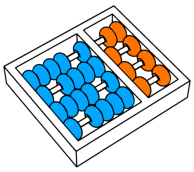
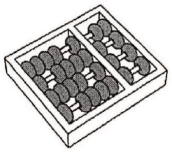


Caroline Castello Letizio

"Método de Modelagem e Geração de Testes para o Ambiente de Ensino à Distância TelEduc"

CAMPINAS
2013



Universidade Estadual de Campinas
Instituto de Computação

Caroline Castello Letizio

"Método de Modelagem e Geração de Testes para o Ambiente de Ensino à Distancia TelEduc"

Orientador(a): **Profa. Dra. Eliane Martins**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

Este exemplar corresponde à versão final da Dissertação defendida por Caroline Castello Letizio, sob orientação de Profa. Dra. Eliane Martins.

Eliane Martins

Assinatura do Orientador(a)

CAMPINAS
2013

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Maria Fabiana Bezerra Muller - CRB 8/6162

L568m

Letizio, Caroline Castello, 1987-
Método de modelagem e geração de testes para o ambiente de ensino à distância TelEduc / Caroline Castello Letizio. – Campinas, SP : [s.n.], 2013.

Orientador: Eliane Martins.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Engenharia de software. 2. Software - Testes. 3. UML (Linguagem de modelagem padrão). 4. Web sites - Testes. 5. Teste baseado em modelos. I. Martins, Eliane,1955-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Modelling and test generation method for the e-learning web environment TelEduc
Palavras-chave em inglês:
Software engineering
Software - Testing
Unified Modeling Language (Computer science)
Web sites - Testing
Model-based testing
Área de concentração: Ciência da Computação
Titulação: Mestra em Ciência da Computação
Banca examinadora:
Eliane Martins [Orientador]
Ana Maria Ambrósio
Claudia Maria Bauzer Medeiros
Data de defesa: 16-12-2013
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 16 de dezembro 2013, pela Banca examinadora composta pelas Professoras Doutoras:

Profª. Drª. Ana Maria Ambrósio
ETE / INPE

Profª. Drª. Claudia Maria Bauzer Medeiros
IC / UNICAMP

Profª. Drª. Eliane Martins
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Método de Modelagem e Geração de Testes para o Ambiente de Ensino à Distância TelEduc

Caroline Castello Letizio[1]

16 de dezembro de 2013

Banca Examinadora:

- Profa. Dra. Eliane Martins (*Orientadora*)
- Profa. Dra. Ana Maria Ambrósio
  Engenharia e Tecnologia Espaciais - INPE
- Profa. Dra. Claudia Maria Bauzer Medeiros
  Instituto de Computação - UNICAMP
- Profa. Dra. Ariadne Maria B. Rizzoni Carvalho
  Instituto de Computação - UNICAMP (Suplente)
- Profa. Dra. Emília Villani
  Divisão de Engenharia Mecânica Aeronáutica - ITA (Suplente)

[1]Suporte financeiro: bolsa CAPES 2011–2012

Resumo

Teste baseado em modelo é uma técnica na qual um sistema é modelado e geram-se testes a partir do modelo. Esta técnica apresenta como uma das maiores dificuldades a modelagem do sistema. Alguns dos empecilhos encontrados ao modelar são o que considerar do sistema e como representar essas considerações no modelo, incluindo em aplicações Web. Sendo assim, este trabalho propõe um método para auxiliar na modelagem de aplicações Web. Esta escolha é motivada pelo fato que esse tipo de sistema tem sido cada vez mais utilizado nos últimos anos e, com a sua evolução, sua complexidade tem aumentado consideravelmente, devido à dinamicidade e interatividade que esse tipo de aplicação oferece. A abordagem proposta é baseada em trabalhos correlatos, para se saber o que modelar da aplicação Web e como criar o modelo, descrevendo-se o passo-a-passo para se gerar os testes a partir dos modelos desenvolvidos e aplicá-los no sistema. A abordagem deste trabalho sugere tanto formas manuais como automáticas para os testes baseados em modelos. O método foi proposto para testar uma aplicação real: o ambiente de ensino à distância TelEduc. Os resultados, obtidos a partir da aplicação da proposta desse trabalho por um analista de teste e pela própria equipe do TelEduc, são analisados, a fim de validar a proposta e auxiliar no processo de desenvolvimento e aplicação dos testes de equipes de qualidade de *software*, principalmente, daquelas que não possuem nenhuma forma de organização de testes.

Abstract

Model based testing is a technique in which the system is modeled and, from this model, tests can be generated. The major difficulty about this technique is to generate the model, since it is problematic to define what should be considered from the system and insert these characteristics into the model. This technique is also used in Web applications which presents the same challenges.

Therefore, the present work proposes a method to assist in the modeling of Web applications. This sort of system has been increasingly used in recent years, where the system's complexity has considerably grown due to the dynamic and interactivity that this kind of application offers. The proposed method is based on related works in order to know what to model from the applications and how to create this model. The method contains the steps to generate tests cases from the created model, and how to execute them in the system. This work also suggests manual and automatic ways to execute the method. The proposal has been applied to test a real application: the e-learning Web environment TelEduc. The results obtained from the method's application by a test analyst and by the TelEduc team, are here analyzed to validate the method and to assist in the process development and tests execution by quality teams, especially those teams where there are no means of tests organization.

Agradecimento

"Obrigada, Senhor Deus, por Sua graça, pelas oportunidades que me aparecem, por iluminar meus caminhos, pela minha vida.

Obrigada, Senhor, pelos meus pais Jose Carlos e Luciane, e minha irmã Camila, que são a minha família, meu porto seguro, são quem me apoiam, me ouvem, me aconselham e me guiam. São meus confidentes e cúmplices. Proporcionam-me oportunidades, carinho, amor, a paz que tenho em minha vida.

Obrigada, Senhor, pelo Hudson, quem sempre tem sido meu ombro amigo, tem me levantado, apoiado. Está ao meu lado e sempre pronto a me ajudar. Ele me enche de ternura e carinho para que eu consiga seguir em frente com meus desafios.

Obrigada, Senhor, pelo Tiago Silveira, amigo de todas as horas e momentos, de horas de estudos e madrugadas de desabafos. Amigo que ajuda e está sempre disposto, que apoia e torce.

Obrigada, Senhor, pela Eliane Martins, que me aceitou para ser sua orientanda quando já não havia mais tempo para escolhas, que me orientou e me guiou nesse trabalho.

Obrigada, Senhor, pela minha família, por meus amigos e por todas as pessoas que colocou em meu caminho, que torceram por mim e me ajudaram de alguma forma para que esse trabalho fosse concluído.

Amém."

Agradeço à Capes pelo apoio financeiro.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Com a evolução da tecnologia, a Web tem sido utilizada, principalmente e cada vez mais, para executar aplicações de qualquer tipo ou área, seja ela de gerenciamento de uma firma até a simulação de pesquisas na área biológica, devido as suas vantagens na distribuição de informação pelas plataformas de qualquer lugar ou pessoa a qualquer momento [1]. Dinamicidade e maior interação do usuário têm sido as principais características das aplicações Web, tornando-as mais complexas. Com isso, vem crescendo a preocupação quanto à qualidade e confiança do sistema, por este ser heterogêneo, distribuído e concorrente. As características das aplicações Web, como implementações estruturadas e comportamento dependente de entradas (do usuário ou do sistema), fazem essas aplicações difíceis de serem compreendidas e implicam em novos desafios para os testes de *software* [1]. Sendo assim, as técnicas convencionais de teste, como geração estática de *scripts* de teste e testes manuais já não comportam a complexidade das aplicações, nem os comportamentos das empresas, em que os analistas de teste só têm acesso ao *software* quando este está projetado e codificado [2].

Um problema em uma aplicação Web pode custar milhões em dinheiro e a perda do negócio. Portanto, todas as entidades de uma aplicação Web devem ser testadas por inteiro para assegurar que a aplicação seja confiável e condizente com as especificações originais [1].

Uma das vertentes da Engenharia de *Software* é a dirigida por modelos, conhecida por Desenvolvimento Dirigido por Modelo (MDD, na sigla em inglês). Tal vertente propõe desenvolver o *software* com foco em modelos. Por envolver uma série de atividades de produção, testes de *software* também são necessários nesse tipo de desenvolvimento. Metodologias de geração de testes atestaram que casos de teste de qualidade poderiam ser derivados a partir dos modelos de desenvolvimento, criando-se a abordagem de teste baseado em modelo (*model based testing* - MBT) [3]. Essa abordagem não é necessariamente exclusiva para o MDD, sendo sua proposta útil para a área de testes em geral. Sendo

assim, o MBT já é usado para testes de aplicações Web.

As primeiras aplicações Web, logo no início do desenvolvimento da *Internet*, eram estáticas, sem alterações nas páginas por parte do usuário. A maior parte apenas possuía navegação de uma página para outra; essas caracterizavam a Web 1.0. A modelagem não era algo complexo, pois apenas precisava representar a página e o link que era utilizado para navegar para outra página. Com a evolução das aplicações Web, o usuário começou a interagir mais com o sistema, ou seja, as entradas e escolhas do usuário passaram a alterar o comportamento do sistema, o que gerou maior dinamicidade na criação e no processo de cada aplicação. Os tipos de modelagem para a Web 1.0 tornaram-se não mais condizentes. Essa evolução da Web, com o rápido avanço para a Web 2.0 e logo para a Web 3.0, transformou os testes baseados em modelo em um desafio, pois além da navegação entre as páginas de uma aplicação, também passaram a ser considerados os elementos dinâmicos da página e os vários tipos de requisições para vários servidores diferentes que uma aplicação pode utilizar.

Em aplicações Web, principalmente as comerciais, a abstração da lógica de uma aplicação por meio de um modelo permite que o processo de produção do sistema seja modificado, independentemente do código do mesmo [4]. O desenvolvimento dirigido por modelo é uma área que está em expansão, na qual os modelos são criados com o objetivo de diminuir a complexidade inerente ao desenvolvimento. A modelagem também ajuda *designers* na fase de implementação da interface gráfica, assim como proporciona suporte para implementação de testes.

Testes baseados em modelos são um tipo de técnica de teste de caixa preta, a qual trata a aplicação sem a necessidade de verificação do código fonte da mesma. Dessa forma, não é necessário assumir nada sobre a estrutura interna da aplicação, pois essa técnica avalia o comportamento dos dados de entrada e saída do sistema [5], com base nos requisitos funcionais e não-funcionais. Há muitas vantagens em se usar esse tipo de teste, entre elas:

- A geração de casos de teste começa cedo no ciclo de desenvolvimento do *software*, o que ajuda a encontrar defeitos (passo, processo ou definição de dados incorretos) logo no início, reduzindo-se o custo de correção de erros;
- A possibilidade de automatizar a geração de testes;
- A redução dos custos na geração dos testes. Neste caso, principalmente em sistemas que mudam frequentemente, um analista de teste necessita somente modificar o modelo do sistema e, com isso, rapidamente recriar seus testes. Assim, não se tem a necessidade de recriá-los manualmente e originar, acidentalmente, erros [6];
- O ganho com testes exploratórios e automatizados, pois, a partir do modelo, o analista de teste tem os caminhos a seguir em um teste exploratório.

A Web hoje nos proporciona páginas dinâmicas com vários *scripts* e programas inseridos apenas no lado cliente da aplicação [7]. O problema encontrado para a geração de testes é que não há uma modelagem eficiente para as aplicações dinâmicas, ou seja, aquelas em que usuários interagem com a aplicação. A maioria das pesquisas tenta criar outros padrões para construir modelos que consigam mostrar essa dinamicidade das aplicações. Tais padrões são baseados nas definições da UML (*Unified Modeling Language*) [8], mas não as utilizam de fato. Máquinas de estado finitas (MEF) e máquinas de estado finitas estendidas (MEFE) têm sido usadas como proposta para o teste baseado em modelo. Uma MEF pode representar a parte do controle de um sistema, tal como um sistema de semáforos. Entretanto, uma MEFE é necessária para sistemas mais complexos, que normalmente possuem controles e partes de dados como protocolos de comunicação [9]. Mesmo assim, esse tipo de modelagem ainda mostra limitações, como a impossibilidade de se mostrar concorrência entre eventos. *Statecharts* também têm sido utilizados para modelagem de aplicações Web, sendo um dos tipos de modelo que possui mais recursos para se representar um sistema, porém mesmo esses diagramas têm mostrado limitações em expressar aspectos específicos e críticos de aplicações Web. Não é possível, por exemplo, representar quem é o responsável, um usuário ou o sistema, por um evento que será recebido pela aplicação [10]. Sendo assim, é necessário encontrar um modelo que consiga superar as principais limitações, ou utilizar conceitos que ajudem a contornar esses problemas, como o modelo de estados da UML.

1.1 Objetivos

A maioria dos projetos e pesquisas estudados e analisados mostra como lidar com a complexidade das aplicações Web e até os resultados que produzem. A maior parte dos relatórios é complexa e parte da premissa de que quem os lê possui grande conhecimento sobre Engenharia de *Software*, tais como documentação e modelos, o que nem sempre é o caso.

Segundo Robinson [2], teste baseado em modelos é uma tecnologia mais complexa do que os testadores estão acostumados a utilizar. Sob o tradicional paradigma de teste, era aceitável contratar pessoas com pouca experiência em testes de *software*. Geração de teste, porém, exige entendimento e experiência em pensar de forma abstrata, entender o produto e entender o ciclo de desenvolvimento. Os testadores enfrentam a complexidade de se criar modelos, resistindo às tentativas de introduzir técnicas baseadas em modelos em *software* em produção. É necessário introduzir de forma gradual e didática a forma como se deve entender a aplicação para se produzir um modelo do mesmo.

Essa defasagem dos trabalhos em ensinar como modelar ocorreu num projeto real entre uma indústria de automóvel, a Unicamp, o INPE e o ITA, na qual essa indústria

começou a fazer testes a partir de modelo de estados em aplicações automotivas. A maior dificuldade foi, justamente, entender como fazer o modelo, o que é um estado e o que é transição, como realmente representar o comportamento do carro com a utilização do modelo.

Sendo assim, haveria uma forma de simplificar o processo de modelagem para geração de testes? Ou seja, haveria alguma forma simples de se ensinar a modelar um sistema e gerar os testes?

Um tipo de aplicação Web muito utilizado atualmente é o de gerenciamento de ensino, como os ambientes de ensino a distância. O TelEduc [55], ambiente de ensino a distância da Unicamp, utilizado por mais de quatro mil instituições do Brasil e Chile, é um sistema de *software* livre e é desenvolvido e mantido pela própria universidade, no Núcleo de Informática Aplicada à Educação (NIED). Sendo um ambiente de ensino, possui vários tipos de ferramentas, como as de comunicação (tais como fórum e correio) e de compartilhamento de arquivos (material de apoio e portfólio), além de gerenciar e editar cursos e pessoas envolvidos no ambiente. Esse sistema é executado utilizando um navegador Web, portanto é uma aplicação Web e é totalmente dinâmico, com utilização de elementos HTML e tecnologia Ajax entre outros componentes. É uma aplicação Web complexa e que não possui uma forma organizada de ser testado. Assim, esse trabalho propõe mostrar o que deve ser considerado de uma aplicação Web para modelar, no formato do sistema TelEduc, e validar esse modelo em relação ao comportamento da aplicação Web, ensinando a gerar testes a partir do modelo e aplicá-los para testar essa aplicação. A maior parte dos trabalhos relacionados a esta pesquisa não tratam do assunto dessa forma, parecendo considerar que os leitores dominem todos os passos da metodologia. Sobre a validação, as pesquisas mostram vários modelos diferentes para sistemas Web, mas não monstram se esses modelos realmente representam o comportamento da aplicação Web. Isto é ressaltado no trabalho de Andrews *et al.* [11], em que a maioria das propostas são aplicadas em sistemas pequenos e de exemplos, não sendo testadas em *software* real.

Como complemento à modelagem, este trabalho também apresenta um guia de como gerar os casos de teste e executá-los em uma aplicação Web.

Como a área que envolve a Web é muito grande e diversificada, este trabalho buscou focar as aplicações com apenas um único servidor, que possuem botões que fazem desnecessário o uso das opções do navegador Web, como é o caso do TelEduc. A metodologia proposta neste trabalho, com os objetivos citados acima, foi utilizada no ambiente por voluntários, produzindo resultados satisfatórios.

1.2 Contribuições

Analisando vários tipos de modelos utilizados pelas técnicas estudadas, pode-se perceber que o modelo de estados da UML na sua versão 2 consegue representar de forma satisfatória e concisa o comportamento de uma aplicação, diferente da máquina de estados clássica. Este último modelo não aceita condições de guarda, o que aumenta o número de estados, tornando-o incompreensível ou até gerador de uma explosão de estados. Portanto, esse trabalho explica como utilizar o modelo de estados da UML para modelar sistemas Web.

Outro ponto relevante do trabalho é a modelagem levando em conta os elementos dinâmicos que modificam uma página da aplicação Web, pois a maior parte dos trabalhos relacionados mostra como modelar o sistema como um todo, ou seja, o fluxo de navegação. Além disso, explica quais partes do sistema serão representadas em um único modelo e a importância da validação dos modelos.

Os resultados do estudo de caso mostraram sucesso da metodologia na explicação da modelagem das partes de uma aplicação e a geração de testes. Os resultados também mostraram que a metodologia produz testes de qualidade em relação à forma como a equipe do ambiente TelEduc testava, pois foram encontrados muito mais defeitos do que no método de teste da equipe.

1.3 Organização do Texto

Este trabalho está organizado da seguinte forma:

- Capítulo 2 – Modelagem de aplicações Web: apresenta os principais aspectos de uma aplicação Web, os desafios da modelagem e os modelos utilizados;
- Capítulo 3 – Teste de Software e Teste Baseado em Modelo: os conceitos fundamentais de teste de *software* e as técnicas existentes são descritos neste capítulo, assim como uma explicação mais detalhada sobre teste baseado em modelo.
- Capítulo 4 – Trabalhos Relacionados: muitos são os trabalhos e pesquisas relacionados à modelagem de aplicações Web e testes baseados em modelo. Os principais trabalhos relacionados são explicados.
- Capítulo 5 – Metodologia: este capítulo explica como modelar aplicações Web, validar o modelo produzido, e como gerar os testes e aplicá-los.
- Capítulo 6 – Estudo de Caso: o estudo de caso realizado foi com o sistema de ensino à distância TelEduc. A metodologia proposta no capítulo 5 foi aplicada e os resultados analisados.

- Capítulo 7 – Conclusão e Trabalhos Futuros: apresenta uma análise dos resultados produzidos e os desafios ainda não resolvidos, que podem ser a base para trabalhos futuros.

Capítulo 2

Modelagem Web

Cada vez mais aplicações Web têm sido utilizadas, aumentando assim, a preocupação na qualidade desse tipo de software.

A modelagem de um sistema é importante para todo o ciclo de vida do software, pois ajuda desenvolvedores, designers e analistas de testes em seus respectivos trabalhos. A seguir, é explicada a evolução das aplicações Web e a importância da sua modelagem. Para esse estudo, foi utilizado o diagrama de estados da UML, que também é explicado ao fim desse capítulo.

2.1 Aplicações Web

Nas últimas décadas, houve grande evolução e desenvolvimento rápido no uso da Web. Com essa evolução, aparecem as aplicações Web.

Entre os principais atributos de uma aplicação Web estão, segundo Pressman [12]:

- Intenso uso de redes: uma aplicação Web está presente em uma rede e atende às necessidades de clientes dos mais variados tipos;
- Concorrência: muitos usuários podem acessar uma aplicação Web no mesmo momento;
- Evolução contínua: aplicações Web evoluem de forma contínua. Diferente de *software* convencional que evolui seguindo um planejamento e/ou ordem cronológica.

Porém, para entender melhor as aplicações Web atuais, é necessário compreender primeiro a sua evolução e a suas estruturas.

Em 1989, a *World Wide Web* era criada por Sir Tim Berners-Lee, o qual estabeleceu o conceito de hipertexto – um texto que associa outras informações, também conhecido

como *link* [13]. A primeira geração de aplicações Web é chamada de Web 1.0, caracterizada por aplicações estáticas e leitura que somente seguiam estritamente uma categorização e nomeação de representações de elementos Web. Para cada requisição, as aplicações eram totalmente "de-para" entre cliente e servidor, ou seja, a requisição tinha que ser respondida pelo servidor. Estas aplicações eram mais voltadas para gerências, nas quais os responsáveis decidiam o fluxo de uso da mesma [14]. Pelo fato da restrição da aplicação ser estática e haver dependência com o servidor, e com o uso da Web como uma chave importante para a construção de uma aplicação, pensou-se em criar um sistema para um usuário mais experiente e com uma dinâmica maior, de modo que surgiu o que se denomina de Web 2.0 [15]. Esta geração incentiva a criatividade, o compartilhamento de informações e a colaboração entre usuários, ou seja, o usuário passa a se envolver mais com a aplicação. Redes sociais surgiram, permitindo que usuários, seja individualmente ou em comunidades, contribuíssem com conteúdo e trocassem conhecimentos na *Internet*. Essa interatividade traz ao usuário um sentimento de pertencer a alguma comunidade, assim como o senso de poder e domínio sobre as aplicações [14], [16].

Com essa evolução dos sistemas Web, o controle dos dados possibilita novas formas de gerência e criatividade na usabilidade e representação das aplicações. A Web passa a ser vista como uma plataforma que fornece serviços e o navegador do usuário representa o cliente que utiliza estes serviços [14].

A dinamicidade e interatividade das aplicações aumentaram e, com isso, usuários podem procurar e armazenar várias formas de dados na *Internet* (fotos, vídeos, textos, arquivos), observar resultados e publicá-los das mais diferentes formas. Com a ideia de compartilhamento e interação, vários são os serviços que a Web 2.0 passa a oferecer, como bate-papos (MSN e Skype), *e-mails* (Gmail, Yahoo, Hotmail), *bookmark* (ferramenta que permite organizar e administrar *links* que já se visitou ou que se planeja visitar sem ter que lembrar o endereço da página), jogos, *wikis*, *blogs*,*sites* de viagem (*sites* que permitem que o usuário explore lugares que se deseja visitar, permitem também compartilhar sugestões dos usuários sobre restaurantes, passeios e até mesmo preço de passagens), vídeos (Youtube), músicas, fotos (Picassa, Kodak Gallery), programação (*sites* que permitem que usuários criem e compartilhem novos códigos de aplicações da Web. Exemplos: Salesforce.com e Yahoo Widget), RSS (*Really Simple Syndication* – agregam *feeds* para que usuários leiam novidades das páginas que possuem RSS), redes sociais (Orkut, Facebook) etc.

Toda essa interação é possível devido às linguagens de programação e novas tecnologias que permitem comunicação de forma assíncrona com o servidor ou até mesmo que já carregam o conteúdo no navegador Web (*browser*) do usuário, de forma que não é necessária, em alguns momentos, uma requisição ao servidor para determinada ação do usuário (a página não precisa ser recarregada totalmente a cada ação do usuário). Como exemplo dessas tecnologias, temos o Ajax (*Asyncronhous Javascript* e XML), Adobe Flex,

Adobe Flash, Microsoft Silverlight, entre outras.

Com toda essa evolução, principalmente das tecnologias da Web semântica integradas que potencializam em larga escala as aplicações Web, inicia-se a Web 3.0 [16]. O princípio desse estágio é baseado na integração e análise de dados de várias fontes dentro dos novos fluxos de informação. A Web 3.0 surge para suprir as limitações da Web 2.0, como a pouca estrutura para gerenciar muitos dados e a incapacidade do sistema de integrar dados de diferentes fontes em diferentes formatos. Por isso, essa nova geração também é chamada de "Web inteligente" [17].

Porém, essa etapa da Web ainda está em desenvolvimento. Pesquisadores, desenvolvedores e até mesmo usuários têm definido a Web 3.0 à sua maneira, mas com um conceito em comum: a personalização da Web. Entretanto, a principal definição da Web 3.0 ainda não emergiu, pois ela está em desenvolvimento pela *World Wide Web Consortium* (W3C)[18] para se tornar realidade [13]. A maior parte das aplicações ainda acontece na versão Web 2.0.

Mesmo com essa evolução, a estrutura da aplicação Web continua a mesma. De forma geral, uma aplicação Web é composta por três partes: o lado cliente, representado por um *browser*; o lado servidor, que utiliza conteúdo tecnológico dinâmico da Web, como linguagem PHP, Java Servlets etc; e uma base de dados, na qual os dados da aplicação que podem ser requisitados pelo cliente são armazenados [19]. O lado cliente faz requisições ao lado servidor, por meio de uma rede, como a *Internet*. O servidor, por sua vez, faz as buscas na base de dados.

A Figura 2.1 representa essa estrutura.

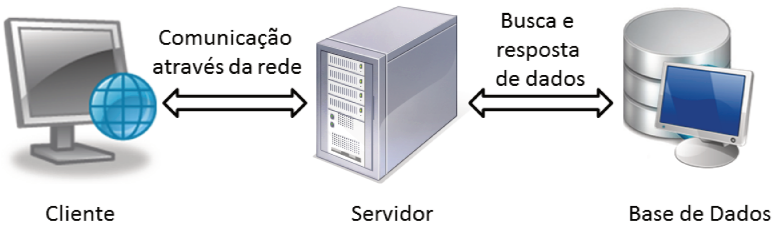


Figura 2.1: Estrutura da Aplicação Web.

