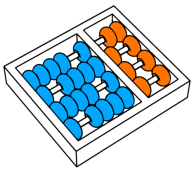
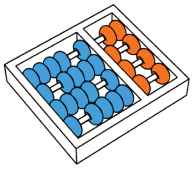


Érika Regina Campos de Almeida

"Geração automática de casos de testes executáveis a partir de casos de teste abstratos para aplicações web"

CAMPINAS
2013



Universidade Estadual de Campinas
Instituto de Computação

Érika Regina Campos de Almeida

"Geração automática de casos de testes executáveis a partir de casos de teste abstratos para aplicações web"

Orientador(a): **Profa. Dra. Eliane Martins**
Instituto de Computação – UNICAMP

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 Érika Regina Campos de Almeida, sob orientação de Profa. Dra. Eliane Martins
Instituto de Computação – UNICAMP.

Assinatura do Orientador(a)

CAMPINAS
2013

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

AL64g

Almeida, Érika Regina Campos de, 1986-
Geração automática de casos de teste executáveis a partir de casos de teste abstratos para aplicações Web / Érika Regina Campos de Almeida. – Campinas, SP : [s.n.], 2012.

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

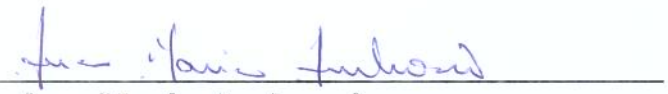
1. Engenharia de software. 2. Software - Testes. I. Martins, Eliane,1955-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

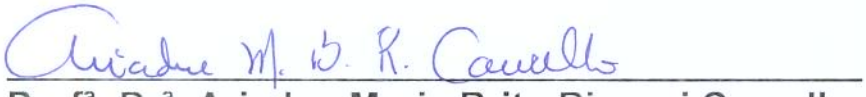
Título em outro idioma: Automatic generation of executable test cases based on abstract ones for Web applications
Palavras-chave em inglês:
Software engineering
Software - Testing
Área de concentração: Ciência da Computação
Titulação: Mestra em Ciência da Computação
Banca examinadora:
Eliane Martins [Orientador]
Ana Maria Ambrósio
Ariadne Maria Brito Rizzoni Carvalho
Data de defesa: 21-05-2012
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

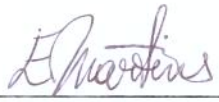
Dissertação Defendida e Aprovada em 21 de Maio de 2012, pela Banca examinadora composta pelos Professores Doutores:



Profª. Drª. Ana Maria Ambrosio
ETE / INPE



Profª. Drª. Ariadne Maria Brito Rizzoni Carvalho
IC / UNICAMP



Profª. Drª. Eliane Martins
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Geração automática de casos de testes executáveis a partir de casos de teste abstratos para aplicações web

Érika Regina Campos de Almeida[1]

21 de maio de 2012

Banca Examinadora:

- Profa. Dra. Eliane Martins
  Instituto de Computação – UNICAMP (Orientadora)
- Profa. Dra. Ana Maria Ambrósio
  Instituto Nacional de Pesquisas Espaciais – INPE
- Profa. Dra. Ariadne Maria Brito Rizzoni Carvalho
  Instituto de Computação – UNICAMP
- Profa. Dra. Maria Beatriz Felgar de Toledo
  Instituto de Computação – UNICAMP (Suplente)

[1]Suporte financeiro de: Bolsa da Capes 2009–2011

Abstract

Automating the generation and execution of test cases meets the major challenge in software development that is doing more with fewer resources. However, to bring together these two processes, it is still necessary to bridge the gap between the specification level of the System under Test (SUT) and its respective implementation. The process of generating test cases usually requires a formal representation of the SUT (based on its specifications) and generates abstract test cases, in the sense that they are on the same detailing level of the specifications. On the other hand, the automatic execution of test cases needs executable test cases, i.e., those that contain implementation details of the SUT to run without manual intervention. So, to use automation from the test design phase to the test execution one, it is necessary to fill the gap between the abstract test case (output artifact of the automatic generation of test cases) and executable one (input artifact for the automatic test execution), once they are at different abstraction levels. Usually, someone with programming skills makes the process of transforming the abstract level to the implementation one, spending much effort and time. In this work, we evaluate the existing literature proposals for automatic mapping of abstract test cases into executable ones that were designed according to Model Driven Testing (MDT), one approach whose aim is to automatically generate software testing artifacts in different levels of abstraction by applying transformation rules. In addition to evaluating the proposals, we chose one to instantiate for Web applications (the kind of applications that have grown most in recent years, either in use as level of complexity), showing what are the steps needed to transform the abstract test cases into executable ones, taking into account that there are specialized libraries to support the writing of the latter one. To evaluate the solution applicability in real Web applications, we conducted case studies using large Web applications (two national ones and another which is worldwide used). In addition, we performed the whole test process (generation of abstract test cases; use of the proposed instantiation for web applications of a method to transform the abstract test cases into executable ones; and running the tests), illustrating the solution feasibility.

Resumo

A automação da geração e da execução de casos de teste está de acordo com o grande desafio na área de desenvolvimento de software que é fazer mais com menos recursos. Entretanto, para unir estes dois processos, ainda é necessário fazer a ponte entre o nível de especificação do Sistema em Teste (SeT) e sua respectiva implementação. O processo de geração de casos de teste usualmente requer uma representação formal do SeT (baseada nos documentos de especificação do SeT) e gera casos de teste abstratos, no sentido que eles estão no mesmo nível de detalhamento das especificações. Já a execução automática de casos de teste necessita de casos de teste executáveis, ou seja, aqueles que contêm detalhes de implementação do SeT para serem executados sem intervenção manual. Para usar automação desde a fase de projeto de teste até a fase de execução, é necessário preencher a lacuna entre o caso de teste abstrato (artefato de saída da geração automática de casos de teste) e o executável (artefato de entrada da execução automática de teste), pois eles estão em níveis de abstração diferentes. Usualmente, alguém com habilidades de programação realiza o processo de transformação do nível abstrato para o de implementação, despendendo muito esforço e tempo. Neste trabalho, avaliamos as propostas para mapeamento automático de casos de teste abstratos em executáveis existentes na literatura que estão de acordo com Teste Dirigido por Modelo (MDT), abordagem cujo intuito é gerar automaticamente artefatos de teste de software em diferentes níveis de abstração aplicando regras de transformação. Além de avaliar as propostas, escolhemos uma delas para instanciar para aplicações Web (as aplicações que mais cresceram nos últimos anos, tanto em uso, como em nível de complexidade), mostrando quais são os passos necessários para transformar os casos de teste abstratos em executáveis, levando em conta que existem bibliotecas especializadas no suporte à escrita deste último. Para avaliar a aplicabilidade da solução em aplicações Web reais, realizamos o estudo de caso usando aplicações Web de grande porte, duas delas de uso nacional e outra disseminada em escala mundial. Além disso, foi realizado o processo completo de teste (geração dos casos de teste abstratos; uso da proposta de instanciação, para aplicações Web, do método para transformar os casos de teste abstratos em executáveis; e execução dos testes), ilustrando a factibilidade da solução.

Agradecimentos

Agradeço primeiramente a Deus por ter iluminado meu caminho em mais uma jornada que certamente foi uma das difíceis de minha vida e por ter me dado força nos momentos de maior dificuldade, de modo que nunca me passou pela cabeça a possibilidade de desistir.

E, como apoio não menos importante, pude contar com meus pais e minha irmã. Os primeiros me incentivaram desde criança a dar valor ao estudo, ensinando-me que, se eu me esforçasse, eu conseguiria alcançar todos os meus objetivos. Foram eles também, juntamente com minha irmã, que sempre estiveram ao meu lado, compartilhando os momentos de alegria (como a qualificação que foi um sucesso, mesmo com a minha irmã sendo a única a fazer perguntas) e de dificuldades.

Além disso, não poderia deixar de agradecer à professora Eliane Martins, minha orientadora neste trabalho e futura orientadora do Doutorado, por sua ajuda e dedicação ao trabalho, pesquisando, lendo artigos e corrigindo meus textos.

Existem também muitas outras pessoas que passaram pela minha vida e me ajudaram, tanto no incentivo quanto na amizade, porque é necessário descansar no final de semana. Agradeço aos amigos por todo apoio.

Por fim, agradeço ao Instituto de Computação (IC) pelo suporte e assistência e à Capes pelo apoio financeiro.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Teste de software é um processo, ou uma série de processos, desenvolvido para certificação de que o software faz o que ele foi projetado para fazer e nada além disso [54].

Ao longo dos anos o teste de software tem se tornado tanto mais difícil quanto mais fácil. Ele está mais difícil devido a variedade de linguagens de programação, sistemas operacionais e plataformas de hardware que estão envolvidos. Mas esses mesmos sistemas operacionais, aliados a outros sistemas de software, também tornam a atividade de teste mais sofisticada do que nunca, pois eles provêm rotinas bem testadas que podem ser incorporadas em uma aplicação sem que o programador a tenha que desenvolver por completo. Interfaces Gráficas de Usuários (GUIs), por exemplo, podem ser construídas a partir de uma biblioteca da linguagem de programação, e, uma vez que tais objetos já foram previamente testados, a necessidade de testá-los como parte da aplicação é reduzida [54].

O processo de desenvolvimento de software também tem passado por intensas transformações nos últimos anos. Novas abordagens de desenvolvimento têm surgido fazendo com que muitos dos problemas encontrados nos projetos anteriores não sejam repetidos. Dentre os problemas mais comuns aos projetos de software podemos destacar: altos custos para evolução, inconsistência entre documentação e sistema final, pouca ou quase que total falta de portabilidade e baixa confiabilidade [7].

A fim de encontrar soluções razoáveis para esses problemas, uma mudança radical de paradigma parece necessária. A tecnologia de objetos introduzida nos anos 80 está saturada e a tecnologia de componentes não melhorou significativamente esta situação, podendo até ser vista como uma complexidade adicional em um domínio que necessita de simplificação [13].

Assim, a principal solução proposta para a nova crise do software parece ser uma mudança do paradigma de composição de objetos para transformação de modelos [13]. A Engenharia de Software dirigida por modelos, mais conhecida como Desenvolvimento

Dirigido por Modelo (MDD), propõe que o processo de criação de software esteja focado em modelos. Estes devem atuar como artefatos de desenvolvimento e não somente como documentação, aumentando o nível de abstração e retirando do desenvolvimento detalhes específicos de ambientes computacionais. Nesse sentido, o grupo OMG (*Object Management Group*) propôs a MDA (*Model Driven Architecture*), cujo objetivo geral é facilitar o desenvolvimento de software dirigido por modelos através do uso de modelos independentes e dependentes de plataforma para gerar uma implementação para execução do sistema de software. Diante do arcabouço conceitual proposto pelo OMG, torna-se possível atingir portabilidade, reusabilidade e interoperabilidade dos modelos [22].

No entanto, mesmo com a mudança do foco de linguagem de programação para modelos no desenvolvimento de software, ainda se faz necessário o processo de teste de software, uma vez que todo desenvolvimento de sistemas de software envolve uma série de atividades de produção, onde as possibilidades de falhas humanas são enormes. Logo, teste de software é um elemento crítico da garantia de qualidade de software e representa a última revisão de especificação, projeto e geração de código [58].

Já há algum tempo, metodologias para geração de testes atestaram que casos de teste de qualidade poderiam ser derivados a partir dos modelos de desenvolvimento, surgindo a abordagem chamada Teste Baseado em Modelo (MBT). Nesta abordagem, a modelagem é feita com base na especificação dos requisitos funcionais do sistema. Os testes gerados servem para averiguar se a implementação está se comportando de acordo com o especificado nos modelos projetados anteriormente. Pode-se citar como vantagens do uso de MBT os seguintes aspectos [7]:

- Diminuição do tempo gasto para geração dos testes;
- Fácil obtenção dos testes em caso de mudanças de requisitos (consequentemente mudanças nos modelos).

Porém, um processo de teste de software para ser considerado completo necessita de muitos outros artefatos além dos casos de teste, artefatos estes que são necessários para especificação, execução e avaliação dos casos de teste. Sabendo disso, uma extensão de MBT, chamada Teste Dirigido por Modelo (MDT), foi proposta para que a geração completa dos artefatos de teste fosse executada de maneira automática através da execução de regras de transformações de modelos [7]. Além da geração de mais artefatos necessários para a atividade de teste, outra diferença significativa de MDT com relação à MBT é que muitas abordagens baseadas em MBT não consideram a distinção entre modelos independentes e específicos de plataforma, os modelos são geralmente projetados de acordo com uma plataforma específica ou são genéricos demais e representam apenas informações independentes de plataforma [24].

