam evidências que os resultados obtidos a partir das abordagens propostas contribuem para o processo da Engenharia de software de testes automatizados na indústria. Considerando o segundo estudo de caso, especificamente, a técnica proposta, inclusive apresentou resultados superiores a uma técnica do estado da arte (*Crawljax*). Além disso, as abordagens também apresentaram como contribuições: automatização do processo de testes na indústria; e com o processo de automatização dos testes é esperada uma melhoria na qualidade do software e uma redução dos custos de desenvolvimento.

Palavras-chave: *User Interfaces*, *widgets*, *Morpheus Web Testing*, *Code Coverage*, *Model Driven Development*

ABSTRACT

NEVES, Romulo de Almeida. AUTOMATION OF THE GENERATION OF TEST CASES: A CASE STUDY IN THE COMPANY EXACTUS SOFTWARE. 70 f. Dissertação – Programa de Pós-graduação em Informática, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2019.

Context: Model Driven Development (MDD) raises the importance of models within the of the software, incorporating them as an part of the final product by means of modeling techniques and code generation. With this, part of the software complexity is hidden inside the code generators. In addition the software is composed of graphical interfaces called User Interfaces (UIs), that are responsible for performing the interaction of the software with the user, thus offering greater flexibility in the execution of tasks. This interface is formed by widgets that allow the use of software such as buttons, menu items, and text boxes. **Problem:** The tests are performed manually, thereby the cost associated with the development of test cases is high. **Objective:** The main objective of this work is to check the efficiency of approaches to the generation of test cases from models in the development process in industry. **Justification:** Ensure software quality, reduce overall software lifecycle time, and costs associated with the tests. **Method:** Develop and elaborate test case generation techniques within different contexts and evaluate its effectiveness for software development processes. Two case studies were conducted: one for the generation of test cases using Model-Driven Testing within a process based on Model-Driven Development; and one for generating test cases using interface components in Web applications. **Results:** Each of the approaches was validated separately and the results indicate evidence that: (i) the generation of cases the Exactus Software MDD process automation improves the efficiency of test approach at Exactus Software; and (ii) the Morpheus Web Testing approach can achieve greater coverage of code compared to thecnical of the art state, because for all the scenarios Morpheus Web Testing was able to achieve a better performance on average. **Conclusions:** In both case studies the coverage values reached represent evidence that the results obtained from the proposed approaches contribute to the process of Engineering automated testing software in the industry. Considering the second case study, specifically, the proposed technique, including results superior to a state-of-the-art technique (Crawljax). Besides that, the approaches also presented the follows contributions: automation of the industry; and with the automation process of the tests an improvement in the quality is expected the software and a reduction in development costs.

Keywords: User Interfaces, widgets, Morpheus Web Testing, Code Coverage, Model Driven Development

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

MDD	Model Driven Development
UI	User Interface
GUI	Grafical User Interface
WUI	Web User Interface
HUI	Handheld User Interface
TS	Teste de Software
MDT	Model Driven Testing
JSF	JavaServer Faces
DSL	Domain-Specific Language
M2T	Model-to-Text
ATL	Atlas Transformation Language
M2M	Model-to-Model
QVT	Query View Transformation
OMG	Object Management Group
OCL	Object Constraint Language
SUT	System Under Test
TI	Tecnologia da Informação
HU	Hospital Universitário
UFS	Universidade Federal de Sergipe
DOM	Dynamic Document Object
SFG	State-Flow Graph
GUI	Graphical User interface
H	Hipótese
MVC	Model-View-Controller
CSS	Cascading Style Sheets
AJAX	Asynchronous JavaScript and XML
JPA	Java Persistence API
API	Aplication Program Interface

SUMÁRIO

1 INTRODUÇÃO

A abordagem de desenvolvimento de software orientado a modelos denominada *Model Driven Development* (MDD) eleva a importância dos modelos dentro do ciclo de vida do software incorporando-os como parte do processo de geração do produto final por meio de técnicas de modelagem e geração de código (LUCRÉDIO, 2009; TOLVANEN et al., 2007).

A principal proposta do MDD é fazer com que o engenheiro ou arquiteto de software não precise interagir com o código fonte diretamente, preocupando-se principalmente com modelos de alto nível (FRANCE; RUMPE, 2007), possibilitando o aumento de produtividade, confiabilidade, manutenibilidade, interoperabilidade das aplicações desenvolvidas, melhorias na manutenção, reutilização e documentação do software (TOLVANEN et al., 2007; KLEPPE et al., 2003; BITTAR et al., 2009).

No processo MDD, os diagramas conceituais não são apenas utilizados como referência para a codificação, mas também são utilizados como artefatos ativos no processo e servem como entrada para ferramentas de geração de código, reduzindo o esforço dos desenvolvedores (KLEPPE et al., 2003). As transformações entre modelos geram sequências de transformações que permitem a implementação completa de um sistema partindo dos seus requisitos (DISTANTE et al., 2007).

Vale destacar que os softwares produzidos pela abordagem do MDD muitas vezes contemplam a utilização de componentes de interfaces gráficas também conhecidas como *User Interface* (UI). As UIs são formadas por elementos (*widgets*) que possibilitam a utilização do software, tais como: *frames*, botões, itens de menu e caixa de texto (SAKAL, 2010; RAUF et al., 2010; MEMON et al., 2001; MARIANI et al., 2012), permitindo assim, uma interação com a aplicação. Nesse sentido, a interface gráfica é denominada de forma diferente dependendo do tipo de aplicação e da plataforma, como por exemplo : *Grafical User Interface* (GUI) para aplicações *desktop*, *Web User Interface* (WUI) para aplicações Web e *Handheld User Interface* (HUI) para dispositivos móveis (SAKAL, 2010).

No entanto, mesmo com a utilização do MDD no processo de desenvolvimento os

softwares podem apresentar defeitos durante a sua execução. Neste contexto, a realização de testes é fundamental, uma vez que, eles permitem garantir a qualidade do produto. O Teste de Software (TS) é uma das principais atividades realizadas para melhorar a qualidade de um software em desenvolvimento. Seu principal objetivo é revelar a presença de erros no software o mais cedo possível no ciclo de desenvolvimento de software, buscando minimizar o custo da correção dos produtos em desenvolvimento (MYERS et al., 2004; PRESSMAN, 1995; ROCHA et al., 2001). DELAMARO et al. (2017) afirmam que uma abordagem de teste bem-sucedida pode diminuir o esforço de teste, contribuir para a melhoria da qualidade do produto e reduzir os custos de manutenção. É importante enfatizar que, na maioria dos casos os testes são realizados de forma manual e o custo associado à elaboração dos casos de teste é alto, já que há uma grande quantidade de interações possíveis dos usuários com as aplicações (NGUYEN et al., 2014), podendo se tornar caros (AHO et al., 2011b). Outros agravantes para a realização dos testes são: a alta complexidade dos sistemas atualmente desenvolvidos e a constante necessidade de sua evolução. Logo é fundamental buscar estratégias para aumentar a qualidade do produto, reduzir o custo e/ou tempo de desenvolvimento e, por fim elevar o padrão de qualidade (MYERS et al., 2004; ROCHA et al., 2001).

Neste contexto a automação do teste de software tem sido vista como uma medida para se melhorar a eficiência da atividade de teste de software e tem se tornado uma das alternativas para obter um produto com um número reduzido de defeitos (BINDER, 2000; KANER, 1997; FEWSTER et al., 2001; HENDRICKSON, 1998; EDWARD, 1999) e, consequentemente uma alternativa para: reduzir os custos, aumentar a eficácia dos testes, garantir a qualidade do software e reduzir o tempo geral do ciclo de vida do software. Sendo assim, a geração de casos de testes a partir de modelos surge como uma estratégia de automatização de testes.

Com base no que foi exposto, o objetivo deste trabalho é verificar a eficácia das abordagens de geração de casos de testes a partir de modelos no processo de desenvolvimento na indústria. Para atingir o objetivo desta dissertação, foram conduzidos dois estudos de casos, e em virtude disso, foram realizadas duas abordagens para a geração de casos de testes a partir de modelos. Sendo assim, as abordagens consistiram em: (i) implantar o processo *Model Driven Testing* (MDT) em um processo MDD desenvolvido pela Exactus Software denominado Celera, utilizando a ferramenta Sikuli através de scripts gerados automaticamente; e (ii) automatizar o processo de geração dos casos de testes funcionais em uma plataforma de desenvolvimento Web denominada Exactus CRM. Para isso, foi elaborada uma estratégia para a geração de casos de teste funcionais automatizados para aplicações Web denominada *Morpheus Web Testing* que utilizam dois *frameworks* que auxiliam os desenvolvedores para o desenvolvimento de WUI, o *JavaServer Faces* (JSF) e o *Primefaces*.

BAKER et al. (2007) definem o MDT como um processo para a geração de artefatos de teste em diferentes níveis de abstração aplicando regras de transformação e contribuindo para aumentar a confiabilidade e produtividade em processos de teste. LIMA et al. (2007) demonstram dois benefícios ao adotar a abordagem MDT: a redução do custo (após implantada) e a automação no processo de geração de casos de teste. Além disso, os modelos de teste podem auxiliar na identificação de erros antes da transformação dos modelos em código (FRANCE et al., 2006). Para isso, a fim de obter benefícios na geração e execução de testes, é necessário que a abordagem MDT refine-se a três tarefas: (i) a geração de casos de teste obtidos através de modelos de acordo com o critério de cobertura; (ii) a geração de oráculos de testes que determinam os resultados esperados de um teste; e (iii) a execução de testes em ambientes de teste obtidos através de modelos. Pode-se observar que a primeira e segunda tarefa são independentes de plataforma, no entanto a terceira deve ser executada em uma plataforma (HILDENBRAND; KORTHAUS, 2004).

Por outro lado, mesmo com o apoio ferramental de automação de testes, os testes em GUI geralmente requerem um trabalho manual, especialmente quando a aplicação sob teste muda rapidamente. Sendo assim, o processo de geração de casos de testes funcionais surge como uma alternativa para automatizar os testes em GUI. Neste processo, os requisitos de teste são estabelecidos a partir da especificação do software, e sua estrutura não é necessariamente considerada. OSTRAND; BALCER (1988) destacam que, o objetivo do teste funcional é encontrar discrepâncias entre o comportamento atual do sistema e o descrito em sua especificação. Assim, consideram-se apenas as entradas e saídas e o testador não tem necessariamente acesso ao código fonte do software.

Neste contexto, é importante destacar as contribuições obtidas com o desenvolvimento do presente trabalho, com isso enfatiza-se: (i) a automatização do processo de testes; (ii) melhoria na qualidade do software; e (iii) redução dos custos de desenvolvimento. Além disso, os resultados indicam evidências que: (i) a geração de casos de teste automático do processo MDD da empresa Exactus Software melhora a eficiência da abordagem de teste na empresa Exactus Software; e (ii) a abordagem *Morpheus Web Testing* consegue alcançar uma maior cobertura de código em comparação as técnicas do estado da arte.

O restante desta dissertação está organizado na seguinte forma: nos Capítulos 2 e 3 são apresentados a fundamentação teórica e os trabalhos relacionados. No capítulo 4 é apresentada a abordagem de geração dos casos de testes no MDT. No capítulo 5 é apresentada a abordagem de geração de casos de testes funcionais automáticos para aplicações Web. Por fim, no capítulo 6, são apresentadas as considerações finais e trabalhos futuros.

2 MODEL DRIVEN DEVELOPMENT

O MDD em conjunto com um processo de automatização dos testes como o MDT tem se tornado uma das alternativas para o desenvolvimento do software e para se obter um produto com número reduzido de defeitos.

Este capítulo tem como objetivo descrever brevemente os conceitos e definições sobre o MDD e MDT, além de apresentar trabalhos relacionados sobre o tema abordado.

Sendo assim, este capítulo está estruturado da seguinte forma: na Seção 2.1 são apresentados os conceitos sobre o MDD; na Seção 2.2 são apresentados os conceitos sobre o MDT; e por fim, na Seção 2.3 são apresentados os trabalhos relacionados ao MDD e MDT.

2.1 CONSIDERAÇÕES INICIAIS

Na Engenharia de Software diferentes paradigmas de desenvolvimento vêm surgindo no intuito de oferecer uma maior produtividade sem perda de qualidade. Um desses paradigmas é o MDD cujo o objetivo é colocar os modelos como artefato central do processo de desenvolvimento, ao invés do código-fonte (ALMEIDA; OLIVEIRA, 2014).

DEURSEN et al. (2000) relatam que a principal diferença entre o processo tradicional de desenvolvimento de software e o MDD são os artefatos gerados ao longo das fases do MDD, os quais são modelos passíveis de transformação especificados através de linguagens específicas de domínio, ou seja, as *Domain-Specific Language* (DSL) o que sugere altos índices de automação entre as fases. Uma DSL é uma linguagem pequena, normalmente declarativa focada em um domínio de um problema particular (DEURSEN et al., 2000; CZARNECKI et al., 2005).

LUCRÉDIO (2009) relata que para possibilitar a criação de modelos, é necessária uma ferramenta de modelagem para a construção dos modelos de domínio na qual os modelos precisam ser semanticamente completos e corretos, pois necessitam também serem compreendidos por um computador. Neste contexto, os modelos servem como entrada para as

transformações que irão gerar outros modelos. Para definir as transformações é necessária uma ferramenta que permita que o desenvolvedor defina regras de mapeamento de modelos para modelos, ou de modelo para código. A Figura 1 ilustra os principais elementos necessários para o processo MDD.

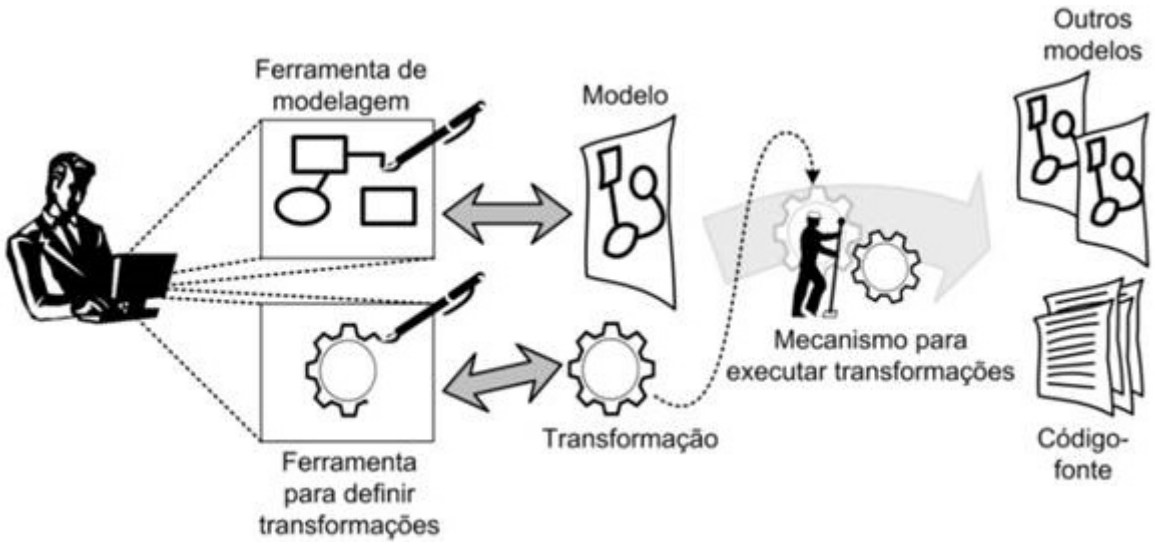


Figura 1: Principais elementos MDD
Fonte: LUCRÉDIO (2009)

No MDD, os modelos fornecem a base para a compreensão do comportamento do sistema e a geração das implementações. Com isso, o objetivo do MDD é reduzir a distância que existe entre o domínio do problema e o domínio tecnológico da solução, utilizando modelos abstratos que protegem os desenvolvedores das complexidades inerentes da plataforma de implementação (FRANCE; RUMPE, 2007). Para isso, o MDD assim como as abordagens orientadas a modelos, utiliza-se de modelos abstratos e transformações (semi-) automáticas de modelos ou de modelos para código (BEYDEDA et al., 2005), conforme ilustrado na Figura 1.

Além dos modelos no MDD, o processo de geração é composto por *templates* que segundo CZARNECKI et al. (2005), são arquivos textos quaisquer instrumentados com condições de seleção e expansão de código. Essas construções são responsáveis por realizar consultas de uma entrada que pode ser um programa, uma especificação textual ou diagramas (CLEAVELAND, 1988). O resultado desta consulta é utilizada como parâmetro para produzir código fonte. A Figura 2 ilustra esse processo, onde cada trecho do *template* Figura2 (B) é processado para consultar a entrada Figura 2 (A) e produzir código Figura 2 (C).