A aplicação Web é composta por páginas identificadas por uma url , que podem ser geradas de forma estática (como o uso simples da linguagem HTML) ou dinâmica (uso de *scripts* e componentes Web, como o Ajax). Essas páginas possuem *interfaces* que são apresentadas no lado cliente, no qual, por meio da interação, faz as requisições ao servidor, que devolve uma requisição com modificação da mesma página ou apresentando uma nova página ao cliente.

2.2 Aspectos a serem Modelados em Aplicações Web

Apesar da estrutura da aplicação Web ser composta, geralmente, por três componentes, modelar todas as requisições entre um componente e outro é praticamente impossível.

Segundo Alalfi, Cordy e Dean [20], existem vários desafios para se modelar sistemas Web, pois esses evoluem de forma rápida, conforme as tecnologias também evoluem. Entre esses desafios, estão:

- O aumento do risco da não interoperabilidade e complexidade da integração dos componentes devido à diversidade e complexidade do ambiente das aplicações Web. A integração de alguns componentes e a aplicação Web, em geral, é extremamente acoplado de forma dinâmica, o que fornece ótima abstração para o entendimento dos desenvolvedores, porém, torna difícil a análise e verificação de testes. Isso ocorre devido às aplicações Web interagirem com muitos componentes que rodam em diversas plataformas de *hardware* e *software* – por exemplo, o lado cliente possui o navegador, linguagem HTML e *applets*. Já o lado servidor, possui Java Servlets e tecnologias como .NET. Todos esses componentes interagem e podem ser encontrados em vários servidores.

- Comportamento dinâmico: muitos componentes do lado cliente são gerados de forma dinâmica, como também a interação entre clientes e servidores pode ocorrer dinamicamente.

- Mecanismos de controle para acesso seguro aos recursos da aplicação: uma aplicação Web pode conter várias formas de entrada de dados, e os usuários podem interagir de formas diversificadas e até complicadas, que a aplicação pode não prever. Isso ocorre devido às aplicações possuírem componentes de banco de dados que podem fornecer os mesmos dados para diferentes usuários.

- Baixa observabilidade de saídas: normalmente, a análise das saídas geradas pela aplicação consiste na observação do documento HTML enviado para o usuário. Há outros tipos de saídas, como alteração do estado do servidor ou do banco de dados, mensagens enviadas para outras aplicações e para outros serviços. Algumas dessas saídas são difíceis de serem rastreadas.

Segundo Alalfi, Cordy e Dean [20], algumas propostas de modelo são dadas na literatura para tentar resolver esses problemas. Para modelagem de conteúdo, há métodos que verificam principalmente o conteúdo estático das aplicações, utilizando modelos UML ou *statecharts*, porém há necessidade de maior exploração da dinâmica das aplicações Web. Em questões de navegação, a literatura mostra que os modelos são baseados na UML,

grafos e *statecharts*. Em relação quando o número de estados do modelo se tornar muito grande, tornando o modelo difícil de ser visualizado e atualizado, a chamada explosão de estados, muitos autores utilizam métodos de modelos híbridos – mistura de dois ou mais modelos em um mesmo método, os quais levam em consideração a aplicação como um todo, modelando todos os níveis de abstração da mesma. A utilização de modelos separados para diferentes níveis da aplicação ajuda a reduzir a complexidade do modelo, como também do problema de tamanho que o modelo pode atingir. Porém, há o problema de o método ter que declarar de forma explícita e cuidadosa a integração entre os modelos.

Conseguir modelar e testar todos os comportamentos de uma aplicação Web é um grande desafio, por isso existem técnicas e propostas para testar algumas partes, alguns componentes e algumas estruturas da aplicação, para otimizar o processo como um todo. Sendo assim, esse trabalho tem como proposta a modelagem para testes do componente do lado cliente, especificamente na dinamicidade da alteração da página, segundo as entradas do usuário.

Como o foco é o lado do cliente, cada página da aplicação pode possuir conteúdo criado de forma estática ou dinâmica. Além disso, deve haver uma estrutura e organização para saber o que e como modelar. Os trabalhos analisados praticamente falam das mesmas divisões que uma aplicação Web pode ter e que deve ser levada em conta para modelagem, como em Winckler e Palanque [10] e em Andrews [21]. Alalfi, Cordy e Dean [20] dividem, de forma mais didática, a aplicação Web em três perspectivas para modelagem: navegação, conteúdo e comportamento, sendo que em cada perspectiva há uma subdivisão:

- Navegação
  - Estática: *links* estáticos para alterar de uma página para outra;
  - Dinâmica: dependendo da entrada, o mesmo *link* pode levar a páginas diferentes;
  - Interativa: interação do usuário com o navegador.
- Conteúdo
  - Estático: não é alterado na página, verifica-se se está correto e completo;
  - Dinâmico: altera a mesma página por meio de *scripts* e componentes.
- Comportamento
  - Segurança: mecanismos de controle de acesso;
  - Processos de execução: execução síncrona ou assíncrona entre cliente e servidor.

Um único modelo que represente todas essas divisões é impraticável, pois se torna impossível de entender devido ao tamanho que aquele pode atingir e o número de relações possíveis. Portanto, nesse trabalho será tratada a navegação estática e dinâmica e o conteúdo dinâmico.

Como já explanado, várias formas são utilizadas para representar aplicações Web, sendo a mais utilizada a máquina de estados, como FSM e *statechart* [21], pois fornecem formas convenientes de representar o comportamento de um *software*, evitando questões associadas com as implementações do código. Sendo assim, a máquina de estados foi escolhida, neste trabalho, para ser aplicada na modelagem das páginas de uma aplicação Web.

2.3 Modelos de Estado da UML

Antes de saber como modelar uma aplicação Web, é necessário entender o que é um modelo de estados, quais seus componentes e qual a sua estrutura, para se obter o maior proveito desse modelo na representação de uma aplicação Web.

O conceito de máquina de estados foi definido na área de matemática conhecida como autômatos finitos (máquina de estados finita) e era aplicado no projeto de circuitos. Em 1960, a máquina de estados começou a ser aplicada no projeto de *software* e depois de 1970 começou a ser uma rotina utilizada em engenharia de *software* [22].

Uma máquina de estados é um sistema em que as entradas atuais e as que já foram utilizadas determinam a saída. O efeito da entrada anterior é representado por um estado [22]. Um diagrama de transição de estados, também conhecido como modelo de estados ou diagrama de estados, é a representação gráfica da máquina de estados. Existem várias variantes dos modelos de estados, como os modelos clássicos de Mealy e Moore e os modelos para notações de orientação a objeto, como *statecharts* de Harel e Gary [23] e a adaptação da UML para esses modelos (modelo de estados da UML).

A máquina de estados da UML tem a proposta de suprir as limitações dos outros tipos de modelos de estados, como as máquinas de estados finitas, estendidas (MEFE) ou não (MEF). Com o modelo de estados da UML, há o conceito de hierarquia de estados aninhados, regiões ortogonais e aumento das definições das ações, as quais podem ser associadas a transições ou aos próprios estados. Entre as limitações das máquinas MEF e MEFE, está o fato de não ser possível mostrar concorrência entre eventos.

Como o modelo de estados da UML possui mais ferramentas e recursos para modelagem e é um dos mais utilizados pela literatura, foi escolhido nesse trabalho para a modelagem das aplicações Web.

Ao utilizar o modelo de estados da UML, é necessário entender o conceito de objeto para se poder representar o seu comportamento de forma correta. Sendo assim, em termos

de aplicação, existem dois tipos de objetos: aquele que sempre responde da mesma maneira a um evento, o qual se chama de independente de estado, e aquele que reage de maneiras diferentes a eventos, chamado de dependente do estado. São nos problemas complexos, que possuem muitos objetos dependentes de estado, que o diagrama de máquina de estados é útil, pois modela de forma eficiente essa complexidade [21].

O diagrama de máquina de estados da UML representa eventos e estados de um objeto, assim como seu comportamento em resposta a um evento. Dessa forma, são mostrados os eventos pelos quais o objeto sofre ação e representa as transições e estados em que o objeto se encontra entre esses eventos [16]. Elas podem expressar tanto o comportamento da parte de um sistema como também os protocolos de uso, tais como protocolos de comunicação (TCP/IP). O modelo de estado comportamental (*behavioral state machine*) especifica o comportamento dos elementos do sistema, sendo esse tipo de máquina utilizado neste trabalho por especificar o comportamento de vários elementos modeláveis de um sistema [24].

A documentação da OMG (OMG – *Object Management Group* - UML) [24] apresenta toda a explicação das notações do diagrama, assim como exemplos de como utilizá-las. Seguem algumas definições importantes:

- Estado: situação ou condição de um objeto em um determinado momento;
- Transição: relacionamento entre dois estados. Ao ocorrer um evento, um objeto muda de um estado para outro. Essa mudança é a transição;
- Evento: ocorrência relevante, ou seja, um dado de entrada que ativa a transição;
- Ação: resultado ou saída que segue um evento;
- Condição de guarda: expressão booleana que é avaliada quando a transição é ativada.

A Figura 2.3 representa uma máquina de estados do campo de *login* da Figura 2.2, na qual se coloca *e-mail* e senha. O objeto é a função do sistema para fazer o *login*: dependendo dos dados colocados para o e-mail e a senha, o *login* será efetuado ou não.

Outra forma de representar diagrama de estados é através da tabela de transição de estados. A tabela é uma outra forma de visualizar o diagrama, e ajuda a verificar se o modelo está correto. A Tabela 2.1 mostra essa tabela. A primeira coluna representa os estados atuais, ou seja, aquele em que se encontra a máquina; as outras colunas representam os próximos estados. As células são preenchidas com os eventos e ações possíveis.

Outra estrutura importante do modelo é a composição de estados. Um estado composto é aquele que possui um compartimento que contém outro diagrama [24]. A Figura 2.4 exemplifica esse caso de composição com ortogonalidade (estados concorrentes),

Figura 2.2: Interface do *login* de um sistema.

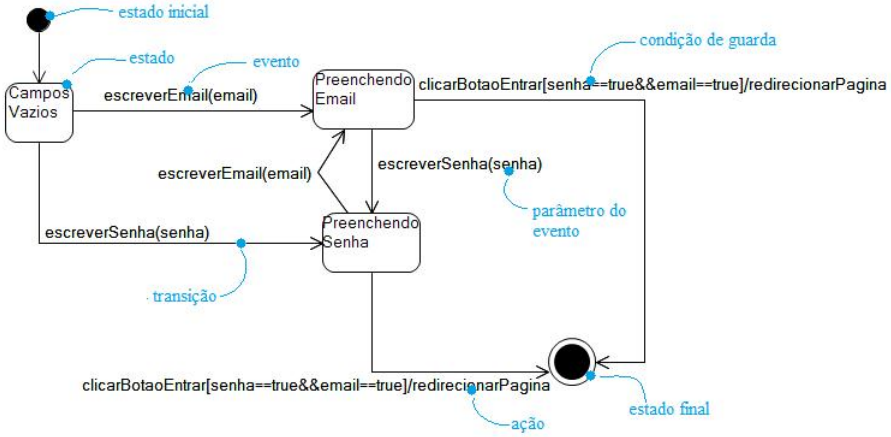


Figura 2.3: Exemplo de máquina de estados e seus componentes.

no caso de um estudante que precisa entregar dois trabalhos de laboratórios (com a condição do trabalho2 só poder ser entregue após o trabalho1) e necessita estudar para a prova final de uma determinada disciplina.

Sendo assim, com as possibilidades de ação e representação de concorrência entre estados, a máquina de estados da UML se mostra eficiente para a modelagem de aplicações Web, onde os estados da aplicação exigem que ações sejam feitas (como o fato de preencher um campo) e a concorrência, como mais de um frame exibindo atividades na página.

Tabela 2.1: Tabela de Transição de Estados do modelo da Figura 2.3.

Estado Atual	Próximo Estado			
	Campos Vazios	Preenchendo E-mail	Preenchendo Senha	Final
Campos Vazios	-	escreverEmail	escreverSenha	-
Preenchendo E-mail	-	-	escreverSenha	clicarBotaoEntrar
Preenchendo Senha	-	escreverEmail	-	clicarBotaoEntrar
Final	-	-	-	-

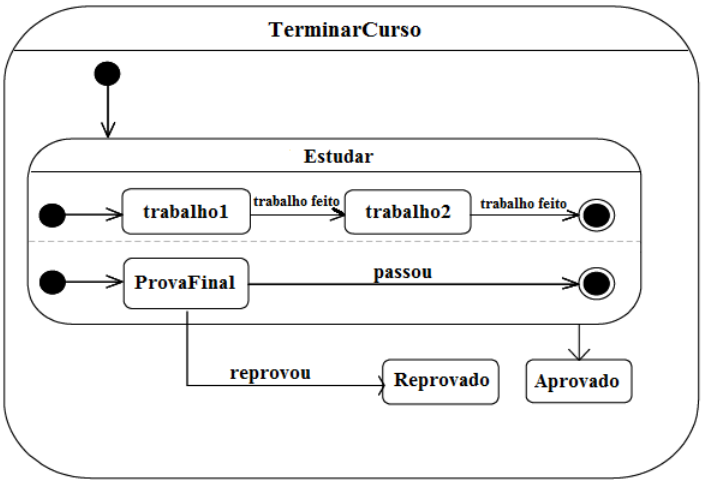


Figura 2.4: Exemplo de composição de estados.

Capítulo 3

Testes de *Software* e Testes Baseado em Modelos

Software com qualidade é uma exigência de qualquer cliente ou usuário. Dentre as atividades do ciclo de vida do desenvolvimento de *software*, está a de verificação e validação, cujo objetivo é garantir a qualidade do produto de acordo com os requisitos do cliente. Uma técnica muito utilizada para verificar e validar o *software* é o teste de *software*. Testes de *software* têm como objetivo executar o sistema para revelar defeitos, ou seja, verificar se o resultado obtido difere do resultado esperado. Neste capítulo, são introduzidos os conceitos básicos de verificação e validação de *software*, assim como os de testes de *software*.

3.1 Verificação e Validação de Software

Um *software* com qualidade é um requisito fundamental cobrado pelos clientes e usuários. Tanto isto é verdadeiro que o termo "qualidade de *software*" significa um conjunto de características que representam as necessidades do cliente que devem ser alcançadas no produto final [25]. Uma das formas de se obter a qualidade do *software* é pela aplicação das técnicas de verificação e validação. A verificação analisa se o *software* segue as especificações e a validação trata de garantir que o *software* atenda as necessidades do cliente [26].

Há duas formas de verificação do *software*: estática e dinâmica. Na primeira, entre os métodos existentes de verificação, há inspeções e revisões do *software*, nas quais se analisa a documentação de requisitos, constroem-se diagramas de *design* e representações do código fonte. Já na segunda forma, cuja verificação pode ocorrer por meio de testes, estes são gerados e executados com dados, as saídas são analisadas e verifica-se se o comportamento está como foi requisitado [26].

Nesse trabalho, aplicamos a segunda forma de verificação e validação: testes de *software*.

3.2 Testes de Software

Antes de entender o que é um teste de *software*, é importante definir os termos *falha*, *defeito* e *erro* [27]:

- Falha: a saída obtida é diferente da saída esperada;
- Defeito: processo ou passo que resulta em dados incorretos;
- Erro: resultado intermediário incorreto ou inesperado na execução do programa.

Sendo assim, entre as várias definições de testes de *software*, a escolhida neste trabalho é a de Binder [22] – em cujos processos de geração de testes este trabalho de mestrado foi baseado. Ela explica que teste de *software* é a execução do código fonte utilizando combinações de entradas e estado para identificar erros.

Os objetivos dos testes, segundo Pressman [12], de forma resumida, são executar o programa com a intenção de identificar erros, revelando a presença de defeitos, e o sucesso está, principalmente, em encontrar defeitos ainda não revelados, ocupando o menor tempo e esforço possíveis.

Os princípios que ajudam a projetar casos de teste eficientes [12] são:

- Todos os testes devem ser rastreáveis com os requisitos do cliente;
- Testes devem ser planejados antes das fases de testes começarem;
- O princípio do Pareto[1] é aplicável em testes de *software*;
- Testes devem começar de partes pequenas e progredir para as maiores;
- Não é possível testar *software* de forma exaustiva;
- Para ser mais eficiente, testes devem ser conduzidos de forma independente do desenvolvimento.

Para que os objetivos e princípios dos testes possam ser alcançados, existem fases e técnicas de teste que ajudam e facilitam o processo de verificação e validação.

[1] O princípio do Pareto implica que 80% de todos os erros descobertos durante os testes serão rastreados para 20% de todos os componentes do programa.

3.2.1 Fases de Teste

Segundo Pressman [12], as fases de teste podem ser vistas seguindo o modelo em espiral de desenvolvimento de *software*, como representa a figura 3.1.

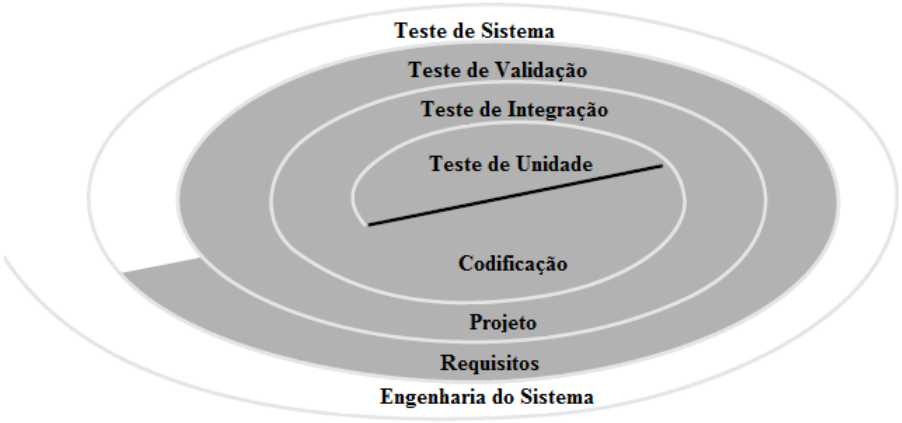


Figura 3.1: Fases de Teste [12].

Inicialmente, os testes de unidade se concentram no teste de cada unidade (componente) do *software* que já possui codificação. O próximo passo é gerar os testes de integração que se concentram no projeto e na construção da arquitetura do *software*. Nesse ponto, ocorrem também os testes de regressão, nos quais testes já criados são executados novamente, para assegurar que as alterações não causaram efeitos inesperados. Após a construção do sistema, começam os testes de validação, nos quais os requisitos estabelecidos como parte da análise são validados contra o *software* que foi construído. E, por último, testes de sistema, com os quais o software é testado como um todo, combinando-se vários elementos, como *hardware*, usuários, performance etc.

Para a geração dos casos de testes para as diferentes fases, diversas técnicas podem ser aplicadas, as quais serão explicadas brevemente a seguir.

3.2.2 Técnicas de Teste

Para criar um teste, no qual haja descrição do que será testado, com quais variáveis, entradas e saídas, é necessário um caso de teste. Sendo assim, um caso de teste é um conjunto de entradas, condições de execução e resultados esperados (saídas) criados para um objetivo particular, como exercitar um caminho específico de um programa ou verificar que um requisito está sendo cumprido [28]. Os casos de teste podem ser abstratos ou concretos.

O caso de teste abstrato (CTA) descreve a lógica do teste sem se vincular aos dados de entrada e saída, ou plataforma ou linguagem de programação [29]. Porém, para se

poder executar um teste, é necessário instanciar um CTA para torná-lo concreto para que, assim, possa ser executado no ambiente de teste.

Para se criar e executar os casos de testes, abstratos e concretos, deve-se decidir como o sistema será tratado, ou seja, se será considerado o código fonte do sistema, criando testes para exercitar essa codificação, ou se o sistema será tratado como um todo, sem se concentrar na implementação do código, mas sim verificando documentação e comportamento. Para cada tratamento do sistema, existem várias técnicas, entre elas testes caixa branca e testes caixa preta. Em testes de caixa branca, técnica também conhecida como testes estruturais, os testes são derivados a partir do conhecimento da estrutura e implementação do *software* [26]. O conjunto de testes deve assegurar que cada ramificação lógica do programa foi seguida, mostrando que cada parte foi executada pelo menos uma vez. De forma mais geral, há uma análise do código do sistema em teste. A compreensão do algoritmo de um componente ajuda a identificar outras ramificações lógicas e novos casos de teste [26].

Nas situações em que são validados os requisitos com o sistema, têm-se a técnica de testes de caixa preta. Nesse tipo de teste, a preocupação está em demonstrar se o sistema funciona ou não apropriadamente. Portanto, os casos de testes são criados a partir da especificação do sistema. O sistema é tratado como uma caixa preta para verificar o comportamento, por meio de determinadas entradas e saídas esperadas. Essa técnica também é chamada de testes funcionais, devido à concentração dos testes estarem na funcionalidade e não na implementação do código [26]. Como é padrão para se definir o teste, o analista de teste deve apresentar entradas e examinar as saídas que o sistema produz. Na situação em que ocorre uma saída inesperada, o teste detectou um problema do sistema. Para instanciar os testes caixa preta, deve-se escolher entradas que tenham alta probabilidade de gerar falhas. Existem diretrizes que podem ajudar em fazer essas escolhas, como as apresentadas por Sommerville [26]:

- Escolha entradas que forcem o sistema a gerar mensagens de erros;
- Projete entradas que sejam maiores que o tamanho de campos ou outros componentes de entrada, por exemplo, *overflow* de *buffers*;
- Repita a mesma entrada várias vezes;
- Force a produção de saídas inválidas;
- Force os resultados a serem muito grandes ou muito pequenos.

Porém, mesmo com essas diretrizes, muitas entradas podem ser geradas, tornando-se impossível utilizar todas nos testes. Para conseguir um conjunto de entradas para obter

o maior número possível de falhas, existem algumas técnicas que auxiliam a obtenção de um conjunto de entradas, por exemplo:

- Partição de equivalência [5]: O domínio de cada parâmetro de uma função é estruturado em classes de equivalência. Para os valores de cada classe, é assumido que a função os trata da mesma forma, sendo assim, apenas um representante de cada classe precisa ser testado;

- Análise de valores limites [5]: Para cada conjunto de entrada, escolhem-se os limites desse conjunto. Por exemplo, o número máximo de caracteres em um campo de texto;

- Testes de todos os pares (*Pairwise Testing*) [30]: As entradas são organizadas em lista ou tabelas, e combinam-se essas entradas ao pares para serem as entradas de um caso de teste. Ou seja, assume-se que há uma boa cobertura dos testes escolhendo-se um par de variáveis, em que uma variável é combinada com algum outra, para formar pares para cada estado.

Sommerville [26] também sugere o teste baseado em cenário para testar se o sistema encontra-se de acordo com os requisitos. Os cenários são criados e casos de teste são originados a partir deles. Para cada cenário, deve-se projetar um conjunto de testes que inclua entradas válidas e inválidas que produzirão saídas válidas e inválidas. No caso dos requisitos serem descritos por casos de uso, deve-se utilizá-los como cenários.

Portanto, os testes caixa preta verificam que as funções do *software* são operacionais, que as entradas são aceitas na devida forma e produzem saídas corretas. Verificam, também, que a integridade das informações externas é mantida.

Deve-se ressaltar que as duas técnicas apresentadas são complementares e não uma alternativa da outra. É também importante salientar que os testes de caixa preta são mais propícios para encontrar funções incorretas ou que estejam faltando, falhas de *interface*, defeitos nas estruturas de dados ou ao acesso as bases de dados, erros de comportamento ou desempenho e erros de inicialização e finalização.

Pressman [12] faz uma análise de possíveis formas de testar alguns tipos de arquitetura, ambientes e aplicações utilizando as técnicas caixa branca e caixa preta. Dentre eles, estão os testes para aplicações Web. O autor afirma que os princípios básicos de teste se assemelham aos de outros tipos de *software*. Sendo assim, ao se tratar de comportamento, a sugestão é justamente utilizar modelos do sistema que sejam possíveis de simular para examinar o comportamento do sistema em consequência de eventos.

3.3 Testes Baseados em Modelo

Como descrito, o objetivo dos testes é encontrar falhas, observar diferenças entre comportamentos de implementação e o que é esperado, baseando-se na especificação do sistema. Teste baseado em modelo é uma variante da técnica de teste caixa preta que utiliza modelos de comportamento do sistema para mostrar o comportamento esperado do sistema e do ambiente [31].

Os métodos convencionais de geração e aplicação de testes, como geração aleatória de testes e aplicação de testes de forma manual, têm encontrado algumas barreiras como, por exemplo, o sistema que está sendo testado apresentar suas funcionalidades alteradas constantemente, o que significa que os testes automatizados criados de forma manual devem ser adaptados para os novos comportamentos, o que pode ser um processo custoso. A geração dos testes manuais produz um número fixo de casos de teste que podem detectar apenas alguns tipos de erros, tornando-se inúteis à medida que os defeitos são encontrados e corrigidos; a aplicação de entradas aleatórias torna difícil o controle e a organização da sequência de dados, podendo diminuir a cobertura dos testes (medição da proporção do sistema que foi exercitado pelos casos de testes, verificando o que falta para ser verificado do sistema) [32]. Testes baseados em modelo de comportamento resolvem esse problema, pois, como já visto, o modelo descreve o comportamento do sistema. Ao percorrer esse modelo, é possível criar casos de teste.

Sendo assim, os testes baseados em modelo são mais eficientes ao permitirem adaptação rápida, caso existam alterações no sistema. Ou seja, conforme o *software* evolui, apenas é necessário atualizar o modelo para se obter os novos testes em concordância com a atualização do sistema.