No entanto, somente gerar os artefatos de teste automaticamente não basta para que a realização da atividade de testes seja feita em pouco tempo. E, como mais testes com menos tempo e recursos é uma exigência do mercado, a automação de teste é vista como uma alternativa para minimizar o tempo e os recursos gastos, sem perda de qualidade do produto [25].

Automação de teste pressupõe a introdução de um alto nível de automação e reuso em cada fase de teste quando aplicável [28]. Assim, ela pode ser entendida de diferentes formas, dependendo do objetivo, tais como:

- *Automação da execução do teste:* esta tem sido objeto de uma grande quantidade de pesquisas nos últimos anos, levando ao surgimento de uma infinidade de notações, ferramentas e *frameworks* para apoiar a execução automática de scripts de teste, incluindo funcionalidades como a captura automática e reprodução, avaliação automática dos veredictos, geração de relatórios dos resultados de teste, etc.

- *Geração automática dos casos de teste:* geração de teste é o processo que permite que testes sejam automaticamente gerados a partir de modelos de sistema ou qualquer outro tipo de representação formal de comportamento ou estrutura do Sistema em Teste (SeT).

Quando se fala na geração automática dos casos de teste, na maioria das situação tais casos de teste gerados são abstratos no de sentido que estão no mesmo nível de abstração dos modelos que representam o SeT. Sendo assim, podemos ver que existe uma lacuna entre a geração dos casos de teste abstratos e a execução automática de teste, pois esta última faz uso de casos de teste executáveis (conhecidos como scripts de teste) que levam em conta detalhes de implementação do SeT.

Usualmente alguém com habilidades de programação (seja da equipe de desenvolvimento ou de testes do projeto) escreve os scripts de teste manualmente com base na especificação dos casos de teste abstratos. De acordo com Feudjio, “escrever testes é agora uma atividade de programação, que requer habilidades diferentes de especificação e execução de teste” [28].

Entretanto, realizar este processo manualmente despende muito tempo e recurso. Dessa forma, indo ao encontro do objetivo de reduzir o tempo de entrega dos sistemas de software de alta complexidade mantendo a qualidade, um problema a ser resolvido na área de verificação e validação é a geração automática de casos de testes executáveis, a partir de casos de teste abstratos, fazendo a ponte entre o nível de abstração da especificação de requisitos e os detalhes de implementação.

1.1 Objetivo

O objetivo deste trabalho é apresentar uma proposta de solução para o problema de geração automática de casos de teste executáveis a partir dos respectivos casos de teste abstratos, levando em conta que esse mapeamento envolve mudanças no nível de abstração dos artefatos envolvidos, indo do nível da especificação do Sistema em Teste (SeT) para o nível de implementação do mesmo.

Uma vez que essa mudança no nível de abstração não é simples e varia muito conforme o tipo de SeT, focaremos este trabalho em um tipo específico de software: aplicações Web. Esta escolha tem uma razão principal: "aplicação Web está entre as classes de sistema de software que mais crescem atualmente" [26]. Estes aplicativos estão sendo usados para apoiar uma ampla gama de atividades importantes: negócios, tais como venda e distribuição do produto; atividades científicas, tais como compartilhamento de informação e proposta de revisão; e atividades médicas, tais como sistemas de diagnóstico. Dada a importância de tais aplicações, aplicações Web com defeito podem causar muitos impactos sobre as empresas, as economias, o progresso científico e saúde. Logo, é importante que elas sejam confiáveis, e, para isso, é essencial que elas sejam validadas.

A metodologia proposta neste trabalho para obter casos de teste executáveis de aplicações Web faz uso de Teste Dirigido por Modelo (MDT), mapeando os casos de testes abstratos em executáveis através de regras de transformação. Particularmente, para scripts de teste de aplicações Web é necessário o uso de algum *framework* que seja capaz de interpretar comandos e executá-los, simulando as ações dos usuários na interface gráfica. O *framework* escolhido foi o Selenium, já utilizado em larga escala na indústria e que tem bibliotecas para as principais linguagens de programação, como Java.

Um detalhe muito importante da metodologia é que ela foi desenvolvida para ser usada na indústria em aplicações Web de grande porte. Por isso mesmo, os três estudos de caso apresentados são aplicações Web reais, uma delas usada em escala mundial e outras duas no Brasil.

1.2 Contribuições

As principais contribuições deste trabalho, com relação à modelagem de aplicações Web, geração de casos de teste abstratos aplicando técnicas de Teste Baseado em Modelo (MBT) e, a partir destes, obtenção dos casos de teste executáveis usando Teste Dirigido por Modelo (MDT), são:

1. Diretrizes para modelagem das aplicações Web em diagramas de classe e estado, sendo que o primeiro é base para a modelagem do segundo. O diagrama de classes

representa as páginas da aplicação Web e suas funcionalidades como um conjunto de classes (páginas) que têm métodos (funcionalidades). A partir do diagrama de classes, deve ser montado o diagrama de estados cujos estados são as classes e as transições são os métodos, estabelecendo assim a modelagem da navegação na aplicação Web.

2. Definição de requisitos que uma metodologia para transformação de casos de teste abstratos em executáveis deve atender.

3. Instanciação de uma metodologia (que faz uso de Teste Dirigido por Modelo (MDT)) já existente para, com base em casos de teste abstratos, mapeá-los em casos de teste executáveis para aplicações Web. Este processo se baseia na modelagem da aplicação Web em diagrama de classe e estado.

4. Além disso, mostramos em três aplicações Web reais que é possível unir MBT (para geração dos casos de teste abstratos) e MDT, agregando valor ao processo de teste, pois, ao se ter um modelo formal do Sistema em Teste (SeT), aumenta-se a qualidade da documentação dos testes e, em caso de mudança nos requisitos, basta apenas que o modelo também seja alterado e os novos testes serão facilmente gerados, diminuindo o tempo de manutenção.

5. A instanciação apresentada foi baseada no trabalho de Javed *et al.* [36]. Entretanto, as regras de transformação estabelecidas por ele na aplicação de MDT foram adaptadas pelos seguintes motivos:

   A regra de transformação do modelo de teste independente de plataforma para o modelo de teste específico de plataforma escrita por Javed *et al.* na linguagem QVT foi reescrita na linguagem ATL, pois não foi encontrada uma ferramenta capaz de executar as regras originais, além de ATL ser o padrão da indústria.

   Já a regra de transformação do modelo de teste dependente de plataforma para o código de teste sofreu adaptações para as bibliotecas empregadas em casos de teste executáveis de aplicações Web.

6. Além disso, as regras de transformação foram devidamente submetidas a um processo de teste. Neste caso, os testes consistiam em executar a transformação com um modelo de entrada e comparar o modelo de saída obtido com o esperado. Para melhor selecionar o conjunto de modelos de entrada utilizamos a metodologia proposta por Fleurey *et al.* que tem por objetivo cobrir os elementos estruturais do metamodelo de entrada.

1.3 Organização

Este trabalho está organizado da seguinte maneira:

- **Capítulo 2 (Teste de Software e Teste Baseado em Modelo):** Apresenta os conceitos básicos de teste de software e de teste baseado em modelo (MBT), uma abordagem de teste de software em que os casos de teste são derivados dos modelos que usualmente descrevem os aspectos funcionais do Sistema em Teste (SeT).

- **Capítulo 3 (Engenharia Dirigida por Modelo):** Apresenta as novas abordagens para desenvolvimento e teste dirigidos por modelo (que compõem a engenharia dirigida por modelo (MDE)), que consiste em, baseado no modelo de desenvolvimento (ou de teste), gerar o código fonte (ou de teste) do SeT. Enquanto a idéia principal de MBT é derivar casos de teste a partir de modelos do SeT, o objetivo de teste dirigido por modelo (MDT) é definir os casos de teste como modelos e então gerar os casos de teste executáveis usando regras transformação [35].

- **Capítulo 4 (Mapeamento de casos de teste abstratos em executáveis):** Apresenta os requisitos que uma metodologia para transformar casos de teste abstratos em executáveis deve satisafazer, analisando a adequação aos requisitos por parte de metodologias existentes na literatura que usam MDT. Além disso, é apresentada a instanciação de uma das metodologias estudadas para transformar casos de teste abstratos de aplicações Web em executáveis usando MDT.

- **Capítulo 5 (Testes das transformações de modelos):** Apresenta o processo de teste das regras de transformação que são empregadas na instanciação para aplicações Web do método para transformar casos de teste abstratos em executáveis.

- **Capítulo 6 (Estudos de Caso):** Apresenta o estudo de caso com três aplicações Web reais e de grande porte para transformar seus casos de teste abstratos em executáveis com a instanciação proposta no Capítulo 4.

- **Capítulo 7 (Trabalhos relacionados):** Apresenta os principais trabalhos da litetura relacionados a MDE, MDT e a aplicação de ambos em aplicações Web. Dessa forma, também comparamos os trabalhos existentes com o que apresentamos nesta dissertação.

- **Capítulo 8 (Conclusão e Trabalhos Futuros):** Apresenta a avaliação dos resultados obtidos e como esperamos que sejam os próximos passos nos trabalhos futuros.

Capítulo 2

Teste de Software e Teste Baseado em Modelo

O ciclo de vida do desenvolvimento de um software não se resume apenas ao desenvolvimento de seu código fonte; ele envolve também a aplicação de atividades de *verificação e validação* (V&V) com o objetivo de garantir um produto de qualidade de acordo com os requisitos dos clientes. Dentre estas atividades destaca-se o *teste de software* que tem por objetivo executar o software para encontrar defeitos.

As aplicações Web, como qualquer outro software, também devem ser testadas. Assegurar a qualidade de aplicações Web é cada vez mais importante devido a sua importância econômica e ao perigo que qualquer defeito pode apresentar (por exemplo, em aplicações bancárias ou ainda de bolsa de valores).

Sendo assim, nas Seções 2.1 e 2.2 deste capítulo, serão detalhados os conceitos de V&V e teste de software com enfoque, quando necessário, para aplicações Web.

2.1 Verificação e Validação (V&V) de software

Verificação e Validação (V&V) é uma das disciplinas de engenharia de software cujo objetivo é ajudar na construção de software com qualidade. Ela pode ser também vista como um conjunto de atividades de análise e teste ao longo do ciclo de desenvolvimento do software para determinar se este realiza suas funções corretamente (e nada além disso) e medir sua qualidade e confibiliadade [65].

Basicamente, podemos dividir a definição desta disciplina em duas partes. *Verificação* se refere ao conjunto de atividades que assegurem que o software implementa corretamente uma função específica. A ela é associada a seguinte frase: "Estamos construindo o produto corretamente?". Já *validação* se refere ao processo de avaliação de um sistema ou componente, durante ou no final do processo de desenvolvimento, para determinar se

satisfaz os requisitos especificados pelo cliente. Podemos associar validação à seguinte frase: “Estamos construindo o produto certo?”.

As técnicas para realização de V&V podem ser divididas em dois grupos: estáticas e dinâmicas [56]. As técnicas estáticas de V&V visam identificar defeitos sem propriamente executar o sistema que está sendo proposto (ou desenvolvido). Algumas técnicas estáticas mais usadas em V&V são: técnicas de inspeção tais como, revisão e inspeção de código; verificação de modelos e execução simbólica. As técnicas dinâmicas de V&V propostas também têm por objetivo revelar defeitos no sistema em desenvolvimento. Entretanto, sua realização é através da execução do sistema, controlando quais são os dados de entrada e observando o comportamento do sistema para verificar se foram ou não observados defeitos. Neste âmbito, as principais técnicas de V&V dinâmicas são as de *Testes de Software*, que têm como objetivo executar o sistema para revelar defeitos, sendo este considerado qualquer resultado obtido diferente do esperado.

2.2 Teste de Software

Teste de software pode ser definido como:

- O processo de executar um programa ou sistema com o objetivo de encontrar defeitos[1] [54].
- O processo de executar um sistema de software e determinar se ele está de acordo com a especificação e executa no ambiente esperado.
- É a execução de código usando combinações de entradas e estados para revelar defeitos.