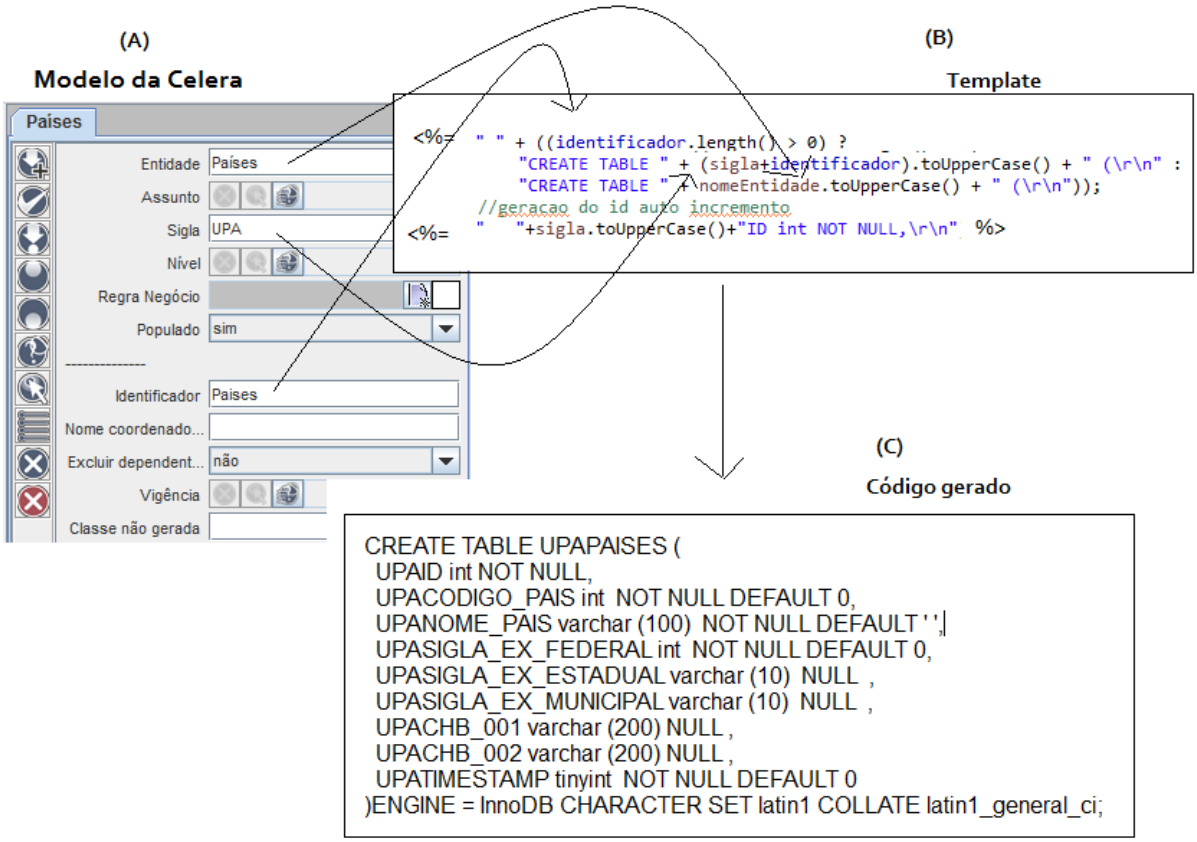


Figura 2: Geração de código baseada em templates

A Figura 3 ilustra o funciomento da abordagem MDD iniciando com uma entrada de especificações realizada pelos desenvolvedores que atuam sobre os modelos a fim de se transformar em código fonte. Neste contexto, alguns *templates* utilizados são:

- *Model-to-Text* (M2T): uma técnica de geração de artefatos textuais a partir de modelos a fim de realizar as transformações necessárias para a geração de casos de teste;
- *Atlas Transformation Language* (ATL): uma linguagem de transformação híbrida e bastante expressiva;
- *Model-to-Model* (M2M): uma técnica de transformação na qual o objetivo é transformar um modelo de entrada em outro modelo;
- *Query View Transformation* (QVT): uma linguagem para transformações entre modelos padronizada pela *Object Management Group* (OMG) com base na *Object Constraint Language* (OCL); e
- Próprias: desenvolvida na própria aplicação;

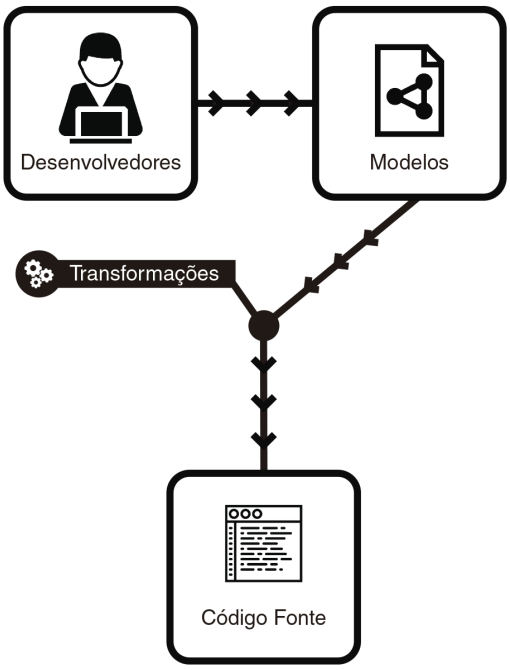


Figura 3: Processo de desenvolvimento MDD

TOLVANEN et al. (2007), KLEPPE et al. (2003), BITTAR et al. (2009), destacam os benefícios para o processo de desenvolvimento utilizando um processo de MDD como:

- **Produtividade:** o tempo do desenvolvimento será melhor aproveitado, pois será gasto na produção de modelos de mais alto nível;
- **Melhorias na manutenção**: no desenvolvimento convencional, a urgência inerente às atividades de manutenção faz com que os desenvolvedores insiram modificações diretamente no código, fazendo com que a documentação logo fique desatualizada;
- **Portabilidade**: um mesmo modelo pode ser transformado em código para diferentes plataformas;
- **Interoperabilidade**: cada parte do modelo pode ser transformada em código para uma plataforma diferente, resultando em um software que executa em um ambiente heterogêneo, porém mantendo a funcionalidade global; e
- **Reutilização:** a reutilização de artefatos de alto nível proporciona maiores benefícios do que a reutilização de código fonte;

SELIC (2003) afirma que o foco do MDD são os modelos, ao invés dos programas de computador. Com isso os modelos são criados utilizando conceitos menos ligados à tecnologia

de implementação subjacente e são mais próximos do domínio do problema em relação às linguagens de programação.

No MDD, a principal idéia é realizar a transformação de modelos de maiores níveis de abstração (domínio do problema) em modelos mais concretos (domínio solução) até se obter o sistema (o código gerado), fazendo com que as futuras modificações do sistema produzido sejam realizadas apenas no modelo mais abstrato (BUARQUE, 2009).

2.2 MODEL DRIVEN TESTING

Uma das características presentes nos testes tradicionais é que, os casos de testes geralmente são escritos manualmente por meio de análise de requisitos (FERNANDES, 2014). Uma das dificuldades de se elaborar os casos de testes de forma manual é a necessidade da utilização de mais recursos e tempo (FERNANDES, 2014). No entanto, quando se utiliza a abordagem MDT, essas necessidades (recurso e tempo) são reduzidas, visto que, o MDT automatiza a geração de casos de teste.

Para isso, inicialmente é desenvolvido um modelo de teste para descrever o comportamento esperado do *System Under Test* (SUT), isto é, o sistema que deverá ser testado. Uma vez finalizada a geração de modelo de teste, os casos de teste são projetados (automaticamente) a partir de modelos baseados em critérios de cobertura que foram selecionados (ABBORS et al., 2009). Sendo assim, BAKER et al. (2007) definem o MDT como uma abordagem para a geração de artefatos de teste, composto por casos de teste, dados de teste e oráculos, através de modelos de desenvolvimento obtidos pela regra de transformação entre modelos e a geração automática.

Neste contexto, JIAO et al. (2006) relatam que o MDT tornou-se um dos fatores para geração e *design* de testes, isso possibilita a geração da base de conhecimento para análise automática de defeitos e localização do problema. Para isso, JIAO et al. (2006) relatam que o MDT deve ser composto pelas seguintes etapas:

- Criação do modelo do sistema de teste SUT alinhado com o modelo de design;
- Geração de caso de teste e oráculo; e
- Execução de casos de teste em modelo de desing executável e/ou sistema em execução, comparando o resultado de teste com o oráculo.

Uma das características presentes na abordagem MDT é que essa abordagem é baseada

em modelos, ou seja um artefato utilizado para a construção do sistema e na comunicação das decisões de design (SELIC, 2003). (LIMA et al., 2007) demonstram dois benefícios ao adotar a abordagem MDT: a redução do custo (após implantada) e a automação no processo de geração de casos de teste. Além disso, os modelos de teste podem auxiliar na identificação de erros antes da transformação dos modelos em código (FRANCE et al., 2006).

BAKER et al. (2007) e KASHYAP; O'REILLY (2012) também relatam benefícios de se utilizar o MDT como: uma melhora nas especificações aliada a uma integração dos testadores nas fases iniciais do processo de desenvolvimento, o aumento dos testes contínuos, uma melhora na qualidade no ato da entrega do produto final, a previsibilidade e redução de riscos e a melhora na eficiência e eficácia dos produtos.

Em contrapartida, a utilização do MDT pode requerer um maior custo durante o período de implementação, além da dificuldade de se balancear entre especificidade do domínio e linguagens de programação genéricas (FRANCE et al., 2006; HILDENBRAND; KORTHAUS, 2004). Essas dificuldades podem influenciar na geração de estudo e pesquisa referente ao tema abordado.

2.3 TRABALHOS RELACIONADOS

A Tabela 1 apresenta um resumo comparativo e o foco da abordagem entre os trabalhos relacionados desta proposta.

Na linha de pesquisa relacionada ao uso do MDT, LI et al. (2006) retratam uma metodologia de MDT para uma aplicação Web, a fim de permitir que usuários definam meta-modelos oferencendo uma interface mais amigável. Este trabalho relata a realização de um experimento como um *plugin* para a IDE Eclipse denominado MDWATP, utilizando um motor de *template* e uma DSL própria. O MDWATP está em desenvolvimento e como trabalhos futuros é proposta a realização de testes do MDWATP em outras aplicações da Web.

ALVES et al. (2008) discutem objetivos e questões sobre como integrar os processos MDD e MDT, apresentando uma proposta de integração a partir de um estudo de caso para sistema de empréstimo de biblioteca, utilizando o motor de *template* ATL e diagramas da UML 2.0. Como trabalhos futuros, é proposto substituir as regras para o motor de *template* QVT.

LAMANCHA et al. (2009) apresentam um MDT para geração automática de casos de teste em um estudo de caso para desenvolvimento de jogos. O motor de *template* utilizado nesse trabalho é o QVT em conjunto com a UML 2.0 com diagramas de classe e sequência. Como trabalhos futuros é proposto incluir outros tipos de processos de geração e automatização.

Tabela 1: Resumo comparativo entre os trabalhos relacionados e a proposta

Autores	Domínio da aplicação	Tipo de modelo utilizado	*Template*
LI et al. (2006)	Aplicações Web. Desenvolvimento de um plugin para a IDE Eclipse	Utilização diagramas UML2 com diagramas de classe e sequência	Próprio
ALVES et al. (2008)	Estudo de caso para controle de empréstimos em uma biblioteca	Utilização diagramas UML 2.0 com diagramas de classe	ATL
LAMANCHA et al. (2009)	Aplicação desktop para desenvolvimento de jogos	Utilização diagramas UML 2.0 com diagramas de classe e sequência	QVT
ALMEIDA; OLIVEIRA (2014)	Aplicação desktop para Hospitais (àrea da saúde)	Utilização de uma DSL própria	Próprio
OLAJUBU et al. (2015)	Estudo de caso industrial para empresas de aviação	Utilização de uma DSL própria e uma modelagem textual semelhante a UML, porém não é descrita como é realizada.	M2T
Trabalho atual	Aplicação desktop para escritórios de contabilidade	Utilização de um template próprio	Próprio

ALMEIDA; OLIVEIRA (2014) assim como este trabalho, promovem uma integração prática do MDD com o MDT denominado Qualitas, como um modelo de processo de desenvolvimento de software orientado a modelos. O processo Qualitas foi desenvolvido no departamento de Tecnologia da Informação (TI) do Hospital Universitário (HU) da Universidade Federal de Sergipe (UFS), utilizando um motor de *template* e uma DSL própria. Como trabalhos futuros, deseja-se validar o modelo proposto em estudos e contextos reais.

OLAJUBU et al. (2015) relatam tentativas preliminares para automatizar a geração de casos de teste a partir de requisitos de modelos de softwares e utilizaram um caso de uso industrial e o motor de *template* M2T, e uma DSL própria. Esse trabalho relata um estudo de caso industrial em sistemas de aviação na empresa GE *Aviation Systems* como uma aplicação desktop utilizando uma técnica de modelagem textual semelhante a UML, no entanto não descreve maiores detalhes sobre a técnica. Como trabalhos futuros é proposto ampliar a geração de casos de teste a partir de novos tipos de requisitos.

Com base no cenário descrito anteriormente é possível afirmar que este trabalho difere de (LI et al., 2006; ALVES et al., 2008; LAMANCHA et al., 2009; ALMEIDA; OLIVEIRA, 2014; OLAJUBU et al., 2015) pois é realizado um processo de geração de *scripts* a serem utilizados pela ferramenta Sikuli como forma de testar automaticamente as GUIs, geradas pelo

processo MDD em sistemas legado. Com isso o MDD está sendo utilizado em conjunto com o Sikuli para automatizar e viabilizar a geração de casos de teste, pois sem esse processo haveria a necessidade de um investimento na contratação de testadores para realizar o processo de teste de todas as telas do sistema. Por fim, é importante destacar que estes trabalhos são referentes ao estudo de caso que aborda a geração de casos de testes no MDT.

3 UI TESTING

Uma das atividades mais comuns para aumentar a confiabilidade dos sistemas é a de teste. Essa atividade envolve executar um sistema com um conjunto de entradas e avaliar se o conjunto de saídas é válido, ou seja, se está de acordo com a especificação. Dentro do contexto de teste de sistema, surge o teste de UI, que é responsável por realizar os testes nas interfaces gráficas.

Sendo assim, este capítulo tem como objetivo descrever brevemente os conceitos e definições referentes as GUIs e WUIs, além de apresentar trabalhos relacionados sobre o tema abordado.

A próxima seção (Seção 3.1) introduz os conceitos sobre o GUI Testing. Na Seção 3.2 são apresentados os conceitos sobre o WUI Testing; a Seção 3.3 apresenta brevemente os conceitos e definições sobre testes baseados em modelos; por fim, a Seção 3.4 são apresentados os trabalhos relacionados ao UI Testing.

3.1 GUI TESTING

As interfaces gráficas denominadas GUIs, são responsáveis por realizar a interação do software com o usuário, por meio de elementos gráficos como: botões, links, caixas de texto e itens de menu, geralmente por meio de mouse ou teclado. Esses elementos gráficos são denominados *widgets*.

MEMON et al. (2001) relatam que cerca de 60% do código fonte das aplicações são destinados a construção das GUIs. Por esse motivo observa-se a necessidade de se realizar os testes nas GUIs e estudar as relações desses testes. No entanto, MEMON et al. (2001), MOREIRA et al. (2013) e AHO et al. (2014) relatam dificuldades de se realizar os testes sobre as GUIs, dentre elas destacam-se:

- A comunicação entre as GUIs e os componentes de *hardware* como mouse e teclado são difíceis de simular;

- Os casos de teste devem explorar muitas combinações dos elementos da interface, o que torna o processo complexo e custoso;
- As características visuais como *layout* de interface não devem afetar o teste; e
- A cobertura de testes deve se basear no número de diferentes combinações de utilização dos componentes de interface gráfica.

Segundo MEMON et al. (2001), as GUIs são compostas por:

- Objetos, que são representados por: *labes*, *tables*, botões, links, text, menus, *combobox*, *checkbox*, *radio buttons*, *calendar*; e
- Propriedades dos objetos que são representadas por: *color*, *font*, *size*, *required*, id, *maxlength*, *mask*.

A Figura 4 (a) ilustra a estrutura de propriedades. MEMON et al. (2001) definem que o valor da propriedade (opcional) é uma *constant draw*, ou seja uma constante escolhida de um conjunto associado com a propriedade em questão: por exemplo a propriedade *background-color* tem um conjunto de valores associado, {*red, blue, black, white*..}. Por fim a Figura 4 (b) ilustra um objeto de botão denominado *Caption* e seu valor atual é *Cancel*.

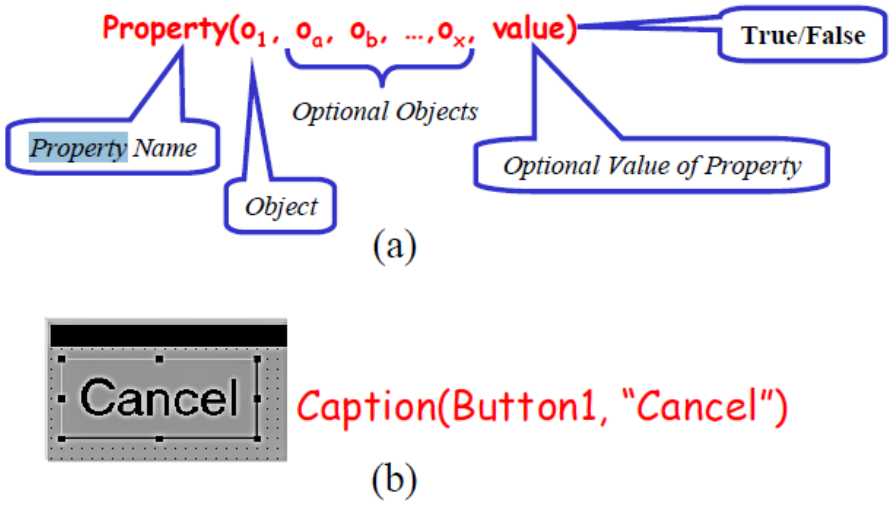


Figura 4: Propriedades de um objeto (Fonte: MEMON et al. 2001)

Neste contexto, MEMON et al. (2001) apresentam os modelos de uso, que também são conhecidos como modelos de interfaces gráficas. Segundo MEMON et al. (2001), para definir os modelos de uso, é necessário identificar os objetos e as propriedades. RAUF et al. (2010) e

MEMON et al. (2001) relatam que existem algumas maneiras de se identificar os objetos e as propriedades.

- O testador examina manualmente as GUIs, identificando todos os objetos e suas propriedades. No entanto essa abordagem pode conter falhas, porque nas GUIs podem existir propriedades ocultas que devem ser identificadas;
- As propriedades e tipos de objetos são extraídos das especificações das GUIs que as descrevem. No entanto essa abordagem pode conter falhas, pois propriedades adicionais podem ter sido introduzidas pela plataforma de implementação da GUI; e
- Examinar as ferramentas e as linguagens de programação e todos os seus tipos de objetos e propriedades identificadas.

MEMON et al. (2001) e AHO et al. (2014) afirmam que, os modelos de uso mais utilizados para a realização dos testes em UI são:

- Modelo baseado em estados; e
- Modelo de uso baseado em engenharia reversa.

No modelo baseado em estados, o comportamento de uma aplicação GUI é representado como uma máquina de estados, nos quais os nós do modelo são chamados de estados GUI, arestas são eventos e interações. Cada evento de entrada pode desencadear uma transição para um estado de máquina. Um caminho é a sequência de estados e eventos da GUI, representando um dado de teste (MEMON et al., 2001; AHO et al., 2014).

No modelo de uso baseado em engenharia reversa, é feita a execução do aplicativo e se observa o comportamento em tempo de execução da UI. O desafio é passar automaticamente pela UI fornecendo dados significativos para os campos de entrada requisitados como usuário e senhas válidos para a tela de login, sem instruções predefinidas do usuário. Geralmente uma intervenção humana é necessária durante o processo de modelagem para alcançar uma boa cobertura com modelos de engenharia dinâmica, o que significa que a modelagem é assistida manualmente por uma pessoa durante o processo de engenharia reversa, ou o modelo inicial gerado é revisado, corrigido e estendido manualmente por uma pessoa após a extração do modelo (AHO et al., 2011a; KULL, 2012).