Segundo Utting, Pretschner e Legeard [31], o modelo que representa o sistema em teste deve ser validado, o que implica que o modelo deve ser tão simples quanto o sistema, fácil para se verificar, modificar e manter. O modelo também deve ser preciso o suficiente para gerar casos de teste eficientes, ou seja, aqueles que podem detectar algum defeito no sistema.

Como os casos de testes são criados a partir do modelo, e conforme o critério escolhido para a geração, os testes sempre terão relevância, não se tornando inúteis conforme os defeitos forem encontrados e corrigidos, pois as entradas para cada teste seguem um critério que pode ser alterado conforme o sistema for sendo modificado.

Outra vantagem é que o modelo pode ser criado no início do ciclo de desenvolvimento do *software*, o que pode mostrar inconsistências nas especificações e impedir defeitos no código [32]. Assim também, o mesmo modelo pode gerar vários casos de teste para as diferentes fases de teste, como os testes de regressão [32].

3.3.1 Níveis de Abstração dos Casos de Teste

Dentro da técnica caixa preta, existem vários métodos de geração de testes que podem gerar tanto CTAs como testes executáveis. A seguir, as definições dos níveis de abstração dos casos de teste:

- Casos de teste abstratos: como descrito na seção Técnicas de Teste, o CTA é aquele em que há apenas descrição da lógica a ser seguida no sistema, sem definição de dados de entrada e independentes da plataforma de execução;
- Casos de testes instanciados: a partir dos CTAs, insere-se dados neles, tornando-os instanciados, para, depois, serem executáveis;
- Casos de teste executáveis: são os testes que possuem dados de entrada e estão no formato específico da plataforma. Assim, o teste está pronto para ser executado de forma automática ou manual.

Em testes baseados em modelo, é comum um teste executável ser originado a partir de um caso de teste abstrato. Inclusive, esse processo já possui ferramentas para que seja automatizado, como em Almeida [3].

3.3.2 Processos dos Testes Baseados em Modelo

Para a geração de testes a partir de modelos, devem-se seguir os seguintes passos (processo baseado no trabalho de Rosaria e Robinson [32]):

- Explorar o sistema que será testado e encontrar e definir o domínio de entradas do sistema

  Nessa etapa, o sistema deve ser avaliado e verificado quais partes serão testadas e quais os aspectos serão considerados no modelo. Como o problema é testar o comportamento do sistema, os aspectos a serem considerados são aqueles que alteram esse comportamento. Esses aspectos, que podem ser chamados de estados, são alterados por possíveis entradas do sistema, que, após serem definidas, permitem a criação do domínio de entradas que serão representadas no modelo.

- Desenvolver o modelo

  Antes de desenvolver o modelo, primeiro é necessário definir o tipo que será utilizado no processo. Dentre os modelos de comportamento, existem, por exemplo, os diagramas de sequência e os diagramas de estado. O modelo de estados permite representar como o sistema irá se comportar com uma determinada entrada em diferentes circunstâncias.

A partir dos estados e entradas verificados na etapa anterior, constrói-se o modelo de estados que representa o sistema.

- Gerar os casos de teste percorrendo o modelo (geração de CTAs)

  Levando-se em consideração que o modelo de estados é abstrato, ou seja, está no mesmo nível das especificações do sistema, os casos de testes gerados por esse modelo também são abstratos, por estarem no mesmo nível de abstração do modelo [3].

  Sendo assim, para criar os casos de testes abstratos, percorre-se o modelo por meio de um critério, pois, assim, há uma orientação de como seguir possíveis caminhos do modelo e de como criar um caso de teste. Vários são os critérios, e cada ferramenta ou método segue um deles, como o critério de cobertura de todos os estados (cada estado deve ser percorrido pelo menos uma vez), cobertura de todas as transições (cada transição deve ser percorrida pelo menos uma vez) [22], etc. Todavia, como o interesse é economizar tempo, esforço e erros humanos, existem muitas ferramentas que percorrem os caminhos de forma automática, nos quais a entrada da ferramenta é o modelo e a saída, os casos de teste. Algumas dessas ferramentas são discutidas nos capítulos de Metodologia e de Trabalhos Relacionados.

- Instanciar os casos de teste e executá-los.

  Após ter os casos de teste abstratos criados a partir do modelo, os testes são instanciados com dados, para criar os testes instanciados. Para instanciar os testes, pode-se utilizar algum método de geração de entrada, como os já citados na subseção Técnicas de Teste desse capítulo, como por exemplo, a análise de valores limites.

  Depois de todos os testes estarem instanciados, são criados os testes executáveis. Os testes executáveis podem ser aplicados no sistema tanto manualmente como automaticamente.

Capítulo 4

Trabalhos Relacionados

Muitos são os trabalhos que tratam da modelagem de aplicações Web para as fases de desenvolvimento do *software*, como *design*, testes e implementação, assim como há aqueles que mostram testes baseados em modelos. Sendo assim, nesta pesquisa, foram levados em consideração os trabalhos que fazem uso de máquina de estados para modelagem – independente do tipo da máquina de estados, aqueles que tratam de testes baseados em modelos utilizando máquina de estados e os processos de validação do comportamento de modelo. Porém, para que um trabalho da área seja considerado, principalmente entre os que tratam de testes baseado em modelos, deve conter a análise de atributos dinâmicos de uma aplicação Web.

As ferramentas que geram modelo de estados, que geram testes e/ou geram modelo e testes não serão descritas, devido ao fato de ser impraticável utilizar todas as ferramentas existentes do mercado para ser possível avaliá-las adequadamente. Outro motivo que gerou a exclusão de tais ferramentas nesta pesquisa é que a maioria delas necessita de licença para poder ser utilizada.

4.1 Modelagem Aplicações Web

Um dos trabalhos mais referenciados por vários artigos e que ressalta o uso da UML é o de Conallen [33]. O autor considera uma aplicação Web um sistema que possui servidor, rede, protocolo HTTP e *browser*. Dessa forma, uma aplicação Web é um sistema de *software*, no qual a entrada do usuário produz efeito no estado das atividades da aplicação. O foco da modelagem encontra-se no fato da aplicação implementar lógica das atividades, e seu uso modificar os estados das mesmas. O objetivo de Conallen foi mostrar possíveis soluções para modelar aplicações Web, focando em componentes significativos da arquitetura da aplicação e como modelá-los. Este trabalho ajuda a entender o que é importante na modelagem, ou seja, o que é relevante ou não na aplicação, além de explicar

algumas formas de como modelar certos objetos quando a UML não possui uma forma de representá-los, por exemplo, utilizando o identificador *stereotype* que a UML fornece. Segundo o autor, primeiro é necessário saber qual o nível de abstração da modelagem a que se quer chegar, ou seja, o nível de detalhes que um modelo precisa ter para trazer benefício a quem for usá-lo. Após a definição do nível, é feito o mapeamento destes artefatos levantados (o que se vai modelar) para os elementos do modelo. Há uma limitação no trabalho, pois o autor apenas exemplifica modelagem utilizando diagrama de classes e não outros tipos de modelo da UML. Porém, a abordagem do estudo é muito interessante quanto à análise da interface. O trabalho de Conallen é concluído com a apresentação de uma forma coerente e completa de integrar a modelagem de elementos específicos da Web com o resto da aplicação, como o nível de detalhe e abstração apropriados para cada utilidade do modelo (para *designers*, implementadores etc.).

Uma abordagem mais específica de modelagem, utilizando *statecharts*, é a de Winckler e Palanque [10], cujo foco é modelar aplicações Web para a fase de *design*. A motivação dos autores nesta pesquisa está no fato das ferramentas existentes e seus métodos oferecerem uma forma muito superficial de representar uma navegação da aplicação e darem suporte a somente uma parte da informação. Há outros métodos existentes, que são tão complexos que não oferecem todos os aspectos necessários para um *design* de uma aplicação. Portanto, para solucionar o problema da modelagem, apresenta-se o *StateWebCharts*, o qual é uma extensão de *statecharts* com novas notações para incluir requisitos do *design* de navegação Web. Para os autores, com o *statechart* não é possível, por exemplo, representar quem (o usuário ou o sistema) é o alvo de um evento recebido pela aplicação.

O trabalho considera os seguintes requisitos como os mais importantes para modelar aplicações Web: modelagem de controle de diálogo (mostrar se é o usuário ou o sistema que despacha eventos sobre a interface); fronteiras do *design* (mostra quais partes da aplicação pertencem realmente a ela e quais não pertencem, como *links* externos); *design* das atividades cliente-servidor (quais partes da aplicação pertencem ao cliente e quais pertencem ao servidor); além de partes de acesso direto do usuário. O *StateWebChart* é composto por estados e eventos. Cada página Web da aplicação é considerada um conjunto de objetos, o qual é associado a um estado. *Links* e objetos interativos que causam transições são representados por eventos. O trabalho mostra as vantagens em se usar este tipo de modelagem, como a distinção clara entre páginas internas e externas da aplicação e o fato de apenas neste tipo de modelagem se conseguir mostrar as funcionalidades de cada lado da aplicação (cliente e servidor).

Outros artigos colaboram para se compreender as formas possíveis de se modelar aplicações Web, como é o caso do trabalho de Leung *et al.* [7], no qual há a análise da complexidade de navegação e do dinamismo de sites Web a partir de *statecharts*. Os autores defendem que com a evolução da *World Wide Web* (WWW), o mapeamento direto

entre elementos apenas por meio de hipertextos (*links*) não é mais possível nas aplicações Web que possuem dinamismo com vários *scripts* e programas. Ou seja, na Web 1.0, em que existiam apenas *links* de uma página a outra, um modelo era fácil de ser criado, porém, com as tecnologias e linguagens que produzem conteúdo dinâmico, isso não é mais possível. Há uma análise de Winckler sobre outros trabalhos relacionados e suas limitações como justificativa para os autores utilizarem sua própria modelagem. O artigo mostra uma análise da navegação Web, explicando o que é uma página, um *hiperlink*, os efeitos de um *browser* e tipos de navegação – como por *frames* e por janelas. Na parte da modelagem, os autores a dividem em duas etapas: na primeira é definido um escopo do sistema e depois é feita a modelagem, na qual, para cada tipo de navegação, é mostrado como esta é feita.

Para definir o escopo do sistema, primeiro é identificado o conjunto de páginas Web de interesse, limitando-o por qualquer página Web que pertença a esse grupo ou páginas diretamente relacionadas por uma página Web do conjunto. Porém, não há um critério definido para essa escolha.

Na parte da modelagem, primeiro é modelada apenas uma página Web. *Hiperlinks* podem ser definidos como seções diferentes para a mesma página. A página é modelada como uma composição de estados e qualquer posição da tela possível é modelada como um subestado da página. Na navegação entre várias páginas distintas, por meio de *hiperlinks*, a abstração é feita individualmente nas páginas. Cada *hiperlink* de uma página para a outra é uma transição de estados. A ativação do hiperlink é um evento que faz ocorrer a transição. Já na navegação baseada em frames, na qual os *frames* dentro de um navegador possibilitam visualização concorrente de várias páginas Web, a sincronização entre *frames* pode ser feita por *scripts* e programas do lado cliente da aplicação. Esta sincronização é modelada por eventos de transmissão do *statechart*. Para o caso de várias janelas, é modelado um *statechart* para cada. A sincronização entre janelas pode ser feita com o mesmo mecanismo usado para os *frames*. Finalmente, o artigo faz referência ao conteúdo dinâmico, àquele em que há interação entre um lado cliente e um lado servidor da aplicação. Os autores consideram esse tipo de modelagem difícil, pois nem todos os estados são conhecidos previamente e o conjunto de possíveis estados pode ser muito grande. A solução é a criação de um modelo para o lado do servidor e outro para o lado cliente.

Mesmo que o artigo de Leung não tenha mostrado uma validação desse modelo e utilizado *statecharts*, as ideias deste trabalho ajudam a pensar em como modelar navegação utilizando *statecharts* e páginas concorrentes que utilizam vários *frames*.

Alguns trabalhos delimitam o que modelar a partir dos *logs* criados pelas aplicações das funcionalidades mais utilizadas pelos usuários, como no caso do trabalho de Mesbah, Deursen e Lenselink [34], o qual descreve uma técnica de *crawling* para aplicações que

utilizam Ajax, analisando as alterações dinâmicas possíveis dos estados da interface do usuário em navegadores Web. Os autores construíram um algoritmo que escanea a árvore do documento DOM , marca os elementos candidatos que são capazes de alterar estados, aciona eventos a esses elementos e constrói uma máquina de estados (na realidade, um grafo) que modela os caminhos de navegação e estados da aplicação Ajax. Para utilização do código apresentado, é necessário ter o código Javascript e o documento DOM da aplicação, portanto, como é utilizado o código fonte, a técnica caracteriza-se como teste caixa-branca. Ao se fazer uso do *crawler*, há apenas a modelagem da parte dinâmica (diferentes *links* que podem ser criados mesmo sem acesso ao servidor) da aplicação, não havendo verificação da parte estática. Outro problema, é que a ferramenta não trata explosão de estados, sendo esta a indicação dos autores para futuros trabalhos.

4.2 Testes Baseados em Modelo

Testes baseados em modelos têm sido uma técnica muito utilizada tanto na área industrial como na acadêmica. Esta seção apresenta aqueles que mostram a modelagem Web com modelos de comportamento e que considera aspectos dinâmicos, gerando os testes originados a partir desses modelos.

Um dos trabalhos que mais reforçam a relevância das limitações das propostas estudadas é o de Alalfi, Cordy e Dean [20], o qual analisa 24 métodos de modelagem e testes baseado em modelos. Segundo os autores, a maioria das propostas verifica o conteúdo estático das aplicações Web e são poucas as que verificam conteúdo dinâmico em termos de correção e completude.

Vários são os trabalhos que utilizam UML e grafos na representação do nível navegacional. Apesar dos autores não sugerirem ou proporem algum tipo de modelo, afirmam que a UML, apesar de ser um padrão de modelagem, pode não ser a melhor escolha para testar e verificar aplicações, pois, muitas vezes, não há como validar os modelos UML. Os autores também ressaltam a modelagem por grafos e afirmam que esse tipo de modelo é mais fácil de produzir e de testar se está ou não correto. Outro ponto levantado pelos autores é que a maior parte das propostas não é validada, ou seja, as propostas não são aplicadas em sistemas reais, ou então os modelos não são avaliados quanto à correção ou se realmente representam o sistema modelado.

A conclusão é a necessidade de um modelo capaz de capturar todas as características de uma aplicação Web em todos os níveis possíveis (estáticos, dinâmicos, com conteúdo e navegação), com uma forma fácil e eficaz de validar o modelo.

A pesquisa que mais influencia este trabalho é a de Marcheto, Ricca e Tonella [35]. Neste estudo, os autores ressaltam o aumento do uso de Ajax nas aplicações e como complica o teste das aplicações Web. Eles fazem uma comparação dos métodos de testes

existentes e ressaltam qual seria o melhor para aplicações dinâmicas. Os métodos são: testes baseados em modelos navegacionais – os quais mostram apenas a navegação em si, ou seja, não mostram interações do sistema entre cliente e servidor, nem o dinamismo de uma página; métodos de cobertura – os quais analisam o código da linguagem, como, por exemplo, Java ou PHP. Este teste, porém, não cobre componentes dinâmicos como o Ajax; e, finalmente, testes caixa preta – que não apresentam nenhum tipo de técnica, apenas consideram a aplicação já pronta. Testes caixa preta são eficazes para testar requisitos e especificações, mas não garantem cobertura completa da estrutura e do comportamento do sistema.

Os autores propõem uma técnica baseada em modelo, na qual se constrói uma Máquina Finita de Estados (FSM, do inglês *Finite-State Machine*), com destaque aos estados dos componentes do lado cliente da aplicação Ajax. Aqui se extraem os caminhos de acordo com um critério de cobertura, geram-se os testes abstratos e, assim, entradas e saídas esperadas são adicionadas, criando testes executáveis. No final, os testes são executados no formato de *scripts*.

A FSM é construída pela análise de possíveis instâncias do modelo DOM, cujos elementos HTML ressaltados são os estados da máquina de estado finita. Também são analisados os eventos do usuário e as mensagens do servidor que alterem o DOM, formando as transições. Então, para cada página da aplicação Web, é construída uma FSM.

Após a modelagem, ferramentas criadas pelos próprios autores são aplicadas para criar os casos de teste. Após aplicar os testes, os autores utilizam injeção de falhas para comparar seu trabalho com outras técnicas existentes e coletam métricas para analisar os resultados. Os autores chegaram à conclusão que sua técnica é válida, porém, na medida do possível, todas as técnicas analisadas, em princípio, deveriam ser utilizadas em conjunto, pois uma complementa a outra nos resultados.

No trabalho de Kung, Liu e Hsia [1], há a explicação de como se testar praticamente todas as partes de uma aplicação Web, desde as requisições HTTP até o fluxo navegacional da aplicação. As partes testadas são: a) o aspecto de objeto, o qual modela as entidades da aplicação como objetos e descreve suas relações de dependência; b) o aspecto comportamental, que descreve a navegação e os comportamentos dos estados dependentes; e c) o aspecto estrutural, que descreve o controle e o fluxo de dados de informação da aplicação.

Os autores realizam um trabalho bem interessante ao mostrar como testar manualmente os três aspectos citados acima. Porém, nenhum dos modelos das partes foi validado. Os autores citam um novo tipo de modelagem para a parte comportamental, chamada *Object State Diagram*, que dizem ser similar a um *statechart*, porém não mencionam quais as diferenças e nem explicam como modelar. Apenas apresentam figuras demonstrativas do estudo realizado.

Alguns trabalhos fazem uso da combinação de modelos para obter um resultado melhor

da semântica da aplicação, como é o caso do estudo realizado por Li, Zhongsheng e He [36], no qual o objetivo é combinar modelo da UML com FSM, para gerar casos de teste automaticamente. Dessa forma, os autores escolheram o diagrama de sequência da UML, pois é possível selecionar as partes de comportamento que se quer testar da aplicação. Já o FSM serve para modelar a especificação da aplicação. Uma combinação gera casos de testes e os valida com a especificação. O trabalho trata a explosão de estados com a construção do *on-the-fly test model* (técnica na qual os testes derivados de um modelo e execução de testes são combinados em um único algoritmo). O modelo UML é transformado em modelo FSM.

Um problema se constitui do fato dos modelos serem feitos conforme as especificações escolhidas pelo usuário, excluindo parte da aplicação a ser validada. A justificativa para isso é a diminuição do número de casos de testes do sistema. Sabe-se que o tempo é curto no desenvolvimento de aplicações, por isso selecionam-se as partes mais importantes para serem testadas primeiro, mas não se descartam as outras partes do sistema. Os diagramas de sequência são utilizados para representar aspectos importantes de interações entre objetos e podem ser utilizados para definir objetivos de teste. Depois, o diagrama de sequência é convertido em FSM para gerar testes automaticamente. Um *on-the-flye test model* é construído e o algoritmo de busca em profundidade é aplicado para gerar os casos de teste.

A maior parte dos artigos estudados mostra como os autores delimitam o que será modelado. No trabalho de Chen *et al.* [37], considera-se a modelagem que leva em conta a interação do *browser*. Ou seja, o objetivo é modelar e gerar testes de aplicações Web, levando em conta o lado dinâmico da aplicação, em termos de fluxo de navegação (entre páginas) e a interação do *browser* utilizado com a aplicação.

O artigo de Chen mostra que primeiro cria-se um grafo de transições de estados para modelar a navegação do usuário dentro da aplicação. Nessa etapa ainda não é examinada a intervenção do *browser* na aplicação, apenas é considerado o fluxo que o usuário pode fazer.

Em seguida, o trabalho exemplifica as possíveis combinações dos botões "Página Anterior" e "Próxima Página" do *browser*, considerando se estão habilitados e desabilitados, criando uma sequência para modelar cada página e comportamento do lado cliente. Nessa sequência, são representadas a "Página Anterior", o comportamento do browser (combinações dos botões de exemplo), a "Página Atual" e a "Próxima Página". Dessa forma, cria-se outro modelo com base no anterior, mas que, considera as interações do *browser* esclarecidas pelas sequências geradas.

Um dos problemas evidenciados no trabalho é o tamanho que os modelos podem atingir. No caso, o autor apenas considerou dois botões do *browser*, porém ressalta que outros devem ser considerados, como o botão de histórico, página favorita etc. Outro

problema está no foco da modelagem estar apenas na navegação entre páginas e não nas alterações possíveis na própria página. E, por último, em nenhum dos modelos, ou preparação dos mesmos, ocorre algum tipo de validação da semântica, considerando-os corretos e válidos para a geração dos testes.

Alguns trabalhos utilizam máquina de estados abstrata. O estudo de Bolis *et al.* [38] faz uso de máquina de estados abstrata para modelar a aplicação. Os autores mencionam um conjunto de ferramentas chamado ASMETA, de suporte para desenvolvedores de máquina de estados abstratas, o qual serve para desenvolver especificações, verificar modelos (*model-cheking*), validar com revisão e simulação e, também, como ferramenta para geração de testes, no caso, ATGT (*ASM Tests Generation*). O trabalho faz uso apenas da ferramenta ATGT, não mencionando se fez uso ou não das ferramentas de validação dos modelos.

Ainda sobre a pesquisa de Bolis, para cada página da aplicação, é construída uma máquina de estados, porém os autores não deixaram claro o que é cada estado de uma página. A única explicação clara é a de eventos: o acionamento de botões, *links*, campos de textos, *checkboxs*, botões de rádio e listas de seleção, assim, entende-se a modelagem da parte dinâmica da aplicação. Para conter a explosão de estados, é feita uma hierarquia de estados, em que uma máquina geral é construída, na qual cada estado é uma página e cada *link* de navegação de uma página para outra é uma transição. Após a criação dos modelos, a ATGT é utilizada para gerar um conjunto de testes de acordo com vários critérios de cobertura. Esses testes são abstratos, sendo utilizado o módulo A2C (tradutor de uma sequência abstrata de teste em um *script* concreto em uma linguagem de *script* selecionada) junto com uma ferramenta chamada Sahi, para criar *scripts* com dados e, dessa forma, aplicar os testes no sistema.

Apesar da grande semelhança do trabalho de Bolis com a presente pesquisa, aquele não possui uma forma para se saber como realmente modelar cada página da aplicação. As seguintes perguntas podem ser feitas: "o que são estados, afinal?" e "quando as transições são disparadas?". Não há uso de condições de guarda, portanto, não se sabe se há criação de caminhos infactíveis nos testes.

Um trabalho muito citado por vários autores é o de Andrews [21]. A proposta é uma técnica, chamada FSMWeb, de geração de testes baseada em modelo de máquina de estado finita. O trabalho realça o problema de explosão de estados e, para lidar com a situação, cria uma hierarquia de máquinas de estados. O nível mais alto dessa hierarquia é a representação da aplicação como um todo. O outro nível são as partes lógicas da aplicação, que os autores denominam como *clusters*. A modelagem considera tanto a parte estática como dinâmica da aplicação.

A geração de testes é feita de duas maneiras. A primeira, para cada FSM, considera suas transições e gera sequências de caminhos, considerando valores de entrada para cada

FSM, ou seja, valores que não relacionem uma FSM com alguma outra. Na segunda forma, são consideradas sequências de agregações de FSMs parciais, considerando que, numa hierarquia que relaciona FSMs, existe uma transição de uma FSM para outra. Ambas as formas de sequências de testes são obtidas por algoritmos de busca em grafos, como aqueles que verificam todos os nós ou todas as arestas.

Os autores alimentam os testes com uma seleção aleatória de possíveis entradas, porém dizem que é possível utilizar permutações das sequências das possíveis entradas ou valores de partição. E depois, com uma ferramenta, geram e executam os *scripts* de teste.

Uma das limitações descrita pelos autores é o da interação do *browser* com a aplicação. Eles afirmam que esta irá aumentar significantemente o número de estados e transições e deve ser um problema tratado de outra forma. Apesar de relatarem ser possível ocorrer sequência de testes infactíveis devido a um modelo incorreto, não tratam da validação do modelo em nenhuma das etapas da técnica.

Como continuação do trabalho anterior, Andrews *et al* [11] apresenta uma ferramenta que constrói máquinas de estados (FSM) para partes da aplicação Web e gera testes a partir dessas máquinas. Na análise dos trabalhos relacionados do artigo, os autores também afirmam, assim como neste trabalho, que a maioria das abordagens não é utilizada em aplicações reais e/ou não foi validada. Andrews, além de explicar o funcionamento da ferramenta, a utiliza em aplicações reais. Porém, como já no trabalho anteriormente analisado, não valida os modelos para a geração de testes.

A ferramenta, como já mencionado, funciona em duas fases. Na primeira fase, há quatro etapas. Primeiro, particiona-se a aplicação em *clusters*, depois define-se as partes lógicas do sistema e, assim, na terceira etapa constrói-se uma FSM para cada *cluster*. Na quarta etapa, constrói-se uma FSM que representa a aplicação inteira, ou seja, a interação entre os *clusters*. Na segunda fase, gera-se os testes para cada FSM construída.

O trabalho faz, ainda, uma análise dos testes gerados com diferentes tipos de entrada de dados e com uma aplicação real de uma universidade.

4.3 Validação dos Modelos

Várias foram as formas pensadas e analisadas para validar semanticamente o modelo da aplicação.

Uma das primeiras ideias era utilizar o modelo executável da UML (xUML) [39] para testar o modelo feito da aplicação. O xUML é um subconjunto da UML 2.0 que permite modelar o sistema em termos semânticos e pode ser testado conforme construído, de forma a traduzir o modelo automaticamente em código fonte. Autores o consideram como a solução do problema entre o desenvolvimento e as atividades de modelagem.