As definições apresentadas remetem à aplicação de V&V com o sistema de software em execução, ou seja, uma técnica dinâmica de V&V. Resumidamente, teste de software é a execução de código fonte usando combinações de entradas e estado para revelar defeitos que possam ter sido introduzidos em qualquer fase do desenvolvimento ou manutenção de sistemas de software e que, em geral, são decorrentes de omissões, inconsistências ou mau entendimento das especificações ou dos requisitos dos sistemas [14].

[1]Neste texto foi adotada a seguinte terminologia com relação às traduções dos termos *fault*, *error* e *failure*: falha (*fault*) pode ser entendida como passo, processo ou definição de dados incorreto (por exemplo, uma instrução ou comando incorreto); erro (*error*) representa as situações em que ocorre diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa constitui um erro; já defeito (*failure*) é a produção de uma saída incorreta com relação à especificação.

Os testes podem ser aplicados desde o nível de módulo até o nível de aplicação e, dependendo do propósito do teste e da especificação de requisitos de software, uma combinação de metodologias de teste pode ser aplicada. Além disso, o teste pode ser *caixa-branca*, quando requer a inspeção do código fonte e a seleção de casos de teste que exercitem partes do código e não de sua especificação, levando em conta mecanismos internos do sistema ou componente; ou *caixa-preta*, quando é usado para determinar se o programa satisfaz aos requisitos funcionais e não-funcionais que foram especificados, ignorando qualquer aspecto de implementação

A documentação dos testes aplicados em cada nível é feita através de casos de teste que são definidos por Binder [17], conforme mostra a Definição 1.

Definição 1 *Um caso de teste especifica o estado pré-teste do Sistema em Teste (SeT) e do seu ambiente, as entradas de teste ou condições, e o resultado esperado. O resultado esperado especifica o que o SeT deve produzir a partir das entradas de teste. Esta especificação inclui mensagens geradas pelo SeT, exceções, valores retornados, e estado resultante do SeT e seu ambiente. Casos de teste também podem especificar as condições iniciais e resultantes para outros objetos que constituem o SeT e seu ambiente.*

Dessa forma, para contextualizar os casos de teste que são usados em aplicações Web apresentamos, nesta seção, quais são as fases de teste e os principais fundamentos necessários para entender os casos de teste utilizados em aplicações Web.

2.2.1 Fases de Teste

A atividade de teste é muito importante para a qualidade do sistema de software e não pode ser deixada para o final do projeto. O modelo V do processo de desenvolvimento de software incorpora a atividade de teste durante todo o ciclo de desenvolvimento. No diagrama do modelo V, apresentado na Figura 2.2.1, o V é executado para baixo e, em seguida, para cima, da esquerda para direita, representando a sequência básica de atividades de desenvolvimento e teste. Além disso, o modelo enfatiza a existência de diferentes fases de teste e indica o modo como elas estão relacionadas com uma diferente fase de desenvolvimento. Tais fases variam desde o teste do código fonte até o teste da aplicação pelos clientes, ou seja, são aplicados tanto teste caixa-branca, como teste caixa-preta. A seguir são apresentados mais detalhes de cada fase:

1. *Teste de unidade*: Teste focado nas menores unidades do software, que podem ser funções, procedimentos, métodos ou classes, para avaliar se os aspectos de implementação de cada unidade estão corretos. Geralmente o caso de teste escrito nesta fase faz uso de *drivers* e *stubs*. Um *driver* é um elemento (classe, programa principal

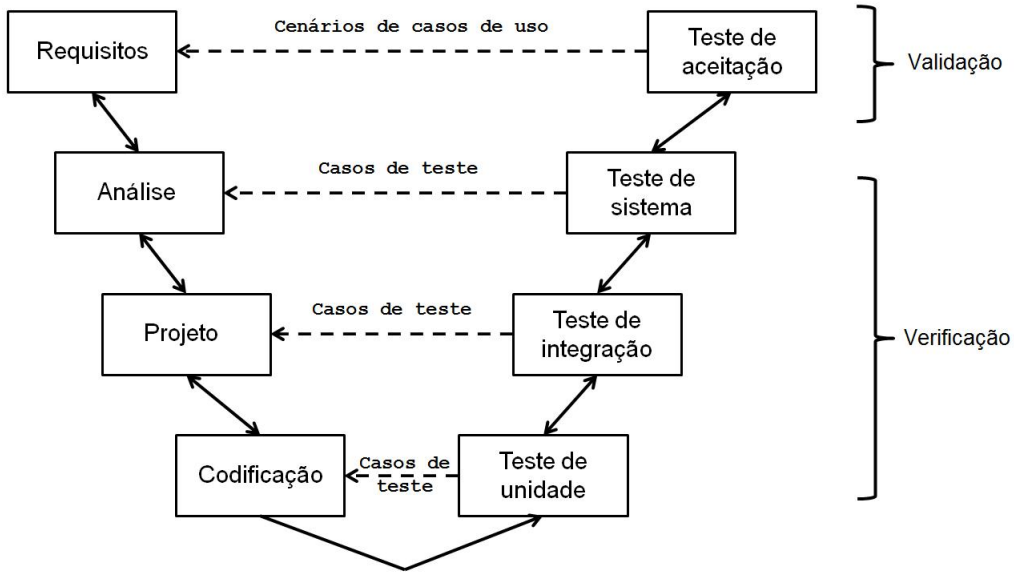


Figura 2.1: Modelo V de desenvolvimento de software.

ou software externo) que aplica os casos de teste à unidade sob teste. O *driver* é responsável por fornecer os dados de entrada, coletar os dados de saída e apresentá-los ao usuário. Um *stub* substitui outras unidades necessárias para a execução da unidade em teste, simulando seu comportamento. A utilização de *drivers* e *stubs* possibilita que a unidade seja testada isoladamente [56].

2. *Teste de integração*: É uma atividade sistemática aplicada durante a integração da estrutura do programa visando descobrir erros associados às interfaces entre os módulos; o objetivo é, a partir dos módulos testados no nível de unidade, construir a estrutura de programa que foi determinada pelo projeto.

3. *Teste de sistema*: Realizado após a integração do sistema, visa identificar erros de funções e características de desempenho, hardware ou base de dados que não estejam de acordo com a especificação.

4. *Teste de aceitação*: Desenvolvedores escrevem testes de unidade para determinar se seus códigos estão fazendo as coisas certas. Da mesma forma, os clientes escrevem testes de aceitação para determinar se o sistema está fazendo a coisa certa. Em outras palavras, o teste de aceitação pode ser entendido como o teste de sistema realizado pelos clientes que desejam se certificar de que seus requisitos foram realmente atendidos.

2.2.2 Tipos de Casos de Teste

Usualmente, a atividade de teste é baseada em um conjunto de casos de teste que contém informações de dados de entrada, pré-condições de execução, procedimento de execução e resultado esperado.

Os casos de teste podem ser classificados em abstratos e executáveis: o primeiro é considerado independente de plataforma e não pode ser diretamente aplicado no Sistema em Teste (SeT), enquanto que o segundo é chamado de específico de plataforma e é aplicado diretamente no SeT. Além disso, é comum os casos de teste executáveis serem derivados dos casos de teste abstratos através de mapeamento dos passos abstratos em estímulos concretos.

Casos de teste abstratos

Um caso de teste abstrato é um cenário envolvendo a comunicação entre o testador e o Sistema em Teste (SeT). Ele especifica uma sequência de estímulos para o SeT, observações para as respostas dadas pelo SeT e como atribuir um veredicto ao resultado do teste baseado nessas observações. Podemos também dizer que um caso de teste abstrato é projetado para exercitar uma sequência de execução específica ou verificar a aderência do SeT a um requisito específico.

A palavra "abstrato" se refere ao fato do caso de teste ser independente de plataforma (e de linguagem de programação) e não poder ser diretamente executado sobre o SeT, necessitando intervenção manual ou um mapeamento para a plataforma específica. A vantagem deste tipo de caso de teste com relação aos executáveis é que este não se preocupa com detalhes de como deve ocorrer a execução.

É comum os casos de teste abstratos serem baseados em modelos de teste. Para máquinas de estado, um caso de teste abstrato consiste de uma sequência de estímulos de entrada seguida dos resultados esperados e dos valores esperados das variáveis de estados observáveis após a ocorrência do estímulo. A Figura 2.2 mostra um exemplo de máquina de estados de um software de caixa eletrônico, enquanto que a Tabela 2.3 ilustra um caso de teste abstrato obtido dessa mesma máquina de estados.

Casos de Teste Executáveis

Um caso de teste executável, também conhecido como script de teste, é, como o próprio nome já diz, a forma executável de um teste. Ele define um conjunto de instruções interpretáveis pelo computador que representam as ações que devem ser executadas e os resultados esperados, usados para identificar qualquer desvio no comportamento obtido.

Casos de teste executáveis são uma alternativa aos casos de teste manuais e apresentam uma grande vantagem com relação a esses, pois podem ser facilmente repetidos em testes

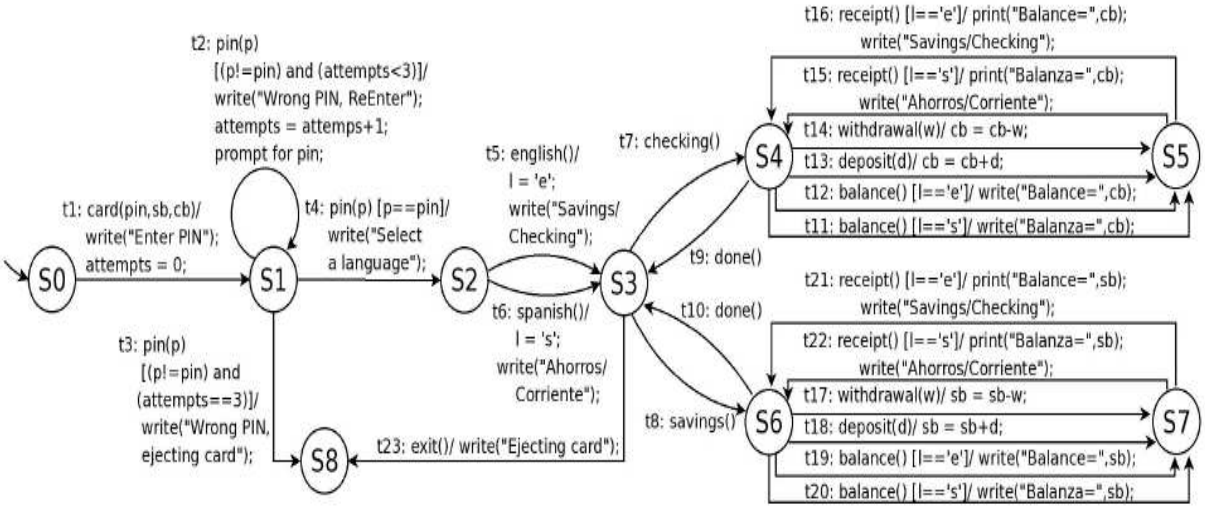


Figura 2.2: Exemplo de uma máquina de estados de um software de caixa eletrônico.

$$\text{seq} = \{\text{checking}(),\ \text{card}(730,\ 840,\ 326),\ \text{pin}(730),\ \text{english}(),\ \text{english}(),\ \text{card}(730,\ 840,\ 326),\ \text{deposit}(207),\ \text{savings}(),\text{english}(),\ \text{deposit}(207)\};$$

Figura 2.3: Exemplo de um caso de teste abstrato de um software de caixa eletrônico.

de regressão[2].

Tais casos de teste podem ser utilizados nas diferentes fases de teste. Nesta seção, estamos interessados em apresentar os principais casos de teste executáveis de forma automatizada utilizado na indústria para teste de unidade e teste de sistema, pois estes são a base para o entendimento dos scripts de teste empregados em aplicações Web.

- **Casos de Teste Executáveis Aplicados em Testes de Unidade**

O testes de unidade consistem normalmente em criar um ou mais unidades, colocá-las em um estado inicial, enviar mensagens a elas e, finalmente, verificar as mudanças ocorridas e o impacto no ambiente do sistema. Em outras palavras, cria-se um *driver* para realizar as configurações, comunicar-se com a unidade em teste e, inclusive, obter o veredicto do teste a partir das respostas obtidas.