Por fim, MEMON et al. (2013) publicaram uma pesquisa relacionada aos modelos de uso, especificamente modelos baseado em eventos, sobre GUI *Ripping*. Essa técnina extrai

dinamicamente modelos baseados em eventos de aplicações GUI para fins de automação de teste. Seu objetivo é fornecer ferramentas para o processo de extração de modelos e geração de teste automatizado, no entanto, não relatam o desafio de fornecer entradas específicas.

3.2 WUI TESTING

As *Web User Interfaces* (WUIs) são definidas como interfaces para aplicações Web em que há uma interação do usuário por meio de um navegador Web, sem o conhecimento do software subjacente (SAKAL, 2010; MEMON et al., 2001). MEMON et al. (2001) definem uma WUI como uma GUI na forma de uma estrutura hierárquica (conforme ilustrado na Figura 5) que consiste em *frames* (que são agrupadores de páginas Web) e com *constraints* entre os objetos de cada página Web. Uma *constraint* pode ser definida como temporal (um evento UI que tem um tempo máximo para ser executado) ou de sincronização (quando se exige que a execução de um evento deve ser concluída antes de executar o próximo evento).

MEMON et al. (2001) afirmam que as WUIs tem todas as características das GUIs, porém as WUIs tem algumas peculiaridades em específico como: *constraint* e portabilidade entre os navegadores Web, o que torna o ambiente mais complexo e desafiador do que o teste de GUI.

Segundo MEMON et al. (2001) , as principais características das WUIs são:

- Conectividade com a internet;
- Orientada a eventos de entrada;
- Contém *frames*;
- Contém páginas e *constraints* entre as páginas;
- Objetos e suas propriedades;
- Possui *constraints* entre os objetos; e
- Os atributos dos objetos.

Uma WUI contém objetos que aceitam a entrada de usuários e uma perspectiva de saída a ser exibida no navegador Web. Exemplos de objetos incluem: caixas de textos, imagens, botões, links, *combobox*, *checkbox*. Os objetos WUI são agrupados em páginas Web que podem ser estáticas ou dinâmicas. As páginas são agrupadas em *frames*, com isso, os agrupamentos

permitem aumentar a usabilidade da WUI, exibindo os objetos relacionados juntos (MEMON et al., 2001).

A figura 5 demonstra uma WUI decomposta em *frames* (f), páginas (p) e objetos (o). Cada *frame* (f1 e f2) contém páginas (p2, p2 e p5), com vários objetos (o1, o2,..o6). Eventos sobre o1, o2 e o3, não podem ser intercalados com eventos sobre os objetos o4, o5 e o6. Percebe-se que os eventos executados em o4, o5 e o6 podem ser intercalados desde que as páginas p4 e p5 sejam exibidas no mesmo *frame*. Essas características de páginas e *frames* podem ser utilizadas para identificar os componentes das WUIs.

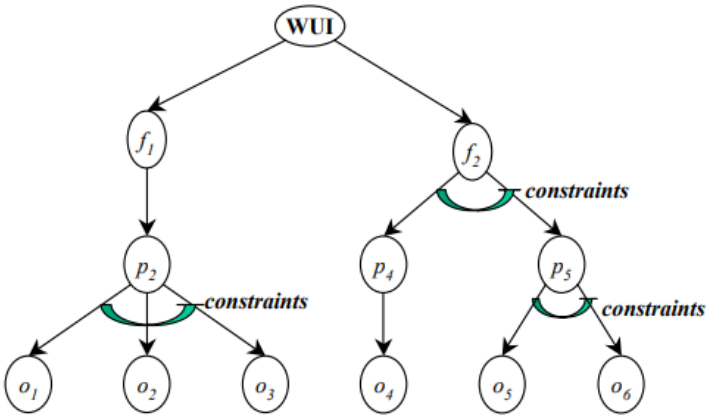


Figura 5: Uma WUI com uma hierarquia de páginas, *frames* e objetos com *constrains* (Fonte: MEMON et al. (2001))

A figura 6 demonstra uma simples WUI, sendo modelado seus objetos (com suas propriedades) e suas *constraints*. A WUI ilustrada na figura 6 contém quatro objetos: *name-label*, *name-field*, *submit-button* e *rest-button*.

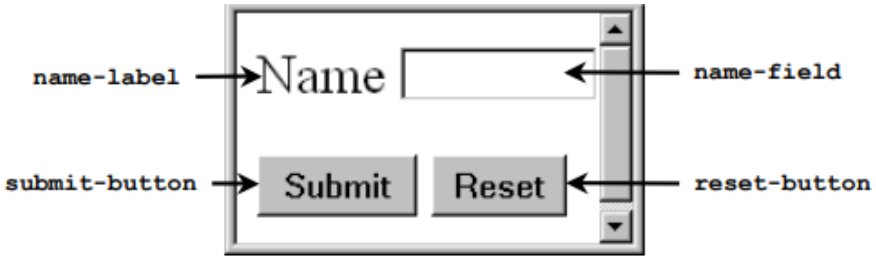


Figura 6: Exemplo de WUI (Fonte: MEMON et al. 2001)

A figura 7 ilustra as propriedades e as características dos objetos. A propriedade *Type* ilustrada nas linhas (4, 6, 8 e 10) descreve o tipo do objeto e seu comportamento. A propriedade *action* demonstrada nas linhas (9 e 11) associam um programa com o objeto em questão, são os

casos dos botões *submit-button* e *reset-button* que tem as ações *POST* e *REST*.

```
1  Frames:  f1 /* A single frame*/
2  Pages:    p1 /*A sigle page*/
3  Objects of p1:
4    name-label: set of properties={type("label"),
5          value("Name"),color("Black"),font("Type Roma") }.
6    name-field: set of properties={type("text-field"),value(""),
7          editable("True")}.
8    submit-button: set of properties={type("button"),
9          caption("Submit"),action("POST")}.
10   rest-button: set of properties={type("button"),
11         caption("Reset"),action("REST")}.
12 Constraints: /**geometric constraints imposed by the
13              HTML code*/
14   {first-object(name-label), after(name-label, name-field),
15      new-line(submit-button),after(submit-button,
16        reset-button)}
```

Figura 7: Propriedades da WUI (Fonte: MEMON et al. 2001)

3.3 TESTE WUI BASEADOS EM MODELOS

MEMON et al. (2001) definem um modelo WUI como uma WUI modelada com um conjunto de objetos *widgets* $O=\{O_1,O_2,...O_n\}$ e um conjunto de propriedades desses objetos $P=\{p_1,p_2,...p_n\}$. O Estado de uma WUI é um conjunto de todas as propriedades de todos os objetos que a WUI contém. Um conjunto válido de estados iniciais é associado a cada WUI.

Na WUI, seu estado não é estático, pois os eventos executados sobre ela podem alterar seus estados com isso, a execução de um evento sobre esse modelo leva a outro estado.

MEMON et al. (2001) definem eventos $E = \{e_1, e_2, e_3....,e_n\}$ de uma WUI como funções de um estado para outro estado. Eventos ocorrem com parte de uma sequência de eventos, $e_1,e_2,e_3....e_n$ onde e_i+1 pode ser executado imediatamente após e_i.

Os eventos podem ser combinados em sequência como $e_1; e_2; e_3; e_4....e_n$ é uma sequência de eventos executáveis para um estado S_0, se existir uma sequência de estados $S_0;S_1;S_3....S_n$ tal que $S_i=e_i(S_1 -1)$ para i=1,....n (MEMON et al., 2001).

Com isso, um conjunto de testes determinam os dados de teste. MEMON et al. (2001) definem dados de teste como: um dado de teste *t* é um par $(S_0, e_1; e_2;....e_n)$ do estado $S_0 \in S_1$, chamado de estado inicial. Um dado de teste é uma sequência de eventos $e_1,e_2,e_3....e_n$.

Se o estado inicial de um dado de teste não é alcançado e/ou a sequência de eventos é ilegal, então o dado de teste não é executado (MEMON et al., 2001).

Neste contexto surge o conceito de controlabilidade que requer que a WUI seja levada

a um estado válido antes de inciar os eventos. Cada WUI está associada a um conjunto distinto de estados, chamados de estados inciais válidos.

Um conjunto de estados S_1 é chamado de conjunto de estados inciais válidos para uma WUI particular se a WUI pode estar em qualquer $S_i \in S_l$ quando é invocada primeiramente (MEMON et al., 2001).

Com base em uma WUI, $S_i \in S_l$, em um estado inicial válido, novos estados podem ser obtidos por meio da ocorrência de eventos S_i. Esses estados são chamados de estados alcançáveis, a partir de S_i.

Um estado S_j é um estado alcançável, se qualquer estado $S_j \in S_l$ e existe uma sequência de eventos executáveis $e_x,e_y,....,e_z$ tal que $S_j = (e_1,e_2,....,e_n)$ (S_i) para qualquer $S_i \in S_l$ (MEMON et al., 2001).

Por fim, uma *constraint* de sincronização temporal $T_1;T_2;T_3;...T_n$ está associada a cada dado de teste WUI, onde T_i é um conjunto de *constraints* de sincronização/temporal no evento e_i (MEMON et al., 2001).

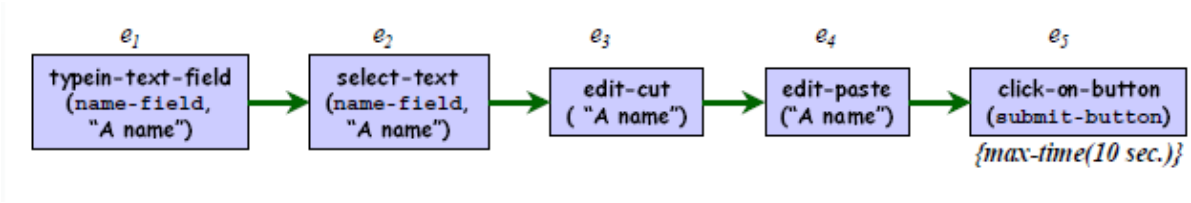


Figura 8: Sequência de eventos WUI (Fonte: MEMON et al. 2001)

Com isso, a sequência de eventos ilustrada na Figura 8 para a WUI ilustrada na Figura 6, em que um dado de teste consiste em 5 eventos, sendo que os eventos e_1 e e_5 disponíveis no navegador Web, enquanto os eventos e_2, e_3 e e_4 são eventos disponibilizados pelo navegador Web. O evento e_5 tem uma *constraint* temporal que impõe um limite sobre o tempo decorrido entre a sua execução e a apresentação dos resultados. Se esse tempo for maior do que 10 segundos, em seguida, deve ser comunicado um erro (MEMON et al., 2001).

3.4 TRABALHOS RELACIONADOS

Existem trabalhos de pesquisa na área de geração de casos de teste para interfaces gráficas: MEMON et al. (2001) foram os primeiros a apresentar um trabalho nesta área. Outras abordagens foram implementadas nesta área, como é o caso dos trabalhos desenvolvidos por Mariani (MARIANI et al., 2012; MESBAH et al., 2012; DALLMEIER et al., 2012).

MEMON et al. (2001) apresentam um *framework* de testes sobre UI's denominado GUITAR, que inclui um método de modelagem baseado em eventos. A ferramenta é usada para gerar modelos de aplicações Java UI para fins de automação dos testes. No processo de geração dos casos de teste, o GUITAR extrai da aplicação informações sobre a estrutura de todas as janelas, *widgets*, assim como os seus atributos e eventos da interface gráfica. A idéia é criar um fluxo de eventos com todas as possíveis interações de eventos na UI. Estes eventos são utilizados para gerar casos de teste de UI que são sequências de eventos de UI. A ferramenta GUITAR também suporta a execução dos casos de teste gerados sobre a aplicação Java UI.

MARIANI et al. (2012) desenvolveram uma ferramenta denominada *AutoBlackTest*. Esta ferramenta realiza o processo de geração de casos de teste a fim de produzir dados de teste de forma incremental a cada interação do usuário de teste. Este processo de geração é dividido em duas partes, na primeira parte a ferramenta irá gerar um modelo de sequência de eventos que podem ser produzidos por meio de uma interação com um UI do aplicativo em teste. Vale ressaltar que a geração do modelo ocorre através da utilização de uma técnica de aprendizado de reforço denominada *Q-learning*. Após o término da primeira parte, inicia-se a geração de um conjunto de dados de teste que cobre as sequências no modelo.

MESBAH et al. (2012) desenvolveram uma ferramenta denominda CrawlJax. Esta ferramenta realiza o processo de geração de casos de teste, analisando automaticamente as mudanças de estado na interface de aplicações Web com a tecnologia Ajax (*Asynchronous JavaScript and XML*). Este processo é dividido em duas etapas: (i) um crawler (controlador) que exercita o código do lado do cliente e identifica os elementos clicáveis que alteram o estado dentro do *Dynamic Document Object* (DOM) construído dinamicamente no navegador. Por fim na segunda etapa (ii) a criação de um grafo de fluxo de estado, denominado *State-Flow Graph* (SFG) que captura os estados DOM dinamicamente, os estados da UI e as possíveis transições realizadas entre eles.

DALLMEIER et al. (2012) apresentam o *WebMate*, uma ferramenta que realiza o processo de geração de casos de teste para aplicações Web. *WebMate* explora a funcionalidade de uma aplicação Web detectando as diferenças entre os navegadores Web e sistemas operacionais. O processo de geração de casos de teste do *WebMate* é composto por quatro etapas: (i) é informada uma URL em que o usuário interage com a aplicação, examinando todos os botões, *links*, formulários ou qualquer elemento manipulado por eventos que pode desencadear uma interação com o usuário. Em (ii), são extraídos os modelos de uso da aplicação em forma de grafo, onde os nós correspondem aos diferentes estados das aplicações e as arestas representam as interações do usuário. Em (iii), são realizados os testes na aplicação Web, a

fim de executar análises, verificar a compatibilidade entre os navegadores e realizar a análise do código e testes de regressão. Por fim em (iv) são relatados os resultados das análises para o usuário.

Diferentemente de MEMON et al. (2001), Mariani *et al.* MARIANI et al. (2012), MESBAH et al. (2012) e DALLMEIER et al. (2012) que utilizam componentes de interface em nível de marcação genérica (*HyperText Markup Language* HTML), este trabalho tem como característica a geração de casos de testes em projetos que utilizam componentes de interfaces complexos, como o *EXtensivle HyperText Markup Language* (XHTML) definidos por *frameworks* de desenvolvimento Web JSF juntamente com componentes de interface (*Primefaces*). Neste contexto, com a utilização do arquivo XHTML em conjunto com o *framework* JSF e os componentes do *Primefaces* destacam as seguintes vantagens: (i) a possibilidade de se obter mais informações do que um arquivo HTML e, com isso, prever e melhorar os níveis de interação dos componentes, e (ii) a possibilidade de se gerar mais casos de testes funcionais e, consequentemente, atingir uma maior cobertura de código. Por fim, é importante destacar que estes trabalhos são referentes ao estudo de caso que aborda a geração de casos de testes funcionais automáticos para aplicações Web.

4 GERAÇÃO DE CASOS DE TESTE NO MDT

Este capítulo tem como objetivo apresentar uma proposta de avaliação de automatização da geração dos casos de testes dentro da empresa Exactus Software. Esta proposta consiste em incorporar o MDT em um processo MDD já existente, denominado Celera.

A próxima Seção 4.1 é apresentado um processo MDD denominado Celera; na (Seção 4.2) é apresentado o processo de geração de casos de teste na Celera.

4.1 CELERA

A fim de promover a utilização do MDD através de especificações, a empresa Exactus Software desenvolveu um processo MDD denominado Celera. Esse processo tem como característica permitir que através de uma entrada de especificações, sejam gerados códigos nas linguagens *Visual Basic* (V.B.) e Cobol, conforme ilustrado na Figura 9.

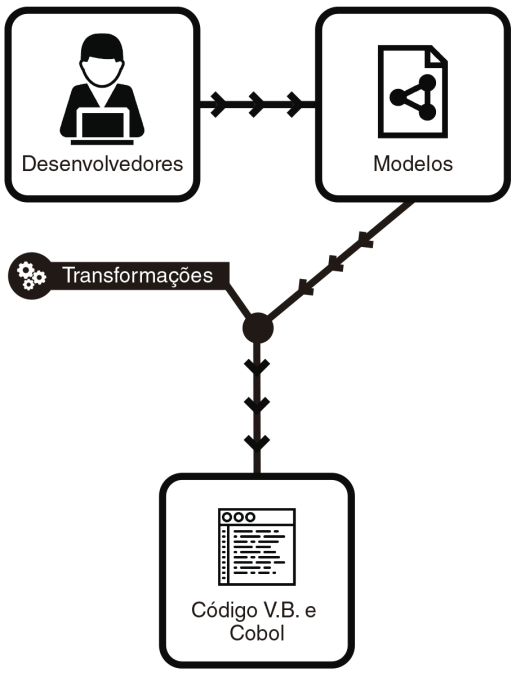


Figura 9: Geração de código V.B. e Cobol pela Celera

A geração de código realizada pela Celera é baseada na especificação dos modelos inseridos no processo MDD em uma estrutura *tree* (árvore), conforme demonstrado na Figura 10, e de acordo com os *templates* de geração desenvolvidos de acordo com as necessidades de implantações de novos casos forem surgindo.

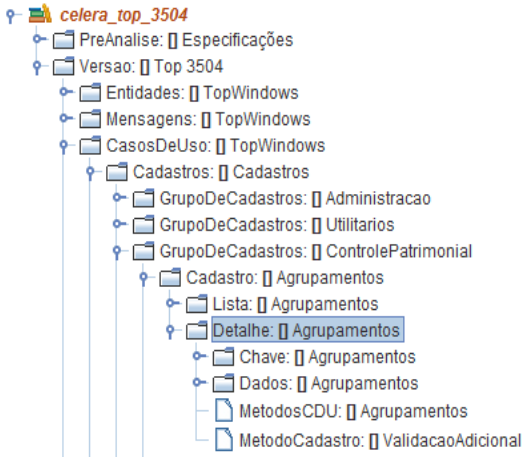


Figura 10: *Tree* da Celera, onde os artefatos são apresentados

Cada modelo possui suas propriedades conforme ilustrado na Figura 11, nelas são especificadas o que serão utilizadas na geração de código, como por exemplo:

- A visibilidade dos componentes;
- Habilitação / Desabilitação do componente;
- Posição na qual o componente será inserido na *Graphical User interface* (GUI);
- Obrigatoriedade do componente;
- Regra de validação do componente a ser gerado;
- Exibição de rótulo (*label*);
- Molduras (posição, largura, comprimemento);
- Definição de filtros (componentes, *labels*, botões); e
- Tipo de componente (botão, caixa de texto, *check*);

Toda a geração de código é realizada, de acordo com as propriedades especificadas na ferramenta Celera, por exemplo: tamanho do campo, tipo do componente (botão, texto, *check*), obrigatoriedade, visibilidade, posição a ser alocada na tela.