A OMG (*Object Management Group*) [40] formalizou o modelo executável da UML,

nomeando-o de fUML, porém, na própria descrição da formalização, há comentários da necessidade de se obter uma ferramenta que consiga executar o fUML.

Apesar das vantagens do xUML, a ideia foi descartada devido ao passo da transformação e tradução do modelo, no qual são necessárias anotações sobre aspectos de arquitetura e um compilador que assimile aspectos de distribuição e desempenho para gerar o código da aplicação. Como não é o foco deste trabalho desenvolver a aplicação final, e sim apenas um modelo válido, foi decidido o uso de uma ferramenta que seja mais fácil apenas para validação, e que, também, não seja necessário o passo de transformar o modelo de estados da UML para um modelo executável. Outro problema nessa abordagem, é a dificuldade em encontrar o compilador que execute o xUML. Os compiladores do próprio criador [39] são pagos, e os que existem de forma gratuita são difíceis de ser utilizados.

Outra forma pensada para validar o modelo foi a simulação, a qual foi escolhida para o presente trabalho. Após testes de ferramentas de simulação, a escolhida foi a Topcased. Uma das ferramentas testadas sem êxito foi a Kieler [41], a qual tem como proposta o *design* baseado em modelo de sistemas complexos. A ferramenta permite editar, navegar e simular os modelos. Apesar de ser uma ferramenta gratuita e com propostas ambiciosas, não foi possível utilizar a simulação, pois várias exceções são geradas, travando a ferramenta. Talvez no futuro, nas novas versões, isso seja corrigido. Um *e-mail* foi enviado ao time, porém a resposta obtida foi de que a parte da simulação está em segundo plano, sendo que há planos de que, no futuro, seja implementada de forma correta. Outra ferramenta testada e rejeitada foi a Xholon [42], também gratuita. Trata-se de um editor, simulador e transformador de modelo em código. Apesar da proposta e dos exemplos que a acompanham, não é trivial de ser utilizada e possui pouca documentação.

Uma ferramenta que se constatou ser muito boa, porém paga, é a MagicDraw [43], ferramenta *CASE* com suporte à colaboração de times, que facilita a análise e *design* de sistemas orientados a objetos e banco de dados. A ferramenta permite editar e validar modelos (quanto à correção e plenitude) e transformação de códigos. Para simulação de modelos, como proposta de validar a modelagem e a semântica do sistema, há uma ferramenta em conjunto, chamada Cameo Simulation, cuja nova versão já está de acordo com a UML executável da OMG.

4.4 Resumo dos Trabalhos

De forma resumida, a Tabela 4.1 traz os trabalhos descritos nesse capítulo. A tabela mostra o modelo utilizado no trabalho, se trata-se de uma nova proposta de modelo ou não, se houve algum tipo de validação no modelo, se foram considerados elementos dinâmicos e aspectos navegacionais da aplicação, se há o cuidado com a explosão de

estados e se há ou não geração de testes a partir do modelo.

Tabela 4.1: Resumo dos Trabalhos Relacionados.

Trabalho	Modelo	Propõe Novo Modelo	Valida o Modelo	Trata Elementos Dinâmicos	Modela Navegação	Trata Explosão de Estados	Gera Testes
Conallen [33] 1999	Diagrama de Classes da UML	-	-	X	X	-	-
Winckler [10] 2003	StateWebCharts	X	-	X	X	-	-
Leung [7] 2000	Statecharts	-	-	-	X	-	-
Mesbah [34] 2012	Grafo	-	-	X	X	-	-
Marchetto [35] 2008	FSM	-	-	X	-	-	X
Kung [1] 2000	Object State Diagram	X	-	X	X	-	X
Li [36] 2009	Diagrama de Sequência da UML e FSM	-	-	X	X	X	X
Chen [37] 2007	Grafo	-	-	X	X	-	X
Bolis [38] 2012	Máquina de Estados Abstratas	-	X	X	X	X	X
Andrews [21] 2005	FSMWeb	X	X	X	X	X	X
Este trabalho	Máquina de Estados da UML	-	X	X	X	-	X

Capítulo 5

Metodologia: da modelagem à geração dos testes

O objetivo deste capítulo é detalhar as etapas de modelagem e geração de testes a partir de um modelo de estados. Sendo assim, nesse capítulo serão analisadas as etapas da metodologia criada para que esse objetivo seja alcançado. Ao final, espera-se conseguir uma alternativa para o problema, ressaltado por Robinson [2], sobre a complexidade da modelagem no processo de testes baseado em modelos.

A automação de todas as partes possíveis do processo é defendida nesse trabalho, porém especificamente neste capítulo não serão envolvidas ferramentas, sendo estas descritas no capítulo Estudo de Caso.

5.1 Metodologia na forma geral

Para iniciar a modelagem de uma aplicação Web, são necessários a documentação de casos de uso e a aplicação Web ou as interfaces das páginas do sistema. Neste trabalho, orienta-se o analista de teste em como testar aquilo que já está produzido em questões de interface, no caso, cada página do seu sistema. Os casos de uso podem indicar quais partes do sistema são as mais importantes e também, futuramente, serem artefatos para saber a cobertura dos testes, ou seja, se os testes cobrem de forma completa os casos de uso. A metodologia, como mostrada pela Figura 5.1, é dividida em sete passos para modelagem, geração de testes, execução destes na aplicação Web e análise dos resultados.

O primeiro passo para iniciar a modelagem é saber quais páginas do sistema serão modeladas. Se uma empresa, por exemplo, segue de forma correta os preceitos de desenvolvimento de *software* defendidos pela engenharia de *software*, todas as etapas do ciclo de vida do *software* possuirão testes, e isto resulta num programa totalmente testado, isto é, cada etapa do desenvolvimento corresponde a uma etapa de preparação dos testes,

para que resulte, então, em testes para diferentes partes do sistema, antes de testá-lo como um todo. Porém, nas situações em que o processo de testes começou tardiamente no ciclo de desenvolvimento do *software*, parece sensato começar os testes pelas partes mais importantes e críticos do sistema, ou seja, nas partes essenciais de funcionamento. Os casos de uso foram utilizados para selecionar essas partes. Portanto, selecionam-se os casos de uso mais importantes do sistema. Um caso de uso pode referenciar uma ou mais páginas da aplicação Web. Assim, os casos de uso devem ser analisados para se descobrir quais as páginas a que cada caso de uso faz referência. Portanto, obtém-se uma lista por ordem de importância, sendo a primeira página da aplicação Web a mais importante e a última a menos importante. Aqui, percebe-se que se cria uma ordem de importância dentro dos casos de uso mais importantes, como uma estratégia de primeiro se testar o que é mais crítico e, conforme o tempo do projeto permitir, testar-se o resto.

A partir da ordem da lista, para cada página nesse segundo passo, faz-se uma análise para se descobrir quais são os estados e quais são as transições dessa página, ou seja, a análise é realizada para se extrair o modelo de estados. Os modelos, normalmente, são editados utilizando uma ferramenta *CASE*.

Após construir os modelos, vem a verificação e validação dos mesmos, sendo este o terceiro passo. Assim, o analista de teste consegue ter certeza de que o modelo foi criado de forma correta e que representa, realmente, a parte do sistema que se está validando, ou seja, o comportamento daquele componente.

Com o modelo pronto e validado, inicia-se outra parte, que é a de criação dos casos de testes.

No quarto passo do método, tem-se a criação dos casos de testes abstratos. O modelo deve ser percorrido e, por meio de critérios de geração de testes, como cobertura de todos os estados ou cobertura de todas as transições – como explicado no capítulo 3, obtêm-se os casos testes abstratos. Nesse passo, a execução pode ser feita tanto de forma manual como automática.

O quinto passo é a instanciação dos casos de testes abstratos e a criação dos casos de testes executáveis. Primeiro, os casos de testes são instanciados com dados. Esses dados são gerados por algum método escolhido pelo analista de testes. Após os casos de testes serem instanciados, é necessário transformá-los em executáveis, seja por *scripts*, ferramentas, ou planilhas para serem executados de forma manual.

Assim que os casos de testes estejam prontos, o sexto passo consiste em executá-los na aplicação Web. Por fim, o passo 7 consiste em verificar e analisar os resultados obtidos.

A Figura 5.1 ilustra os passos da metodologia. As seções a seguir apresentam cada um desses passos mais detalhadamente.

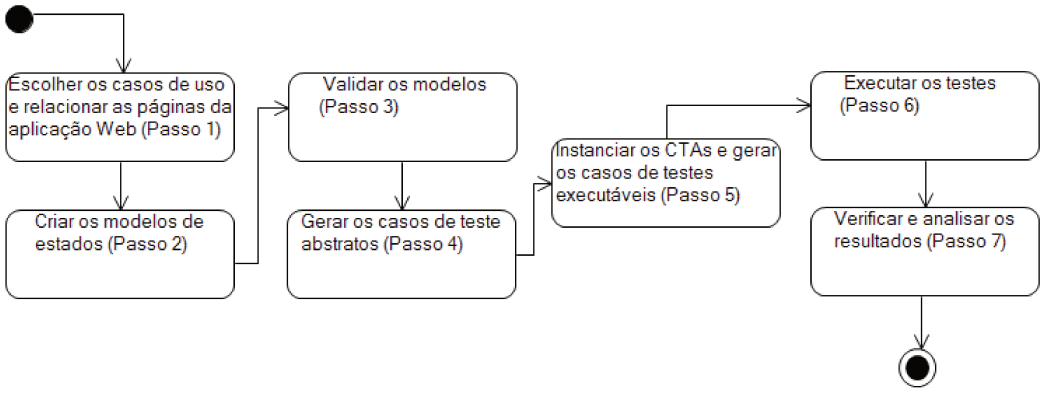


Figura 5.1: Passo-a-passo da metodologia.

5.2 Escolha dos Casos de Uso e Lista das Principais Páginas

Como descrito por Sommerville [26], caso de uso é uma técnica baseada em cenário para extração dos requisitos. Essa técnica identifica as interações individuais com o sistema, assim como quem está envolvido nela, ou seja, os atores. Atores representam um conjunto de pessoas ou dispositivos que operam o sistema. Mais formalmente, atores são qualquer pessoal ou dispositivo que se comunica com o sistema ou produto que é externo ao sistema [12]. Todo ator tem um ou mais objetivos ao utilizar o sistema.

Um caso de uso encapsula um conjunto de cenários, e cada cenário é um conjunto de acontecimentos. Se o cenário possui mais de um conjunto, deve haver um cenário para a interação normal e mais os cenários para as possíveis exceções. Um caso de uso pode ser descrito no formato de texto, em liguagem natural ou com o uso de diagramas da UML, como por exemplo, diagrama de estados. Neste trabalho, parte-se da descrição textual dos casos de uso, pois esta é a forma de documentação existente para o sistema TelEduc, o alvo desta proposta. Resumidamente, um caso de uso é uma narrativa descritiva de como os atores interagem com o sistema [12].

Uma análise nos casos de uso deve ser feita para verificar as funcionalidades mais importantes do sistema, pois como Pressman [12] afirma, o primeiro passo ao testar um sistema é testar cada tarefa de forma separada. O ideal é testar tudo da sua aplicação, mas por onde começar? Ou então, se o tempo não for suficiente, o que testar? Os testes podem começar pelos componentes mais importantes do sistema, aqueles que são mais críticos ou que realizam tarefas essenciais. Com a análise dos casos de uso, cria-se a lista das funcionalidades mais importantes e o que estiver nessa lista é por onde serão iniciados os testes.

Tabela 5.1: Resumo do exemplo de caso de uso

Nome	*Login* no sistema
...	
Descrição	Usuário faz o *login* para ter acesso ao sistema inteiro
...	
Fluxo Principal	1) O usuário clica no link "*login*" no canto superior direito da tela; 2) O sistema exibe uma caixa com os campos "e-mail" e "senha"; 3) O usuário preenche os campos e clica em "Entrar". A senha deve conter no mínimo 4 caracteres e todos maiúsculos; 4) Sistema valida os dados, se os dados estiverem incorretos, execte o fluxo de exceção E1. O sistema exibe o nome do usuário no canto da página.
Fluxo de Exceção	E1) O sistema redireciona para a página de erro; E2) O sistema deve mostrar qual o motivo do erro; E3) Usuário clica em "Voltar" para retornar à página principal.

Na situação de não existir documentação, o gerente do projeto deve ser consultado para saber quais as principais funcionalidades, ou as páginas do sistema devem ser abertas para se navegar e levantar o que é mais importante.

Como exemplo para explicação do método, a Tabela 5.1 descreve um caso de uso, de forma resumida, para realizar um *login* em um sistema. As Figuras 5.2, 5.3, 5.4 e 5.5 mostram as interfaces desse sistema.

Figura 5.2: Página inicial do sistema, com o *link login*.

O caso de uso da Tabela 5.1 mostra os eventos que devem ocorrer nas telas do sistema. Por exemplo, o item 1 da descrição descreve que o usuário deve clicar em *login*. A tela da Figura 5.2 mostra que há esse *link*. Assim, os eventos descritos são as ações nas telas do

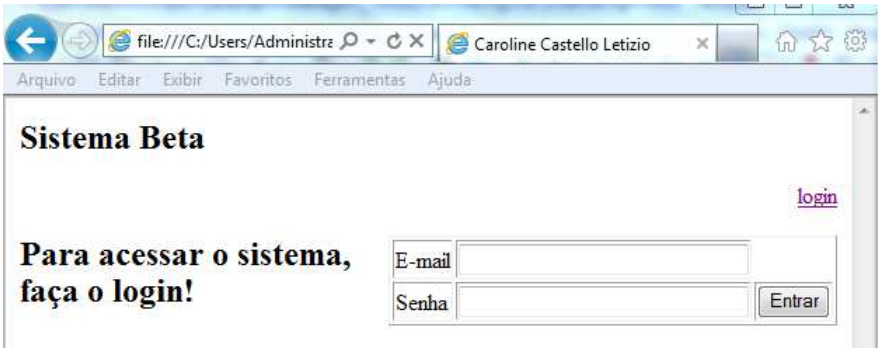


Figura 5.3: Página inicial com a caixa para efetuar o *login*.

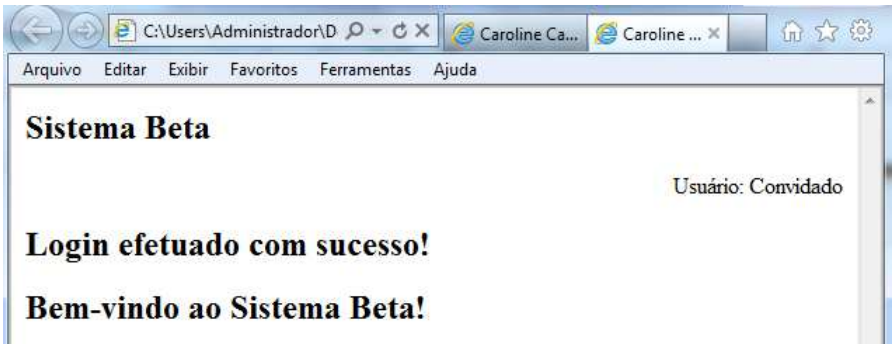


Figura 5.4: Página inicial ao realizar o *login*.

sistema. Outro detalhe é que um caso de uso pode ser equivalente a uma ou mais páginas do sistema. No caso do exemplo dado, o caso de uso equivale a três páginas do sistema (a Figura 5.3 é igual à 5.2, porém com o conteúdo dinâmico, que é a aparição da caixa de *login*).

5.3 Criação dos Modelos

A primeira pergunta a se considerar para criar o modelo é: o que modelar? Como explicado no capítulo Modelagem de Aplicações Web, o modelo a ser utilizado neste trabalho é o diagrama de estados da UML e o caso de uso nos indica qual o limite do modelo, ou seja, até que parte do sistema aquele modelo irá representar. Uma funcionalidade do sistema pode ser descrita por mais de um caso de uso, por exemplo, se um sistema de compras possui dois casos de uso: um com o fluxo de fechar a compra e pagar com cartão de crédito e outro com o fluxo de continuar a compra após fechar o carrinho de compras. Neste caso, haverá um modelo de estados para cada caso de uso. Dessa forma, o modelo não ficará muito grande por englobar todas as situações com o carrinho fechado para

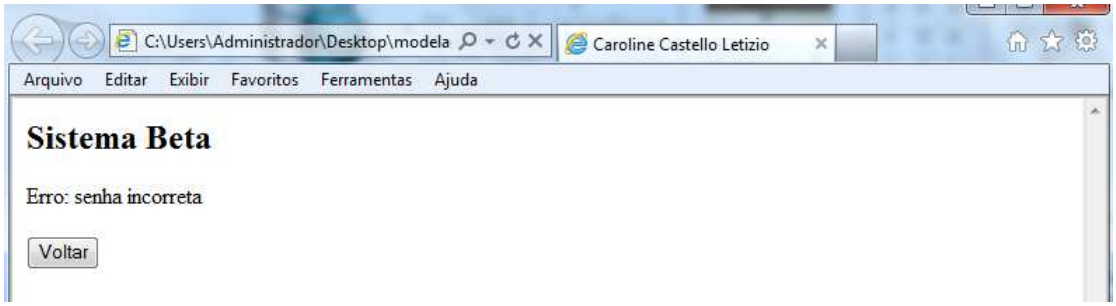


Figura 5.5: Página de erro.

finalizar a compra. Porém, o caso de uso não mostra exatamente o que se quer modelar, ou seja, todas as possibilidades de entrada do usuário no sistema, botões existentes etc., pois é nessa parte que se verifica a importância da navegação das telas do sistema para se construir um modelo mais completo.

Como analisado nos trabalhos relacionados, no capítulo 4, é necessário modelar os aspectos dinâmicos de uma página. Isso está bastante evidente no trabalho de Alalfi, Cordy e Dean [20], o qual ressalta a necessidade de um modelo que mostre os aspectos dinâmicos, devido à evolução da Web. Se no caso de uso mais de uma página estiver envolvida, a navegação entre essas páginas também será modelada.

Como o caso de uso é um modelo funcional, não mostra, necessariamente, a dinamicidade de um sistema, e é isso que um diagrama de estados permite visualizar, ou seja, mostra uma visão dinâmica do sistema. A UML inclui em seus diagramas de estado notações para ilustrar eventos e estados de casos de uso [16].

Mas como saber o que é um estado, transição, saída, etc.? Analisando alguns trabalhos, dois se mostraram mais coerentes e atuais quanto a essas questões. A próxima subseção explica esses trabalhos e a conclusão tirada a partir deles.

5.3.1 Definição de Estados e Transições

Para se conseguir uma definição de estado e transições para modelagem de sistemas Web, vários foram os trabalhos estudados, dentre eles, o de Marchetto, Ricca e Tonella [35] e o de Winckler e Palanque [10]. A seguir uma explicação resumida das definições dada pelos autores e, por fim, as definições usadas neste trabalho.

O trabalho de Marchetto, Ricca e Tonella [35] analisa sistemas que utilizam tecnologia Ajax (a qual torna a aplicação dinâmica). Para gerar os testes a partir de modelos são utilizadas engenharia reversa e técnica de Web *crawling*, nas quais são investigados os elementos dinâmicos da página com uma ferramenta *crawler*, sendo necessário o código do sistema. Os autores mostram que a ideia em um teste baseado em estados é que os estados

dos componentes do lado do cliente de uma aplicação Ajax precisam ser levados em conta durante a fase de teste. Portanto, considera-se que um estado é uma possível instância do documento DOM, ou seja, elementos HTML (que podem ser alterados de acordo com a interação do usuário) do lado cliente de uma página web, e seus valores correspondentes são usados para construir um modelo de estado finito. Em outras palavras, um estado é todo o elemento dinâmico da página (área de texto, formulários, elementos prontos como um calendário etc). As transições são associações de mudança de estado produzida por eventos da interface (por exemplo, o clique de um botão).

Outra pesquisa que traz as definições de estado e transição é a de Winckler e Palanque [10]. Apesar dos autores mostrarem uma ferramenta própria, explicam quais elementos devem ser considerados pela ferramenta para montar o modelo de estados. Um estado é considerado uma abstração de um container (aquilo que armazena todo o conteúdo a ser exibido em uma aplicação Web, como as páginas HMTL) para objetos, ou seja, o documento DOM com seus elementos HMTL, sejam eles gráficos ou executáveis. De uma forma similar, as transições são representadas por agentes que ativam esses objetos. A maior parte dos estados é uma estrutura básica que representa informação. Há aqueles que representam conteúdo gerado de forma dinâmica durante uma execução do sistema (como a execução de pressionar um botão da tela do sistema). Existem, também, os estados que representam informações que são acessíveis por transições, mas não fazem parte do projeto em desenvolvimento, e sim de algum projeto externo. Esse tipo de estado, geralmente, representa um *site* externo.

Os eventos possuem classificação nesse trabalho segundo o agente que os "engatilha": usuário, como um clique de um *mouse*, e sistema, como um método que altera a atividade de um estado. Essas classificações são propagadas para as transições. De forma resumida, transições são definidas tanto de usuários ou sistema e são os eventos que alteram o comportamento da aplicação Web.

Os autores ressaltam que, na modelagem, todos os tipos de estado devem aparecer, porém pode-se criar uma notação própria para cada tipo. Em relação à existência de *frames* em uma página, a sugestão na modelagem é utilizar regiões ortogonais, em que cada região representa um *frame* individual.

As limitações desses dois trabalhos estão no fato do analista de teste ter que recorrer ao código fonte do sistema, ou seja, há necessidade de saber detalhes sobre aspectos internos das páginas (no caso deste trabalho, utiliza-se as páginas geradas para extrair o modelo, não necessitando de detalhes das construções das páginas). No caso de Marchetto, Ricca e Tonella [35], é necessário o acesso ao documento DOM e a ferramenta de *crawler* para investigar os elementos dinâmicos. Já no trabalho de Winckler e Palanque [10], também deve-se ter o acesso para identificar os *frames*, por exemplo. Outra limitação do trabalho de Marchetto, Ricca e Tonella [35] é que considera apenas elementos Ajax, e uma aplicação

pode conter outras tecnologias que tornam o lado cliente mais interativo, como Active-X. Porém, as definições dos elementos para serem modelados e como são modelados são muito importantes.

Portanto, com a análise desses dois trabalhos, foram estabelecidas as definições a seguir para este trabalho:

Definição 1: *Um estado é toda estrutura dinâmica dentro de uma página e toda nova página que pode ocorrer dentro de uma funcionalidade do sistema.*

Definição 2: *Transições são os eventos e ações que alteram os estados e que fazem ocorrer modificações no comportamento da página.*

5.3.2 Explicando o Exemplo

Considerando o caso de uso da Tabela 5.1, é descrito o comportamento dinâmico da página ao aparecer o campo *login* e, se houver erro nos dados, deve aparecer uma página de erro e a opção "Voltar". A Figura 5.6 exemplifica o modelo criado a partir do caso de uso e da navegação das telas:

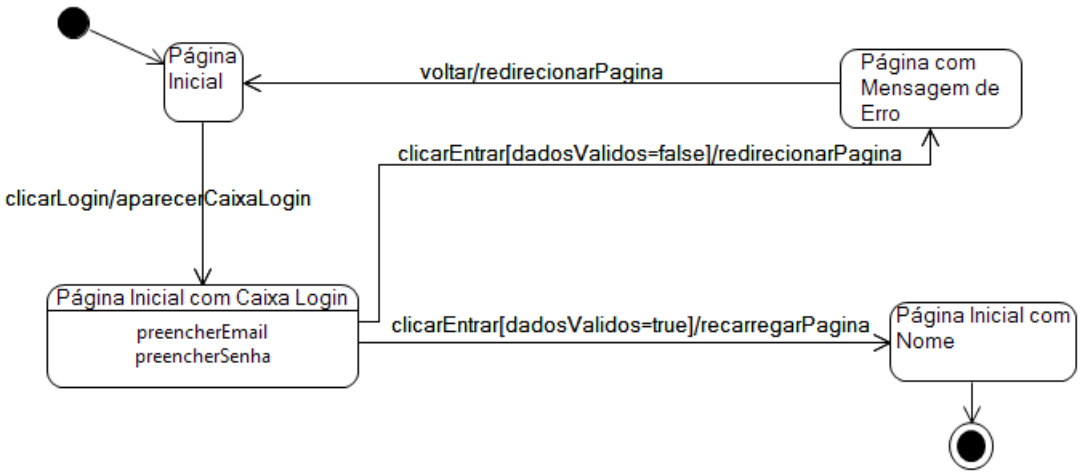


Figura 5.6: Modelo de estados a partir do caso de uso.

A partir das definições da subseção anterior, ao verificar o exemplo da Figura 5.6, é possível observar quatro estados: "Página Inicial", "Página Inicial com Caixa Login", "Página Inicial com Nome" e "Página com Mensagem de Erro". Note-se que "Página Inicial com Caixa Login" ainda está na página inicial, porém com o comportamento modificado, sendo que antes não havia uma caixa e agora há. Assim com base na modelagem de *frames* proposta por Winckler e Palanque [10], modela-se a alteração da página, mas sem utilizar regiões ortogonais, apesar da possibilidade de utilizar esse recurso da modelagem UML. Dentro desse estado, existem as ações internas: "preencherEmail" e

"preencherSenha". Ações internas são outra particularidade da UML na versão 2 que a diferencia dos outros modelos existentes. Dessa forma, sabe-se que, quando o modelo estiver no estado "Caixa Login", há duas ações a serem executadas.