O *framework* mais amplamente utilizado para a realização de testes de unidade para a linguagem de programação Java é o *JUnit*. Ele é um *framework Open Source* para automação de testes de unidade na linguagem Java, originalmente escrito por Kent Beck e Erich Gamma [15]. Seus principais objetivos são eliminar o teste e *debug* "clássicos"

[2]Processo de teste das mudanças em um sistema de software visando garantir que as mudanças não afetarem o código feito anteriormente.

(aqueles nos quais são colocados comandos de escrita para testar a saída ou expressões de *debug*), e tornar os testes fáceis de escrever (eles podem ser reusáveis e extensíveis) e executar [15].

Para realizar um caso de teste executável de uma classe utilizando o *framework JUnit*, o desenvolvedor deve escrever o código fonte dos testes que achar necessário para os métodos da classe em teste. Isso deve ser feito em forma de outra classe que estenda (relação de herança amplamente utilizada em orientação a objetos) a classe abstrata *TestCase*, como ilustrado na Listagem 2.1. Tais testes podem ser repetidos quantas vezes necessário e são amplamente utilizados na indústria, pois permitem que algumas falhas no código fonte do software já sejam percebidas na fase de desenvolvimento do ciclo de vida de tal software, gerando uma economia no projeto, já que o custo de um defeito aumenta conforme o avanço de fase do projeto.

Apesar de aqui ser apresentado apenas o *framework* de teste *JUnit* para linguagem de programação Java, cabe destacar que, atualmente, a maioria das linguagens de programação possui *frameworks* para teste de unidade.

```
1  import junit.framework.TestCase;

3  public class TestATM extends TestCase {
     public void testCaseModel() {
5      ATM.checking();
       Card.card(730, 840, 326);
7      ATM.pin(730);
       Language.english();
9      Language.english();
       Card.card(730, 840, 326);
11     ATM.deposit();
       ATM.savings();
13     Language.english();
       ATM.deposit();
15   }
   }
```

Listagem 2.1: Exemplo de caso de teste executável no formato do JUnit

- **Casos de Teste Executáveis Aplicados em Teste de Sistema de Aplicações Web**

Uma forma possível de fazer automatização de teste de software é através da técnica "captura e repetição" que, através de uma ferramenta, registra as ações executadas por um usuário na interface gráfica de um aplicativo e converte as partes nos scripts de teste que pode ser executado quantas vezes se desejar. Um exemplo de ferramenta que aplica essa técnica é o Selenium IDE (*Integrated Development Environment*) [2]. Ele funciona como

um *add-on* do navegador Firefox e apresenta uma interface fácil de usar para desenvolver e executar casos de teste individuais, ou mesmo *suites* de teste de aplicações Web [27].

Apesar de simples e prático, o uso individual desta técnica é muito limitado e tem um alto custo de manutenção, pois os *scripts* contêm os dados para os testes, de modo que qualquer alteração nos dados implica em mudanças no *script*.

Uma alternativa para a limitação dessa técnica é seu uso em conjunto com técnica *data-driven* (que permite que um mesmo teste seja executado com diferentes dados de entrada), porque, desta forma, dados de teste são armazenados em arquivos separados dos *scripts*. Esta técnica também permite que o *designer* de teste e implementador trabalhem em diferentes níveis de abstração; o *designer* apenas prepara os arquivos com os dados de teste, sem se preocupar com questões técnicas de automação de teste [27].

A técnica *data-driven* pode ser aplicada através do Selenium-RC (*Remote Control*) que, além disso, também permite que os *scripts* possam ser executados em vários navegadores e não somente no Firefox, como faz o Selenium IDE [2].

O Selenium-RC permite ao desenvolvedor de automação de teste usar uma linguagem de programação, como Java, para a máxima flexibilidade e extensibilidade para o desenvolvimento da lógica de teste. Por exemplo, se o aplicativo em teste retorna um conjunto de resultados, e se o programa de testes automatizados precisa executar testes em cada elemento do conjunto de resultados, o suporte da linguagem de programação pode ser usado para percorrer o conjunto de resultados, chamando os comandos de Selenium para testar cada item.

Os testes criados com o Selenium-RC são testes que utilizam o JUnit, de forma similar aos testes de unidade. Sua execução acontece da mesma forma que os testes de unidade, solicitando-se sua execução via JUnit. No entanto, existe uma diferença: para que o teste execute, é necessário que exista uma instância de um servidor Selenium executando na mesma máquina que contém os testes que serão executados [60].

Um exemplo do uso do Selenium-RC pode ser visto na Listagem 2.2. Nela é apresentado um caso de teste executável que faz acesso à página do buscador Google, digita como critério de busca *instituto de computação unicamp* e, após a página de resultados ser carregada, clica no primeiro *link*. Para isso, são usados dois métodos: *i*) o método `setUp` é padrão do *framework* JUnit e é sempre executado antes de cada teste, sendo que, no nosso exemplo, ele é usado para abrir a página do buscador Google; *ii*) já o método `testSelenium` contém os passos que são executados depois que a página do buscador Google é carregada, que no caso são os passos de busca e escolha de um resultado.

```
package com.example.tests;

import com.thoughtworks.selenium.*;
import java.util.regex.Pattern;

public class Selenium extends SeleneseTestCase {
  public void setUp() throws Exception {
    setUp("http://www.google.com.br/", "*chrome");
  }
  public void testSelenium() throws Exception {
    selenium.open("/");
    selenium.type("q", "instituto de computacao unicamp");
    selenium.click("btnG");
    selenium.click("//ol[@id='rso']/li[1]/h3/a[1]/em[1]");
    selenium.waitForPageToLoad("30000");
  }
}
```

Listagem 2.2: Exemplo de caso de teste executável no formato do Selenium

2.3 Teste Baseado em Modelo

Teste baseado em modelo é uma modalidade de teste que se baseia em modelos de comportamento que representam o comportamento desejado de um sistema e, possivelmente, o comportamento de seu ambiente. Pares de entrada e saída do modelo da implementação são interpretados como casos de teste para esta implementação: a saída do modelo é a saída esperada do Sistema em Teste (SeT).

O uso de modelos é motivado pela observação de que, tradicionalmente, o processo de testes tende a ser não-estruturado, sem muitos detalhes, irreproduzível e não-documentado. A idéia é que a existência de um artefato que represente explicitamente o comportamento desejado pode ajudar a mitigar as implicações destes problemas [64].

De certa forma, todo teste realizado é baseado em modelos. A criação de modelos não é uma nova habilidade, uma vez que os testadores sempre constroem modelos, ainda que informais, para executar os testes e confrontar os resultados e comportamento obtidos com o que se espera. Porém, considera-se como definição de teste baseado em modelo aquele cujo modelo é formal e em que há possibilidade de tornar automática a geração de casos de teste.

Ainda é importante observar dois outros detalhes sobre teste baseado em modelo:

- Os modelos do SeT devem ser simples e fáceis de serem verificados, pois, caso contrário, o esforço para validar o modelo poderá ser maior do que validar o SeT [64];

- O modelo descrevendo o SeT é usualmente abstrato, no sentido que ele está no mesmo nível das especificações desse sistema. Sendo assim, os casos de teste obtidos também são considerados abstratos, uma vez que estão no mesmo nível de abstração do modelo.

2.3.1 Representação do Sistema em Teste (SeT) na Forma de Modelos

Modelagem, seja em engenharia de software ou em qualquer outra disciplina, significa o processo de criação de uma representação de um sistema real ou fenômeno, resumindo e simplificando o seu comportamento. Modelos sempre descrevem o sistema ou fenômeno a partir de um certo ponto de vista. Os modelos podem se concentrar em qualquer coisa entre a lógica interna e o comportamento de entrada e saída perceptível ao usuário. Além disso, modelos podem ser apresentados usando qualquer uma das inúmeros notações diferentes, como os diagramas de UML.

Quando falamos de teste de software, e engenharia de software em geral, o modelo fundamental no qual muitos outros modelos são baseados é a máquina de estado finito (do inglês *finite state machine* (FSM)). Isto fica evidente quando analisamos a forma como a maioria dos sistemas de software é entendida, utilizada e testada - em qualquer momento, um sistema parece estar em um estado específico, em que um conjunto finito de entradas é permitido. Quando o sistema recebe uma das entradas permitidas, ele executa uma ação e passa para outro estado, em que um outro conjunto de entradas é então permitido. Sistemas que correspondam a essa descrição são conhecidos como sistemas reativos e, por exemplo, as aplicações Web se encaixam nesta categoria.

2.3.2 Geração dos Casos de Teste

Uma vez que o modelo do Sistema em Teste (SeT) já foi feito e validado, devem ser então gerados os casos de teste abstratos. Entretanto, para isso deve ser estabelecido um critério de seleção de testes, que consiste basicamente de regras para determinar quais subconjuntos de comportamentos do modelo devem ser escolhidos de forma que os comportamentos equivalentes no sistema sejam testados e confrontados com os resultados esperados.

A escolha do critério é uma questão complexa e que envolve diversos fatores. De maneira geral, esta escolha deve atender um compromisso que evite um custo elevado de uso e que revele o maior número possível de falhas, com atenção àquelas de maior probabilidade de ocorrência e severidade.

Há diversos tipos de critérios, que variam de acordo com os elementos requeridos

por cada um: funcionalidades do sistema; exercício de elementos estruturais do modelo (como as transições de um diagrama de estados); utilização de conceitos probabilísticos ou somente aleatoriedade; perfis de operação do sistema; classes de defeito no modelo.

Após a escolha do critério de teste, é possível gerar um conjunto de casos de teste capaz de atendê-lo. Tal atividade pode ser feita manualmente, mas costuma ser feita com a ajuda de ferramentas que lêem o modelo de teste de acordo com o formato estabelecido.

Capítulo 3

Engenharia Dirigida por Modelo

Uma importante mudança de paradigma está acontecendo no campo da engenharia de software que pode ter consequências importantes sobre a forma como os sistemas de informação são construídos e mantidos. A idéia central da composição de objetos está sendo progressivamente substituída pela noção de transformação de modelos. Pode-se ver estes tanto como continuidade ou como ruptura. A idéia de sistemas de software compostos de objetos interligados não está em oposição à idéia de ciclo de vida do software visto como uma cadeia de transformações do modelo [20].

A Engenharia Dirigida por Modelo (MDE) é uma nova abordagem que tem por objetivo aliviar os problemas de complexidade de plataforma, combinando linguagens de modelagem e transformações de modelos [61] para gerar automaticamente código a partir de modelos.

Nesta nova visão, o desenvolvedor se preocupa com a qualidade dos modelos, de modo que eles reflitam corretamente o artefato que se deseja obter. Entretanto, não basta que tais modelos estejam condizentes; é preciso também que as transformações de modelo estejam corretas, de modo a garantir que o processo de automação seja confiável; caso contrário, todo processo será comprometido. Citando Fleurey *et al.*, "uma única transformação errada pode tornar todo o processo de desenvolvimento baseado em modelo vulnerável [29]", ainda mais que, uma vez pronta, a transformação será utilizada por muitas vezes.

Tais modelos podem ter várias formas, mas como uma forma de padronizar o uso de MDD, em 2003, foi proposta pela OMG (*Object Management Group*) a MDA (*Model Driven Architecture*), uma abordagem padronizada aberta e independente de fornecedor, cujo principal objetivo é padronizar diversos modelos para que as empresas os usem para representar aplicações [7]. Tal abordagem será detalhada na Seção 3.1.

A filosofia de MDD também pode ser aplicada para o processo de teste de software. Aliando as práticas de MDD com a abordagem de Teste Baseado em Modelo (MBT), que

usa modelos para geração automática de casos de teste, temos o Teste Dirigido por Modelo (MDT) que gera artefatos de teste, e não somente casos de teste, de acordo com regras de transformações entre modelos. Dentre as vantagens da utilização de MDT em relação a MBT a principal é que, em MBT, os casos de teste são derivados a partir dos modelos de desenvolvimento fracamente conectados, sendo estes normalmente incompletos, sem informações necessárias para os testes (restrições, casos alternativos, dentre outros). Com a utilização das práticas de MDD, onde modelos são o centro do desenvolvimento, tais informações poderão ser naturalmente incorporadas [7]. Mais detalhes sobre MDT podem ser vistos na seção 3.2.