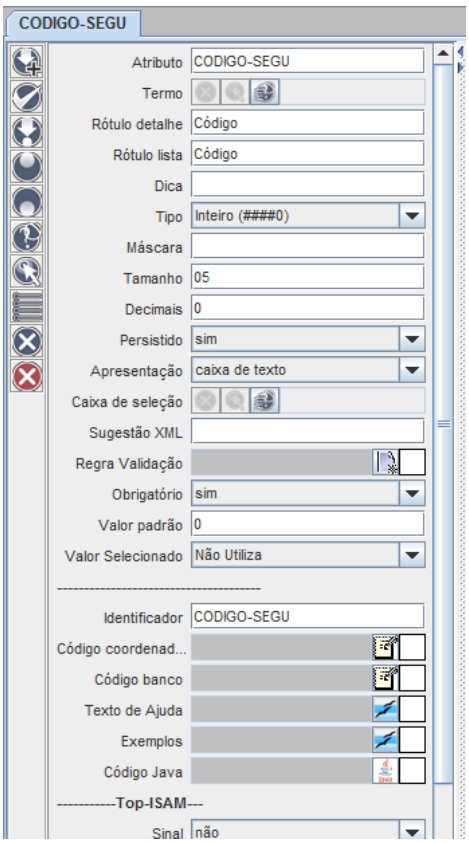


Figura 11: Especificação dos modelos pela Celera

Com a utilização da Celera pela Exactus Software, destacam-se as seguintes vantagens:

- **Produtividade**: o desenvolvimento do software é constituído de projeto de alto nível e codificação. Os modelos são produzidos antes da codificação, servindo de auxílio às tarefas de desenvolvimento e manutenção;
- **Manutenibilidade**: a manutenção restringe-se principalmente aos *templates*, e não no código gerado;
- **Corretude**: além do gerador não introduzir erros acidentais, como erros de digitação, o gerador permite que a identificação de erros conceituais, sejam verificados em um nível mais alto de abstração; e
- **Comunicação**: com a utilização da Celera, diversos profissionais possuem meios mais efetivos para a comunicação, uma vez que os modelos são mais abstratos do que os códigos. Neste contexto, especialistas de domínio têm um papel mais ativo no processo, podendo utilizar diretamente os modelos para identificar os conceitos do negócio.

No entanto destacam-se desvantagens como:

- O custo para se projetar, implementar e manter um processo MDD.
- A dificuldade de se balancear entre especificidade do domínio e linguagens de programação genéricas;
- A complexidade do processo MDD com as transformações e geradores de código, pois tratam-se de artefatos inerentemente mais difíceis de construir e manter;
- O processo MDD exige profissionais com habilidades na construção de linguagens, na Celera, nas transformações e nos geradores de código; e
- O desenvolvimento dos artefatos específicos do processo MDD exige profissionais com habilidades nas transformações e nos geradores de código.

4.2 GERAÇÃO DOS CASOS DE TESTE NA CELERA

Foi conduzido um estudo de caso com a Celera em que foi incluído o MDT com o objetivo de gerar *scripts* a serem utilizados pela ferramenta Sikuli, de acordo com os modelos especificados e com os *templates* desenvolvidos, conforme ilustrado na Figura 12, em que os desenvolvedores atuam com uma entrada de especificações sobre os modelos a fim de se transformar em códigos V.B., Cobol e *scripts* Sikuli.

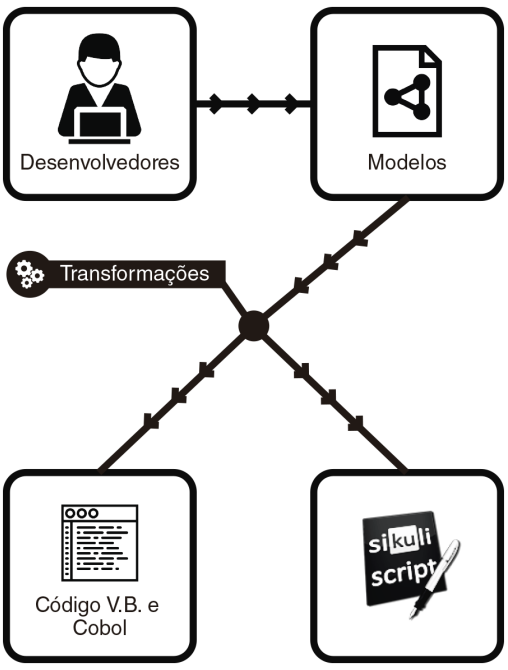


Figura 12: Modelos e transformações atuam no processo Celera

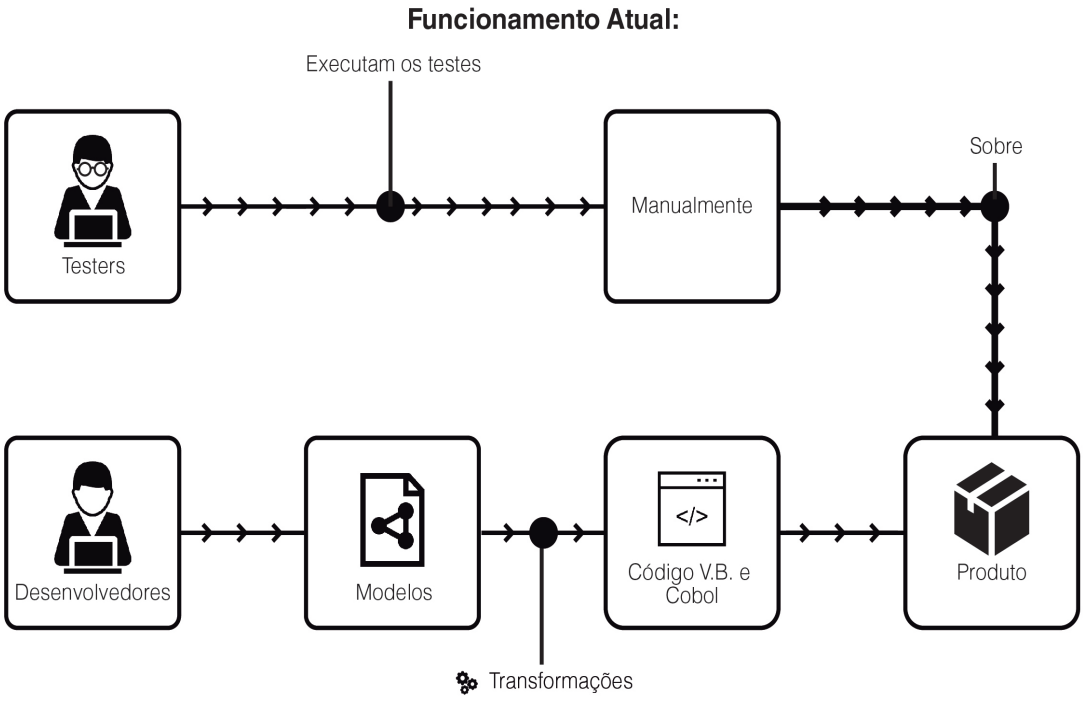


Figura 13: Modelo de teste de desenvolvimento atual

A Figura 13 ilustra o processo atual de desenvolvimento e teste realizado na Exactus Software, e a Figura 14 ilustra a proposta deste trabalho. No processo atual, conforme ilustrado na Figura 13 os desenvolvedores especificam os modelos, que são transformados em código V.B. e Cobol, e ao final os testadores executam os testes manualmente sobre o produto gerado. No entanto, conforme ilustrado na Figura 14 a proposta deste trabalho é automatizar o processo de testes, incluindo no processo MDD o MDT. Uma das características deste processo é que os desenvolvedores especificam os modelos que são transformados em código V.B., Cobol e *script* Sikuli. Ao final os testadores especificam os modelos de casos de teste que serão utilizados como entrada, na qual serão transformados em *scripts* Sikuli pela Celera a fim de realizarem os testes automaticamente sobre os produtos produzidos pela Celera.

Para realizar os testes sobre o produto gerado pela Celera será utilizada a ferramenta Sikuli. O Sikuli tem como característica a utilização de *screenshots* (imagens) para gerar testes utilizando padrões de imagens. Esses testes são direcionados a eventos que podem ocorrer através de dispositivos de entrada (mouse e teclado) em uma GUI. Nesse sentido o uso do Sikuli deve contribuir com essa limitação e auxiliar atividades de manutenção desse sistema.

Com isso, a proposta de gerar os testes automaticamente no processo MDD deve-se pela necessidade de diminuir o tempo e os custos na elaboração dos testes feitos pelos desenvolvedores e *testers*. Sendo assim, a fim de minimizar o tempo com a execução dos

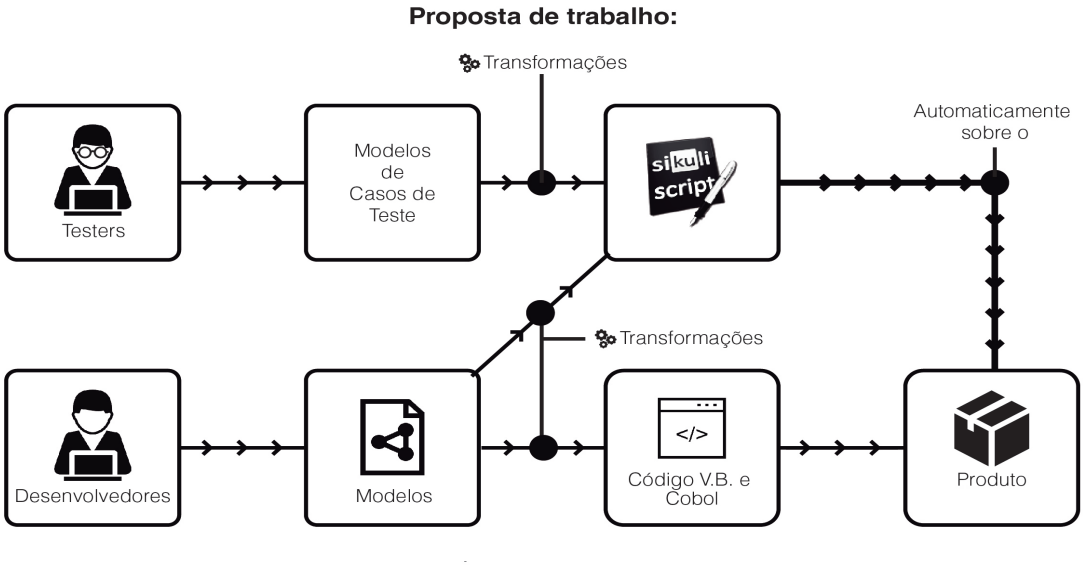


Figura 14: Proposta de modelo de teste e desenvolvimento deste trabalho

testes, este trabalho tem como objetivo automatizar o processo dos testes realizados, visto que o processo de confecção dos *scripts* Sikuli se tornem automáticos através de um processo MDD. Esses *scripts* serão gerados pela Celera através das especificações inseridas nos modelos, transformações e executadas na ferramenta Sikuli.

4.2.1 PROCESSO DE GERAÇÃO DOS CASOS DE TESTE

SOMMERVILLE; SAWYER (1997) e DELAMARO et al. (2017) definem casos de teste como uma forma de estabelecer as entradas a serem informadas pelo testador (manualmente ou com apoio ferramental) e os resultados esperados a partir dessa ação.

Os casos de teste são compostos por entradas, os passos e por fim os oráculos, que segundo (SOMMERVILLE; SAWYER, 1997; DELAMARO et al., 2017) são mecanismos que possibilitam definir a saída ou comportamento esperado de uma execução.

Considerando o contexto da implementação do processo MDT na empresa Exatus Software, neste trabalho os casos de teste são obtidos através de transformações de dois tipos de modelos, os modelos de teste gerados pelos testadores em uma *Fit Table* e os modelos especificados na Celera.

A Figura 15 ilustra os modelos definidos pela Celera, conforme demonstrado na Figura 15 (A) e as *Fit Tables* representado na Figura 15 (B) que são responsáveis por incluir os campos

de interação e os valores de dados específicos inseridos durante a execução de cada caso de teste.

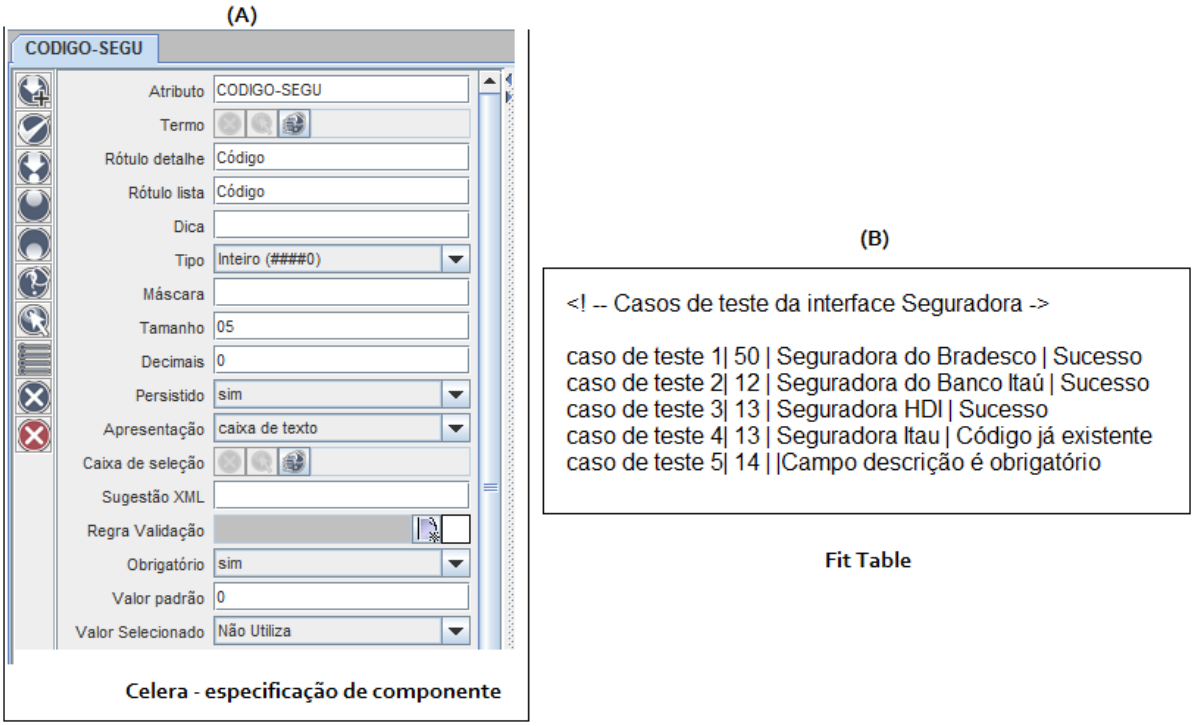


Figura 15: Modelos da Celera e as Fit Tables

As entradas são definidas pelas *Fit Tables* (arquivos texto) nos casos de teste. Devem conter as informações de valores (válidos e inválidos) para a realização dos casos de teste. As *Fit Tables* devem definir todos os casos de teste a serem executados de acordo com a GUI na qual será realizado os testes. As *Fit Tables* são definidas pelos Analistas de Negócio em conjunto com os testadores da empresa Exactus Software.

Os modelos definidos pela Celera são estabelecidos pelos desenvolvedores em conjunto com os arquitetos de software de acordo com as especificações para cada caso de uso a ser desenvolvido a fim de fazer um vínculo dos elementos das *Fit Tables* com os compenentes da interface da tela.

Com isso, os desenvolvedores especificam os atributos de domínio a serem utilizados em cada GUI. Os atributos de domínio utilizados serão: o tamanho do componente, o tipo (VarChar, inteiro, decimal, date, char) e o tipo de apresentação (caixa de texto, botão, check ou uma combo). A definição dos atributos deve ser realizada conforme ilustrado na Tabela 2.

Tabela 2: Propriedades a serem especificadas na Celera

Propriedade	Descrição
Tipo	Define o tipo do campo (Inteiro, Decimal, VarChar, Senha, Date, Char).
Tamanho	Define a quantidade de caracteres do componente.
Decimais	Se o campo for um decimal, define quantas casas de decimais serão permitidas.
Persistido	Define se o campo será persistido em banco de dados, ou não será persistido.
Apresentação	Define a forma de apresentação do componente (Exemplo: combo, check, botão, caixa de texto).
Valor padrão	Define o valor padrão que será informado, se o campo for numérico, pode ser especificado um valor 0, se o campo for do tipo texto (VarChar), pode ser uma String vazia (espaço em branco).
Obrigatório	Define se o campo é obrigatório ou não.
Código Java	Informar blocos de código(s) java a ser utilizado para auxiliar na geração de *scripts* Sikuli. Utilizado apenas quando não for possível gerar 100% do código.

A Figura 16 ilustra o processo de especificação dos atributos de domínio produzida pela Celera para definirem uma GUI. Nesta figura, por exemplo, são especificados os campos código e descrição, na qual Figura a 16 (A) ilustra a especificação do atributo de domínio referente ao campo código. A Figura 16 (C) demonstra a especificação do atributo de domínio descrição a serem utilizados na produção da interface seguradora, conforme demonstrado na Figura 16 (B).