O estado "Página Inicial com Nome" também se refere à mesma página inicial que possuía apenas o *link* "*login*", porém, como o usuário efetuou o *login* com dados corretos, houve novamente uma alteração da página inicial, agora com o nome do usuário no local do *login*. No caso do estado "Página com Mensagem de Erro", representa-se outra página do sistema, que faz parte, entretanto, do caso de uso exemplificado na Tabela 5.1. Então, essa página também ganha espaço no modelo, sendo um estado com transições que chegam a ele e vão para outros estados.

Nesse mesmo exemplo, também são mostrados os eventos (ações de entrada) que são *links* e botões, como o *link* "*login*" e o botão "Entrar". Os eventos são acompanhados das ações de saída, que são respostas esperadas pelo acontecimento dos eventos. Geralmente, são ações que o sistema efetua. Também é possível notar as condições de guarda, o que ajuda na navegação do modelo. As condições de guarda são outra particularidade dos modelos da UML que permitem diminuir a quantidade de estados. Se há utilização do modelo clássico, o qual não permite representar condições de guarda, corre-se um risco maior de ocorrência do problema de explosão de estados, uma vez que cada guarda teria que ser transformada em um estado.

5.4 Validação dos Modelos

Em Alalfi, Cordy e Dean [20] é ressaltada a importância da validação do modelo, ou seja, se ele está completo e correto para se gerar os testes. A forma sugerida pelos autores é a verificação de modelo (*model checking*) a qual faz uma análise estática, formal do modelo quanto a sua estrutura. Porém, há outra forma não estática e menos complexa do que *model checking*? Sim, há algumas técnicas de execução do modelo, como o modelo executável da UML [39] e compiladores de máquinas de estados, como o SMC [44]. Entretanto, essas técnicas existentes ou necessitam que um *driver* seja criado para executar um modelo, ou, no caso do modelo executável, um compilador que interpreta o modelo que possui especificações da arquitetura o verifica e faz a transformação em código, por isso não são utilizadas nesse trabalho.

A técnica proposta adotada neste trabalho foi a de simulação do modelo de estados. A simulação do modelo representa o comportamento do modelo, ou seja, é possível analisar e gerenciar os eventos, transições, ações e variáveis e visualizar e rastrear a execução. Com a simulação, que pode ser feita de forma manual ou automática, é possível explorar e testar o comportamento dinâmico dos modelos [45].

Tabela 5.2: Tabela de Transição de Estados do modelo da Figura 5.6.

Estado Atual	Próximo Estado			
	Página Inicial	Página Inicial com Caixa Login	Página Inicial com Nome	Página com Mensagem de Erro
Página Inicial	-	clicarLogin/ aparecerCaixa Login	-	-
Página Inicial com Caixa Login	-	-	clicarEntrar [dadosValidos= true]/ recarregarPagina	clicarEntrar [dadosValidos= false]/ redirecionarPagina
Página Inicial com Nome	-	-	-	-
Página com Mensagem de Erro	voltar/ RedirecionarPagina	-	-	-

O importante da simulação é percorrer cada estado e cada transição pelo menos uma vez, para compreender o comportamento que está sendo representado. Portanto, é possível validar se o modelo satisfaz aos requisitos do sistema, ou seja, se os cenários descritos pelos casos de uso estão sendo representados pelo modelo.

Para fins de exemplificação, vamos considerar novamente o modelo da Figura 5.6. Para realizar a simulação manualmente, será necessária a tabela de transições de estados (TTE) do modelo. A Tabela 5.2 mostra a TTE do modelo da Figura 5.6. Para acompanhar a simulação, o modelo será utilizado junto com a tabela, para mostrar como é feita a validação, os estados e transições serão realçados (coloridos). Nas figuras que se seguem, o estado em negrito com fundo colorido (ou cor vermelha de fundo roseado) mostra o estado atual da simulação do modelo, e a linha pontilhada (ou cor verde), as possíveis transições. A Figura 5.7 mostra o início da simulação do modelo da Figura 5.6.

No início da simulação, tem-se como início o estado “Página Inicial” e como possível transição a ocorrer “clicarLogin/aparecerCaixaLogin”. No caso da simulação manual, o responsável deve ir marcando o trajeto que já percorreu (no caso, a transição é riscada) e ficar atento ao que falta ocorrer. A Tabela 5.3 mostra como fica a tabela conforme a simulação da Figura 5.7. Qual o significado dessa simulação da Figura 5.7? Essa imagem significa que, na página inicial do sistema, a única opção de clique ou de ação é o link

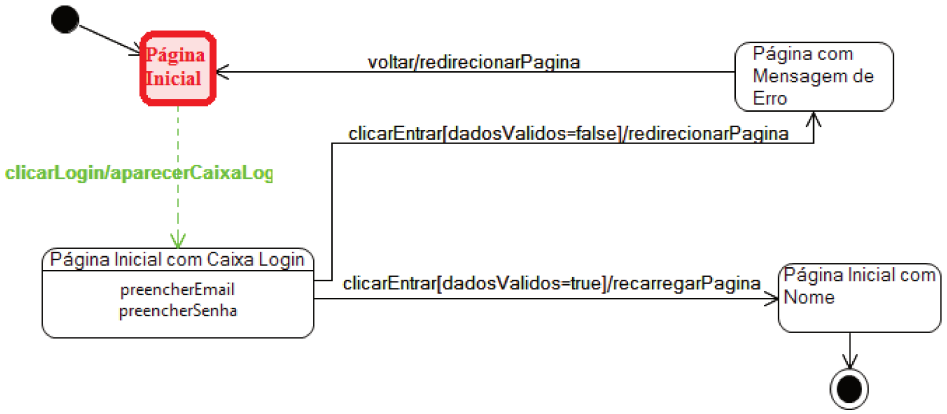


Figura 5.7: Início da simulação do modelo.

"*login*" e que, ao clicar, deve aparecer uma caixa com campos para se fazer o *login*. Essa interpretação representa o comportamento do sistema. Caso haja algum *link* ou botão, além do *login*, é possível prever se falta alguma coisa no modelo.

O próximo passo da simulação é seguir a possível transição. A Figura 5.8 e a Tabela 5.4 representam essa etapa:

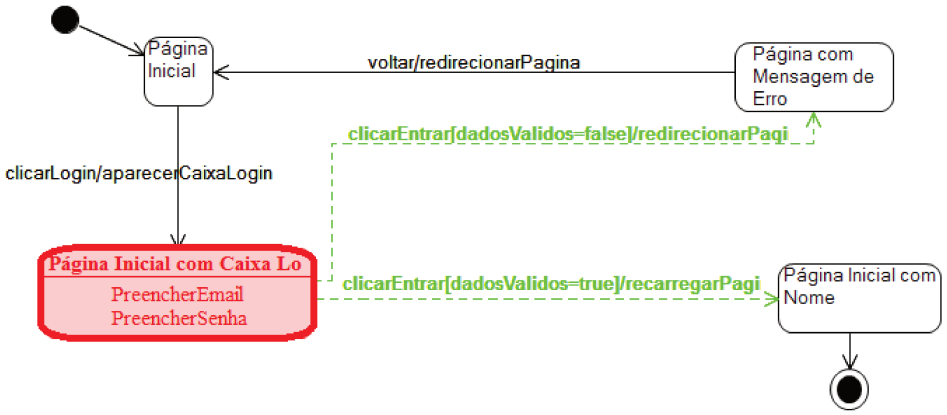


Figura 5.8: Segunda etapa da simulação.

A Figura 5.8 mostra dois caminhos possíveis a partir do estado "Página Inicial com Caixa Login": aquele em que os dados incluídos eram corretos e o outro com algum dado incorreto. O responsável pela validação deve seguir os dois caminhos, um de cada vez, a fim de verificar qualquer possibilidade de comportamento do sistema e se o modelo condiz com o caso de uso.

Tabela 5.3: Início da simulação do modelo com tabela.

Estado Atual	Próximo Estado			
	Página Inicial	Página Inicial com Caixa Login	Página Inicial com Nome	Página com Mensagem de Erro
Página Inicial	-	**clicarLogin/ aparecerCaixa Login**	-	-
Página Inicial com Caixa Login	-	-	clicarEntrar [dadosValidos=true]/ recarregarPagina	clicarEntrar [dadosValidos=false]/ redirecionarPagina
Página Inicial com Nome	-	-	-	-
Página com Mensagem de Erro	voltar/ RedirecionarPagina	-	-	-

Tabela 5.4: Continuação da simulação do modelo com tabela.

Estado Atual	Próximo Estado			
	Página Inicial	Página Inicial com Caixa Login	Página Inicial com Nome	Página com Mensagem de Erro
Página Inicial	-	~~clicarLogin/ aparecerCaixa Login~~	-	-
Página Inicial com Caixa Login	-	-	**clicarEntrar [dadosValidos=true]/ recarregarPagina**	clicarEntrar [dadosValidos=false]/ redirecionarPagina
Página Inicial com Nome	-	-	-	-
Página com Mensagem de Erro	voltar/ RedirecionarPagina	-	-	-

Tabela 5.5: Término da simulação do modelo com tabela.

Estado Atual	Próximo Estado			
	Página Inicial	Página Inicial com Caixa Login	Página Inicial com Nome	Página com Mensagem de Erro
Página Inicial	-	~~clicarLogin/ aparecerCaixa Login~~	-	-
Página Inicial com Caixa Login	-	-	~~clicarEntrar [dadosValidos= true]/ recarregarPagina~~	~~clicarEntrar [dadosValidos= false]/ redirecionarPagina~~
Página Inicial com Nome	-	-	-	-
Página com Mensagem de Erro	~~voltar/ RedirecionarPagina~~	~~-~~	-	-

Supondo que a pessoa que criou o modelo houvesse esquecido de tratar o erro, ao validar o modelo e rever o caso de uso, o responsável deve ficar atento a toda possibilidade de erro que pode ocorrer por determinada ação do usuário ou até mesmo do sistema.

A simulação termina quando todas as transições e estados tiverem sido confirmadas, como mostra a Tabela 5.5. Caso haja erro no modelo, uma atualização deve ser feita, e se reinicia a simulação para validação do modelo.

5.5 Geração dos Casos de Testes Abstratos (CTAs)

Os testes abstratos são criados a partir de critérios utilizados para se percorrer o modelo e obter caminhos que se tornam casos de teste. Existem muitas ferramentas e métodos que geram testes abstratos a partir de modelos, e, neste trabalho, é totalmente recomendada a geração automática de testes a partir de modelos, ou seja, utilizando ferramentas. A ferramenta utilizada nessa pesquisa é explicada no capítulo Estudo de Caso. Porém, esses testes também podem ser obtidos de forma manual. Lembrando que manualmente há mais chances de ocorrerem erros.

Para geração de forma manual, recomenda-se a estratégia proposta por Binder [22].

Considere o modelo da Figura 5.9 (o mesmo modelo da Figura 5.6, com as transições enumeradas para facilitar o entendimento com a árvore gerada). O autor explica que, após a geração do modelo, cria-se uma árvore de transições baseada nesse modelo. A árvore, como representada na Figura 5.10, inclui todos os caminhos de ida e volta, ou seja, sequência de transições que começam e terminam no mesmo estado e caminhos simples que começam no estado inicial e terminam no estado final. Se houver um *loop*, considera-se apenas uma interação desse *loop*.

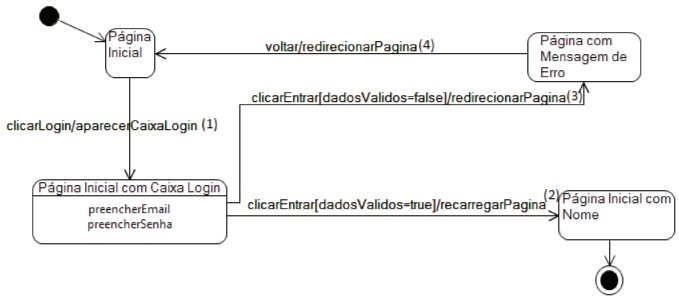


Figura 5.9: Modelo de estados do caso de uso da Tabela 5.1, com transições enumeradas.

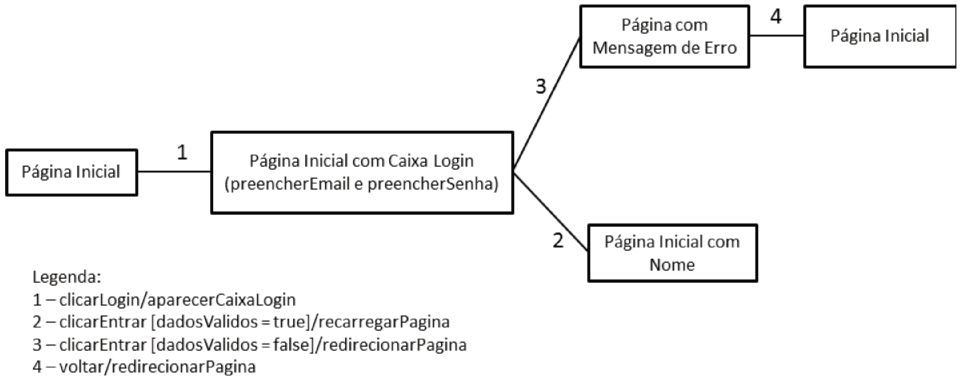


Figura 5.10: Árvore de transições obtida a partir do modelo de estados da Figura 5.6.

A transformação do modelo para árvore de transições ocorre da seguinte forma: a raiz da árvore é representada pelo estado inicial, como na Figura 5.9 – o estado "Página Inicial". Uma aresta é desenhada para cada transição que sai do nó inicial - no caso, apenas a transição "clicarLogin/AparecerCaixaLogin" - com outros nós sendo adicionados para cada estado resultante, ou seja, o próximo nó será o estado "Página Inicial com Caixa Login" e as transições "clicarEntrar(dadosValidos=false)/redirecionarPagina" e "clicarEntrar(dadosValidos=true)/recarregarPagina", como as próximas arestas . Um nó folha é marcado como "final" se o estado representado já houver sido desenhado, no caso "Estado Inicial" e "Página Inicial com Nome", que é o estado final do modelo. Mais nenhuma

transição é adicionada no nó final. Este procedimento é repetido até que todos os nós-folhas sejam finais. Quando existir condição de guarda, pelo menos duas transições são necessárias para as condições, uma transição para que a condição seja verdadeira e outra transição para a condição falsa – como no exemplo há as transições para cada condição de guarda. Se ocorrerem várias variáveis que podem fazer a condição ser verdadeira, deve-se transcrever uma transição para cada variável, exercitando todas. Os estados que possuírem ações internas devem ser representados com essas ações, para, na geração de testes, serem colocadas no teste.

A estrutura da árvore depende da ordem em que as transições foram inseridas. O autor sugere que as transições sejam inseridas na árvore no sentido horário, a partir da ordem em que aparecem, partindo do topo de cada estado, ou seja, a primeira transição será a que aparece logo na parte superior do estado, seguindo em sentido horário. Dessa forma, obtém-se um caminho transversal em largura das transições de cada estado. Com a árvore obtida, é possível aplicar os algoritmos de busca em grafos para se obter os testes. Ainda é possível fazer a aplicação, de forma manual, na qual cada teste é obtido iniciando no nó raiz e terminando em um nó folha. Os resultados esperados são as sequências de estados e ações os quais correspondem aos eventos da sequência de teste.

Na proposta de Binder [22], o formato de um caso de teste abstrato é {estado; transição; estado; transição, ...}. No caso da árvore da Figura 5.10, têm-se dois casos de teste:

1. {Página Inicial; clicarLogin/aparecerCaixaLogin; Página Inicial com Caixa Login (preencherEmail, preencherSenha); clicarEntrar[dadosValidos=true]/ recarregarPagina; Página Inicial com Nome}

2. {Página Inicial; clicarLogin/aparecerCaixaLogin; Página Inicial com Caixa Login (preencherEmail, preencherSenha); clicarEntrar[dadosValidos=false]/ redirecionarPagina; Página com Mensagem de Erro; voltar/redirecionarPagina; Página Inicial}

Um detalhe a se destacar é que o modelo do exemplo é pequeno. Conforme o modelo aumenta de tamanho, a árvore também aumenta, assim como suas possíveis combinações. Por isso, incentiva-se o uso de ferramentas para a geração dos testes a partir de modelo.

5.6 Geração dos Casos de Testes Executáveis

Para mostrar que o sistema em teste atende aos requisitos, critérios de entrada e saída de dados devem ser definidos. Há a geração de possíveis entradas no sistema e saídas esperadas e, assim, define-se os dados para instanciar os testes.

Para a instanciação dos dados, existem critérios para testes de caixa preta que são definidos em termos de possíveis valores de entradas e saídas do sistema. Algumas técnicas,

como análise de valores limites, partições de equivalência são utilizados [5]. Após a definição dos critérios, criam-se os testes executáveis para realizá-los no sistema.

As próximas subseções explicam como criar os dados de entrada e as saídas esperados do sistema, a instanciação dos casos de teste com os dados criados, a geração e execução dos testes, analisando, assim, os resultados obtidos.

5.6.1 Geração de Dados para os Casos de Testes

Partição de equivalência e valores limites são algumas das técnicas mais utilizadas na literatura para definir dados para os testes. Sendo assim, foram utilizadas neste trabalho.

Como descrito no capítulo 3, a partição de dados é utilizada para definir classes de equivalência e conjunto de dados [5]. O princípio da técnica está em definir um domínio de entrada do componente que está sendo testado e dividi-lo em um número finito de partições de equivalência, supondo que os dados pertencentes a uma partição têm capacidade de revelar os mesmos tipos de problemas e sendo uma classe de equivalência representante de um conjunto de estados válidos e inválidos para uma dada condição de entrada.

No exemplo do modelo da Figura 5.6, como descrito nas condições de *e-mail* e senhas no caso de uso da Tabela 5.1, temos duas partições de equivalência para os campos "*e-mail*" e "senha": (1) valores válidos para ambos os campos, como a estrutura para e-mail tradicional [nome@dominio.com] e para a senha a estrutura XXXX , sendo X qualquer caractere do alfabeto e (2) valores inválidos para os campos, como quando o *e-mail* não possui o "@" e/ou a senha tiver algum caractere minúsculo.

Já na análise de valores limites, tem-se a identificação dos limites das classes de equivalência, por exemplo, valores que possuam o valor mínimo e o máximo possível, um arquivo vazio, um valor vazio, um valor que possa resultar num estouro de uma pilha etc. No caso das partições encontradas no parágrafo anterior, um exemplo seria na segunda classe, no campo senha, teríamos que exercitar senhas com três caracteres, senha com 50 caracteres (já que não há um valor máximo), uma senha com apenas um caractere minúsculo e seis dígitos e uma senha com apenas um caractere maiúsculo e seis dígitos.

Para definir as saídas esperadas, basta olhar as saídas das transições. Por exemplo, no modelo da Figura 5.6, a transição "clicarEntrar(dadosValidos=false)/RedirecionarPagina" sai do estado "Página Inicial com Caixa Login" e vai para o estado "Página com Mensagem de Erro". Assim, a saída é o redirecionamento da página para a tela com a mensagem de erro.

Uma tabela com os valores de entrada e as saídas esperadas pode ser feita. A Tabela 5.6 exemplifica esses valores:

Tabela 5.6: Exemplo de valores de entrada e saídas esperadas.

Tipo de partição	Campo	Valor de entrada	Saída esperada
Valores Válidos	E-mail	teste@dissertacao.com	Login no sistema
	Senha	ABCD	
	E-mail	teste@teste2.com.br	Login no sistema
	Senha	TESTE2	
Valores Inválidos	E-mail	teste.com	Mensagem de erro
	Senha	ABCD	
	E-mail	teste@dissertacao.com	Mensagem de erro
	Senha	ABC	
	E-mail	teste@teste2.com.br	Mensagem de erro
	Senha	abCd	

5.6.2 Instanciação dos Casos de Teste

Após os valores serem definidos, instanciam-se os casos de teste com os dados. Para isso, utilizam-se os CTAs já criados.

O caso de teste abstrato possui o formato {estado inicial, transição1, estado2, etc}, como visto na sessão Geração de Testes Abstratos. A transição é composta por evento e ação (capítulo Modelagem Web), sendo assim, o CTA pode ser escrito com a seguinte sequencia de passos:

1. Estado inicial. Ocorrer evento transição 1. Deve ocorrer ação de saída da transição 1
2. Estado 2. Ocorrer evento transição 2. Deve ocorrer ação de saída da transição 2
3. ...

Assim, instancia-se o CTA com dados para gerar o caso de teste instanciado. A forma genérica fica assim:

1. Estado inicial. Ocorrer evento com dado de entrada da transição 1. Deve ocorrer ação de saída esperada da transição 1
2. Estado 2. Ocorrer evento com dado de entrada da transição 2. Deve ocorrer ação de saída esperada da transição 2
3. ...

Considere o exemplo dado:

CTA: {Página Inicial; clicarLogin/aparecerCaixaLogin; Página Inicial com Caixa Login (preencherEmail, preencherSenha); clicarEntrar[dadosValidos=true]/recarregarPagina;

Página Inicial com Nome} Utilizando os valores da Tabela 5.6 para os dados válidos, gera-se o seguinte caso de teste instanciado:

1. Página Inicial, clicar no link "*login*", o sistema deve mostrar a caixa de *login*.
2. Na caixa de *login*, preencher e-mail = teste@dissertacao.com e preencher senha = ABCD, clicar no botão "Entrar", o sistema deve recarregar a página.
3. Página inicial com nome.

Como a Tabela 5.6 possui outro conjunto de *e-mail* e senha válidos, outro caso de teste instanciado, semelhante ao de cima, deve ser gerado.

Após a geração dos casos de teste instanciados com dados válidos, geram-se os casos de testes instanciados a partir do CTA de dados inválidos.

Ao final da criação de todos os casos de testes com dados, obtém-se um conjunto de casos de testes para serem transformados em executáveis, ou seja, específicos da plataforma do sistema em teste e serão executados de forma manual ou automática.

5.7 Execução dos Testes

A execução dos testes, como já mencionado no capítulo Testes de *Software* e Testes Baseados em Modelos, pode ocorrer de forma manual ou de forma automática.

A partir dos casos de teste instanciados, cria-se um roteiro denominado casos de testes executáveis. O formato e o tipo dos casos de teste executáveis são feitos conforme a preferência de trabalho do analista de teste. Aqui, é indicado o uso de planilhas, pois além da possibilidade de se relacionar a planilha de testes abstratos com os instanciados, é possível visualizar de uma forma melhor os testes, além das possibilidades de cálculos e estatísticas que as planilhas oferecem.

A partir dos casos de teste instanciados, cria-se uma tabela que deve conter o evento, ou seja, a entrada que deve ser realizada para o sistema devolver uma resposta. Após, se verifica se o evento possui algum parâmetro, isto é, um valor que deve ser preenchido com um dado; para isso, também há uma coluna na tabela para o parâmetro. Se há parâmetro, há um valor; já que o caso de teste já está instanciado, coloca-se uma coluna para o valor. Também deve haver a saída esperada nessa tabela. Nesta etapa, acrescenta-se uma coluna a mais para a verificação da saída ocorrida, ou seja, a saída que realmente o sistema mostrou. Esse resultado é o que será comparado com a saída esperada para ver se o sistema devolveu a resposta correta, mostrando se o teste "passou" ou "não passou", colocado na coluna "Resultado do Teste". A Tabela 5.7 exemplifica essa planilha, para os seguintes casos de teste instanciados do exemplo desse capítulo:

Tabela 5.7: Casos de teste.

Caso de Teste	Evento	Parâmetro	Valor	Saída Esperada	Saída Observada	Resultado do Teste
1	clicarLogin	-	-	Aparecer caixa "login"		
	preencherEmail	email	teste@dissertacao.com	-		
	preencherSenha	senha1	ABCD	-		
	clicarEntrar	-	-	Redirecionar página "login sucesso". Aparecer página com o nome do usuário		
2	clicarLogin	-	-	Aparecer caixa "login"		
	preencherEmail	email	teste.com	-		
	preencherSenha	senha1	ABCD	-		
	clicarEntrar	-	-	Redirecionar página erro. Aparecer mensagem de e-mail errado		
	clicarVoltar	-	-	Redirecionar para página inicial		

- Caso de teste instanciado 1:
  1. Página Inicial, clicar no *link* "*login*", o sistema deve mostrar a caixa de *login*.
  2. Na caixa de *login*, preencher *e-mail* = teste@dissertacao.com e preencher senha = ABCD, clicar no botão "Entrar", o sistema deve recarregar a página.
  3. Página inicial com nome.
- Caso de teste instanciado 2:
  1. Página inicial, clicar no *link* "*login*", o sistema deve mostrar a caixa de *login*.
  2. Na caixa de *login*, preencher *e-mail* = teste.com e preencher senha = ABCD, clicar no botão "Entrar", redirecionar a página.
  3. Página com mensagem de erro; clicar em "Voltar", redirecionar a página.
  4. Página inicial.

Esses casos de teste executáveis são os testes para execução de forma manual. Caso o analista de teste deseje automatizar os casos de teste, basta transformá-los em *scripts* específicos na plataforma escolhida para testes.

5.8 Análise dos Resultados

Conforme os testes são executados, os resultados são registrados. No caso da planilha da Tabela 5.7, as saídas observadas são colocadas no campo “Saída Observada”.