Tanto em MDA como em MDT são utilizadas as linguagens de modelagem para representar o sistema de software e seu ambiente, e as regras de transformações entre modelos. Tais regras são empregadas para transformar um Modelo *A* em um Modelo *B* de acordo com o tipo de abordagem que se está tratando, e um aspecto muito importante dessas regras é a linguagem em que elas são descritas. Para ajudar no entendimento das principais linguagens utilizadas para modelagem do software e para descrição das regras de transformação entre modelos, as Seções 3.3 e 3.4 apresentam brevemente cada uma delas.

Dessa forma, este capítulo irá apresentar os principais fundamentos de MDA e MDT, juntamente com dois aspectos muito importantes da utilização dessas duas metodologias, que são as linguagens de modelagem e as regras de transformação entre modelos. Tais regras são muito importantes, pois, uma vez escritas, elas podem ser aplicadas em muitos sistemas de software, bastando que seus modelos estejam de acordo com o modelo de entrada.

3.1 *Model Driven Architecture* (MDA)

A idéia principal de MDA está na utilização das linguagens de modelagem como linguagens de programação, e não apenas como linguagens de projeto. Com isso, os modelos de software deixam de ser apenas artefatos de documentação, para serem os artefatos principais no processo de desenvolvimento do software [49]. Um modelo é uma representação simplificada de algum conceito, com o objetivo de observação, manipulação e entendimento sobre tal conceito [52]. No desenvolvimento de software, modelos são criados com o objetivo de diminuir a complexidade inerente ao desenvolvimento.

MDA define uma abordagem para a especificação do sistema de software que separa a especificação da funcionalidade do sistema da especificação da implementação dessa funcionalidade em uma plataforma de tecnologia específica. A abordagem MDA e os padrões designados por ela permitem que o mesmo modelo de funcionalidades do sistema seja realizado em múltiplas plataformas, através de normas de mapeamento auxiliares, ou por meio de mapeamentos pontuais para plataformas específicas, e permite que diferentes

aplicações sejam integradas explicitamente relacionando seus modelos, possibilitando a integração e interoperabilidade e apoiando a evolução do sistema uma vez que plataformas de tecnologia são instáveis [39].

Dos modelos concretos, todos, com exceção do Modelo de Negócio, são modelos de sistema. Cada um desses modelos representa o sistema a partir de um ponto de vista diferente. O Modelo de Requisitos, também conhecido como Modelo Independente de Computação (CIM), é a representação mais abstrata do sistema; já o Modelo Independente de Plataforma (PIM) é menos abstrato, pois apresenta parte da lógica do sistema e algumas considerações técnicas. O PIM representa um refinamento do Modelo de Requisitos. Um Modelo Específico de Plataforma (PSM) é um refinamento de um PIM e é descrito em termos específicos de uma plataforma onde o sistema é implementado. Os Modelos Físicos representam artefatos físicos utilizados no desenvolvimento ou execução do sistema como, por exemplo, arquivos ou nós computacionais [49].

Vale ressaltar que todos os modelos (CIM, PIM e PSM) são instâncias de um metamodelo que especifica os elementos do modelo e como podem ser as relações entre esses elementos.

Uma utilização simples da abordagem MDA no desenvolvimento de software consiste na modelagem de uma aplicação em uma linguagem de modelagem independente de plataforma (i.e. UML). O modelo independente de plataforma (PIM) pode então ser traduzido em um modelo específico de plataforma (PSM) através da escrita de especificações de transformação que são mapeamentos entre o PIM e alguma linguagem de implementação (por exemplo, Java) [36]. A transformação entre modelos é feita através das chamadas regras de transformações ou mapeamentos. Este processo é ilustrado na Figura 3.1.

3.2 Teste Dirigido por Modelo (MDT)

A maioria das abordagens de Teste Baseado em Modelo (MBT) não faz a separação entre modelos independentes de plataforma e os dependentes de plataforma, ou seja, ou tais modelos são adaptados para uma plataforma específica, ou eles são mais abstratos a este respeito, levando em conta somente informações do modelo independente de plataforma [33].

Para solucionar este problema em MBT, pode ser aplicado o Teste Dirigido por Modelo (MDT) que segue a filosofia MDA. A mesma abstração, em termos de modelagem independente e específica de plataforma e geração de código do sistema pode ser aplicada para os modelos de teste, mas, neste caso, o modelo independente de plataforma é denominado PIT (*Platform Independent Testing Model*), enquanto o modelo específico de plataforma é conhecido por PST (*Platform Specific Testing Model*).

Além disso, os modelos de teste podem ser transformados a partir dos modelos do

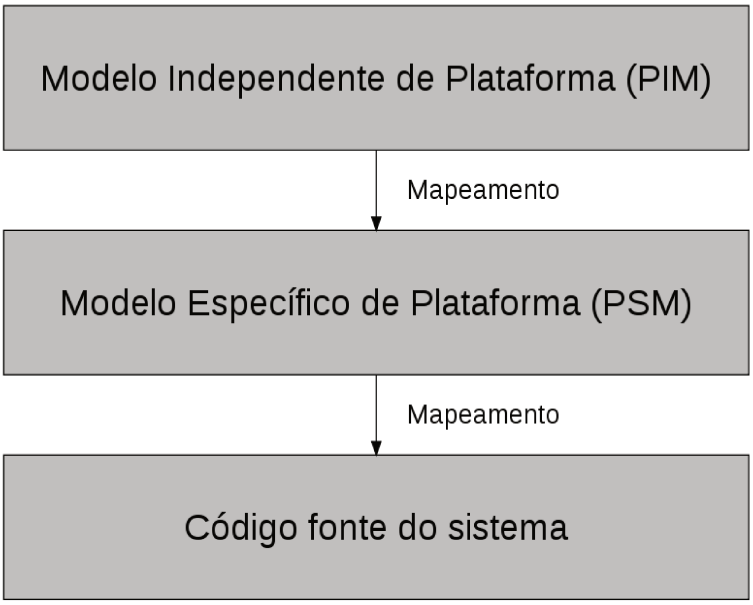


Figura 3.1: Processo de transformação entre os modelos na abordagem MDA (retirado de [23]).

sistema de software. Isso permite a integração inicial dos testes no processo geral do ciclo de desenvolvimento do software. Uma vez que o modelo do software tenha sido definido no nível PIM (Modelo Independente de Plataforma), um modelo de teste independente da plataforma (PIT) pode ser derivado deste. Em seguida, o modelo PIT pode ser transformado, tanto para diretamente testar o código fonte, como para um modelo de teste específico da plataforma (PST). Outra forma de obter o PST é derivá-lo do PSM (Modelo Específico de Plataforma), assim como o PIM pode ser transformado no PIT [23].

Após cada etapa de transformação, o modelo de teste pode ser aperfeiçoado e enriquecido com propriedades específicas do teste. Mais ainda, algumas questões do teste, como, por exemplo, informações de controle de teste e implantação têm de ser manualmente adicionadas ao modelo de teste. Em UML, tais informações são representadas por anotações que representam conceitos no PSM e são aplicadas a elementos do PIM para indicar como esses elementos devem ser transformados [57].

No final do processo, o modelo de teste pode ser finalmente transformado em código executável de teste a partir do PIT ou do PST. Tal processo pode ser visto na Figura 3.2.

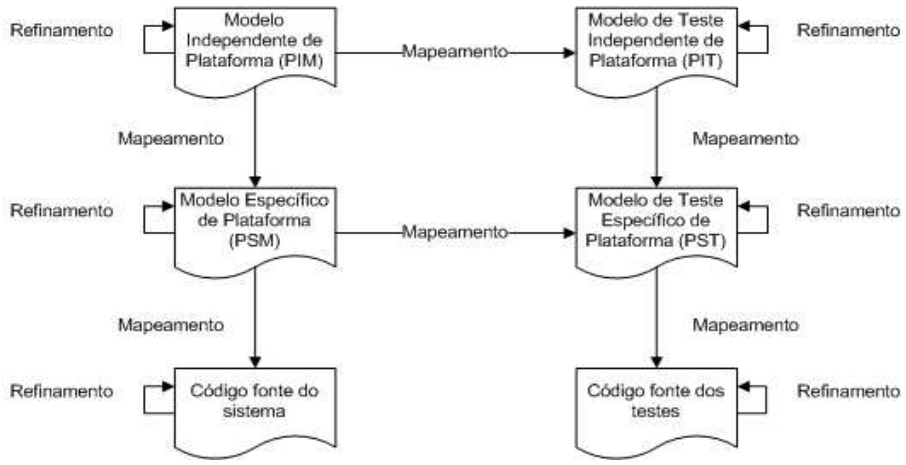


Figura 3.2: Processo de transformação entre os modelos na abordagem MDT (retirado de [23]).

3.3 Linguagens de modelagem empregadas nas abordagens MDA e MDT

Os modelos de entrada empregados nas metodologias MDA e MDT normalmente seguem alguns padrões impostos, seja pela OMG ou mesmo pelo uso comum na indústria.

Em MDA, os modelos são os principais artefatos, integrados no processo de desenvolvimento através de uma cadeia de transformações de PIM para PSM até aplicação codificada. Para possibilitar isso, a abordagem MDA requer que modelos sejam expressos em uma linguagem baseada em MOF (*Meta-Object Facility*). Isso garante que os modelos possam ser armazenados em um repositório *MOF-compliant*, analisados e transformados por ferramentas compatíveis, e processados em XMI[1] para o transporte em uma rede. Isso não restringe os tipos de modelos que se pode usar - as linguagens baseadas em MOF atualmente modelam estrutura, comportamento (de muitas formas diferentes), e dados das aplicações; UML (*Unified Modeling Language*) e CWM (*Common Warehouse Metamodel*), padrões da OMG, são bons exemplos de linguagens de modelagem baseadas em MOF.

Basicamente, MOF é o alicerce do ambiente padrão para indústria da OMG, onde os modelos podem ser exportados a partir de uma aplicação, importados para a outra, transportados através de uma rede, armazenados em um repositório e depois recuperados, mapeados em diferentes formatos (incluindo XMI), transformados e utilizados para gerar o código da aplicação. Estas funções não estão restritas a modelos estruturais, ou até

[1]Formato padrão de XML, colocado pela OMG, para a transmissão e armazenamento de modelos em banco de dados.

mesmo para os modelos definidos na UML - modelos de comportamento e modelos de dados também podem participar neste ambiente, assim como linguagens de modelagem não-UML, contanto que todos eles sejam baseados em MOF [31].

3.4 Transformações com os modelos das abordagens MDA e MDT

Nesta seção serão apresentadas as transformações que podem ser realizadas com os modelos de MDA e MDT. São dois os tipos de transformação: (i) de um modelo para outro, apresentada na Seção 3.4.1; e (ii) a transformação de um modelo para texto, detalhada na Seção 3.4.2.

3.4.1 Transformação Modelo-Modelo

A transformação de modelos pode ser definida como um processo que realiza a conversão de um modelo em outro modelo do mesmo sistema. Estes modelos podem ser instâncias do mesmo metamodelo ou de metamodelos diferentes. Além disso, as transformações de modelo podem ser classificadas em dois tipos de acordo com a variação no nível de abstração [38]:

- Transformação horizontal: é uma transformação em que os modelos de entrada e saída estão no mesmo nível de abstração. Transformações horizontais são realizadas para apoiar a evolução do modelo. Dois exemplos de evolução do modelo são: *i*) adicão novos recursos e *ii*) reestruturação para melhorar características já existentes.

- Transformação vertical: é uma transformação em que os modelos de entrada e saída estão em nível de abstração diferentes. Refinar um modelo ou realizá-lo em uma linguagem de programação são exemplos de transformações verticais.

O tipo básico de transformação modelo-modelo é a transformação PIM-PSM (ou PIT-PST, no caso de MDT); no entanto, antes de gerar o modelo PSM, diversas transformações no modelo PIM podem ser necessárias. Portanto, os modelos manipulados por essas transformações podem ser denominados modelo de entrada e saída, origem e destino, ou ainda esquerda e direita, para o caso de transformações bidirecionais [49].

A especificação de transformação entre modelos pode ser feita através de uma linguagem puramente declarativa, uma linguagem puramente imperativa, ou uma combinação entre esses dois tipos, formando uma linguagem híbrida [49].