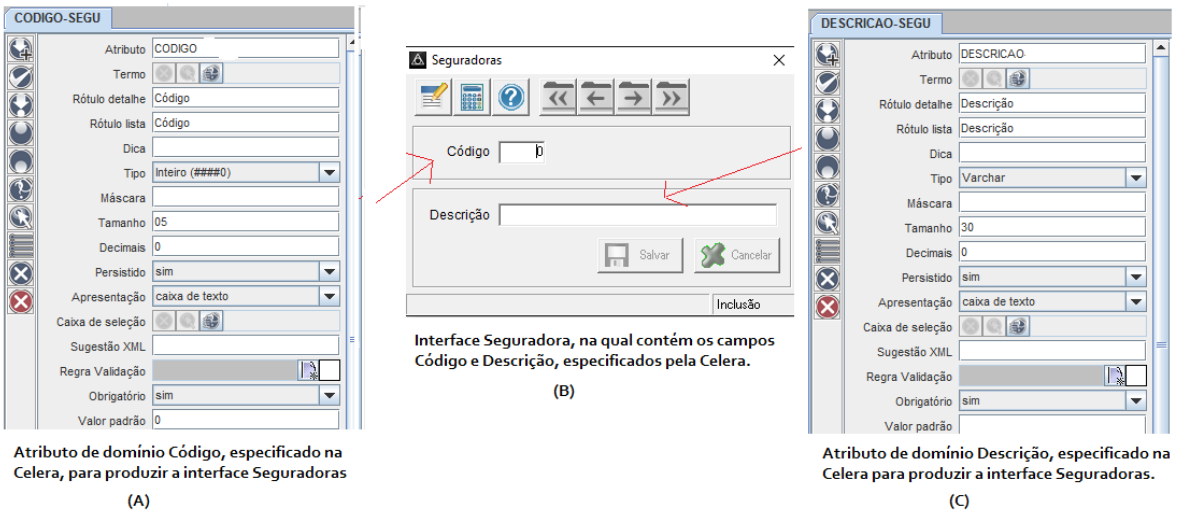


Figura 16: Modelo de entrada especificado na Celera

Os próximos componentes de casos de teste a serem estabelecidos são os passos, que são definidos pelos modelos da Celera. Em termos de teste de software, os modelos da Celera são importantes pois definem os passos do caso de teste. Outro fator a ser destacado é que a elaboração dos modelos da Celera já fazem parte do processo de MDD empregado pela Exactus Software. Por isso, esse passo não difere do processo que já é utilizado na empresa, apenas sendo aproveitado para a geração de casos de teste.

Por fim, os oráculos serão obtidos através das especificações definidas nas *Fit Tables* pelos testadores, conforme ilustrado na Figura 17, onde é demonstrado as saídas esperadas pelos casos de teste.

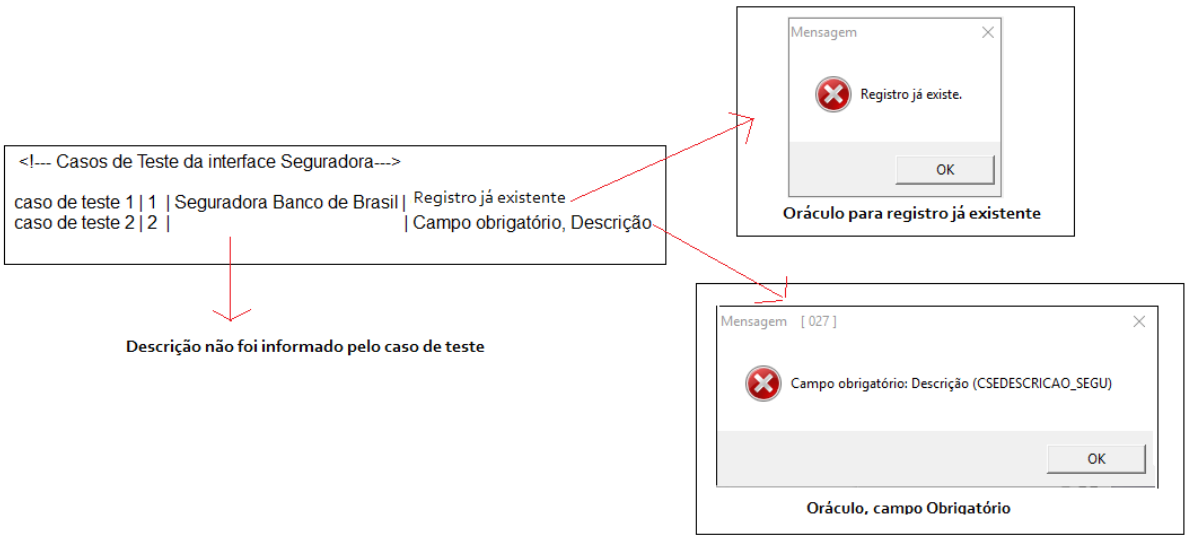


Figura 17: Oráculos

Com isso, o processo de automatização dos testes dos sistemas legados produzidos pela Celera, ocorre através das especificações dos casos de teste e dos atributos de domínio já introduzidos pela Celera. Neste contexto, o processo Celera produzirá *scripts* Sikuli utilizando instruções definidas em *templates*. Essas instruções serão responsáveis por realizar consultas de entrada nas *Fit Tables* e nos modelos da Celera, a fim de serem transformados em códigos para *scripts* Sikuli. Sendo assim todo o processo de geração de *scripts* Sikuli ocorrerá através dos modelos que serão utilizados dentro do processo MDT e dos casos de teste desenvolvidos pelos testadores.

A Figura 18 ilustra todo o processo da definição dos elementos que serão utilizados na produção do *scripts* Sikuli pelo processo Celera. Cada trecho do *template* será processado para consultar os casos de testes e os modelos definidos na Celera para produzir o código correspondente, a partir de dois modelos.

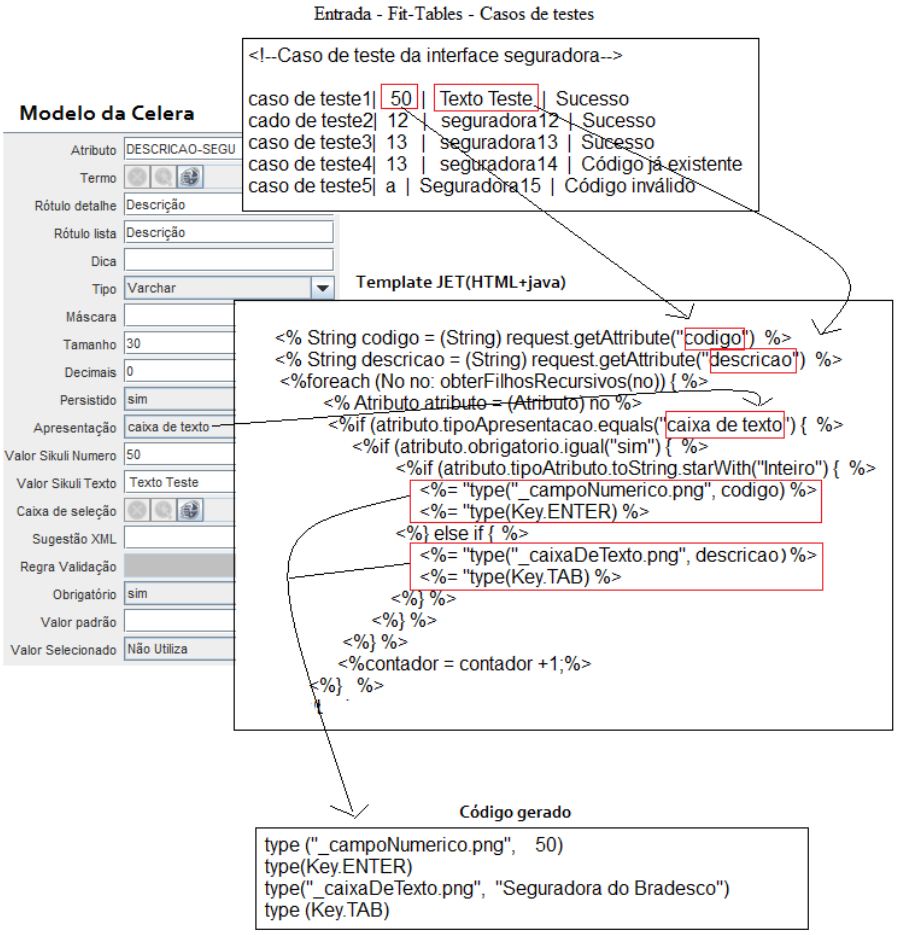


Figura 18: Geração de script Sikuli baseada em *templates*

Os modelos da Celera, são utilizados para iniciar as transformações, nesses modelos os desenvolvedores especificam os atributos de domínio a serem utilizados na produção dos *scripts* Sikuli. O segundo modelo utilizado na geração de código é uma *Fit Table* (arquivo texto) de casos de teste que devem conter as informações de valores (válidos e inválidos) a serem introduzidas pelo Sikuli para a realização dos casos de teste.

Após da definição dos modelos da Celera e das *Fit Tables*, é definido o processador de *templates* em um formato *Java Emitter Templates* (JET) com código Java, responsável por instanciar o *template* com base nas entradas. Esse elemento é encarregado por consultar as entradas e retornará o código *script* Sikuli.

Por fim é necessário verificar os oráculos dos casos de teste conforme ilustrado na Figura 19, na qual o processo de geração é composto pelos modelos da Celera, pelas *Fit Tables* e o processador de *templates*. Os modelos da Celera são responsáveis por repassar para o processador de *template* as informações referentes a obrigatoriedade, descrição e se o campo

é único ou não. As *Fit Tables* representadas pelos *scripts* dos casos de teste, são responsáveis por repassarem para o processador de *template* o caso de teste a ser utilizado na geracão dos oráculos. Por fim, o processador de *templates* é responsável por gerar os oráculos através definições dos atributos de domínio no processo MDD e dos *scripts* de entrada dos casos de teste.

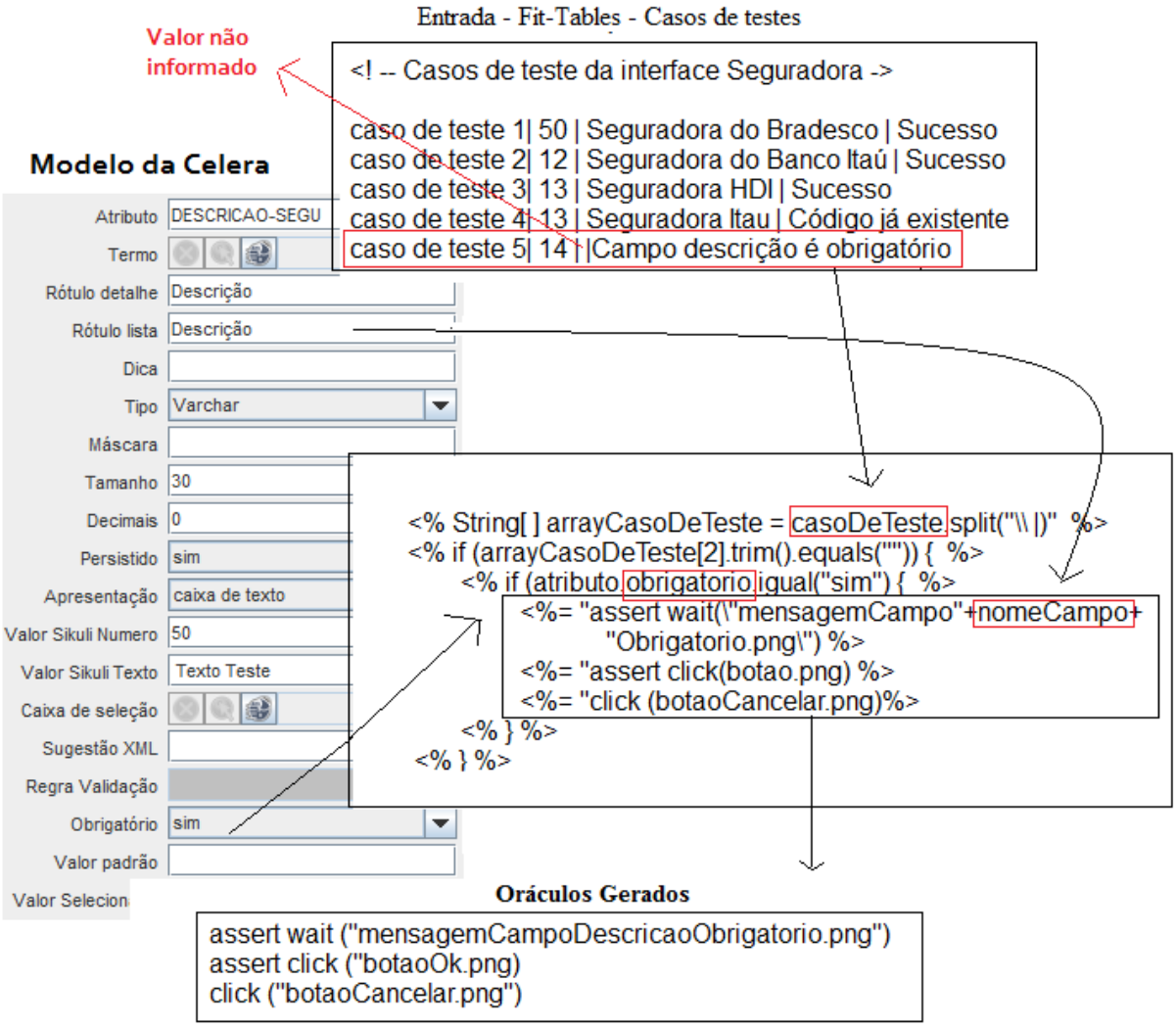


Figura 19: Geração de oráculos

A Figura 20 ilustra a utilização dos oráculos e seus repectivos *scripts* Sikuli produzidos pelo processo.

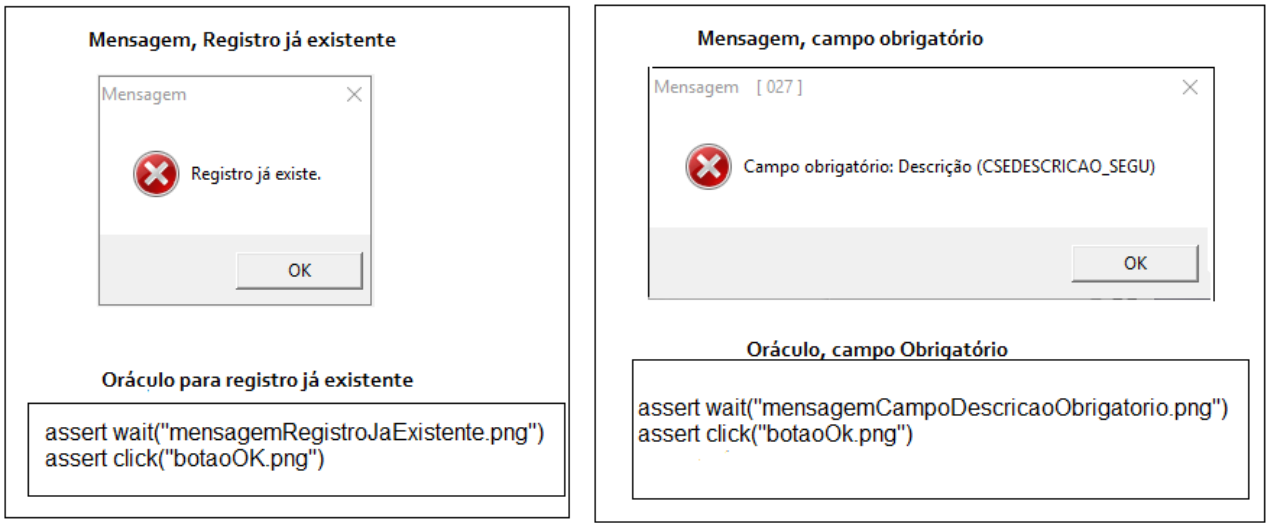


Figura 20: Oráculos da aplicação

4.2.2 ESTUDO DE CASO

No contexto deste estudo, foi realizado um estudo de caso como forma de validar esta proposta. Para isso, foram definidas as seguintes Hipóteses (H).

- $\mathbf{H}_0$: O uso da geração de casos de teste automático do processo MDD da Exactus Software não aumenta a eficiência da abordagem de teste da empresa.

- $\mathbf{H}_1$: O uso da geração de casos de teste automático do processo MDD da Exactus Software aumenta a eficiência da abordagem de teste da empresa.

Neste trabalho, a eficiência relatada em $\mathbf{H}_0$ e $\mathbf{H}_1$ será medida a fim de :

- Verificar o quanto os casos de testes gerados neste processo automático serão úteis para a aplicação obtida através do processo MDD;
- Avaliar a qualidade dos casos de teste gerados automaticamente; e
- Avaliar o resultado do processo de automatização de teste em relação ao manual.

4.2.3 METODOLOGIA

Considerando as hipóteses $\mathbf{H}_0$ e $\mathbf{H}_1$ descritas anteriormente, para testar a validade das hipóteses foi conduzido um estudo de caso com o processo de MDT dentro da empresa

Exactus Software contemplando um determinado conjunto de funcionalidades da aplicação sendo desenvolvida.

A eficiência considerando $\mathbf{H}_0$ e$\mathbf{H}_1$ vai ser medida inicialmente utilizando a métrica de *Code Coverage* dos casos de testes gerados automaticamente.

Com isso, foram implementados módulos de geração de casos de teste para as seguintes funcionalidades:

- **Seguradora**: uma GUI que contém dois campos (Código e Descrição). A definição e a sequência dos campos na GUI é demonstrado abaixo:
  - Código: campo deverá ser obrigatório, do tipo numérico (no máximo 4 dígitos), único e sua forma de apresentação deverá ser do tipo caixa de texto; e
  - Descrição: campo deverá ser obrigatório, do tipo VarChar como no máximo 30 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto.

- **Moeda**: uma GUI que contém cinco campos (Código, Sigla, Descrição, Tipo e Frequência). A definição e a sequência dos campos na GUI é demonstrado abaixo:
  - Código: campo deverá ser obrigatório, do tipo Varchar com 1 caracter apenas e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Sigla: campo deverá ser obrigatório com no máximo 8 caracteres, do tipo VarChar e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Descrição: campo deverá ser um campo obrigatório, do tipo VarChar, com no máximo 30 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Tipo: campo deverá ter apenas um caracter, do tipo VarChar, é obrigatório e sua forma de apresentação deverá ser do tipo caixa de seleção; e
  - Frequência: campo deverá ter apenas um caracter, do tipo VarChar, é obrigatório e sua forma de apresentação deverá ser do tipo caixa de seleção.

- **Fornecedor**: uma GUI que contém 12 campos (Código, Nome, Endereço, Bairro, Município, UF, Cep, Fone, Fax, Contato, CNPJ/CPF e Insc. Estadual). A definição e a sequência dos campos na GUI é demonstrado abaixo:

  - Código: campo deverá ser um valor numérico, é obrigatório, com no máximo 15 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Nome: campo deverá ser obrigatório, do tipo VarChar com no máximo 50 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Endereço: campo deverá ser obrigatório, do tipo VarChar com no máximo 50 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Bairro: campo deverá ser obrigatório, do tipo VarChar com no máximo 20 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Município: campo deverá ser obrigatório, do tipo VarChar com no máximo 30 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - UF: campo deverá ser obrigatório, do tipo VarChar com no máximo 2 caracteres e sua forma de apresentação deverá ser do tipo caixa de seleção;
  - Cep: campo deverá ser obrigatório, do tipo Inteiro, com no máximo 8 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Fone: campo deverá ser obrigatório, do tipo Inteiro, com no máximo 8 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Fax: campo deverá ser obrigatório, do tipo Inteiro, com no máximo 8 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - Contato: campo deverá ser obrigatório, do tipo VarChar com no máximo 20 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto;
  - CPF / CNPJ: campo deverá ser do tipo numérico, obrigatório com no máximo 14 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto; e
  - Insc. Estadual: campo deverá ser do tipo VarChar, obrigatório com no máximo 15 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto.