Após a execução dos casos de testes, as saídas observadas são comparadas com as saídas esperadas, para obter um veredito se o caso de teste “passou” ou “não passou”. Um caso de teste que ”passou”é aquele em que as saídas esperadas realmente ocorrem, ou seja, a saída do caso de teste está correta em relação ao comportamento esperado. Já com um caso de teste que ”não passou”, ocorre o contrário: as saídas esperadas não ocorreram, isto é, o sistema não teve o comportamento esperado na execução da sequência do teste.

Na situação de um caso de teste que não passou, ou seja, naquela em que ocorreu algum tipo de erro, deve ser analisado e verificado em que passo do teste ocorreu a falha. Por exemplo, ao executar o caso de teste 1 da Tabela 5.7, colocar uma senha incorreta e o sistema aceitar como correta, ocorreu um erro de verificação do sistema; esse é o passo que foi identificado como errado. Após a identificação do defeito, esse deve ser reportado para a equipe de desenvolvimento. Ao reportar a falha, o analista de teste deve lembrar-se de colocar o passo-a-passo que se seguiu até o defeito ocorrer, os dados utilizados no teste, qual a versão do sistema que estava sendo testada e detalhar o máximo possível da falha que ocorreu.

Dessa forma, completa-se o método de testes proposto.

Capítulo 6

Estudo de Caso

O estudo de caso neste trabalho tem como objetivo mostrar a qualidade da metodologia quanto a ajudar uma equipe de um *software* Web a se organizar em seus testes, saber modelar as partes de seu sistema, encontrar mais erros no sistema do que os métodos de teste que já utilizam e verificar o desempenho da validação dos modelos.

O sistema escolhido para o estudo foi o ambiente de ensino a distância TelEduc. Para avaliar o método proposto, duas formas de aplicação ocorreram: na primeira, um analista de teste já habituado com testes baseados em modelo aplicou a metodologia e obteve os resultados quanto à validação dos modelos e falhas encontradas no sistema. Na segunda, a equipe do próprio ambiente TelEduc foi instruída quanto à metodologia e a aplicou, obtendo também resultados. A princípio, apenas a pessoa responsável pela qualidade do TelEduc aprenderia a abordagem desse projeto, porém toda a equipe de desenvolvimento mostrou interesse em aprender, resultando que todos os integrantes foram instruídos e utilizaram o método apresentado.

Para saber a qualidade e o grau de satisfação dos usuários do TelEduc, um questionário foi aplicado perguntando sobre as dificuldades do sistema e sugestões para aprimoramento da abordagem. Um questionário é uma forma de investigação que ajuda a documentar os benefícios [46] e pontos negativos de um estudo. Neste capítulo, o estudo de caso é apresentado segundo as diretrizes de Kitchenham, Pickard e Pfleeger [46]:

- Definir as hipóteses, ou seja, os efeitos que se espera do método. No caso, espera-se que o método proposto ajude equipes a implementar ou melhorar o processo de testes, obtendo bons resultados e uma forma prática de se recriar os testes caso o sistema mude, apenas alterando os modelos feitos;
- Definir o projeto piloto – no caso, o ambiente de ensino a distância TelEduc;
- Identificar os métodos de comparação. Nessa etapa, o método de comparação é como a equipe do TelEduc testava o ambiente com a nova proposta de criação de

testes;

- Minimizar os efeitos de fatores de confusão. Por exemplo, não se pode comparar os resultados de uma pessoa experiente no método com os de um principiante. Neste trabalho, não foi encontrado fator de confusão;

- Planejar o estudo de caso. Nessa etapa, deve-se identificar todas as questões a serem resolvidas para que a avaliação ocorra da melhor forma possível. O estudo de caso foi planejado tanto para o especialista poder aplicar o método como para a equipe do TelEduc. Nesse segundo caso, organizou-se o estudo com aulas expositivas do método e a aplicação prática;

- Monitorar o estudo de caso em relação ao planejamento. O progresso e resultados devem ser coletados e comparados sempre com o planejamento, para saber se tudo está ocorrendo conforme o esperado;

- Analisar e reportar os resultados. Isso é feito na última seção deste capítulo.

6.1 TelEduc

TelEduc, como mencionado na Introdução, é um ambiente de ensino a distância. É um sistema que possui vários tipos de ferramentas, como as de comunicação e de compartilhamento de arquivos, além de gerenciar e editar cursos e pessoas envolvidos no ambiente.

A documentação do sistema é bem escassa e a que existe está muito desatualizada. O sistema possui documentos de casos de uso, propostas de *layout*, glossário, especificação de requisitos, regras de negócio, padronização de codificação e relatórios descrevendo problemas de acessibilidade do ambiente. Todos esses documentos datam do ano de 2009.

Além do problema da documentação para o desenvolvimento e manutenção do sistema, a equipe não possui um procedimento sistemático de testes, ou seja, os testes são feitos totalmente na forma *ad hoc*, no qual uma pessoa da equipe separa um tempo para ficar navegando no sistema, realizando atividades e descobrindo possíveis erros, tudo feito de forma manual. Não há um plano para saber o que pode ser feito ou o que já foi feito ao se testar a aplicação. Essa foi a principal razão para a escolha desse sistema para ser o estudo de caso deste projeto de mestrado. E também, devido ao fato da autora deste projeto ter trabalhado na equipe e conhecer as dificuldades com os testes.

6.2 Aplicação da Metodologia pelo Analista de Testes

Como explicado no início do capítulo, a aplicação do método ocorreu de duas formas. Na primeira, um analista habituado com rotina de testes baseado em modelo e com a metodologia proposta a aplicou no ambiente TelEduc e obteve resultados com a validação dos modelos e com falhas encontradas no sistema. Achou-se necessário verificar as dificuldades que alguém que já possuía experiência com esse tipo de teste teria, além de aprimorar possíveis partes do método para poder expor para pessoas com pouca ou sem experiência em modelagem de sistemas e em testes de *software*.

No capítulo Metodologia, foi explicado o passo-a-passo do método proposto para se obter o modelo de estado da parte do sistema que se deseja testar até a geração e aplicação de teste. A Figura 6.1 retoma a figura do capítulo Metodologia com as ferramentas para automação dos de alguns passos incluídas, para orientar o leitor nas próximas seções desse capítulo. As ferramentas são explicadas durante o capítulo.

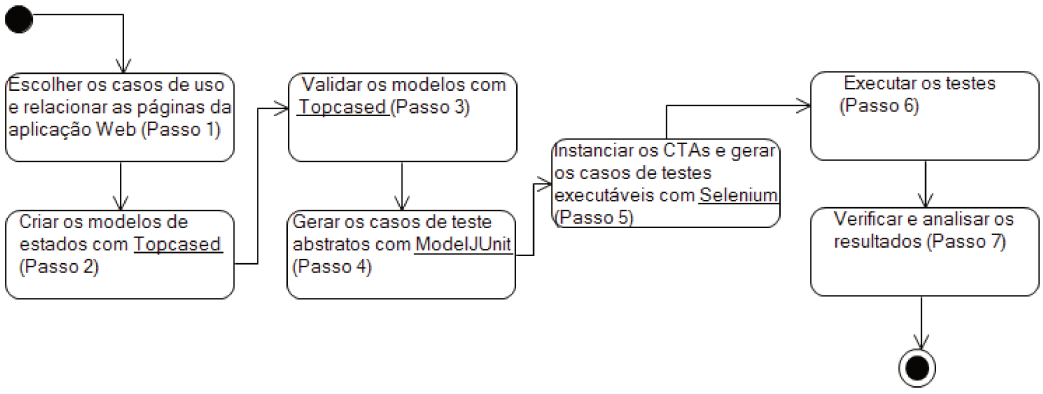


Figura 6.1: Passo-a-passo do método proposto

6.2.1 Escolha dos Casos de Uso

O sistema TelEduc possui 58 casos de uso. Em contato com a equipe de desenvolvimento, foi obtida a lista com os principais casos de uso do ambiente. Um detalhe importante a ser ressaltado, como já mencionado, é que os casos de uso nunca foram atualizados desde a primeira versão no ano de 2009. Nesse tempo, até os dias atuais, o sistema já passou por várias transformações, inclusive alterações muito significativas em sua interface do usuário. Porém, os casos de uso ainda mostram até onde vão os limites de uma determinada ferramenta do ambiente TelEduc, sendo muito úteis para delimitar os modelos, já que eles

descrevem as funcionalidades das páginas dessas ferramentas. As interfaces do usuário do sistema, na sua versão atual, foram utilizadas para a construção de cada modelo.

A Figura 6.2 mostra o diagrama de casos de uso do TelEduc, no qual apenas os principais casos de uso do ambiente aparecem, para que haja facilidade de leitura.

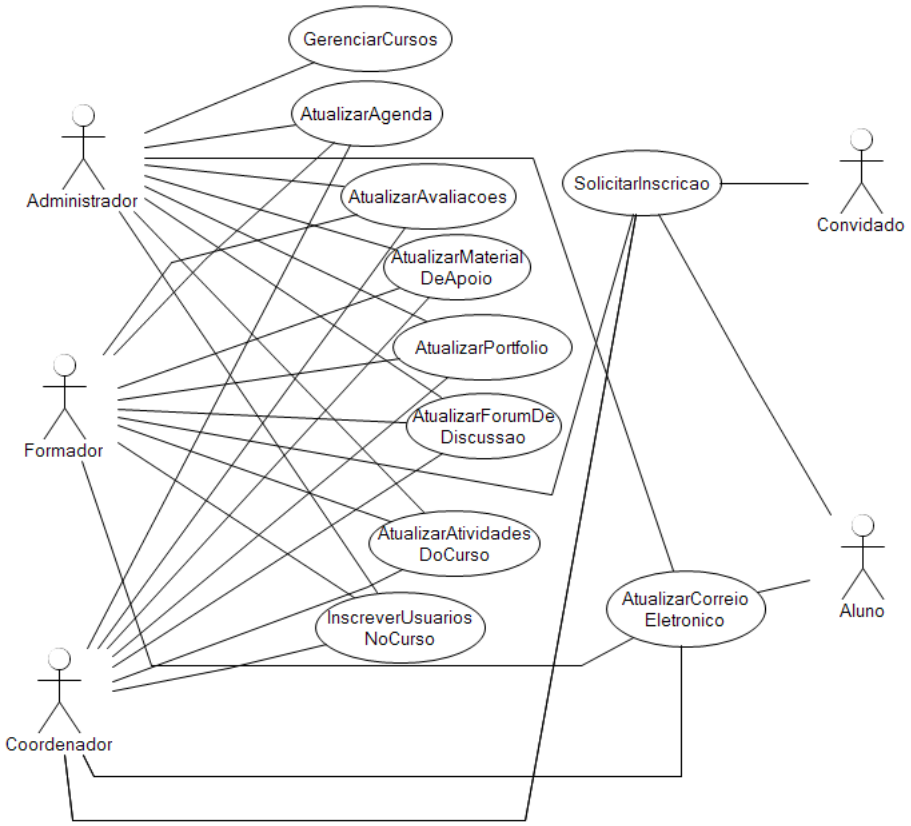


Figura 6.2: Diagrama dos principais casos de uso do sistema TelEduc.

Dos 10 casos de uso que foram relatados como os mais importantes, como constam na Figura 6.2, quatro foram escolhidos para a aplicação da metodologia. Não houve um critério de escolha, pois todos são de igual importância para a aplicação TelEduc, exceto o caso de uso Inscrever Usuário no Curso, o qual foi escolhido por ser o menor e o primeiro a ser utilizado na abordagem. Ou seja, foi um caso de uso escolhido para testar e aprimorar a metodologia. Os casos de uso escolhidos foram:

- Atualizar Agenda – o objetivo desse caso de uso é descrever o registro de informações CRUD (*create, retrieve, update, delete*) persistentes do sistema sobre a agenda do curso.

Tabela 6.1: Informações dos modelos das ferramentas selecionadas.

Modelo	#Estados	#Transições	#Eventos	#Expressões atômicas	CCS
M1: Solicitar Inscrição	9	13	13	6	34
M2: Inscrever Usuário no Curso	19	45	45	18	110
M3: Atualizar Agenda	18	46	46	17	111
M4: Atualizar Material de Apoio	25	56	56	24	138

- Atualizar Material de Apoio – o objetivo desse caso de uso é descrever o registro de informações CRUD (*create, retrieve, update, delete*) persistentes do sistema sobre os materiais de apoio relacionados a um curso.
- Inscrever Usuário no Curso – o objetivo desse caso de uso é realizar a inscrição de um aluno, formador ou convidado em um determinado curso do TelEduc.
- Solicitar Inscrição – o objetivo desse caso de uso é realizar a solicitação de cadastro de novos usuários ao sistema TelEduc.

6.2.2 Criação dos Modelos

Foi criado um modelo para cada caso de uso. O Apêndice A mostra o modelo do caso de uso "Inscrever Usuários no Curso"e a Tabela 6.1 mostra algumas informações referentes a cada modelo, os quais estão na ordem da sua criação, ou seja, o modelo M1 foi criado antes do M2, M3 e M4, e assim por diante. Essa ordem será importante na seção Validação dos Modelos.

Também foi calculada para cada modelo a respectiva complexidade, por meio da métrica CCS - *Cyclomatic Complexity of State Machine* [47]. É possível calcular essa métrica diretamente da máquina de estados, levando em consideração o número de eventos e expressões atômicas (condições de guarda) de cada transição. A fórmula usada é CCS = |T| + |E| + |AG| + 2, onde T é o conjunto de transições, E é o conjunto de eventos e AG é o conjunto de expressões atômicas.

Uma característica interessante é que o número de transições aumenta conforme a quantidade de elementos e componentes dinâmicos que uma ferramenta possui, pois quanto maior o número de elementos dinâmicos (os estados da aplicação), mais ações

Tabela 6.2: Quantidade de validações de cada modelo.

Modelo	Quantidade de validações
M1: Solicitar Inscrição	4
M2: Inscrever Usuário no Curso	2
M3: Atualizar Agenda	2
M4: Atualizar Material de Apoio	1

podem ocorrer em determinada interface. Por exemplo, o modelo M2 possui 19 estados e 45 transições, já o modelo M3 tem 18 estados, porém 46 transições, ou seja, um estado a menos e uma transição a mais em relação a M2. Como visto no capítulo Modelagem Web, é importante modelar os aspectos dinâmicos de uma página, assim como as relações de uma página com outra, influenciando a quantidade de estados e transições. Esse fato é o que influencia na diferença de estados e transições entre as ferramentas "Agenda" e "Inscrever Usuário no Curso".

A modelagem de cada caso de uso foi realizada com a ferramenta Topcased [48]. A Topcased é uma ferramenta *CASE* desenvolvida para sistemas em desenvolvimento, inclusive para aplicações críticas. É uma ferramenta gratuita que permite uma modelagem fortemente focada em rastreabilidade dos artefatos do desenvolvimento do sistema, como também as simulações das modelagens. Seu maior problema é a alta curva de aprendizagem inicial, porém, uma vez que se aprende, torna-se uma grande aliada para o desenvolvimento de *software*. A escolha dessa ferramenta deve-se à animação do modelo, pois ocorre de uma forma dinâmica e interativa, o que facilita a compreensão do comportamento do modelo. A animação é feita com cores e uma tabela de eventos, na qual o usuário escolhe o próximo evento que irá ocorrer e a ferramenta mostra o efeito desse evento.

6.2.3 Validação dos Modelos

Conforme cada modelo era construído com a ferramenta, era validado, e se ocorressem erros, estes eram corrigidos no modelo e a animação ocorria novamente. A Tabela 6.2 mostra o número de vezes que cada modelo foi validado: Um ponto de verificação importante nesta tabela é que o número de validações caiu conforme o especialista ficava mais acostumado com o sistema, suas características e a forma como modelá-lo. Ou seja, o modelo M1 que possui uma quantidade inferior de estados foi validado mais vezes em relação ao modelo M4. O modelo M1 foi modelado anteriormente ao modelo M4 e as diretrizes ainda estavam sendo compreendidas no momento da modelagem. Outro fator interessante é que a complexidade do modelo também não é um fator relevante para ocorrer erro na modelagem, pois o modelo M4 é o mais complexo em relação aos outros e foi

o que teve que ser validado menos vezes.

Apesar de o último modelo ter sido validado apenas uma vez, os outros precisaram de mais de uma interação de validação, pois continham erros na modelagem, o que mostra que a animação é muito útil na verificação do comportamento e na correção do modelo. Por meio dela foi possível verificar que havia algum comportamento inesperado ou faltante e também transições que faltavam ser inseridas no modelo.

6.2.4 Geração dos Casos de Teste Abstratos

Após a validação de cada modelo, a ferramenta ModelJUnit [49] foi utilizada para gerar os testes abstratos.

A ModelJUnit é, na verdade, uma biblioteca que estende o conceito de testes de unidade da linguagem Java, o JUnit, para testes baseados em modelo. É uma ferramenta gratuita, que permite escrever máquinas de estados finitas simples ou estendidas como classes da linguagem Java, gerando casos de testes e métricas de cobertura do modelo (como cobertura de transições e estados). A Figura 6.3 mostra um exemplo do trecho de uma das máquinas descritas para uso na ferramenta ModelJUnit. A entrada da ferramenta, portanto, é a máquina de estados descrita na linguagem Java, e a saída é um conjunto de casos de teste.

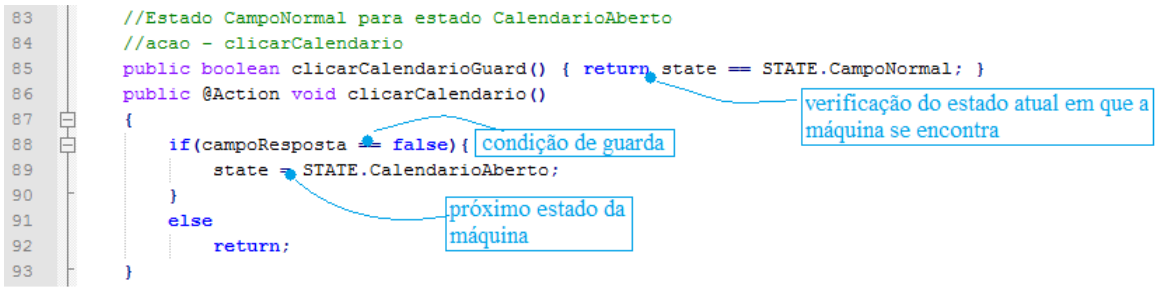


Figura 6.3: Exemplo de trecho de código de uma máquina de estados na ferramenta ModelJUnit.

Para gerar os testes, a ModelJUnit disponibiliza três algoritmos:

- *Random Walk*: percorre o modelo de forma aleatória, gerando caminhos aleatórios;
- *Greedy Walk*: também utiliza geração aleatória de caminhos, porém, a cada novo caminho a preferência será de uma transição não utilizada até o momento;
- *Lookahead Walk*: percorre o modelo avaliando N-níveis do mesmo de ações ou transições mais vantajosas.

Tabela 6.3: Casos de teste abstratos gerados pela ferramenta ModelJUnit.

Modelo	#Casos de teste abstratos
M1: Solicitar Inscrição	4
M2: Inscrever Usuário no Curso	8
M3: Atualizar Agenda	9
M4: Atualizar Material de Apoio	26

Para este trabalho, foi utilizado o algoritmo *Greedy Walk*, para obter uma cobertura de todas as transições do modelo, onde a preferência da escolha da próxima da transição é para aquela que ainda não foi escolhida. A garantia da cobertura foi possível com a métrica de cobertura de transições que a ModelJUnit disponibiliza no relatório de testes gerados.

É importante ressaltar aqui que todos os caminhos gerados foram verificados de forma manual para saber se todos eram válidos. Ou seja, para ter a certeza de que não havia caminhos infactíveis gerados. Isso ocorreu pois, apesar da ModelJUnit permitir o uso de condições de guarda no código, não foi possível verificar como são tratadas essas condições de guarda pela ferramenta, não garantindo, dessa forma, que um caminho impossível de ocorrer não fosse gerado pela ferramenta.

A Figura 6.4 mostra um exemplo de saída da ferramenta ModelJUnit, ou seja, os casos de teste.

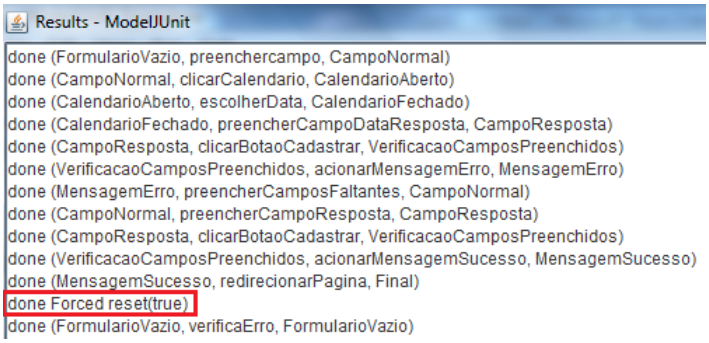


Figura 6.4: Exemplo de saída da ferramenta ModelJUnit. A linha *Forced reset (true)* separa o caso de teste anterior do próximo.

A Tabela 6.3 mostra a quantidade de casos de teste abstratos gerados para cada modelo feito do sistema TelEduc.

Tabela 6.4: Quantidade de casos de testes instanciados.

Modelo	#Casos de teste instanciados
M1: Solicitar Inscrição	18
M2: Inscrever Usuário no Curso	20
M3: Atualizar Agenda	13
M4: Atualizar Material de Apoio	34

Com relação a complexidade de cada modelo (métrica CCS), pode ser observado que pode haver uma relação entre complexidade do modelo e quantidade de casos de teste gerados, onde quanto maior a complexidade, maior o número de casos de teste abstratos gerados. Como não há uma quantidade suficiente de dados para comparação, essa relação não pode ser realmente afirmada. Porém, como observado, no modelo M1, de CCS igual a 34, temos 4 casos de teste, e em M4, um modelo mais complexo que M1, de CCS igual a 138, tem-se 26 casos de teste abstratos.

6.2.5 Instanciação e Concretização dos Casos de Testes

Para cada caso de teste abstrato, foi utilizada a técnica de valores limites para instanciar os casos de testes, como explicado nos capítulos de Testes Baseados em Modelos e Metodologia. A Tabela 6.4 mostra a quantidade de casos de testes instanciados criados. Nessa situação, o que influencia na quantidade de casos testes instanciados não é a complexidade do modelo, nem seu tamanho, e sim, a quantidade de entradas de dados possíveis para os estados. O modelo M3 possui mais botões do que campos de entrada de dados, como uma caixa de texto. O que não ocorre em M2, que é um formulário, ou seja, mais campos para menos botões. Sendo assim, há mais valores limites e partições para M2 do que para M3. Entretanto, o tamanho do modelo também pode influenciar nas possibilidades de caminhos diferentes que podem ser percorridos no sistema, criando mais casos de teste, como foi observado nos casos de testes abstratos.

Após a instanciação dos casos de testes, houve a concretização destes. Devido às condições de tempo para a realização do trabalho, alguns casos de testes foram projetados para serem executados de forma automática e outros, a maioria, de forma manual. Os casos de testes concretos para automatização são *scripts* que foram criados para serem executados na ferramenta Selenium [50]. Essa ferramenta possui duas versões: *plugin* para o navegador Mozilla Firefox e uma biblioteca que pode ser adicionada em várias linguagens, como Java, C++ etc. Nesse trabalho, foi utilizada a Selenium IDE – *plugin* para o navegador Mozzila Firefox. A Figura 6.5 mostra um exemplo desse script.

Na Figura 6.5, pode ser observado que o *script* Selenium segue uma série de passos; cada um com um comando que faz com que a Selenium consiga interpretar e executar

na interface do sistema como se fosse realmente um usuário. No script, é possível colocar verificações e assertivas. Uma verificação ocorre quando apenas desejar verificar uma determinada característica, sem que essa seja, necessariamente, a saída esperada após todos os comandos. No caso de uma assertiva, essa representa a saída final esperada de uma série de comandos. Por exemplo, em um teste em que se clica no link "*login*" e deve ser aberta uma caixa para colocar os dados de *login*, se o *login* for realizado com sucesso, aparece a mensagem de que o usuário está no sistema. Nesse caso, ocorre uma verificação para saber se houve a abertura da caixa para dados, e há uma assertiva para se verificar se o *login* foi efetuado ou não com sucesso. No exemplo da figura, ocorre apenas uma assertiva.

Como resultado final, o *script* da Selenium retorna se o caso de teste passou ou não passou, que é possível ser verificado no canto esquerdo da tela.

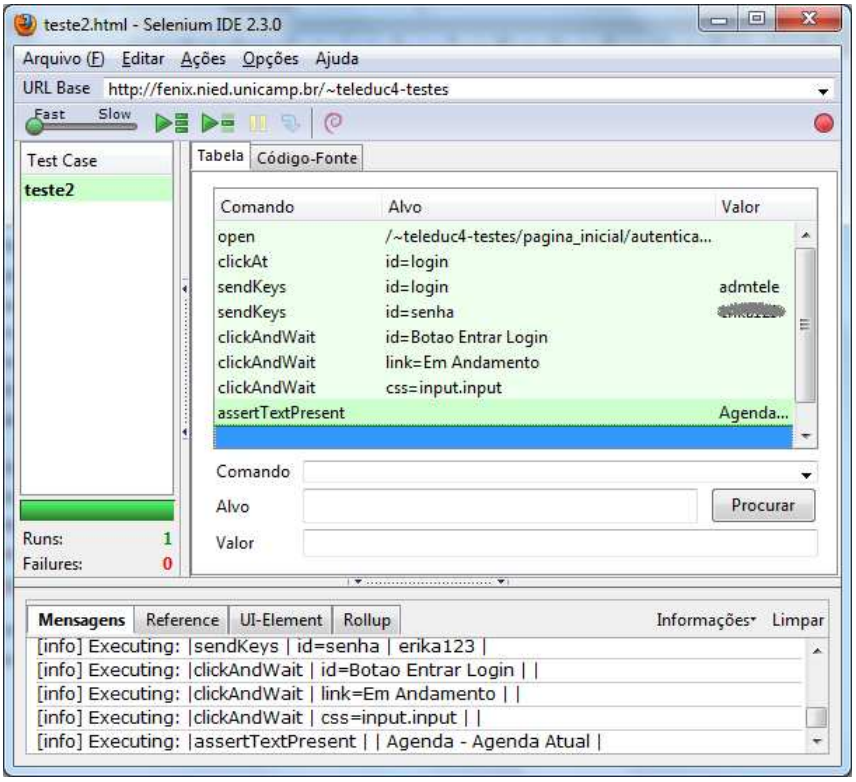


Figura 6.5: Exemplo de *script* da ferramenta Selenium.