Transformação de modelos é um componente crítico de Engenharia de Software dirigida por modelo. A OMG reconheceu isto e, consequentemente, criou o *Query/-Views/-Transformations* (QVT) RFP (*Request for Proposal*), para procurar uma solução compatível com o padrão MDA: UML, MOF, OCL, etc. Várias respostas foram dadas por empresas e instituições de pesquisa. Eles evoluíram durante três anos uma única proposta, que foi chamada de QVT. Algumas outras propostas evoluíram paralelamente ao processo da OMG. *Atlas Transformation Language* (ATL) é uma delas [37].

ATL e QVT compartilham alguns elementos em comum, uma vez que, inicialmente, eles compartilharam o mesmo conjunto de requisitos definidos no QVT RFP. Entretanto, os requisitos atuais de ATL mudaram com o tempo conforme a linguagem amadurecia [37].

O padrão QVT define uma especificação híbrida para transformações. De um lado, há uma parte declarativa, que prevê mecanismos para definir transformações como um conjunto de relações, que devem se manter entre os elementos do modelo de um conjunto de modelos candidatos (ou seja, modelos de entrada e saída). Esta parte declarativa pode ser dividida em duas camadas de acordo com o nível de abstração: a camada relacional, que fornece notação gráfica e textual de uma especificação declarativa de relações, e a camada de núcleo, que fornece uma definição mais formal de transformações. Por outro lado, QVT também tem uma parte imperativa que define mapeamentos operacionais que se estendem da parte declarativa com implementações imperativas. Esta parte é utilizada quando é difícil fornecer uma especificação meramente declarativa de uma relação [62].

ATL também é uma linguagem de transformação híbrida. Ela contém uma mistura de construções declarativas e imperativas. Encoraja-se um estilo declarativo de especificação de transformações. No entanto, às vezes é difícil fornecer uma solução completa declarativa para um dado problema transformacional. Nesse caso, os desenvolvedores podem recorrer aos recursos imperativos da linguagem [49].

As transformações ATL são unidirecionais, que operam em modelos de entrada e produzem modelos de saída. Durante a execução de uma transformação, o modelo de entrada pode ser navegado, mas não são permitidas alterações. O modelo de saída não pode ser navegado. Uma transformação bidirecional é implementada como um par de transformações: uma para cada direção. Os modelos de entrada e saída em ATL devem ser expressos no formato de serialização XMI [5].

Uma transformação ATL pode ser decomposta em três partes: cabeçalho, *helpers* e regras. O cabeçalho é usado para declarar as informações gerais, tais como o nome do módulo (é o nome de transformação e deve corresponder ao nome do arquivo), os modelos de entrada e saída e importação das bibliotecas necessárias. *Helpers* são sub-rotinas utilizadas para evitar a redundância de código. Por fim, as regras são o coração das transformações ATL, porque elas descrevem como os elementos de saída (com base no metamodelo de produção) são produzidos a partir de elementos de entrada (com base

no metamodelo de entrada) [5].

Um exemplo de uso da linguagem de transformação ATL consiste em transformar o modelo de famílias no modelo de pessoas, com a seguinte regra: para formar o nome completo de uma pessoa, deve ser aplicado ao primeiro nome dela o sobrenome da família. Além disso, separa-se as pessoas em homens e mulheres, de acordo com sua classificação na família (i.e. pai, mãe, filho, filha, etc.). O modelo de entrada, em formato XMI, é ilustrado na Listagem 3.1, da mesma forma que o modelo de saída pode ser visto na Listagem 3.2. Já a regra de transformação no formato da linguagem ATL é mostrada na Listagem 3.3. Nela podemos ver claramente a separação em cabeçalho, *helpers* e regras.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="
    Families">
  <Family lastName="March">
    <father firstName="Jim"/>
    <mother firstName="Cindy"/>
    <sons firstName="Brandon"/>
    <daughters firstName="Brenda"/>
  </Family>
  <Family lastName="Sailor">
    <father firstName="Peter"/>
    <mother firstName="Jackie"/>
    <sons firstName="David"/>
    <sons firstName="Dylan"/>
    <daughters firstName="Kelly"/>
  </Family>
</xmi:XMI>
```

Listagem 3.1: Modelo de entrada - Modelo de Famílias

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="
    Persons">
  <Male fullName="Jim March"/>
  <Male fullName="Brandon March"/>
  <Male fullName="Peter Sailor"/>
  <Male fullName="David Sailor"/>
  <Male fullName="Dylan Sailor"/>
  <Female fullName="Cindy March"/>
  <Female fullName="Brenda March"/>
  <Female fullName="Jackie Sailor"/>
  <Female fullName="Kelly Sailor"/>
</xmi:XMI>
```

Listagem 3.2: Modelo de saída - Modelo de Pessoas

```
-- @path Families=/Families2Persons/Families.ecore
-- @path Persons=/Families2Persons/Persons.ecore

module Families2Persons;
create OUT : Persons from IN : Families;

helper context Families!Member def: familyName : String =
  if not self.familyFather.oclIsUndefined() then
    self.familyFather.lastName
  else
    if not self.familyMother.oclIsUndefined() then
      self.familyMother.lastName
    else
      if not self.familySon.oclIsUndefined() then
        self.familySon.lastName
      else
        self.familyDaughter.lastName
      endif
    endif
  endif;

helper context Families!Member def: isFemale() : Boolean =
  if not self.familyMother.oclIsUndefined() then
    true
  else
    if not self.familyDaughter.oclIsUndefined() then
      true
    else
      false
    endif
  endif;

rule Member2Male {
  from
    s : Families!Member (not s.isFemale())
  to
    t : Persons!Male (
      fullName <- s.firstName + ' ' + s.familyName
    )
}

rule Member2Female {
  from
    s : Families!Member (s.isFemale())
  to
    t : Persons!Female (
      fullName <- s.firstName + ' ' + s.familyName
    )
}
```

Listagem 3.3: Regra de transformação do modelo de famílias para o modelo de pessoas

3.4.2 Transformação Modelo-Texto

O exemplo mais conhecido de transformações do tipo modelo-texto é a geração de código-fonte, que recebe um modelo de entrada (e.g.: um modelo de classes da UML), e realiza a geração dos arquivos de código-fonte para uma linguagem (e.g.: Java, C++). Transformações do tipo modelo-texto podem ser consideradas casos especiais de transformações do tipo modelo-modelo. Para isto, basta que seja definido um metamodelo para expressar a linguagem de programação desejada. Assim, o código fonte gerado pela transformação é representado através de um modelo que posteriormente é armazenado em forma de arquivo texto [49].

Capítulo 4

Mapeamento de casos de teste abstratos em executáveis

Atualmente, muitos gerentes de software são pressionados a fazer cada vez mais com menos tempo e recursos. Como a atividade de teste faz parte do ciclo de desenvolvimento do software, tal problema também se aplica a ela. Como uma resposta à necessidade de economizar recursos e tempo, a automação de teste tem alcançado mais enfoque na indústria. Entretanto, mesmo executando automaticamente os casos de teste, ainda é necessário especificá-los e mapeá-los, de alguma forma, em uma linguagem executável (i.e., criar scripts de teste), tarefa que exige um tempo considerável de trabalho de um analista de teste.

Quando se especifica casos de teste, seja através de um documento de texto ou de ferramentas específicas, usualmente se trata de casos de teste abstratos, que estão no mesmo nível da especificação funcional do Sistema em Teste (SeT). Os casos de teste abstratos podem ter dados concretos ou apenas uma especificação das regras envolvidas em um determinado dado de entrada. Além disso, já existem metodologias para geração de tais casos de teste, como a já apresentada na Seção 2.3 chamada Teste Baseado em Modelo (MBT).

Já os casos de teste executáveis, conhecidos como scripts de teste, levam em conta detalhes da implementação do SeT, conforme detalhado na Seção 2.2.2 do Capítulo 2.

Podemos ver que existe uma grande diferença entre os dois tipos de caso de teste (abstrato e executável) e, dado que gerar casos de teste abstratos é um problema que já possui algumas soluções, o grande desafio é como mapear, de forma automática, os casos de teste abstratos em executáveis para que se possa tirar proveito de todos os benefícios providos pela automação da execução de teste.

Sendo assim, neste capítulo iremos descrever os requisitos que uma metodologia para transformação de casos de teste abstratos em executáveis deve atender e analisar algumas

propostas existentes na literatura para verificar se alguma delas atende aos requisitos, de modo que possamos instanciar para aplicações Web. Além disso, serão analisadas as metodologias existentes na literatura e que atendam aos requisitos especificados para que possamos instanciar uma delas para mapear casos de teste abstratos de aplicações Web em executáveis.

4.1 Requisitos

O uso de Teste Dirigido por Modelo (MDT) requer a definição de modelos de teste a partir dos quais será derivado o código de teste. Uma das formas possíveis e mais práticas de se obter tais modelos é derivá-los dos modelos de desenvolvimento, como propõem Dai *et al.* [23]. Entretanto, seja por questões de confidencialidade ou simplesmente pelo fato do desenvolvimento de software não aplicar Desenvolvimento Dirigido por Modelo (MDD), não é sempre que a equipe responsável pelos testes tem acesso aos modelos de desenvolvimento para derivar os modelos de teste. Por esses motivos, o primeiro requisito levantado para a escolha do formato do modelo de teste foi que ele não esteja diretamente ligado aos modelos de desenvolvimento, pois desejamos estender a metodologia a todas as equipes de teste de aplicações Web.

Dado que este requisito não é o único para escolha do modelo de teste (que, no nosso caso, representa um caso de teste abstrato), e que este processo pode ser visto como a escolha de um produto, levantamos uma lista com outros requisitos para tal modelo de teste:

1. Deve contemplar os principais elementos de um caso de teste [6]: dados de entrada, passos, resultados esperados;

2. Deve permitir, se necessário para o teste, a especificação de passos que devem ser executados antes do teste para deixar a aplicação Web em um estado conhecido;

3. Deve permitir, se necessário para o teste, a especificação de passos que devem ser executados depois do teste para voltar a aplicação Web em seu estado inicial;

4. Deve possibilitar a organização dos casos de teste em suites de teste, que organizam logicamente os casos de teste, normalmente por funcionalidade em teste.

Além da escolha do modelo de teste usado para representar os casos de teste abstratos, o uso de MDT requer regras de transformação para mapear primeiramente o modelo de teste independente de plataforma no modelo de teste espefíco de plataforma e, em seguida, gerar a partir deste último o código de teste (casos de teste executáveis). Sendo assim,

temos o requisito que as regras de transformação devem tanto estar disponíveis como ser executáveis com ferramentas *open source*.

Finalmente, o último requisito é que o código de teste gerado esteja em uma linguagem de programação que permite a realização de testes automatizados de aplicações Web.

4.2 Metodologias que atendem aos requisitos

Para atender tais requisitos, procuramos na literatura algumas linguagens e formatos que se adequassem as nossas necessidades. Consideramos a possibilidade de usar TestML [30], uma linguagem baseada em XML e independente de ferramenta, que foi desenvolvida para especificar descrições de teste. Esta linguagem foi desenvolvida para suprir as demandas de uso de Teste Baseado em Modelo (MBT) de sistemas embarcados da indústria automotiva. Por esse motivo, a linguagem é muito específica para as aplicações desenvolvidas neste ramo da indústria, sendo que, se desejássemos utilizá-la, teríamos que fazer muitas adaptações.

Ali *et al.* [4] propõem uma abordagem cujo modelo de teste, denominado "árvore de teste", é derivado de máquinas de estado. Por esse motivo, o modelo está muito ligado a este tipo de diagrama e, apesar de nossos estudos de caso se basearem em máquinas de estados, não desejamos que o modelo de teste seja específico para um tipo de diagrama.

Já Alves *et al.* propõem um *framework* que une Desenvolvimento Dirigido por Modelo (MDD) e Teste Dirigido por Modelo (MDT) com os seguintes objetivos: máxima automatização no desenvolvimento do software e dos testes e maior controle do software para evolução. Entretanto, apesar de apresentarem um estudo de caso envolvendo a linguagem de programação Java, que permite a escrita de casos de teste executáveis para aplicações Web, a proposta de Alves *et al.* é focada em derivar os modelos de teste a partir dos modelos de desenvolvimento, e, portanto, não atende ao Requisito 1. Além disso, o *framework* não está disponibilizado, de modo que não poderíamos aplicá-lo.