- **Tipo do Item**: uma GUI que contém dois campos (Código e Descrição). A definição e a sequência dos campos na GUI é demonstrado abaixo; e
  - Código: campo deverá ser obrigatório, do tipo numérico (no máximo 4 dígitos), único e sua forma de apresentação deverá ser do tipo caixa de texto; e

  - Descrição: campo deverá ser obrigatório, do tipo VarChar como no máximo 30 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto.

- **Classificação Contábil**: uma GUI que contém dois campos (Código e Descrição). A definição e a sequência dos campos na GUI é demonstrado abaixo.

  - Código: campo deverá ser obrigatório, do tipo numérico (no máximo 2 dígitos), único e sua forma de apresentação deverá ser do tipo caixa de texto; e

  - Descrição: campo deverá ser obrigatório, do tipo VarChar como no máximo 30 caracteres e sua forma de apresentação deverá ser do tipo caixa de texto.

4.2.4 RESULTADOS

Para validar a proposta deste trabalho com o uso de uma abordagem MDT, por meio da geração de casos de teste para a ferramenta Sikuli, onde dois modelos são transformados e utilizados nos sistemas, foi realizado um estudo de caso com a avaliação da métrica de *Code Coverage*.

A métrica de *Code Coverage* será adotada para medir a qualidade dos casos de teste produzidos do código V.B. produzido pela Celera, a fim de verificar os códigos referentes as camadas de negócio e persistência. Será descartado no entanto os códigos referentes à camada de visão e os componentes (de terceiros e os que foram desenvolvidos em outras linguagens, porém de domínio da própria Exactus Software) utilizados no sistema produzido pela Celera.

Para este estudo foram selecionados cinco GUIs e os resultados parciais desse estudo de caso são sumarizados na Tabela 5 sendo demonstrada a cobertura do código V.B. gerada pelo Celera:

- **Seguradora:** 77 linhas (das 78 existentes) foram cobertas pelos casos de testes totalizando 98% de código coberto;

- **Moeda:** 101 linhas (das 107 existentes) foram cobertas pelos casos de testes totalizando 94,39% de código coberto;

- **Fornecedor:** 211 linhas (das 222 existentes) foram cobertas pelos casos de testes totalizando 95,04% de código coberto;

- **Tipo do Item:** 95 linhas (das 99 existentes) foram cobertas pelos casos de testes totalizando 95,95% de código coberto; e

- **Classificação Contábil:** 94 linhas (das 98 existentes) foram cobertas pelos casos de testes totalizando 95,91% de código coberto.

Tabela 3: Cobertura do código V.B. gerado pela Celera

GUI	Total de Linhas	Total de linhas Cobertas	% de cobertura
Seguradora	78	77	98%
Moeda	107	101	94,39%
Fornecedor	222	211	95,04%
Tipo do Item	99	95	95,95%
Classificação Contábil	98	94	95,91%

Considerando os resultados acima, podem ser observadas evidências que suportam a hipótese alternativa $\mathbf{H}_1$. No entanto, mais estudos devem ser conduzidos.

4.2.5 DISCUSSÃO

Diferentemente do trabalho atual, os trabalhos de LI et al. (2006), ALVES et al. (2008) e LAMANCHA et al. (2009) não utilizam a métrica de *Code Coverage* como forma de avaliação dos casos de teste pois, os trabalhos estão focados em construir modelos de casos de teste utilizando diagramas de UML 2.0. Com isso, os resultados são demonstrados utilizando diagramas de sequência para os casos de testes desenvolvidos pelos testadores de acordo com cada GUI a ser realizado o teste.

Enquanto LI et al.(2006), ALVES et al. (2008) e LAMANCHA et al. (2009) apresentam os resultados em diagramas de sequência da UML 2.0, ALMEIDA; OLIVEIRA (2014) relatam que o trabalho está em andamento e que apenas as fases de revisão bibliográfica e a elaboração da proposta foram concluídas, com isso não é relatado nenhum resultado parcial ou final neste trabalho.

Diferentemente dos trabalhos citados acima cujo os resultados são demonstrados em diagramas da UML 2.0 e do trabalho atual que utiliza a métrica de *Code Coverage* em sua avaliação, OLAJUBU et al. (2015) apresentam os resultados em uma tabela, indicando se os casos de testes produzidos pelo processo MDT desenvolvido obtiveram sucesso ou falha.

Com isso, é importante destacar que esse trabalho utiliza o Sikuli, para realizar os testes de acordo com cada GUI e seus casos de testes respectivos, para os sistemas legados produzidos pela Celera. O Sikuli é utilizado neste trabalho devido a uma versão antiga da ferramenta *Visual Studio* adquirida pela empresa Exactus Software, que não tem suporte de teste unitário e *framework* de teste por isso, o Sikuli se mostrou como uma alternativa. Uma dificuldade observada no uso da ferramenta Sikuli, é a instabilidade de interface, porém neste caso, o processo MDD segue um padrão bem estabelecido e definido pela empresa Exactus Software fazendo com que o ponto fraco do Sikuli seja resolvido devido ao processo MDD.

5 GERAÇÃO DE CASOS DE TESTES FUNCIONAIS PARA APLICAÇÕES WEB

Este capítulo tem como objetivo apresentar uma proposta de avaliação de automatização da geração dos casos de testes dentro da empresa Exactus Software. Esta proposta consiste em gerar casos de testes funcionais para aplicações Web.

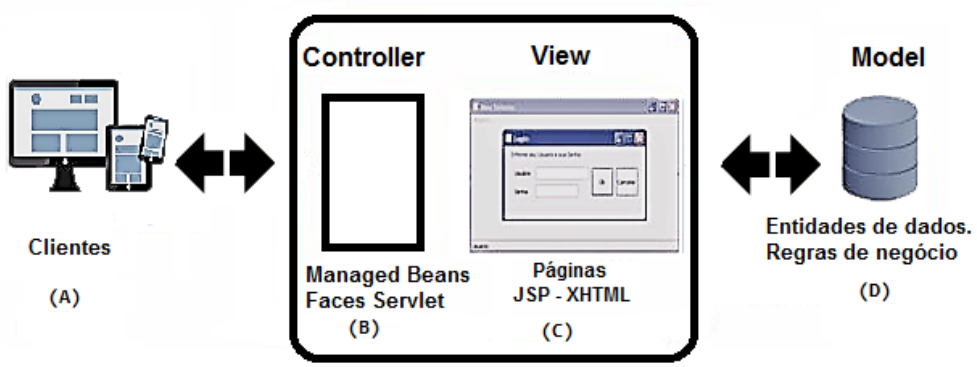
Na próxima Seção 5.1 são apresentados os conceitos e definições do JSF e do *Primefaces*; e na seção (Seção 5.2) é apresentado o processo de geração de casos de testes funcionais sobre um projeto Web denominado Exactus CRM.

5.1 JSF E PRIMEFACES

O JSF é um *framework* para desenvolvimento de aplicações Web com Java. KEHE et al. (2013) relatam que o JSF foi criado devido a necessidade de desenvolver aplicações Web visando ganho de desempenho, flexibilidade, reuso, facilidade de manutenção e eficiência na construção de interfaces Web dinâmicas.

Desta forma, o *framework* foi baseado no padrão de projeto *Model-View-Controller* (MVC). Esse padrão de projeto visa separar o desenvolvimento do sistema em três camadas: *Model*, *View* e *Controller*. A primeira camada, *Model*, é responsável por representar os objetos da camada de negócio e fornecer ao controlador o acesso aos dados. A segunda camada, *View*, é responsável por encaminhar as ações do usuário para a camada *Controller* e demonstrar os resultados obtidos em uma interface gráfica. Por fim a terceira camada, *Controller*, faz a ligação das camadas *Model* e *View* interpretando as solicitações do usuário e retornando a visualização adequada à solicitação (KEHE et al., 2013; LUCKOW; MELO, 2010) conforme detalhada na Figura 21.

Figura 21: Padrão de Projeto MVC



Na Figura 21 nota-se que no JSF a camada *View* é formada por páginas JSP ou XHTML (conforme detalhado na Figura 21 (C)) que utilizam componentes que respondem a eventos solicitados pelos clientes (conforme detalhado na Figura 21 (A)) declarados por meio de tags. Esses componentes são associados a um *Managed Bean* ou *Faces Servlets* (conforme detalhado na Figura 21 (B)) identificado através de anotações que permitem gerenciar o redirecionamento de páginas, a utilização de validadores, a comunicação com outras camadas como, por exemplo, a *Model* (conforme detalhado na Figura 21 (D)) e, por fim, a injeção de dependência (JUNWU; JUNLING, 2010).

Outra característica do JSF é a utilização da tecnologia *Facelets* como definições de interfaces. Essas interfaces, denominadas *Template* de páginas, incluem os componentes de telas sendo que alguns destes componentes fazem parte do pacote JSF e outros são distribuídos em bibliotecas que são oferecidas por diversas empresas que estendem as funcionalidades dos componentes já oferecidos pela especificação como: *Apache Tobago*, *PrimeFaces*, *RichFaces* e *IceFaces*.

Atualmente, uma das bibliotecas mais populares é o *Primefaces*. Essa biblioteca oferece inúmeras opções de componentes tanto para as aplicações Web quanto Mobile (LUCKOW; MELO, 2010), no entanto, sua utilização é possível em projetos de código aberto ou comerciais, desde que respeitados os termos da licença.

O *Primefaces* oferece uma integração com o *Cascading Style Sheets* (CSS) (um mecanismo para adicionar estilo a um documento Web) e seus componentes aplicam funções em JavaScript e *Asynchronous JavaScript and XML* (AJAX) para implementar as funcionalidades que se aproximam dos objetos de uma *Graphical User Interface* (GUI) em aplicações desktop, oferecendo todos os eventos necessários que uma interface gráfica necessita.

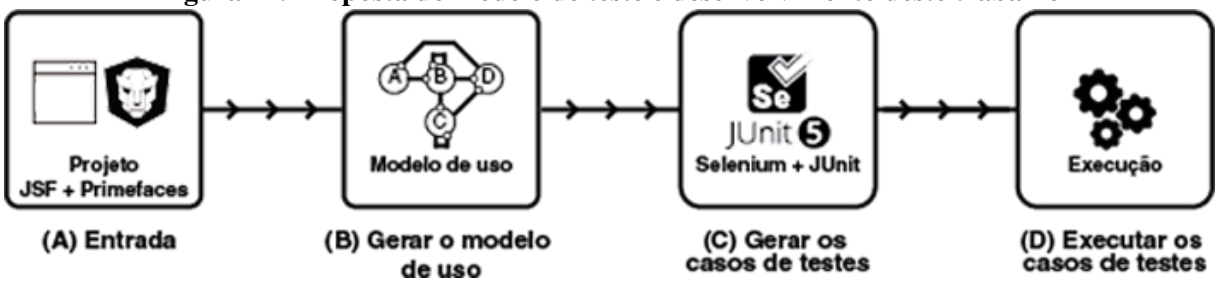
5.2 GERAÇÃO DE CASOS DE TESTE NO EXACTUS CRM

Foi conduzido um estudo de caso com o Exactus CRM com o objetivo de gerar casos de teste funcionais automáticos para aplicações Web. Para isso, foi elaborada uma estratégia através do desenvolvimento de uma ferramenta denominada *Morpheus Web Testing* que consiste na geração de casos de teste utilizando dois *frameworks* que auxiliam os desenvolvedores na construção de aplicações web: JSF e *Primefaces*.

Essa geração de casos de testes foi dividida em quatro passos, conforme demonstrado na Figura 22.

Durante o primeiro passo, a Entrada (veja Figura 22(A)), é necessária a entrada de um projeto que utiliza os *frameworks* JSF e *Primefaces*. Após a inserção do projeto, é iniciado o processo de geração de modelo de uso (veja Figura 22 (B)). Neste segundo passo, são extraídas da aplicação Web informações sobre a estrutura de todas as janelas, *widgets*, assim como os seus atributos e eventos da interface gráfica. Estes eventos são utilizados para gerar casos de teste da WUI que são sequências de eventos de WUI. Finalizado o segundo passo, é iniciada a geração de casos de teste (veja Figura 22 (C)), em que devem ser utilizados os *frameworks JUnit* e *Selenium WebDriver* para a geração de casos de testes. Por fim, no último passo (veja Figura 22 (D)), é realizada a execução dos casos de testes.

Figura 22: Proposta do modelo de teste e desenvolvimento deste trabalho



Na etapa de entrada, foi desenvolvido um software que identifica os objetos e as propriedades de uma WUI. Sendo assim, o *Morpheus Web Testing* utiliza estratégias diferentes para cada tipo de entrada, conforme demonstrado na tabela tabela 4 e detalhado abaixo.

- Para os objetos do tipo: *InputTextArea*, *InputNumber*, *InputMask*, *InputText*, *Passoword*, *TextEditor*, *ckEditor* , ou *Editor*, são utilizadas as estratégias de inserção de texto como: números, textos gerados de forma aleatória e textos em branco, ou seja, sem nenhum conteúdo;

- Para os objetos do tipo: *Button*, *CommandButton*, *ComandLink*, *Link* e *LinkButton*, as estratégias elaboradas foram o clique sobre estes objetos;
- Para o objeto do tipo *SelectBooleanButton*, *SelectBooleanCheckBox*, *SelectOneButton*, *SelectOneRadio*, *SelectOneMenu*, *SelectOneListBox*, *SelectManyButton*, *SelectManyMenu* ou *SelectManyCheckBox*, a estratégia a ser utilizada será de seleção; e
- Para os objetos do tipo *calendar* foram adotadas as estratégias de inserção de datas aleatórias, números inteiros e texto vazio.

Tabela 4: Componentes e estratégias utilizadas pelo *Morpheus Web Testing*

Componente	**Estratégia**					
	Inserção de números inteiros e aleatórios	inserção de textos aleatórios	Inserção de textos em branco	Clique	Seleção	Inserção de datas aleatórias
InputTextArea	X	X	X			
InputNumber	X	X	X			
InputMask	X	X	X			
InputText	X	X	X			
Passoword	X	X	X			
TextEditor	X	X	X			
Editor	X	X	X			
ckEditor	X	X	X			
Button				X		
CommandButton				X		
ComandLink				X		
Link				X		
LinkButton				X		
SelectBooleanButton					X	
SelectBooleanCheckBox					X	
SelectOneButton					X	
SelectOneMenu					X	
SelectOneListBox					X	
SelectManyButton					X	
SelectManyMenu					X	
SelectManyCheckBox					X	
calendar	X		X			X

Vale destacar que, para as estratégias de inserção de textos e números inteiros aleatórios, caso a propriedade maxLenght não seja especificada no componente um tamanho padrão de 50 caracteres será atribuído.

Durante a segunda etapa, os objetos inseridos dentro de um elemento de formulário da WUI selecionadas são identificadas e automaticamente associadas a uma estratégia (definida na etapa 1) a serem utilizadas na interface. Vale ressaltar que, para cada tipo de objeto identificado, diferentes tipos de estratégias serão utilizadas.

O processo de geração automática dos casos de testes (com apoio ferramental) é realizado através de instruções definidas em *templates*. CZARNECKI et al. (2005) definem *templates* como arquivos textos quaisquer, instrumentados com condições de seleção e expansão de código. Essas instruções são responsáveis por realizar consultas de uma entrada que podem ser um programa, uma especificação textual ou diagramas e, como resultado, é possível obter o parâmetro para produzir o código fonte (CLEAVELAND, 1988).

Neste trabalho, o processo de geração dos casos de teste ocorre através de uma entrada do código de uma página Web (XHTML) e, através desta página Web, o processador de *template* consegue utilizar as informações referentes aos elementos e as propriedades da WUI a fim de produzir código Java a ser utilizado em conjunto com o *framework Selenium WebDriver* para a execução dos testes, conforme demonstrado na Figura 24.

Seja o exemplo do elemento de uma WUI h:inputText demonstrado na linha 2 da Figura 24(A). É possível perceber que este elemento irá atribuir um valor para a classe ”MeuForm”, atributo ”textoSimples”. Uma vez definido o elemento WUI, o *template* conseguiu identificá-lo através do código el.getName().equals(”inputText”) conforme demonstrado na linha 6 Figura 24(C) que, por sua vez, irá gerar um código Java em que será atribuído o valor de Texto 1 através do comando element.sendKeys(”Texto 1”), demonstrado na Figura 24(D) linha 5 e inserida durante a execução do teste conforme demonstrado na Figura 24(B).

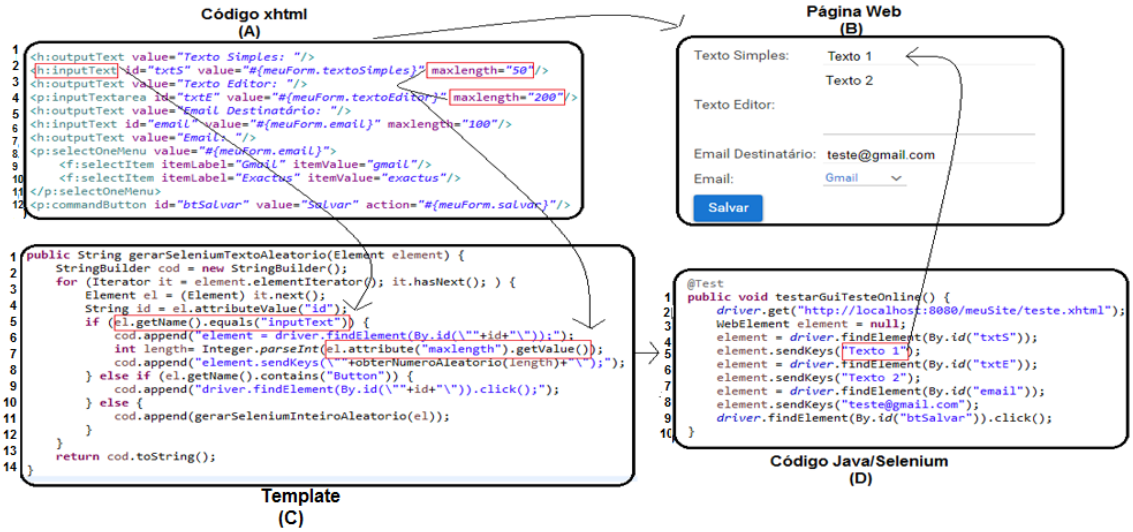
A Figura 23, ilustra um trecho de código Java gerado pelo processo definido acima (veja Figura 24) em que, para cada campo definido pelo termo driver.findElement(By.id(”id do campo”)) serão inseridos os valores que se encontram nos termos element.sendKeys(”valor”), como por exemplo: na Figura 23, linha 6 é definido driver.findElement(By.id(”nome”)), em que o valor nome, refere a propriedade Id, de um objeto do tipo inputText localizado em uma página xhtml. Após o *Selenium WebDriver* localizar o objeto e a propriedade na página Web, é inserido um valor, que pode ser (um texto aleatório, número aleatório ou um texto em branco) na página Web através do comando element.sendKeys(”um texto”), conforme demonstrado na Figura 23, linha 7. Por fim, para os objetos do tipo botão, conforme indicado na Figura 23, linhas 18 e 19, são geradas instruções do tipo clique.