Como explicado, a maioria dos testes foram executados de forma manual. Nesse caso, utilizou-se planilhas para organização dos testes, como explicado no capítulo Metodologia. A Figura 6.6 mostra um dos testes criados para o modelo M1.

# Caso de Teste	Evento	Parâmetros	Valores	Saída Esperada
1	preencherCampo	login	Caroline	pagina não se altera
		senha	Caroline	
		confirmacaoSenha	Caroline	
		Nome	Caroline Letizio	
		RG	43990562-X	
		sexo	feminino	
		email	caroline.letizio@gmail.com	
		telefone		
		endereco	Rua Jose	
		cidade	São Paulo	
		estado	SP	
		profissao	estudante	
		local de trabalho	faculdade	
		escolaridade	superior	
		informacoes adicionais	Oi	
	clicarCalendario			abrir calendário
	escolherData	data	18/05/1987	selecionar data e fechar calendário
	preencherCampoResposta		6	
	clicarBotaoCadastrar			dados preenchidos corretamente
	acionarMensagemSucesso			aparecer mensagem de sucesso
	redirecionarPaginaLogin			redirecionar a página

Figura 6.6: Exemplo de planilha com os casos de teste instanciados.

6.3 Resultados dos testes realizados pelo Analista de Testes

Após a execução dos testes, fez-se a coleta de dados e a análise dos erros. É interessante ressaltar que foram encontrados erros no sistema TelEduc de duas formas: enquanto o modelo era criado a partir dos casos de uso e da interface do usuário e pela execução dos testes. A Tabela 6.5 mostra esses resultados. Os defeitos encontrados durante a modelagem são, na maioria, erros de interface do usuário. Esse tipo de defeito, não necessariamente, é encontrado por um teste automatizado. Também, não necessariamente, encontrado apenas durante a modelagem, e sim, por execução manual. Porém, como ocorreram erros durante a modelagem, mostra-se outro ponto positivo da modelagem ser conferida com a interface, pois mais defeitos podem aparecer.

Um exemplo de defeito encontrado enquanto o modelo era criado foi na ferramenta Agenda: ao excluir uma agenda anterior, voltar para a tela inicial e criar uma agenda nova, a caixa de título da nova agenda aparece no final da página, quando o certo é aparecer logo abaixo do botão “Nova Agenda”. A Figura 6.7 representa essa falha.

Nos testes executados manualmente, foram encontrados vários tipos de defeitos, alguns graves, por exemplo, a ferramenta Agenda, ao escolher a opção de excluir uma agenda e confirmar a mensagem, a página é direcionada para outra com erro “página não encontrada, erro 404”, ao invés de voltar para a página que mostra todas as agendas. Para

Tabela 6.5: Resultados da execução dos testes encontrados pela metodologia.

Modelo	Identificador do total de defeitos	Defeitos modela-gem	Defeitos testes	Total
M1: Solicitar Inscrição	EM1	1	2	3
M2: Inscrever Usuário no Curso	EM2	4	1	5
M3: Atualizar Agenda	EM3	1	6	7
M4: Atualizar Material de Apoio	EM4	3	2	5

Tabela 6.6: Defeitos reportados pela equipe do TelEduc.

Ferramenta	Identificador do total de defeitos	Defeitos reportados
M1: Solicitar Inscrição	ER1	1
M2: Inscrever Usuário no Curso	ER2	1
M3: Atualizar Agenda	ER3	1
M4: Atualizar Material de Apoio	ER4	0

saber se os defeitos encontrados pelo analista haviam sido ou não reportados pela equipe TelEduc, foram levantados os defeitos já reportados pela equipe através do acesso autorizado ao ambiente de gerenciamento de bugs da equipe. A Tabela 6.6 mostra os defeitos reportados pela equipe das mesmas ferramentas testadas e da mesma versão do sistema.

Da ferramenta Agenda, o erro ER1 que havia sido reportado pela equipe, registrado no sistema de gerenciamento de bugs, foi o mesmo encontrado pelos testes executados e está entre os encontrados em EM1. Já os erros reportados ER2 e ER3 não haviam sido detectados pelos testes, portanto não estão entre EM2 e EM3, pois os erros de ER2 e ER3 eram a verificação do ".com" no e-mail. Foi verificado que, nos testes criados pelo analista de teste, não havia sido considerado, como uma possível entrada, um e-mail sem o ".com" no final. Sendo assim, os erros de ER2 e ER3 não foram detectados por EM2 e EM3.

6.4 Aplicação e Resultados da Metodologia pela equipe TelEduc

Toda a equipe do TelEduc, composta de quatro pessoas, concordou em participar do *workshop* organizado para ensinar a metodologia. Porém, devido ao tempo e às atividades dos participantes, três fizeram apenas até a validação do modelo e um seguiu até o final com a metodologia, porém, conseguiu apenas criar parte dos testes e aplicá-los. Mas

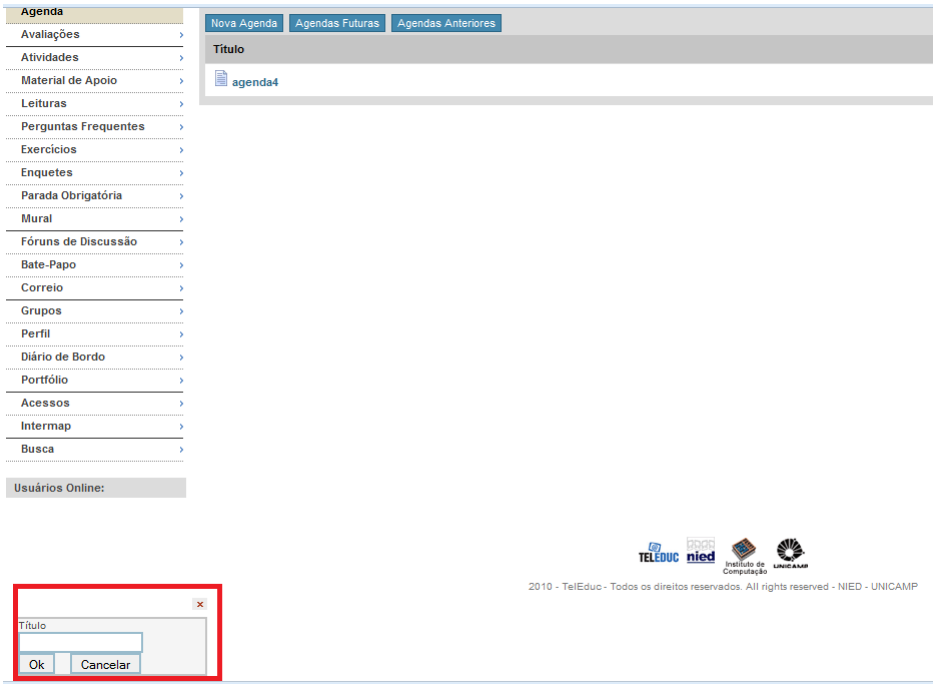


Figura 6.7: Erro da caixa de título para criar uma nova agenda.

muitos resultados puderam ser obtidos.

O *workshop* criado teve como objetivo ensinar modelo de estados da UML e a instrução das ferramentas utilizadas para validar os modelos e gerar os testes. Foram necessários encontros para ensinar sobre a modelagem, pois os participantes são alunos da graduação que, ou não tiveram a disciplina de Engenharia de Software ainda, ou porque não haviam aprendido sobre modelo de estados. Apenas um participante já havia visto esse tipo de modelagem, mas não na área de computação, e sim, na área da engenharia elétrica. Foram realizados nove encontros, sendo que os quatro primeiros foram para instrução da modelagem, o quinto para validação do modelo, o sexto para geração de testes abstratos, o sétimo para geração dos testes executáveis. O oitavo e nono encontros foram realizados para eles escolherem uma ferramenta do TelEduc da lista de casos de uso mais importantes, modelar essa ferramenta, validar, gerar e aplicar os testes. Como o *workshop* foi organizado nos horários possíveis dos participantes, ficou muito restrita a aplicação total da metodologia. Não foi aplicada a metodologia já diretamente no ambiente de desenvolvimento do sistema TelEduc pois cada participante possui um papel na equipe, sendo a responsável por teste apenas uma única pessoa. Porém, como todos os participantes mostraram interesse em aprender a metodologia, foi aplicado o *workshop* a todos.

Ao final, um questionário foi passado para avaliar a metodologia. Esse questionário possuía perguntas como, o grau de dificuldade de cada etapa, como avaliaram a metodolo-

Tabela 6.7: Resultados gerados pela equipe do TelEduc.

Participante	#Validações modelo	#Erros na modelagem	Erros na execução	Erro já reportado pela equipe?
1	2	0	1	Não
2	1	1	0	Não
3	2	1	0	Não
4	1	1	0	Não

Tabela 6.8: Avaliação da metodologia pela equipe do TelEduc.

Participante	Nível da modelagem e validação	Nível da geração dos CTAs	Nível da geração dos testes executáveis	Nível da metodologia no geral	A metodologia e útil para o processo de testes?
1	Difícil	Fácil	Médio	Médio	Sim
2	Médio	Fácil	Fácil	Médio	Sim
3	Fácil	Médio	Médio	Médio	Sim
4	Médio	Difícil	Médio	Médio	Sim
Total	Médio	Fácill	Médio	Médio	Sim

gia no todo, se houve importância na validação do modelo, se foram encontrados defeitos durante a modelagem e se foi encontrado defeito durante a execução dos testes.

Sobre a validação do modelo, todos os participantes encontraram erros em suas modelagens através da animação do modelo, corrigindo-os e refazendo a animação. Dois participantes afirmaram que tiveram que animar duas vezes seus modelos e os outros dois, apenas uma.

Três participantes encontraram erro durante a modelagem. O participante que executou os testes (foram 8 testes executados) encontrou 1 erro de execução. Todos os participantes afirmaram que seus erros não haviam sido reportados previamente na ferramenta de gerenciamentos de erros da equipe.

A Tabela 6.7 resume os resultados descritos acima.

Na avaliação da metodologia e de suas partes, foi pedido para classificar entre fácil, médio e difícil a dificuldade de cada passo. Depois, eles avaliaram se acharam útil a metodologia para o processo de testes. A Tabela 6.8 sumariza esses resultados.

6.5 Análise dos Resultados

Com os resultados obtidos tanto pelo analista de testes como pela equipe do TelEduc, foi possível mostrar que a validação do modelo é muito útil e eficaz para avaliar se o modelo realmente representa o comportamento do sistema e se irá gerar testes condizentes com o programa. A metodologia, de forma geral, foi aprovada e obteve bons resultados, pois apenas 1 dos defeitos encontrados já havia sido reportado pela equipe de desenvolvimento, sendo encontrado um total de 20 defeitos em apenas 4 casos de uso, sendo que o sistema conta com 58 casos de uso.

Um dos fatores que ajudou a encontrar defeitos foi o uso de critérios de testes funcionais, como partição e valores limites para instanciar os testes. Com isso, foi possível testar entrada de dados inesperados pelo sistema, mas que têm alta porcentagem de chances de ocorrer. Inclusive, isso foi um dos comentários do participante que encontrou o defeito com a execução dos testes, pois foi no uso de caractere especial que ele encontrou o defeito.

Foi possível observar, também, a importância do conhecimento da modelagem para saber como interpretar o sistema para modelá-lo, assim como validar a semântica do modelo. Como reportado por Robinson [2], e descrito no capítulo de Introdução, testes baseados em modelo é uma tecnologia não muito utilizada e ainda incomum para testadores. A dificuldade está, justamente, na modelagem do sistema. Mesmo com a explicação do passo a passo da modelagem, é importante que o responsável pelo modelo obtenha conhecimentos, estudando máquina de estados para ter mais noção do que é possível realizar ou não e, também, adquirir maior experiência para poder modelar algo mais complexo, como aplicações Web com elementos dinâmicos.

Outro aspecto que pode afetar o bom desempenho da metodologia é o tempo que se tem para testar o sistema. Se a pessoa responsável pela modelagem e pelos testes possui experiência, isso não será problema. Porém, para pessoas que ainda não têm familiaridade com modelo de estados, e precisa aprender um pouco mais, a modelagem pode demorar um pouco. Mas, o tempo leva à experiência, que leva à eficiência da aplicação do método.

Sendo assim, mesmo com pouco tempo para ocorrer a aplicação da metodologia pela equipe do TelEduc com o acompanhamento da analista de teste, a pessoa responsável pela parte de testes do sistema TelEduc reportou que irá aplicar a metodologia no seu processo de testes para melhorar a qualidade da aplicação.

Dessa forma, para aplicar a metodologia para a rotina de testes, é importante o uso de ferramentas que auxiliem nesse processo. Duas sugestões seriam para gerenciamento de testes e bugs e para execução automática dos testes. No caso de gerenciamento, existem boas ferramentas gratuitas, como Testlink [51]. Para o caso de bugs, existe a Bugzilla [52] e Mantis [53]. Há ferramentas pagas, como o Jira [54]. Para execução automática de testes para aplicações Web, como já explicado, uma ferramenta muito utilizada é a

Selenium [50], com a qual é possível criar *scripts* e guardá-los para os testes de regressão. Uma vez os *scripts* prontos, é possível guardá-los para realizar testes de regressão.

Capítulo 7

Conclusões e Trabalhos Futuros

Este capítulo apresenta as conclusões tiradas a partir do Estudo de Caso, e depois, os objetivos para os trabalhos futuros.

7.1 Conclusões

A proposta deste trabalho foi mostrar o método de teste baseado em modelo aplicado a sistemas Web. Devido a uma das principais dificuldades estar na modelagem, os objetivos foram: apresentar o modelo de estados, levantar as características dos sistemas atuais da Web, realçar o que deve ser considerado para o modelo, além de indicar como colocar as características consideradas no diagrama de estados. Utilizou-se o diagrama de estados da UML, devido à sua facilidade e maior potencial para modelagem e menor limitação nas representações do sistema, ao contrário do que ocorre nos modelos de estado clássicos, que não permitem o uso de condições de guarda. A partir do modelo pronto e validado semanticamente, para determinar se estava de acordo com o comportamento do sistema, os casos de testes foram gerados a partir dos modelos, instanciados com dados e aplicados ao sistema.

No estudo de caso, o método foi aplicado por um analista de testes, que propôs este método, e pela equipe do ambiente de ensino à distância TelEduc, que foi o sistema utilizado para avaliar a proposta deste trabalho. A partir dos resultados obtidos, tornou-se evidente a importância da validação dos modelos, pois todos foram revisados mais de uma vez. Além disso, a proposta encontrou mais defeitos que a técnica utilizada pela equipe TelEduc, a qual era totalmente *ad hoc*. Assim, foi possível observar a diferença entre uma equipe de qualidade usar ou não uma técnica de testes, com a qual, com a organização dos testes, há melhores resultados para a qualidade do *software*.

Dentre as dificuldades encontradas no método proposto, está a modelagem, fato ocorrido no início dos *workshops* com a equipe do TelEduc. Porém, com as explicações, exem-

plificações e treinos realizados, a modelagem se tornou mais fácil, com a equipe propondo outras formas de se modelar com o diagrama de estados da UML. Outra dificuldade foi o uso da ferramenta Topcased [48], ferramenta *CASE* utilizada para modelagem e animação do modelo, devido à pouca documentação existente. Porém, com o uso frequente desta ferramenta, o processo passou a não ser tão complicado, e com a familiaridade é muito provável que sua utilização deixe de ser realmente um problema.

Com os resultados obtidos e a partir da análise feita na seção Análise de Resultados do capítulo 6, e pelo sistema do estudo de caso ser real e utilizado no mercado, o método se mostrou viável e possível de ser aplicado em outros ambientes e *software* Web.

7.2 Trabalhos Futuros

Ao se desenvolver o método, muitos detalhes foram observados, mostrando aspectos importantes que precisam ser considerados. Entre eles, a necessidade da atualização dos casos de uso e de outras documentações do *software*. Com o caso de uso atualizado, é necessário verificar a cobertura de testes sobre ele para garantir que todos os requisitos do sistema foram atendidos. Outro detalhe é a baixa oferta de ferramentas que auxiliam a validação dos modelos utilizando animação ou simulação do modelo. Também houve dificuldade em encontrar ferramentas para gerar casos de testes abstratos, pois algumas eram muito difíceis de serem utilizadas e outras não eram completas ou não aceitavam máquina de estados da UML, levando à desistência de sua aplicação pelo usuário.

Uma verificação importante que deve ser realizada é o problema da explosão de estados do modelo. Apesar do estudo de caso deste trabalho não ter apresentado esse problema, é sabido que este problema é inevitável, quando se testam sistemas ou páginas mais complexas. A forma encontrada para reduzir este problema foi considerar modelos por caso de uso, e não para o sistema completo. Uma extensão do trabalho, portanto, é o uso de outras técnicas que evitem esse problema.

Complementando a referência às ferramentas de gerenciamento e execução automática de testes, planeja-se a demonstração destas, explicando seu uso. No final, espera-se criar um *framework* que modele, valide, gere testes e os execute, tornando o ciclo da técnica de teste baseado em modelo para sistemas Web completo e automático.

Referências Bibliográficas

[1] D. C. Kung, C-H. Liu, and P. Hsia, "An object-oriented Web test model for testing Web applications", Proceedings First AsiaPacific Conference on Quality Software, 2000, pp. 537–542.

[2] H. Robinson, "Obstacles and opportunities for model-based testing in an industrial software environment", 1st European Conference on Model-Driven Software, 2003.

[3] E. R. C. Almeida, "Geração Automática de Casos de Testes Executáveis a partir de Casos de Teste Abstratos para Aplicações Web", Universidade Estadual de Campinas, 2012.

[4] R. Mohan, M. A. Cohen, and J. Schiefer, "A State Machine Based Approach for a Process Driven Development of Web-Applications", in 14th International Conference, CAiSE 2002 Toronto, Canada, May 27–31, 2002 Proceedings, 2002, pp. 52–66.

[5] P. Baker, D. Zhen, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams, "Model-Driven Testing Using the UML Testing Profile", in in Model-Driven Testing Using the UML Testing Profile, New York, New York, USA: Springer Berlin Heidelberg, 2007, pp. 7–13.

[6] S. R. Dalal, a. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice", Proceedings of the 21st international conference on Software engineering - ICSE '99, vol. 1999, no. May, 1999, pp. 285–294.

[7] K. R. P. H. Leung, L. C. K. Hui, S. M. Yiu, and R. W. M. Tang, "Modeling Web navigation by statechart", Proceedings 24th Annual International Computer Software and Applications Conference. COMPSAC2000. IEEE Comput. Soc, 2000, pp. 41–47.

[8] OMG, "UML." [Online]. Available: http://www.uml.org/.

[9] A. Kalaji, R. M. Hierons, and S. Swift, "A Search-Based Approach for Automatic Test Generation from Extended Finite State Machine (EFSM)", 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques, 2009, pp. 131–132.

[10] M. Winckler and P. Palanque, "StateWebCharts: A formal description technique dedicated to navigation modelling of web applications", in in Interactive Systems. Design, Specification, and Verification, A. J. Jorge, J. N. Nunes, and J. F. Cunha, Eds. Springer Berlin Heidelberg, 2003, pp. 61–76.

[11] A. a. Andrews, J. Offutt, C. Dyreson, C. J. Mallery, K. Jerath, and R. Alexander, "Scalability issues with using FSMWeb to test web applications", Information and Software Technology, Jan. 2010, vol. 52, no. 1, pp. 52–66.

[12] R. S. Pressman, "Software Engineering A Practitioner's Approach", Seventh Edition. New York, New York, USA: McGraw-Hill, 2010, p. 930.

[13] M. M. I. Pattal, Y. Li, and J. Zeng, "Web 3.0: A Real Personal Web! More Opportunities and More Threats", in 2009 Third International Conference on Next Generation Mobile Applications, Services and Technologies, 2009, pp. 125–128.

[14] S. S. C. Shang, Y.-L. Wu, and O. C. L. Hou, "An Analysis of Business Models of Web 2.0 Application", in 2009 Sixth International Conference on Information Technology: New Generations, 2009, pp. 314–319.

[15] G. S. Chakravarthy and R. R. Barde, "Web 2.0 applications - an approach to testing", in 2008 International Conference on Innovations in Information Technology, 2008, pp. 195–198.

[16] C. Larman, "Utilizando UML e Padrões", Terceira Edição. Porto Alegre: Bookman, 2007, p. 696.

[17] F. L. F. Almeida and J. M. R. Lourenço, "eCreation of Value with Web 3.0 Technologies", in 6th Iberian Conference on Information Systems and Technologies (CISTI), 2011, pp. 1–4.

[18] T. Berners-Lee and J. Jaffe, "World Wide Web Consortium (W3C)", [Online]. Available: http://www.w3.org/. [Accessed: 02-May-2013].

[19] M. Tatsubori and K. Takahashi, "Decomposition and Abstraction of Web Applications for Web Service Extraction and Composition", in Web Services, 2006. ICWS '06. International Conference, 2006, pp. 859–868.

[20] M. H. Alalfi, J. R. Cordy, and T. R. Dean, "Modeling methods for web application verification and testing State of the art", Software Testing, Verification And Reliability, 2009, vol. 19, no. 4, pp. 265–296.

[21] A. a. Andrews, J. Offutt, and R. T. Alexander, "Testing Web applications by modeling with FSMs", Software And Systems Modeling, Jan. 2005, vol. 4, no. 3, pp. 326–345.

[22] R. V. Binder, "Testing Object-Oriented Systems Models, Patterns, and Tools", First Edition. Addison-Wesley Professional, 2000, p. 1191.

[23] D. Harel and E. Gery, "Executable Object Modeling with Statecharts", Computer, 1997, vol. 30, no. 7, pp. 31–42.

[24] OMG, "OMG Unified Modeling Language TM (OMG UML), Superstructure", Object Management Group (OMG®), 2009, pp. 541–603.

[25] A. R. C. Rocha, J. C. Maldonado, and K. C. Weber, "Qualidade de Software: Teoria e Prática". Texas: Prentice Hall, 2001, p. 303.

[26] I. Sommerville, "Software Engineering", Eighth Edition. England: Addison-Wesley Publishers Limited, 2007, p. 865.

[27] M. E. Delamaro, J. C. Maldonado, M. Jino, "Introdução ao Teste de Software", Primeira Edição. Elsevier, 2007, pp 1-7.

[28] C. Kaner, "What Is a Good Test Case?", in Software Testing Analysis And Review Conference (STAR), 2003, pp. 1–16.

[29] A. Hartman, K. A., and K. Nagin, "A Test Execution Environment Running Abstract Tests for Distributed Software", in Software Engineering and Applications (SEA), 2002, pp. 5–10.

[30] "Pairwise Testing", [Online]. Available: http://www.pairwise.org. [Accessed: 07-May-2013].

[31] M. Utting, A. Pretschner, and B. Legeard, "A Taxonomy of Model-Based Testing", Hamilton, 2006, pp. 1–18.

[32] S. Rosaria and H. Robinson, "Applying models in your testing process", Information and Software Technology, vol. 42, no. 12. Microsoft Corporation, , Sep-2000, pp. 815–824.

[33] J. Conallen, "Modeling Web application architectures with UML", Communications of the ACM, vol. 42, no. 10, New York, New York, USA, 1999, pp. 63–70.

[34] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax -Based Web Applications through Dynamic Analysis" ACM Transactions on the Web, Mar. 2012, vol. 6, no. 1, pp. 1–30.

[35] A. Marchetto, F. Ricca, and P. Tonella, "A case study-based comparison of web testing techniques applied to Ajax web applications", International Journal on Software Tools for Technology Transfer, Oct. 2008, vol. 10, no. 6, pp. 477–492.

[36] L. Li, Q. Zhongsheng, and T. He, "Test purpose-based test generation for web applications", in 2009 First International Conference on Networked Digital Technologies, 2009, pp. 238–243.

[37] S. Chen, H. Miao, B. Song, and Y. Chen, "Towards Practical Modeling of Web Applications and Generating Tests", in 2010 4th IEEE International Symposium on Theoretical Aspects of Software Engineering, 2010, no. 2007, pp. 209–217.

[38] F. Bolis, A. Gargantini, M. Guarnieri, E. Magri, and L. Musto, "Model-Driven Testing for Web Applications Using Abstract State Machines", in ICWE 2012 International Workshops: MDWE, ComposableWeb, WeRE, QWE, and Doctoral Consortium, 2012, pp. 71–78.

[39] S. J. Mellor and M. J. Balcer, "Executable UML: A Foundation for Model-Driven Architecture". Boston: Addison-Wesley Longman Publishing Co., 2002, p. 368.