Já a proposta de Javed *et al.* [36], que gera testes de unidade para as linguagens de programação Java e Smalltalk a partir de diagramas de sequência, apresenta um metamodelo de entrada que se adequa aos nossos requisitos do modelo de teste. Pode-se questionar o fato deste metamodelo de entrada ser derivado de um diagrama de sequência, mas a estrutura montada por eles se ateve somente aos elementos mais simples de um diagrama de sequência, como é a sequência de passos de um caso de teste abstrato.

Mais ainda, Javed *et al.* disponibilizaram os metamodelos de entrada e saída, bem como as regras de transformação, sendo que essas estão escritas em linguagens comuns de Engenharia Dirigida por Modelo (MDE).

Desse forma, o trabalho de Javed *et al.* se assemelha ao nosso, exceto pelas particularidades de aplicações Web que devem ser inseridas nos casos de teste da linguagem

de programação Java e por estarmos lidando com teste caixa-preta ao invés de teste caixa-branca.

Sendo assim, nós nos basearemos na proposta deles para transformar modelos de teste (casos de teste abstratos) de aplicações Web em casos de teste executáveis (testes de unidade no formato JUnit).

O processo de transformação de casos de teste abstratos em executáveis proposto por Javed *et al.* e ilustrado na Figura 4.1, foi planejado seguindo as diretrizes de Teste Dirigido por Modelo (MDT): um modelo de teste independente de plataforma (PIT) é transformado em um modelo de teste específico de plataforma (PST); este, por sua vez, deve ser mapeado em código de teste, incorporando, inclusive, os dados de teste.

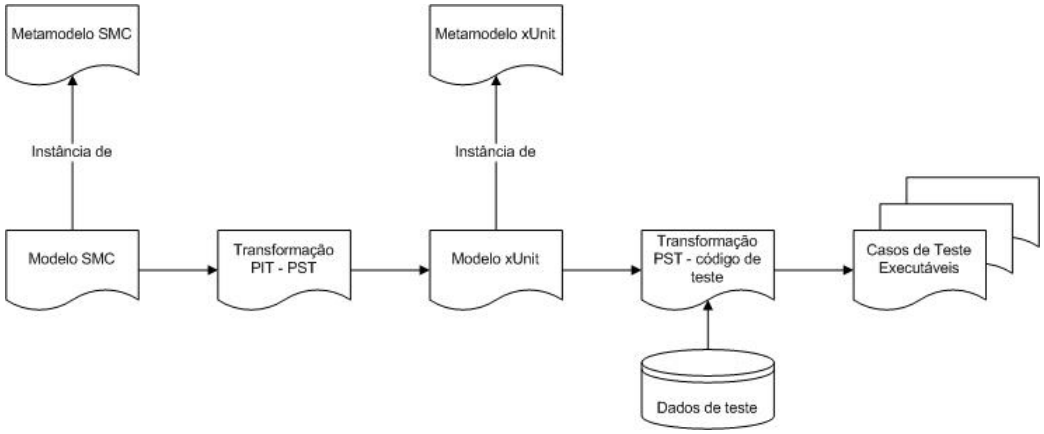


Figura 4.1: Processo de transformação de casos de teste abstratos em executáveis.

O PIT proposto por Javed *et al.* é chamado de SMC (*Sequence of Method Calls*) e representa uma sequência de chamada de métodos, enquanto que o PST é denominado xUnit por representar a família de casos de teste de unidade. Já as regras de transformação PIT-PST e PST-código de teste (em formato de teste de unidade para Java ou Smalltalk) são escritas nas linguagem QVT e Mofscript, respectivamente.

4.3 Aplicações Web

"Um novo termo foi adicionado ao vocabulário de tecnologia nos últimos anos: aplicações Web" [21]. Uma aplicação Web é um programa que roda inteira ou parcialmente em um ou mais servidores Web e que pode ser visualizada pelos usuários através de um *Web site* [10]. Apesar de ter uma definição simples, muitos conceitos estão envolvidos em uma aplicação Web, como o próprio termo *Web site*. Abaixo descrevemos alguns deles:

- Uma *página Web* é a informação que pode ser vista em uma janela do navegador.

Ela ser pode armazenada como um HTML estático ou ser dinamicamente gerada por software, como um JSP ou um Java Servlet.

- Um *Web site* é uma coleção de páginas Web e os componentes de software associados, que são relacionados semanticamente por conteúdo e sintaticamente por links e outros mecanismos de controle.

Recentemente, esse tipo de sistema de software cresceu extraordinariamente ao redor do mundo [40]. Aplicações Web fornecem um mecanismo eficiente para a construção de pontos de vista diferentes para bancos de dados corporativos. Por exemplo, enquanto os clientes usam uma livraria online através de interface Web particular, gestores ou técnicos podem visualizar a mesma informação através de uma aplicação Web diferente (e, obviamente, direitos de acesso diferentes) em uma Intranet. No entanto, essas visões são mais do que pontos de vista de banco de dados simples, pois envolvem diferentes caminhos de navegação, índices, etc [59].

Mais ainda, elas são populares devido à universalidade de navegadores web, e a conveniência de usar um navegador web como um cliente. A capacidade de atualizar e manter aplicações Web sem distribuir e instalar software em potencialmente milhares de computadores de clientes é uma das principais razões para sua popularidade, assim como o fato de ter compatibilidade com muitas plataformas. Os exemplos mais comuns de aplicações Web são: webmail, *e-banking*, *e-business* (vendas no varejo *online*), noticiários *online* e wikis.

4.4 Transformação de casos de teste abstratos em executáveis para aplicações Web

Nesta seção apresentados como foi a instanciação dos artefatos de MDT propostos por Javed *et al.* [36] (modelos de teste e regras de transformação) para gerar casos de teste executáveis que sejam adequados para teste de aplicações Web, ou seja, casos de teste executáveis cujo formato é diferente dos providos por Javed *et al.*, pois estes se limitam a teste de unidade para as linguagens Java e Smalltalk.

4.4.1 Metamodelo SMC (*Sequence of Method Calls*)

O metamodelo SMC (*Sequence of Method Calls*), proposto por Javed *et al.* [36], representa um diagrama de sequência, com suas interações, mensagens, classes, parâmetros e valores esperados. O diagrama de sequência é um dos diagramas de interação da UML (i.e., aqueles que descrevem como um grupo de objetos colaboram em um determinado

comportamento) que mostra a interação entre objetos com a preocupação de documentar os métodos executados ao longo do tempo. Um diagrama de seqüência possui duas dimensões: vertical, representando o tempo; e horizontal, representando os diferentes objetos.

Mesmo que a idéia inicial dos autores desse metamodelo tenha sido deixá-lo em concordância com um diagrama de sequência, tal metamodelo pode ser expandido para representar casos de teste abstratos. Por exemplo, o metamodelo SMC pode representar também os casos de teste abstratos obtidos por outro tipo de modelo, como o diagrama de estados[1].

Mais ainda, quando lidamos com geração de teste manual, mas que também envolva chamadas de métodos, o metamodelo SMC também se aplica.

Descrição do metamodelo SMC

O metamodelo SMC é ilustrado na Figura 4.2. Mesmo tendo sido criado com base em um diagrama de sequência, ele ignora os aspectos mais complexos de tal diagrama (como, por exemplo, conectores), sendo basicamente composto por interações, mensagens, classes, parâmetros e valores esperados. A seguir explicaremos cada item e suas respectivas relações.

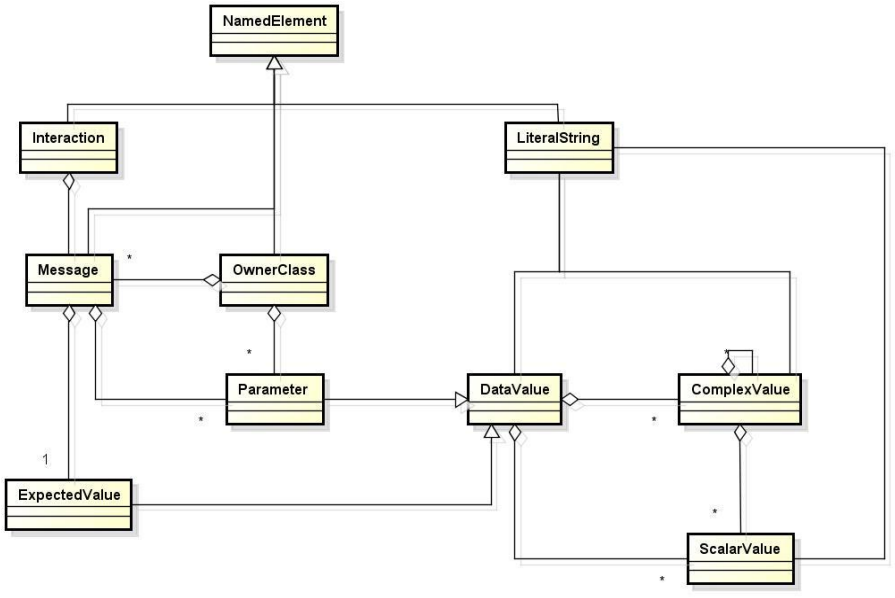


Figura 4.2: Estrutura do metamodelo SMC [36].

[1]O diagrama de máquina de estados tem como elementos principais o estado, que modela uma situação em que o elemento modelado pode estar ao longo de sua existência, e a transição, que leva o elemento modelado de um estado para o outro. Tanto o diagrama de estados como o diagrama de sequência são considerados diagramas comportamentais da UML.

- Elemento nomeado (*Named element*): representa um elemento que é nomeado;

- Interação (*Interaction*): representa uma parte de um diagrama de sequência;

- Mensagem (*Message*): fica contida em uma interação e faz parte do subconjunto de chamadas de métodos do diagrama de sequência selecionado pelo testador. Uma mensagem pode ter parâmetros (*Parameter*) e resultados esperados (*Expected value*);

- Parâmetro (*Parameter*): elemento de uma mensagem que representa os valores de parâmetro que esta espera receber. Pode ser de um tipo escalar (*Scalar value*) ou complexo (*Complex value*);

- Valor esperado (*Expected value*): também é um elemento de uma mensagem, opcional, que indica se esta tem um resultado esperado a ser verificado. Assim como o parâmetro, pode ser do tipo escalar ou complexo;

- Classe proprietária (*Owner class*): representa a classe a que método pertence, ou seja, é a classe que recebe a mensagem. Ela possui parâmetros para seus construtores que são necessários para criar uma instância dessa classe no caso de teste executável;

- Valor escalar (*Scalar value*): é um valor atômico que não contém nenhum outro dado. Em linguagens de programação, como Java ou C++, podem ser *int*, *float*, *String*, etc.;

- Valor complexo (*Complex value*): são valores que contém outros valores, como uma estrutura de dados.

Este modelo, parcialmente ilustrado na Listagem 4.1, é descrito no formato "Ecore", que é a linguagem padrão para especificação de metamodelos da ferramenta Eclipse [3].

```
<?xml version="1.0" encoding="ASCII"?>
<ecore:EPackage xmi:version="2.0"
  name="smc" nsURI="http://smc" nsPrefix="smc">
  <eClassifiers xsi:type="ecore:EClass" name="NamedElement">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name">
      <eType xsi:type="ecore:EDataType"
        href="http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
    name="Model" eSuperTypes="//NamedElement">
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="interaction" upperBound="-1" eType="//Interaction"
      containment="true" eOpposite="//Interaction/owner"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
    name="Interaction" eSuperTypes="//NamedElement">
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="owner" lowerBound="1" eType="//Model"
      eOpposite="//Model/interaction"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="message" upperBound="-1"
      eType="//Message" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
  name="Message" eSuperTypes="//NamedElement">
  ...
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
    name="Parameter" eSuperTypes="//DataValue">
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="owner" lowerBound="1" eType="//Message"
      defaultValueLiteral="no" eOpposite="//Message/parameter"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
    name="ExpectedValue" eSuperTypes="//DataValue">
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="owner" lowerBound="1"
      eType="//Message" eOpposite="//Message/expectedValue"/>
  </eClassifiers>
  ...
</ecore:EPackage>
```

Listagem 4.1: Formato “Ecore” do metamodelo SMC

Modelagem de Aplicações Web para obtenção do Modelos SMC

Para que um caso de teste abstrato de uma aplicação Web seja escrito no formato SMC, precisamos definir a correlação entre os elementos do metamodelo SMC e os elementos de um caso de teste normalmente aplicado a este tipo de aplicação.