É importante enfatizar que, a abordagem *Morpheus Web Testing* é um software de

Figura 23: Exemplo de código Java gerado

```java
@Test
public void instrutorTextoAleatorio() {
      driver.get(
        "http://localhost:8080/meuSite/instrutor.xhtml");
      WebElement element = null;
      element = driver.findElement(By.id("nome"));
      element.sendKeys("pclwdgsfxwndfiyv");
      element = driver.findElement(By.id("email"));
      element.sendKeys("cdddoomuofnjmbse");
      element = driver.findElement(By.id("telefone"));
      element.sendKeys("tvjqgbkosj");
      element = driver.findElement(By.id("celular"));
      element.sendKeys("hhghkzljlg");
      element = driver.findElement(By.id("instrutorLogin"));
      element.sendKeys("cmukmqfeyuxejyob");
      element = driver.findElement(By.id("instrutorSenha"));
      element.sendKeys("nquwehuero");
      driver.findElement(By.id("btSalvarInstrutor")).click();
      driver.findElement(By.id("btLimparCamposInstrutor")).click();
}
```

Figura 24: Geração de código Java com o *framework* Selenium baseado em *templates*



código aberto, sendo disponível para acesso atravé dos endereço [1]. Além disso, a abordagem *Morpheus Web Testing* foi desenvolvida utilizando os seguintes *frameworks*:

- **Java**: JDK versão 1.8; e
- **Dom4j**: Uma biblioteca Java de código aberto para trabalhar com XML, XPath e XSLT.

5.2.1 AVALIAÇÃO

No contexto deste estudo, foi realizado um estudo de caso como forma de validar esta proposta. Para isso foram definidas as seguintes Hipóteses $\mathbf{H}_0$ e $\mathbf{H}_1$.

- $\mathbf{H}_0$: O processo de geração de casos de teste funcionais automáticos para aplicações Web que utilizam *frameworks* JSF e componentes de interface *Primefaces* não conseguem alcançar uma maior cobertura de código em comparação com uma técnica do estado da arte.

- $\mathbf{H}_1$: O processo de geração de casos de teste funcionais automáticos para aplicações Web que utilizam *frameworks* JSF e componentes de interface *Primefaces* conseguem alcançar uma maior cobertura de código em comparação com uma técnica do estado da arte.

5.2.2 METODOLOGIA

Considerando as Hipóteses $\mathbf{H}_0$ e $\mathbf{H}_1$ descritas anteriormente, para testar e validar a hipótese, foi conduzido um estudo de caso afim de verificar o uso de um processo de geração de casos de teste funcionais automáticos para aplicações Web que utilizam *frameworks* de desenvolvimento Web como o JSF e *Primefaces*, utilizando a métrica de *Code Coverage* para analisar a cobertura de código. Para este estudo foi utilizado um projeto Web denominado Exactus CRM e, duas ferramentas de geração de casos de teste automático para aplicações Web, *Morpheus Web Testing* e *CrawlJax* (técnica do estado da arte), com o objetivo de realizar uma avaliação comparativa entre ambas aplicações.

Com isso, os casos de testes foram gerados utilizando o Exactus CRM. Vale destacar que o software utiliza os *frameworks* descritos a seguir:

- **JSF**: versão 2.2.8;

[1] https://github.com/raneves/morpheus.git

- ***Primefaces***: versão 6.0;
- ***Hibernate - Java Persistence API* (JPA)**: versão 5.0;
- ***Maven***: versão 3.1;
- **cobertura**: versão 2.1;
- ***jetty***: versão 9.4.9;
- **Java**: JDK versão 1.8;
- ***selenium-java***: versão 3.7.1; e
- **MySql**: versão 5.5.

O Exactus CRM utiliza o desing *Pattern* MVC, e é divido em 4 camadas descritas a seguir:

- **Camada *Data Access Object* (dao)**: Camada responsável por conter toda a lógica de acesso e execução referente ao banco de dados;
- **Camada *Plain Old Java Object* (pojo)**: Camada responsável por fazer o mapeamento das classes para as tabelas do banco de dados, ou seja, uma entidade JPA representada pelo padrão de projeto denominado JavaBeans (a classe deverá possuir atributos privados, um construtor padrão sem argumentos e métodos *getters* e *setters* para cada atributo);
- **Camada managedBean**: Contém códigos responsáveis por realizar o *back-end* da aplicação como regras de negócio e validações em geral. Esta camada recebe os dados digitados pelos usuários através das páginas XHTML, processa e devolve os resultados da operação para a página; e
- **Camada util** : Camada responsável por conter códigos referentes as validações, como: cpf, cnpj, emails, cep e telefone. Por fim, contém a *Aplication Program Interface* (API) de envio de emails.

Vale ressaltar que, o Exactus CRM contém um total de 207 classes e 11415 linhas de código. A quantidade de classes (veja Tabela 5 coluna 2) e o total de linhas de código (veja Tabela 5 coluna 3) de cada camada estão descritas a seguir:

- **camada dao**: 65 classes e 1618 linhas de código;

- **camada managedBean**: 39 classes e 4857 linhas de código;
- **camada util**: 4 classes e 171 linhas de código; e
- **camada pojo**: 99 classes e 4769 linhas de código.

5.2.3 RESULTADOS

Conforme mencionado anteriormente, para validar a proposta deste trabalho foi realizado um estudo de caso afim de verificar o uso de um processo de geração de casos de testes funcionais automáticos para aplicações Web que utilizam dois *frameworks* que auxiliam os desenvolvedores para o desenvolvimento de WUI, o JSF e o *Primefaces*. Para a realização deste estudo, foram gerados 91 casos de teste pela abordagem do *Morpheus Web Testing* sobre o Exactus CRM. Como referencial foi utilizada a abordagem do *CrawlJax*, neste caso, também medindo a cobertura alcançada pelo Exactus CRM quando executada com o *CrawlJax*. Vale destacar que ambos os projetos (Morpheus Web Testing e o CrawlJax) utilizaram a métrica de *Code Coverage* como forma de avaliação deste estudo.

Sendo assim, comparando os resultados entre as abordagens do estado da arte, neste caso a do *CrawlJax* com o *Morpheus Web Testing*, pode demonstrar evidências que suportam a hipótese $\mathbf{H}_1$.

Os resultados deste estudo estão sumarizados na Tabela 5, demonstrados no Gráfico I (%) e descritos a seguir.

- **Camada dao**: 1062 linhas (das 1618 existentes) foram cobertas pelos casos de testes, totalizando 66% de cobertura. Enquanto na abordagem do *CrawlJax*, 543 linhas (das 1618 existentes) foram cobertas pelos casos de testes, totalizando 34% de cobertura. Foi aplicado o teste estatístico de normalidade *Shapiro-Wilk* sobre as duas abordagens, e os resultados apresentam evidências de que a cobertura com ambas abordagens não seguem uma distribuição normal, (w=0.87864 e p-value=$1.196e^{-05}$) para o *Morpheus Web Testing*, e (w=0.89158 e p-value=$3.438e^{-05}$) para a abordagem do *CrawlJax*. Sendo assim, em uma comparação entre as coberturas das abordagens do *Morpheus Web Testing* e a do *CrawlJax*, foram observadas diferenças estatisticamente relevantes, segundo o teste estatístico *Man-Whitney Wilcox*, (w=26042, p-value=0.0001478);
- **Camada pojo**: 1200 linhas (das 4769 existentes) foram cobertas pelos casos de testes, totalizando 25% de cobertura. Enquanto na abordagem do *CrawlJax*, 1081 linhas (das 4769 existentes) foram cobertas pelos casos de testes, totalizando 23% de cobertura.

Foi aplicado o teste estatístico de normalidade *Shapiro-Wilk* sobre as duas abordagens, e os resultados apresentam evidências de que a cobertura com ambas abordagens não seguem uma distribuição normal, (w=0.848242 e p-value=7.137e^{-05}) para o *Morpheus Web Testing*, e (w=0.92879 e p-value=4.637e^{-05}) para a abordagem do *CrawlJax*. Sendo assim, em uma comparação entre as coberturas das abordagens do *Morpheus Web Testing* e a do *CrawlJax*, não foram observadas diferenças estatisticamente relevantes, segundo o teste estatístico *Man-Whitney Wilcox*, (w=5122.5, p-value=0.5825);

- **Camada managedBean**: 2295 linhas (das 4857 existentes) foram cobertas pelos casos de testes, totalizando 48% de cobertura. Enquanto na abordagem do *CrawlJax*, 2234 linhas (das 4857 existentes) foram cobertas pelos casos de testes, totalizando 46% de cobertura. Foi aplicado o teste estatístico de normalidade *Shapiro-Wilk* sobre as duas abordagens, e os resultados apresentam evidências de que a cobertura com ambas abordagens não seguem uma distribuição normal, (w=0.92668 e p-value=0.01408) para o *Morpheus Web Testing*, e (w=0.89555 e p-value=0.001648) para a abordagem do *CrawlJax*. Sendo assim, em uma comparação entre as coberturas das abordagens do *Morpheus Web Testing* e a do *CrawlJax*, não foram observadas diferenças estatisticamente relevantes, segundo o teste estatístico *Man-Whitney Wilcox*, (w=854.5, p-value=0.3495);

- **Camada util**: 109 linhas (das 171 existentes) foram cobertas pelos casos de testes, totalizando 64% de cobertura. Enquanto na abordagem do *CrawlJax*, 77 linhas (das 171 existentes) foram cobertas pelos casos de testes, totalizando 46% de cobertura. Foi aplicado o teste estatístico de normalidade *Shapiro-Wilk* sobre as duas abordagens, e os resultados apresentam evidências de que a cobertura com ambas abordagens seguem uma distribuição normal, (w=0.95568 e p-value=0.7518) para o *Morpheus Web Testing*, e (w=0.84161 e p-value=0.2001) para a abordagem do *CrawlJax*. Sendo assim, em uma comparação entre as coberturas das abordagens do *Morpheus Web Testing* e a do *CrawlJax*, não foram observadas diferenças estatisticamente relevantes, segundo o teste estatístico *t-Student*, (t=2.2479, df=4.272 e p-value=0.08358); e

- **Todas as camadas**: 4641 linhas (das 11415 existentes) foram cobertas pelos casos de testes, totalizando 41% de cobertura. Enquanto na abordagem do *CrawlJax*, 3935 linhas (das 11415 existentes) foram cobertas pelos casos de testes, totalizando 35% de cobertura. Foi aplicado o teste estatístico de normalidade *Shapiro-Wilk* sobre as duas abordagens, e os resultados apresentam evidências de que a cobertura com ambas abordagens não seguem uma distribuição normal, (w=0.93908 e p-value=1.261e^{-7}) para o *Morpheus Web Testing*, e (w=0.90323 e p-value=2.445e^{-10}) para a abordagem do *CrawlJax*. Sendo

assim, em uma comparação entre as coberturas das abordagens do *Morpheus Web Testing* e a do *CrawlJax*, foram observadas diferenças estatisticamente relevantes, segundo o teste estatístico *Man-Whitney Wilcox*, (w=26042, p-value=0.0001478);

Vale destacar que o intervalo de confiança do teste estatístico aplicado sobre ambas abordagens foi de 0,95 além disso, o *Morpheus Web Testing* e a abordagem do *CrawlJax* não foram capazes de gerar códigos de testes para alcançar algumas linhas de código, como o caso das camadas pojo e maganagedBean que são associadas aos métodos *getters()* e *setters()* das classes. Outro caso encontrado são as camadas util e dao, uma vez que todos os métodos que pertencem as classes destas camadas possuem tratamento de *exceptions*, e em muitos casos não foi possível simulá-las.

Tabela 5: Cobertura do código do Exactus CRM

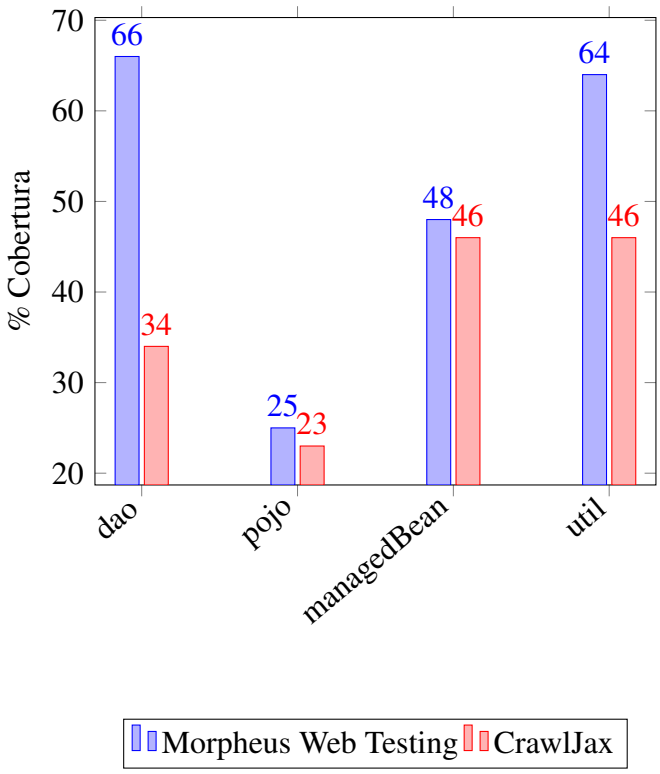
Camada	Total de Classes	Total de Linhas	Total de linhas Cobertas pelo Morpheus Web Testing (%)	Total de linhas Cobertas pelo CrawlJax (%)
dao	65	1618	1062 (66%)	543 (34%)
pojo	99	4769	1200 (25%)	1081 (23%)
managedBean	39	4857	2295 (48%)	2234 (46%)
util	4	171	109 (64%)	77 (46%)
Total	**207**	**11415**	**4641 (41%)**	**3935 (35%)**

Neste contexto, considerando os resultados descritos acima, podem ser observadas evidências que suportam a hipótese $\mathbf{H}_1$ descrita anteriormente, pois, é possível observar que, para todos os cenários o *Morpheus Web Testing* conseguiu atingir um desempenho melhor na média, tanto para a camada dao, pojo, managedBean e a util. Outro fator importante a ser destacado é que, foram realizados testes estatísticos comparando as camadas entre as abordagens *Morpheus Web Testing* e o *CrawlJax*, e foram observadas diferenças estatísticas especificamente na camada dao, no entanto não foram observadas diferenças estatísticas nas camadas pojo, managedBean e util. Por fim, foi realizado um teste estatístico sobre todas as camadas entre ambas as abordagens, e os resultados demonstraram diferenças estatísticas.

5.2.4 DISCUSSÃO

Este estudo apresentou um processo de geração de casos de teste funcionais automáticos para aplicações Web. Para isso, foi elaborada uma estratégia de geração de casos de teste utilizando dois *frameworks* que auxiliam os desenvolvedores na construção de aplicações Web: JSF e *Primefaces*.

Gráfico I % Cobertura Morpheus X CrawlJax



Vale ressaltar que, a abordagem do *CrawlJax* obteve uma média de 35% de cobertura dos casos de teste, enquanto a ferramenta *Morpheus Web Testing* obteve uma média de 41% de cobertura dos casos de teste, e os resultados mostram evidências estatisticamente relevantes de que a cobertura foi diferente entre ambas as abordagens, e com isso trazendo evidências para o suporte da hipótese alternativa $\mathbf{H}_1$. Um dos fatores que contribuíram para que a abordagem do *Morpheus Web Testing* obtivesse uma maior cobertura em relação ao *CrawlJax*, ocorreu devido a utilização de componentes de interfaces XHTML no processo de geração dos casos de testes que foram definidos pelo *framework* JSF juntamente com os componentes de interface do *Primefaces*.

Convém salientar que, na abordagem do *Morpheus Web Testing* a camada dao atingiu uma diferença significativa de cobertura, se comparado ao *CrawlJax* e um dos fatores que determinaram essa disparidade foi que, ao identificar os componentes da página XHTML, a abordagem *Morpheus Web Testing* sintetiza múltiplos casos para inserir, enquanto a abordagem do *CrawlJax* dispara cliques sobre os *links* ao invés de inserir valores aleatórios. Outro ponto importante a se destacar é que as camadas pojo e managedBean foram pouco influenciadas, potencialmente porque possuem muitos métodos *getters* e *setters* associados as classes.

É importante enfatizar que, a abordagem do *CrawlJax* utiliza componentes genéricos do HTML para realizarem a geração dos eventos que são associados a cada componente e,

em termos de geração de casos de teste, é gerado um teste para executar uma ação. Porém o *Morpheus Web Testing*, não utiliza a marcação genérica do HTML, mas como está utilizando o arquivo XHTML do JSF há mais informações do que um arquivo HTML em comum.

De fato, dado que na abordagem do *Morpheus Web Testing* ocorre um processo de avaliação de uma aplicação utilizando uma abstração de mais alto nível e, através das páginas XHTML é possível identificar um componente genérico como uma div do tipo menu dentro do código, devido a tag do componente de interface e consequentemente, ocorre a possibilidade de identificar níveis de interações mais elaborados que não são identificados caso contenha informações de um determinado elemento HTML dentro da div. Outro ponto importante a ser destacado é que, com um menu é possível prever um tipo de interação que em uma div não se preveria, ou seja, com um menu seria necessário um simples clique, com um botão direito ou esquerdo para abrir outras ações, ao contrário da div, que não seria possível determinar outras opções.