[40] "OMG - Object Management Group", [Online]. Available: http://www.omg.org.

[41] "Kieler." [Online]. Available: http://www.informatik.uni-kiel.de/rtsys/kieler/. [Accessed: 02-May-2013].

[42] "Xholon." [Online]. Available: http://www.primordion.com/Xholon/index.html. [Accessed: 02-May-2013].

[43] "MagicDraw." [Online]. Available: http://www.nomagic.com/. [Accessed: 02-May-2013].

[44] C. Rapp, "SMC - The State Machine Compiler", http://smc.sourceforge.net/.

[45] "Sparx Systems." [Online]. Available: http://www.sparxsystems.com/enterprise_architect_user_guide/9.3/model_simulation/model_simulation.html. [Accessed: 02-May-2013].

[46] B. Kitchenham, L. Pickard, and S. Pfleeger, “Case Studies for Method and Tool Evaluation” Software, IEEE, 1995, vol. 12, no. 4, pp. 52–62.

[47] S. Wagner and J. Jan, “Model-Based Identification of Fault-Prone Components”, in 5th European Dependable Computing Conference, 2005, pp. 435–452.

[48] “Topcased”, Aerospace Valley, http://www.topcased.org/.

[49] M. Utting, G. Perrone, J. Winchester, S. Thompson, R. Yang, and P. Douangsavanh, “ModelJUnit.” Department of Computer Science, University of Waikato, Waikato, New Zealand, 2009.

[50] J. Huggins, J. Bakken, A. Barantsev, E. Beans, D. Berrada, J. Bevam, T. Bhaduria, D. Burns, H.-B. Chais, J. Evans, D. Fabulich, P. Grandjean, A. Goucher, J. Harty, A. Hatvani, H. Philippe, D. Hunt, A. Kemparaj, O. Ifland, L. InmanSemerau, S. Kasatani, L. Laskin, J. Lebya, P. Lightbody, K. Menard, E. Messeri, R. Patterson, F. Reynaud, and K. Rosenvold, “SeleniumHQ.” Open QA, 2013.

[51] F. Mancardi, “TestLink - User Manual.” pp. 1–52, 2010.

[52] “Bugzilla.” Mozilla, http://www.bugzilla.org/.

[53] “Mantis Bug Tracker”, http://www.mantisbt.org/.

[54] “Jira”, Atlassian, https://www.atlassian.com/software/jira.

[55] ”TelEduc”, NIED, http://www.teleduc.org.br/.

Apêndice A

Modelagem de um caso de uso do TelEduc

Como descrito no capítulo Estudo de Caso, este apêndice mostra a modelagem de uma das ferramentas do sistema TelEduc, na forma de diagrama de estados, e a validação desse modelo. Por questões didáticas, será ilustrada apenas uma parte do modelo.

Retomando a explicação do capítulo 5, para criar um modelo é necessário analisar o caso de uso e as interfaces da ferramenta que esse caso de uso descreve.

O caso de uso exemplificado neste apêndice é "Inscrever Usuários no Curso". O trecho escolhido é o que descreve a inscrição dos formadores no curso, ou seja, inscreve as pessoas que serão responsáveis por este. A Figura A mostra parte da descrição desse caso de uso retirada da própria documentação do sistema TelEduc. As áreas destacadas da Figura A são referentes aos fluxos para inscrição de formadores.

As Figuras A.2, A.3, A.4, A.5 e A.6 representam as telas do sistema apresentadas para inscrever os formadores do curso.

Na seção "Criação dos Modelos" do capítulo Metodologia, é explicado como o caso de uso limita o modelo, ou seja, o modelo irá representar apenas o que está descrito naquele caso de uso. Assim, um ponto relevante para a modelagem é que a ação de gerenciar formadores, possível a partir do botão "Gerenciamento de Formadores", visível na figura A1, não será incluída na modelagem, pois existe um caso de uso específico para essa ação. Porém, como a tela de gerenciamento aparece como ação final da inscrição feita com sucesso, apenas o direcionamento a ela deve aparecer no modelo.

Com as definições abaixo retiradas do capítulo Metodologia, é possível identificar os elementos que devem aparecer no modelo:

Definição 1: *Um estado é toda estrutura dinâmica dentro de uma página e toda nova página que pode ocorrer dentro de uma funcionalidade do sistema.*

Definição 2: *Transições são os eventos e ações que alteram os estados e que fazem*

Fluxo Principal	P01. O usuário do sistema acessa um determinado curso P02. O usuário do sistema solicita o recurso "Inscrever Formadores". P03. O sistema apresenta um formulário requisitando o nome, e-mail e login de um ou mais formadores. P04. O usuário informa os dados requisitados e clica em "Inscrever". P05. O sistema realiza a inscrição deste usuário
Fluxos Alternativos	A01 [P02]. O usuário do sistema solicita o recurso "Inscrever Alunos". A01.1 O sistema apresenta um formulário requisitando o nome, e-mail e login de um ou mais alunos. A01.2 O fluxo alternativo retorna ao fluxo principal em P04 A01 [P02]. O usuário do sistema solicita o recurso "Inscrever Convidados". A02.1 O sistema apresenta um formulário requisitando o nome, e-mail e login de um ou mais convidados. A02.2 O fluxo alternativo retorna ao fluxo principal em P04 A03[P04]. O usuário clica em "Cadastrar por arquivo". A03.1 O sistema apresenta um formulário requisitando um arquivo CSV(Comma-Separated Values). A03.2 O usuário clica em "Enviar Arquivo". A03.3 O sistema ativa a caixa para seleção do arquivo. A03.4. O usuário clica em "Inscrever". A03.5. O sistema registra os usuários. A04[P04]. O usuário clica em "Usuários já cadastrados". A04.1 O sistema apresenta um formulário para busca de usuários.

Figura A.1: Parte do caso de uso "Inscrever usuários no curso".

ocorrer modificações no comportamento da página.

Assim, a partir das figuras das telas e do caso de uso, é possível notar que o formulário, os botões "Inscrever", "Cadastrar por Arquivo", "Gerenciamento de Formadores" e "Voltar" são elementos que alteram os comportamentos das páginas, portanto devem aparecer no modelo. Na tela de inscrição por arquivo, o botão "Selecionar Arquivo" e a saída obtida ao se pressionar esse botão também devem ser considerados na modelagem, assim como as telas de erro e a mensagem de sucesso. A Figura A.7 mostra o modelo criado.

A.1 Validação do Modelo

O modelo representado na Figura A.7 já está em sua versão final. Porém, antes de se chegar a ela, houve uma primeira versão, a qual estava incorreta. A Figura A.8 representa essa primeira versão.

O método abordado no capítulo 5 explica que, após o modelo estar pronto, uma validação do mesmo deve ser feita. A forma para se validar o modelo se dá por meio da simulação, na qual cada transição e estado é percorrido pelo menos uma vez, para verificar se o comportamento do sistema está sendo representado de forma correta. A partir da simulação do modelo da Figura A.8 com a ferramenta Topcased, foi possível verificar um

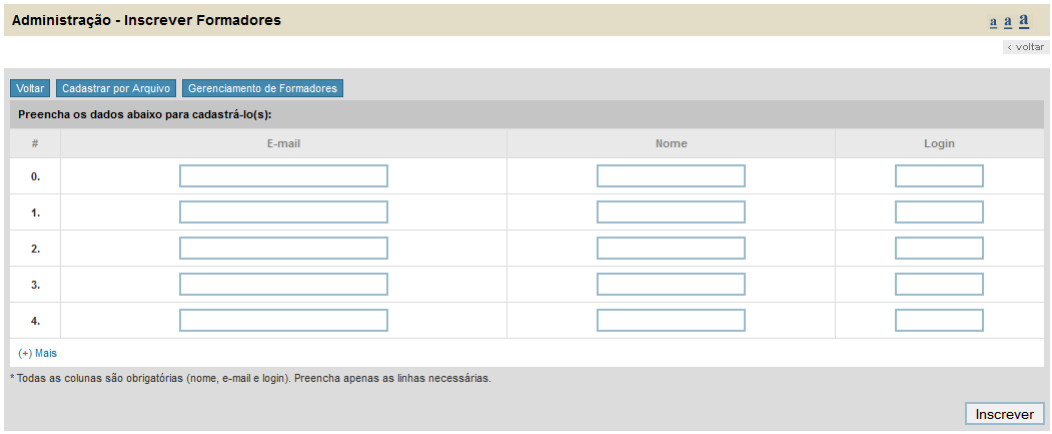


Figura A.2: Tela inicial da inscrição de formador.

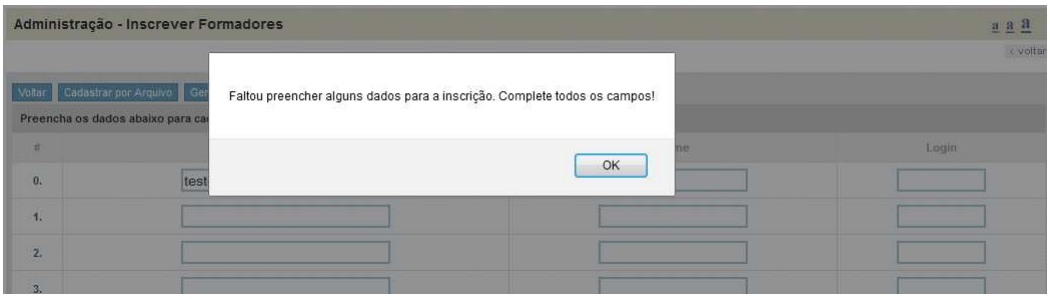


Figura A.3: Tela de erro para informação inválida em algum campo de preenchimento.

erro e chegar a uma nova versão. As Figuras A.9 e A.10 mostram parte da simulação. A Figura A.9 representa o modelo na tela com o formulário para inscrever formadores (Figura A.2). A Figura A.10 representa o momento que o usuário deve preencher os campos do formulário. Deve-se notar que há apenas a opção de apertar o botão "inscrever", se os campos estiverem corretos, sendo que não está demonstrado o que ocorre se algum campo estiver faltando ou errado, ou seja, falta o estado que representaria a tela da Figura A.3. Outro detalhe importante é o fato do caso de uso não descrever o que ocorre no caso de algum campo errado, portanto a validação também ajuda a atualizar e corrigir os casos de uso.

Sendo assim, foi adicionado um estado e suas transições para poder representar o comportamento do sistema, caso algum campo esteja errado ou faltando. Após essa correção, uma nova simulação foi feita, para poder verificar, novamente, se o modelo está representando o comportamento do sistema. A Figura A.11 mostra a validação do modelo com a correção feita. Dessa vez, a simulação mostrou que o modelo está de acordo com o esperado, sendo esta (Figura A.7) a versão final.

Figura A.4: Tela para inscrever formadores a partir de um arquivo de extensão csv.

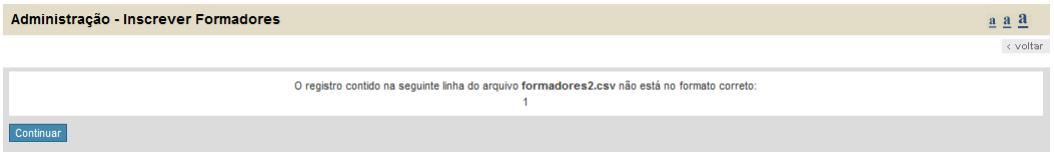


Figura A.5: Tela de erro se o arquivo csv não estiver correto.

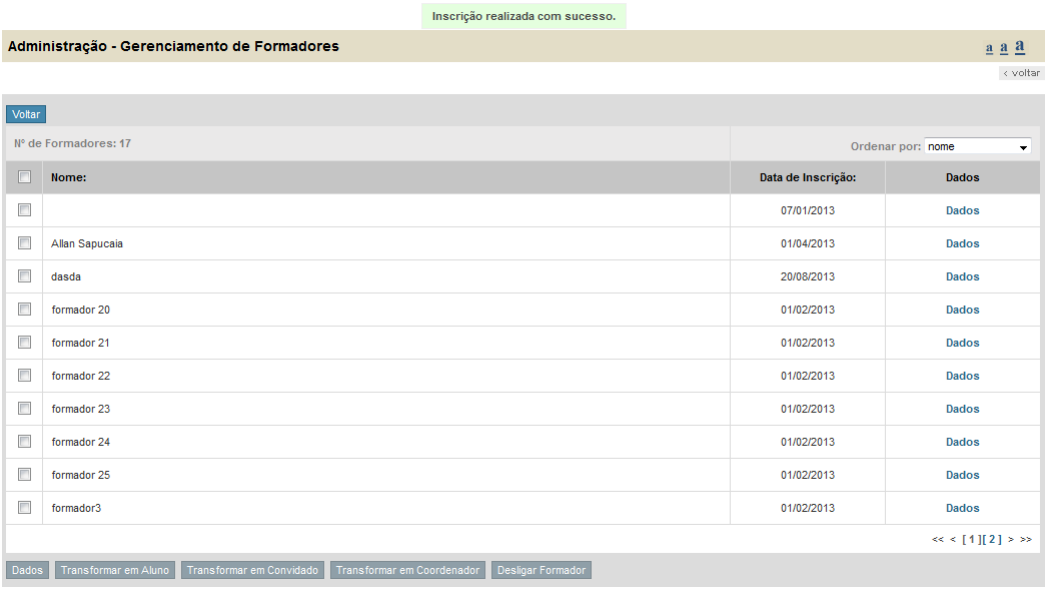


Figura A.6: Tela de sucesso de inscrição, tanto pelo formulário como pelo arquivo de extensão csv.

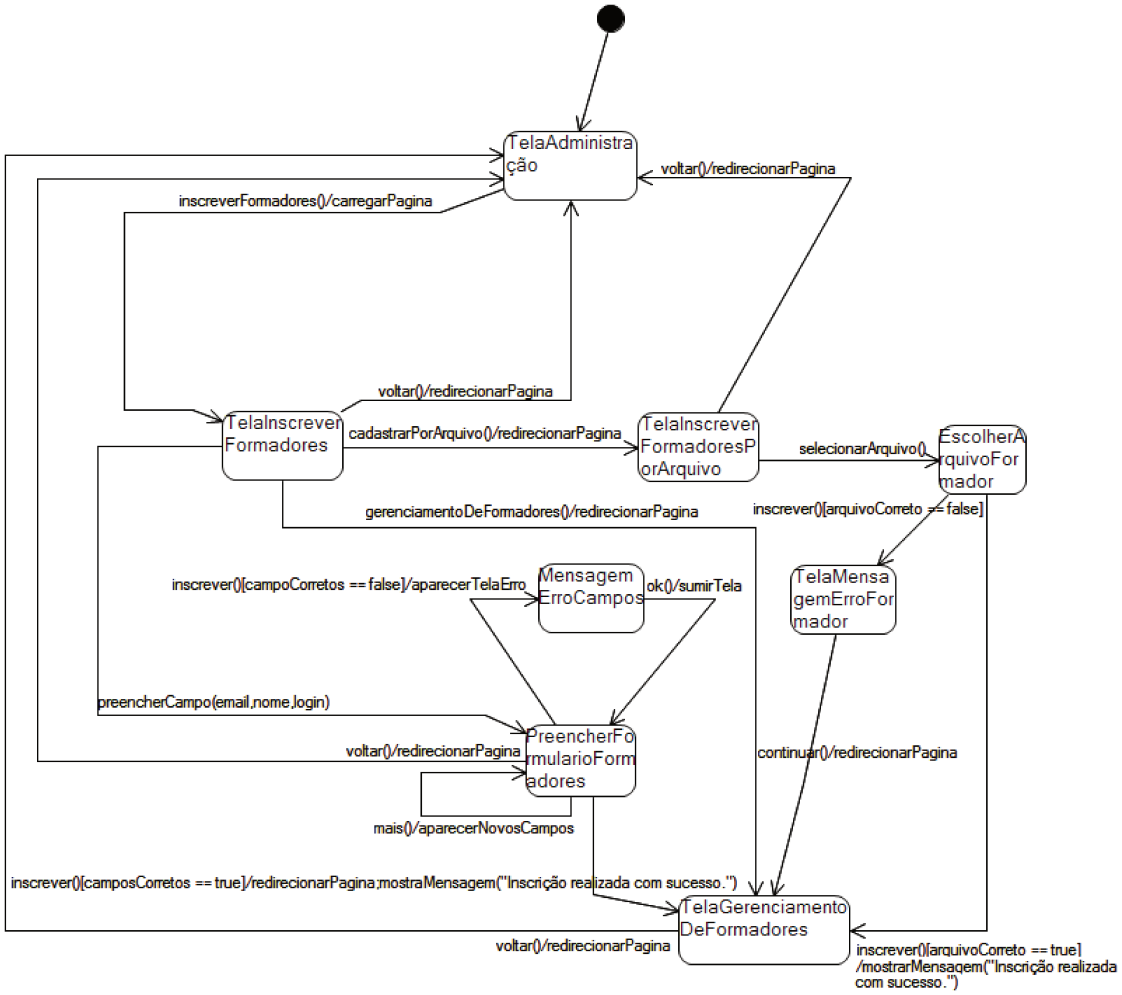


Figura A.7: Modelo de estados referente à inscrição de formadores no curso.

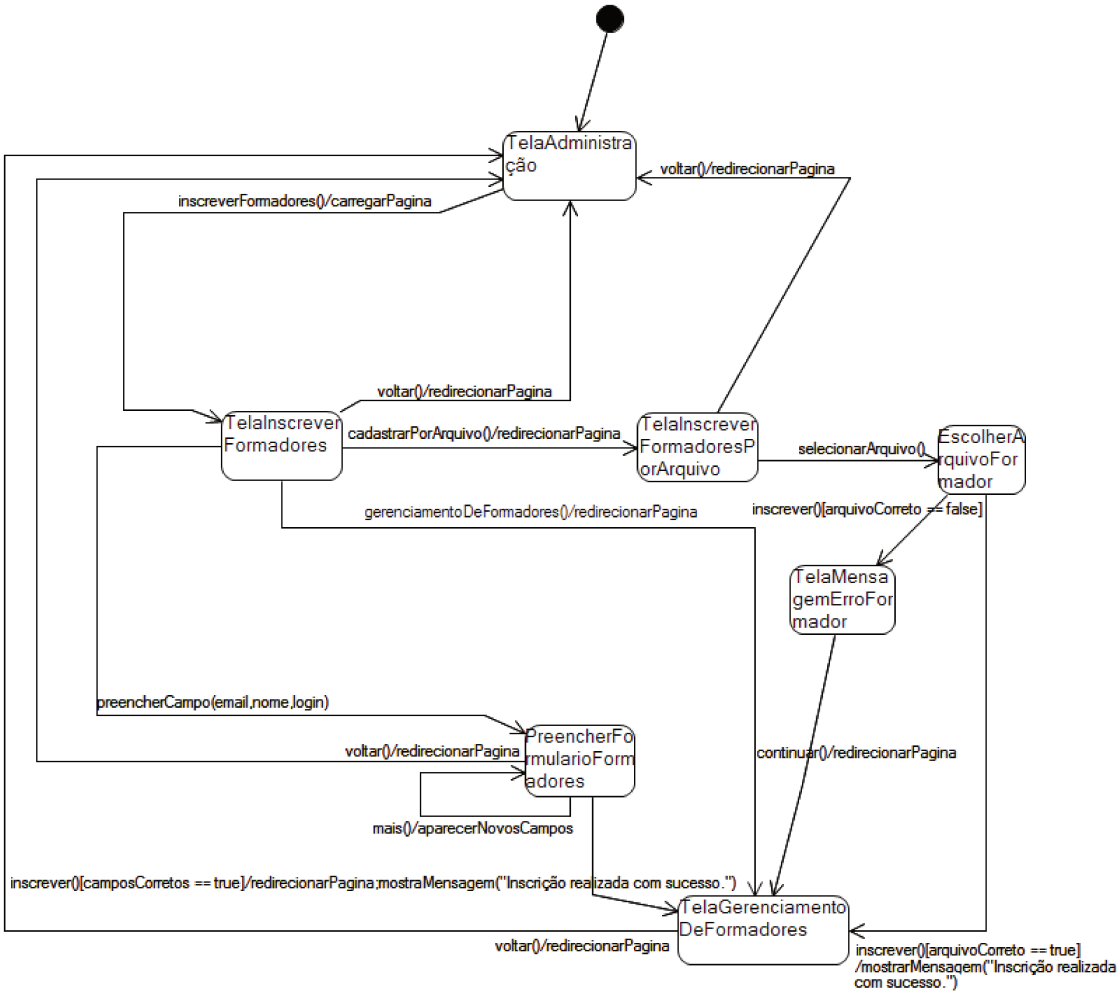


Figura A.8: Primeira versão do modelo de inscrição de formador no curso.

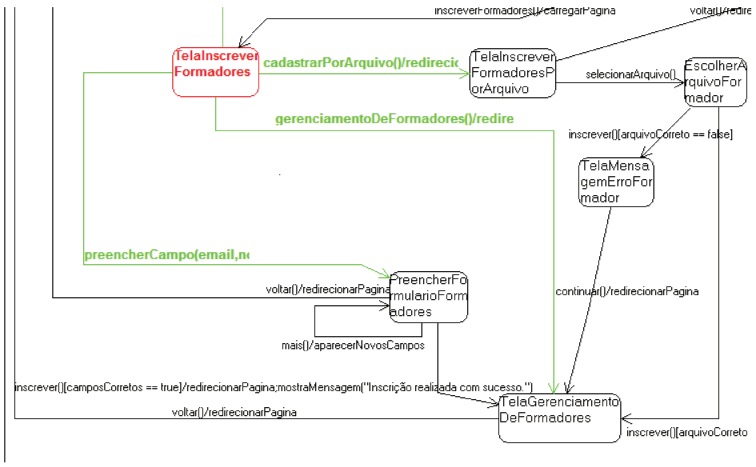


Figura A.9: Início da simulação do modelo da Figura A.8.

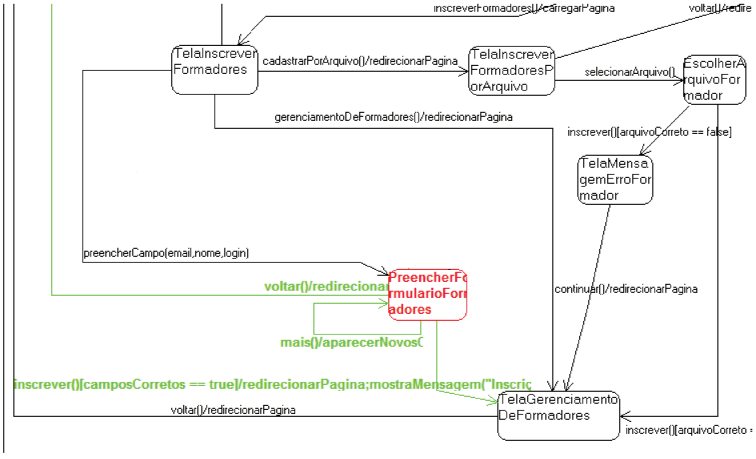


Figura A.10: Continuação da simulação do modelo da Figura A.8.

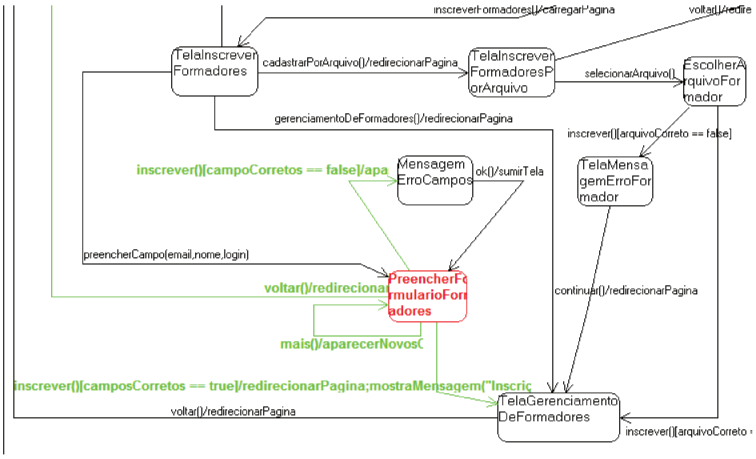


Figura A.11: Validação da versão final do modelo.

Apêndice B

Glossário

Ajax (Asynchronous JavaScript and XML)
Tecnologia que permite comunicação assíncrona entre cliente e servidor.

Applet
Aplicação pequena que executa uma tarefa específica dentro de outra aplicação maior.

DOM (Document Object Model)
Documento que define a estrutura lógica de documentos (HTML e XML) e as formas como esses são acessados e manipulados.

Hipermídia
Hipermídia é uma ligação de conteúdos como imagens e vídeos com outros conteúdos. Extensão de hipertexto.

Hipertexto
Paradigma de interface do usuário que mostra documentos dinamicamente, sendo usualmente utilizado como uma forma de organização de material que contém referência cruzada automática para outros documentos chamados *links*.

HTML
Linguagem de programação de página Web.

INPE
Instituto Nacional de Pesquisas Espaciais

Java Servlet

Módulo de programa Java, servidor que processa e responde às requisições do lado cliente.

Teste Exploratório

Teste exploratório é uma técnica interativa, na qual se explora o *software*. O analista de teste não possui muitas informações do que irá testar e como. A partir da experiência de uso do *software*, encontram-se novos métodos de testar o mesmo.

URL

"*Uniform Resource Locator*" – endereço de localização da página na rede/servidor.

Web Crawler

Programa que "navega"por páginas Web e verifica informações para criar um índice de busca. Também conhecida como *spider*.