Dessa forma, enquanto a abordagem do *CrawlJax* utiliza componentes de interface em nível identificado de marcação HTML genérica, neste trabalho são utilizados componentes de interfaces complexos definidos por meio do *Primefaces*, sendo assim é possível identificar mais estados da aplicação.

Outro fator importante a ser destacado é que, diferentemente da abordagem atual, o *AutoBlackTest* desenvolvido por (MARIANI et al., 2012), que utiliza duas estratégias para realizar o processo de geração de casos de testes (i) a utilização de uma técnica de aprendizado denominada Q-learnig, e (ii) múltiplas heurísticas para abordar alguns casos em comum como: preenchimento de formulários e persistência de arquivos. Por outro lado, o *Morpheus Web Testing* utiliza a semântica de componentes de interfaces do XHTML no processo de geração dos casos de testes. Deve-se salientar que, efetivamente para se comparar o trabalho atual com (MARIANI et al., 2012) seriam necessários mais experimentos e, tal fato pode ser conduzido em trabalhos futuros.

Convém destacar que, na área de geração de casos de teste para aplicações web, dois trabalhos relacionados (*CrawlJax* e *WebMate*) reportaram o uso da estrutura HTML de uma página web para gerar os testes. Páginas HTML possuem um conjunto limitado de componentes de interface por padrão, sendo eles: elementos de tipo *input* de texto, botões, *links* entre outros. Na ferramenta *Morpheus Web Testing*, por outro lado, foi proposto o uso de componentes de interface mais complexos a partir dos *templates* XHTML que utilizam o *framework Primefaces* do JSF para gerar os casos de teste. Nos *templates* XHTML, são especificados componentes com um nível maior de abstração, como por exemplo: *widget* de tipo calendário, *inputs* com

campos de validação, *links* e botões associados aos *input*, entre outros.

Dessa forma, neste trabalho, foi investigado se o uso de componentes de interface em um nível maior de abstração como menus, poderiam gerar uma maior cobertura em estratégias de geração de casos de teste, em comparação ao uso de elementos de interação do HTML. A nossa hipótese de pesquisa geral é de que, quando é identificado um *widget* do tipo *calendar*, associados a um campo de texto, especificados no *template* XHTML com o *framework Primefaces*, é possível prever níveis de interação mais elaborados (considerando a relação entre os componentes, estratégias de validação e campos de mensagens) em comparação á identificação separada dos elementos HTML, sem relação dos *links* de uma *widget* com os *inputs* e sem a identificação de estratégias de validação, potencialmente recuperáveis no XHTML com o *framework Primefaces*.

Em termos de gerais, significa que, ao se obter um código HTML gerado de uma interface web que utiliza os *frameworks* JSF e *Primefaces*, o resultado seria um HTML contendo um elemento UL com sub-elementos LIs no entanto, ao analizar um componente *Primefaces*, esta UL potencialmente seria um menu. Uma vez que sabemos que é um menu, em teoria sabemos quais possibilidades de interação que o usuário teria com aquele componente, sendo assim, com esta informação adicional quando ao visualizar que o componente é um menu, em teoria há possibilidade de: (i) obter mais informações do que um arquivo HTML e, como isso, prever e melhorar os níveis de interação dos componentes, e (ii) a possibilidade de se gerar mais casos de testes funcionais e, consequentemente, atingir uma maior cobertura de código.

É preciso considerar que, na abordagem *Morpheus Web Testing*, não foram exercitados todas as possibilidades de componentes de interface do *framework Primefaces*. Os componentes que foram considerados no processo de geração de casos de teste foram: *input* com mascáras e mensagens de validação; *widget* de calendário; *checkbox*; *links*; *combobox*; área de texto; botões de radio. Para cada tipo de *widget*, foi estabelecido um conjunto de casos de teste, diferentes possibilidades de entrada, considerando os mecanismos de interação específicos de cada *widget* e a integração dos mesmos na aplicação web.

Por fim, o potencial de generalização da abordagem *Morpheus Web Testing*, ou seja, a melhoria em relação ao *CrawlJax* está relacionado ao fato de utilizar componentes de interface como entrada e não simplesmente código HTML. Tal potencial significa que ao utilizar a mesma abordagem com outros *frameworks* ou abstrações de componentes de interface como o *JavaScript* ou *Angular* como entrada, possivelmente os resultados semelhantes seriam observados.

6 CONSIDERAÇÕES FINAIS

Esse trabalho apresentou uma forma para automatizar a geração de casos de teste a partir de modelos na empresa Exactus Software. Nesse sentido, foram realizados dois estudos de casos no processo de geração de casos de testes: (i) geração de casos de teste a partir de modelos definidos em um processo MDT; e (ii) a geração de casos de testes funcionais a partir de modelos de aplicações web que utilizam componentes de interface através de arquivos XHTML.

No primeiro estudo de caso, foi implantado o processo MDT em um processo MDD desenvolvido pela Exactus Software denominado Celera, utilizando a ferramenta Sikuli através de scripts gerados automaticamente. Para isso, uma transformação baseada em templates é aplicada para automatizar geração de casos de teste, através de especificações de requisitos definidas no processo Celera, em modelos de entrada. Os resultados indicam que há evidências que o uso da geração de casos de teste automático do processo MDD da Exactus Software aumenta a eficiência da abordagem de teste da empresa.

No segundo estudo de caso, ocorreu a automatização do processo de geração dos casos de testes funcionais na plataforma de desenvolvimento Exactus CRM. Para isso, foi elaborada uma estratégia para a geração de casos de teste funcionais automatizados para aplicações Web que utilizam dois *frameworks* que auxiliam os desenvolvedores para o desenvolvimento de WUI, o JSF e o *Primefaces*. Os resultados indicam que há evidências que o *Morpheus Web Testing* consegue alcançar uma maior cobertura de código em comparação a técnica do estado da arte, pois para todos os cenários o *Morpheus Web Testing* conseguiu alcançar um desempenho melhor na média.

Além disso, em termos de abrangência da abordagem proposta, a ferramenta *Morpheus Web Testing* pode ser aplicada a qualquer projeto JSF que utilize o *framework Primefaces* de componentes de interface. A avaliação da abordagem de geração de casos de teste da ferramenta *Morpheus Web Testing* foi conduzido em um ambiente real, neste caso empresarial, em uma aplicação que está sendo utilizada em produção e, que representa uma melhoria na prática.

Os resultados indicam que há evidências que o *Morpheus Web Testing* consegue alcançar uma maior cobertura de código em comparação as técnicas do estado da arte, pois para todos os cenários o *Morpheus Web Testing* conseguiu alcançar um desempenho melhor na média. Embora a avaliação tenha apresentado evidências que a abordagem supera o estado da arte, mais avaliações precisam ser realizadas para verificar a capacidade de generalização da abordagem.

Vale ressaltar ainda que, nesse trabalho os estudos de caso foram conduzidos em um ambiente real, neste caso empresarial, aplicações que estão sendo utilizadas em produção e, que representa uma melhoria na prática.

Este trabalho apresentou as seguintes contribuições para o tópico de automatização da geração de casos de testes:

- Automatização do processo de testes da empresa Exactus Software dos sistemas legados gerado pela Celera;
- Automatização do processo de testes da empresa Exactus Software dos sistemas legados gerado pela Celera;
- Os modelos representados pelas *Fit Tables*;
- Automatização do processo de testes, em aplicações Web que utilizam os *frameworks* JSF e o *Primefaces*; e
- O processo de automatização dos testes atingiu uma melhoria na qualidade do software, redução dos custos de desenvolvimento, diminuição do tempo geral do ciclo de vida do software e por fim, permitir que os testes acompanhem a evolução e desenvolvimento de um ambiente de teste completo.

As sugestões de trabalhos futuros são:

- Produzir outros estudos do uso de MDD junto com a ferramenta *GUI Testing* Sikuli, considerando a questão dos padrões de Interface e de padrões que o Sikuli utilizará na execução dos testes;
- Estender o estudo de caso e tentar viabilizar outras técnicas para tentar automatizar ainda mais o processo de geração de casos de teste; e

- Realizar o mesmo estudo com outros tipos de componentes do JSF como: *RichFaces* e *IceFaces*, além disso, utilizar outras ferramentas de geração de casos de testes funcionais como o *WebMate*, *GuiTTAR* ou o *AutoBlackTest*.

A seguir serão descritas as limitações deste projeto. Ao total foram levantadas quatro limitações.

- O estudo de caso conduzido não contempla todas as funcionalidades para o MDT desenvolvido;
- O estudo de caso foi conduzido utilizando apenas uma aplicação WUI que utiliza os *frameworks* JSF e Primefaces, neste caso o Exactus CRM;
- O estudo de caso foi conduzido utilizando apenas uma ferramenta de geração de casos de testes funcionais como o CrawlJax;e
- Na abordagem *Morpheus Web Testing*, não foram exercitados todas as possibilidades de componentes de interface do *framework Primefaces*.

REFERÊNCIAS

ABBORS, F.; TRUSCAN, D.; LILIUS, J. Tracing requirements in a model-based testing approach. In: **Advances in System Testing and Validation Lifecycle, 2009. VALID '09. First International Conference on**. [S.l.: s.n.], 2009. p. 123–128.

AHO, P.; MENZ, N.; RÄTY, T. Enhancing generated java gui models with valid test data. In: IEEE. **IEEE Conference on Open Systems (ICOS'11)**. [S.l.], 2011. p. 310–315.

AHO, P. et al. Automated java gui modeling for model-based testing purposes. In: IEEE. **8^{th} International conference on Information technology: New generations (ITNG'11)**. [S.l.], 2011. p. 268–273.

AHO, P. et al. Murphy tools: Utilizing extracted gui models for industrial software testing. In: IEEE. **7^{th} IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW'14)**. [S.l.], 2014. p. 343–348.

ALMEIDA, C. C. de J.; OLIVEIRA, A. A. de. Qualitas: A proposal of process model development software driven models. In: **Proceedings of the 7th Euro American Conference on Telematics and Information Systems(EATIS'14)**. New York, NY, USA: ACM, 2014. p. 28:1–28:4.

ALVES, E. L.; MACHADO, P. D.; RAMALHO, F. Uma abordagem integrada para desenvolvimento e teste dirigido por modelos. In: **2nd Brazilian Workshop on Systematic and Automated Software Testing**. [S.l.: s.n.], 2008.

BAKER, P. et al. **Model-driven testing: Using the UML testing profile**. [S.l.]: Springer Science & Business Media, 2007.

BEYDEDA, S. et al. **Model-driven software development**. [S.l.]: Springer, 2005.

BINDER, R. V. **Testing object-oriented systems: models, patterns, and tools**. [S.l.]: Addison-Wesley Professional, 2000.

BITTAR, T. J. et al. Web communication and interaction modeling using model-driven development. In: **Proceedings of the 27th ACM International Conference on Design of Communication(SIGDOC '09)**. New York, NY, USA: ACM, 2009. p. 193–198.

BUARQUE, A. d. S. M. **Desenvolvimento de software dirigido por modelos: Um foco em engenharia de requisitos**. Tese (Doutorado) — Universidade Federal de Pernambuco, 2009.

CLEAVELAND, J. C. Building application generators. **IEEE software**, IEEE, v. 5, n. 4, p. 25–33, 1988.

CZARNECKI, K. et al. Model-driven software product lines. In: ACM. **Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications**. [S.l.], 2005. p. 126–127.

DALLMEIER, V. et al. Webmate: a tool for testing web 2.0 applications. In: ACM. **Proceedings of the Workshop on JavaScript Tools**. [S.l.], 2012. p. 11–15.

DELAMARO, M.; JINO, M.; MALDONADO, J. **Introdução ao teste de software**. [S.l.]: Elsevier Brasil, 2017.

DEURSEN, A. V.; KLINT, P.; VISSER, J. Domain-specific languages: An annotated bibliography. **Sigplan Notices**, v. 35, n. 6, p. 26–36, 2000.

DISTANTE, D. et al. Model-driven development of web applications with uwa, mvc and javaserver faces. In: BARESI, L.; FRATERNALI, P.; HOUBEN, G.-J. (Ed.). **Web Engineering: 7th International Conference, ICWE 2007 Como, Italy, July 16-20, 2007 Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 457–472.

EDWARD, K. Integrated, effective test design and automation. **Software Development**, v. 21, n. 2, p. 36–38, 1999.

FERNANDES, G. F. D. Geração automática de casos de teste a partir de requisitos. 2014.

FEWSTER, M. et al. Common mistakes in test automation. In: **Fall Test Automation Conference (FTAC'01)**. [S.l.: s.n.], 2001.

FRANCE, R.; RUMPE, B. Model-driven development of complex software: A research roadmap. In: **Future of Software Engineering, 2007. FOSE '07**. [S.l.: s.n.], 2007. p. 37–54.

FRANCE, R. B. et al. Model-driven development using uml 2.0: promises and pitfalls. **Computer**, IEEE, v. 39, n. 2, p. 59–66, 2006.

HENDRICKSON, E. The differences between test automation success and failure. **Proceedings of STAR West**, 1998.

HILDENBRAND, T.; KORTHAUS, A. A model-driven approach to business software engineering. In: CITESEER. **Proceedings of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI 2004)**. [S.l.], 2004. v. 4, p. 18–21.

JIAO, Y. et al. Towards model-driven methodology: a novel testing approach for collaborative embedded system design. In: IEEE. **2006 10th International Conference on Computer Supported Cooperative Work in Design**. [S.l.], 2006. p. 1–5.

JUNWU, X.; JUNLING, L. Developing crm system of web application based on javaserver faces. In: IEEE. **2^{nd} International Workshop on Education Technology and Computer Science (ETCS'10)**. [S.l.], 2010. v. 1, p. 766–769.

KANER, C. Improving the maintainability of automated test suites. In: **International Software Quality Week (ISQW'97)**. [S.l.: s.n.], 1997.

KASHYAP, A.; O'REILLY, S. **Lean Approach Through Model-Based Testing**. [S.l.]: Testing Experience Magazine, 2012.

KEHE, W. et al. Design and implementation of fire administrative management system based on jsf and ejb3. 0. In: IEEE. **5^{th} International Conference on Computational and Information Sciences (ICCIS'13)**. [S.l.], 2013. p. 555–558.

KLEPPE, A. G.; WARMER, J. B.; BAST, W. **MDA explained: the model driven architecture: practice and promise**. [S.l.]: Addison-Wesley Professional, 2003.

KULL, A. Automatic gui model generation: State of the art. In: IEEE. **23rd IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'12)**. [S.l.], 2012. p. 207–212.

LAMANCHA, B. P.; USAOLA, M. P.; GUZMAN, I. G. R. de. Model-driven testing in software product lines. In: **Software Maintenance, 2009. ICSM 2009. IEEE International Conference on**. [S.l.: s.n.], 2009. p. 511–514.

LI, N. et al. A framework of model-driven web application testing. In: **30th Annual International Computer Software and Applications Conference (COMPSAC'06)**. [S.l.: s.n.], 2006. v. 2, p. 157–162. ISSN 0730-3157.

LIMA, H. S. et al. Automatic generation of platform independent built-in contract testers. In: **SBCARS**. [S.l.: s.n.], 2007. p. 47–60.

LUCKOW, D. H.; MELO, A. A. de. **Programação Java para a WEB**. [S.l.]: Novatec Editora, 2010.

LUCRÉDIO, D. Uma abordagem orientada a modelos para reutilização de software. **INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO UNIVERSIDADE DE SÃO PAULO**, p. 37, 2009.

MARIANI, L. et al. Autoblacktest: Automatic black-box testing of interactive applications. In: IEEE. **15th IEEE International Conference on Software Testing, Verification and Validation (ICST'12)**. [S.l.], 2012. p. 81–90.

MEMON, A. et al. The first decade of gui ripping: Extensions, applications, and broader impacts. In: IEEE. **20th Working Conference on Reverse Engineering (WCRE'13)**. [S.l.], 2013. p. 11–20.

MEMON, A. M. et al. **Comprehensive Framework for Testing Graphical User Interfaces**. [S.l.]: University of Pittsburgh Pittsburgh, 2001.

MESBAH, A.; DEURSEN, A. van; LENSELINK, S. Crawling ajax-based web applications through dynamic analysis of user interface state changes. **ACM Trans. Web**, ACM, New York, NY, USA, v. 6, n. 1, p. 3:1–3:30, mar. 2012. ISSN 1559-1131.

MOREIRA, R. M.; PAIVA, A. C.; MEMON, A. A pattern-based approach for gui modeling and testing. In: IEEE. **24th IEEE International Symposium on Software Reliability Engineering (ISSRE'12)**. [S.l.], 2013. p. 288–297.

MYERS, G. J. et al. **The art of software testing**. [S.l.]: New Jersey: John Wiley & Sons, Inc., 2004.

NGUYEN, B. N. et al. Guitar: an innovative tool for automated testing of gui-driven software. **Automated Software Engineering**, Springer, v. 21, n. 1, p. 65–105, 2014.

OLAJUBU, O. et al. Automated test case generation from domain specific models of high-level requirements. In: ACM. **Proceedings of the 2015 Conference on research in adaptive and convergent systems**. [S.l.], 2015. p. 505–508.

OSTRAND, T. J.; BALCER, M. J. The category-partition method for specifying and generating fuctional tests. **Communications of the ACM**, ACM, v. 31, n. 6, p. 676–686, 1988.

PRESSMAN, R. S. **Engenharia de software**. [S.l.]: Makron books São Paulo, 1995.

RAUF, A. et al. Automated gui test coverage analysis using ga. In: IEEE. **7^{th} International Conference on Information Technology: New Generations (ITNG'10)**. [S.l.], 2010. p. 1057–1062.

ROCHA, A. R. C. d. et al. Qualidade de software: teoria e prática. **São Paulo: Prenttice Hall**, 2001.

SAKAL, M. Gui vs. wui through the prism of characteristics and postures. **Management**, v. 5, n. 1, p. 003–006, 2010.

SELIC, B. The pragmatics of model-driven development. **IEEE software**, IEEE Computer Society, v. 20, n. 5, p. 19, 2003.

SOMMERVILLE, I.; SAWYER, P. **Requirements engineering: a good practice guide**. [S.l.]: John Wiley & Sons, Inc., 1997.

TOLVANEN, J.-P.; POHJONEN, R.; KELLY, S. Advanced tooling for domain-specific modeling: Metaedit+. In: **Sprinkle, J., Gray, J., Rossi, M., Tolvanen, JP (eds.) The 7th OOPSLA Workshop on Domain-Specific Modeling, Finland**. [S.l.: s.n.], 2007.