Primeiramente, os elementos básicos de uma aplicação Web são suas páginas, que permitem que as funcionalidades do sistema sejam exercitadas. Este comportamente se assemelha ao de uma classe que possui métodos que provêm funcionalidades, de forma que um diagrama de classes seria adequado para modelar a estrutura de uma aplicação Web.

Em um trabalho relacionado, Bellettini *et al.* [16] sugerem uma abordagem para modelagem e geração de casos de teste para aplicações Web com base em diagramas de classe e estados, sendo que o primeiro é aplicado para representar a estrutura e os componentes da aplicação Web, enquanto que o segundo modela a navegação.

O diagrama de classe proposto por Bellettini *et al.* leva em conta não somente páginas Web, mas também seus componentes, como forms, scripts, etc., e tal diagrama é derivado, por engenharia reversa, do código fonte. O mesmo se aplica para o diagrama de estados, com a diferença que neste acrescenta-se as informações de fluxo de navegação entre os componentes da aplicação Web.

A proposta apresentada por eles poderia ser aplicada conjuntamente como nosso trabalho, mais especificamente para geração dos casos de teste abstratos. Entretanto, em nosso caso, levamos em conta o fato que muitas vezes a equipe de teste não tem acesso ao código fonte e, portanto, não podemos aplicar diretamente a proposta de Bellettini *et al.*.

Sendo assim, adaptamos os modelos propostos por Bellettini *et al.* para levar em conta apenas as páginas Web e suas funcionalidades. A Figura 4.3 ilustra o metamodelo do diagrama de classe que utilizamos neste trabalho. Neste metamodelo, as páginas Web contém funcionalidades que por sua vez apresentam parâmetros que precisam ser preenchidos para acionar a respectiva funcionalidade.

Com relação ao diagrama de estados, também desejamos empregá-lo para representar a navegação da aplicação Web. Winckler *et al.* [67] também propõem o uso de modelo de estados para modelar a navegação de uma aplicação Web, de uma maneira que se adequa ao nosso contexto de não ter acesso ao código fonte e não precisar representar todos os componentes, apenas a página Web e suas funcionalidades. Para eles, cada página Web é considerada um agrupamento de objetos e está associada a um estado. Os links e outros objetos interativos que acionam transições são representados por eventos.

Neste trabalho, como em Winckler *et al.*, também adotamos que cada página Web é um estado. Já as transições são as funcionalidades que podem ser acionadas pela página Web e que são representadas pelos métodos no diagrama de classes. Dessa forma, conseguimos aqui ver a clara relação entre os elementos dos dois diagramas (de classes e

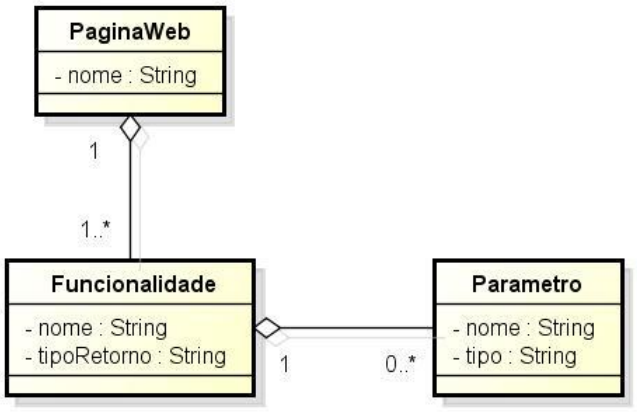


Figura 4.3: Metamodelo dos diagramas de classe para modelagem da estrutura de aplicações Web.

estados): as páginas Web, que são as classes no diagrama de classes, tornam-se os estados no diagrama de estados; já as funcionalidades da página Web, que são os métodos do diagrama de classe, são as transições do diagrama de estados. Além disso, cada diagrama tem sua função: enquanto o diagrama de classes representa a estrutura da aplicação Web, o diagrama de estados detalha o fluxo de navegação.

Especificados os diagramas a serem empregados na modelagem de aplicações Web, detalhamos aqui o roteiro para criação de um caso de teste abstrato no formato SMC:

1. Baseado na especificação e na interface gráfica da aplicação Web, elabore um diagrama de classes, onde uma classe representa uma página da aplicação Web e cada operação da classe, com seus respectivos parâmetros, representa uma funcionalidade daquela página e seus dados de entrada.

2. Se for usado um diagrama de estados para geração dos casos de teste abstratos, então:

   Crie o diagrama levando em conta que os estados correspondem a uma classe do diagrama de classes, enquanto que as transições estão relacionados aos métodos de uma classe, indicando uma mudança de página na aplicação Web quando uma funcionalidade é executada.

   Derive os casos de teste abstratos da ferramenta (usualmente através de uma ferramenta de Teste Baseado em Modelo). Tais casos de teste devem estar no formato SMC, onde as mensagens destes são equivalentes aos métodos do diagramas de classe e tais mensagens possuem uma classe proprietária e parâmetros (também de acordo com o diagrama de classes).

3. Caso contrário, então:

   Escreva (ou obtenha por outra técnica) os casos de teste abtratos da aplicação Web em formato SMC.

4. Além disso, em aplicações Web ainda podem ser aplicados os valores esperados (elemento "Valor esperado" do metamodelo SMC), que normalmente serão mensagens de sucesso ou falha; ou o preenchimento de um determinado campo baseado nos dados de entrada. Neste caso, mapeie tais validações para operações do diagrama de classes e, consequentemente para passos do caso de teste.

Exemplos de um diagrama de classes e de um passo de seu caso de teste para uma aplicação Web de controle de produtos podem ser vistos nas Figuras 4.4 e 4.5. Podemos observar que as classes não têm relação umas com as outras, pois a navegabilidade entre elas é representada pelo diagrama de estados.

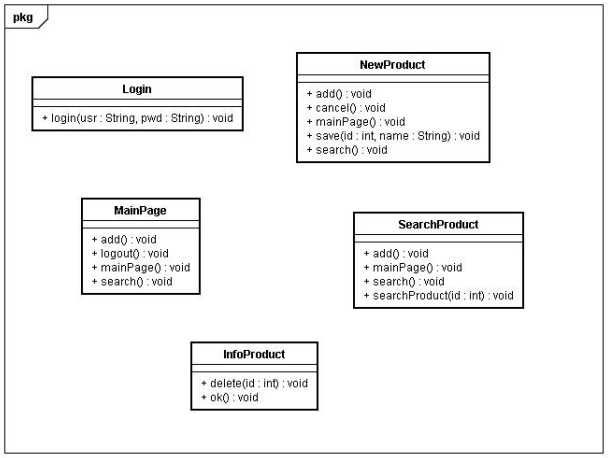


Figura 4.4: Sistema de controle de produtos: diagrama de classe.

```
<message name="login" order="4">
  <OwnerClass name="Login"/>
  <parameter name="usr" setter="" type="String"/>
  <parameter name="pwd" setter="" type="String"/>
</message>
```

Figura 4.5: Formato SMC do evento de entrada "login".

Dessa forma, executando este processo para todos os fluxos da aplicação Web, obtém-se os casos de teste abstratos em formato SMC.

4.4.2 Metamodelo xUnit

O metamodelo xUnit representa a família de *frameworks* utilizada para teste de unidade. Segundo Javed *et al.* [36], não existia um metamodelo que representasse tal família anteriormente; então baseando-se no estudo da arquitetura de casos de teste escritos com o apoio de diferentes *frameworks* de teste de unidade, derivaram tal metamodelo. O *framework* mais conhecido para teste de unidade é o JUnit.

Em nosso trabalho, usaremos as instâncias deste metamodelo como Modelos de Teste Específico de Plataforma (PST), pois já definimos que nossos casos de teste terão formato JUnit com as particularidades para aplicações Web.

A Tabela 4.1 apresenta os elementos do metamodelo xUnit e a relação entre ele e o metamodelo SMC, enquanto que a Figura 4.6 ilustra a relação entre os elementos.

Tabela 4.1: Mapeamento de elementos do metamodelo SMC para elementos do metamodelo xUnit.

Elementos SMC	Elementos xUnit
Modelo SMC	O modelo SMC com suas respectivas interações gera uma `suite de teste`.
Interação	Cada interação implica em um `caso de teste`.
Mensagem	Para cada mensagem, uma `asserção` e um `método` (dentro da asserção) são gerados.
Parâmetro	Para cada parâmetro em uma mensagem do modelo SMC, é criado um `parâmetro` no método do modelo xUnit.
Valor esperado	É o mesmo caso de *Parâmetro*. Para o valor esperado de uma mensagem do modelo SMC, é criado um `valor esperado` no método do modelo xUnit.
Classe proprietária	Se uma mensagem do modelo SMC possui uma classe proprietária, haverá uma `classe proprietária` correspondente no método do modelo xUnit.
Valor escalar	Valor escalar
Valor complexo	Valor complexo

4.4.3 Regras de transformação

Javed *et al.* [36] definem regras de transformação SMC-xUnit (PIT-PST) e xUnit-código de teste (PST-código de teste), usando, no primeiro caso, QVT (*Query/View/Transformation*) e, no segundo, Mofscript. Apesar de disponibilizar os arquivos correspondentes a tais regras, não foi possível encontrar uma ferramenta que executasse a transformação

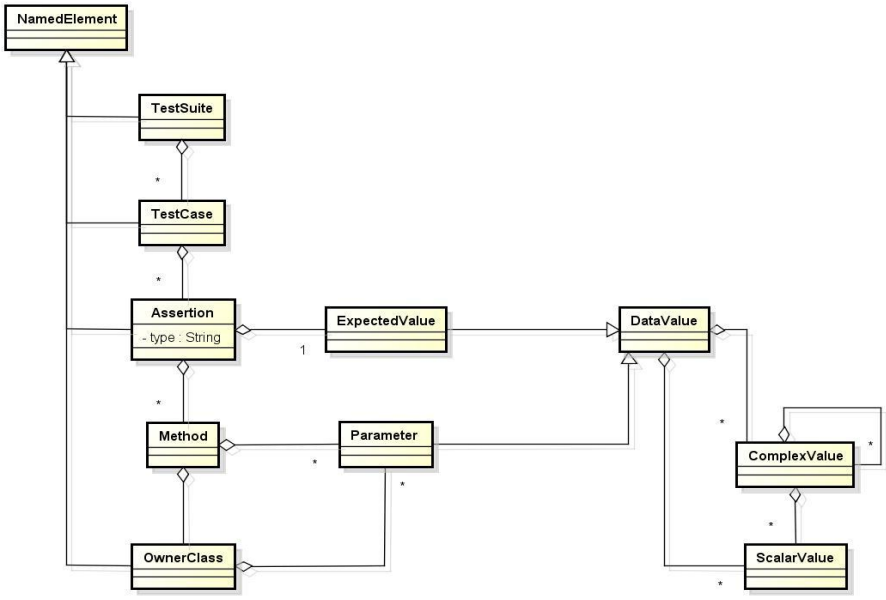


Figura 4.6: Estrutura do metamodelo xUnit [36].

escrita em QVT. Por esse motivo, e por ATL (*Atlas Transformation Language*) ser mais difundida na indústria que QVT e possuir mais documentação e ferramentas de apoio, reescrevemos a transformação QVT em ATL.

Além disso, realizamos algumas correções nas transformações após aplicar o processo de teste, uma vez que Javed *et al.* não o tinham feito formalmente. Tal processo de teste é descrito no Capítulo 5.

A seguir descrevemos as transformações de modelo, apresentando tanto o mapeamento de instâncias do metamodelo SMC em instâncias do metamodelo xUnit, que envolve os modelos independente e específico de plataforma, como os detalhes da transformação entre instâncias do metamodelo xUnit e o código de teste no formato JUnit para aplicações Web envolvendo a API da ferramenta Selenium.

Transformação de modelos SMC para modelos xUnit

Baseado nos casos de teste no formato SMC, obter os correspondentes no formato Selenium não é uma tarefa simples. Existe uma lacuna entre os dois formatos de teste: o primeiro está em um nível mais abstrato do que o segundo, porque tem uma conexão direta com a especificação de aplicação web, enquanto o formato executável deve levar em conta detalhes do desenvolvimento a fim de desencadear ações em elementos da interface de usuário da aplicação web.

Sendo assim, é necessário aplicar a transformação PIT-PST para realizar a mudança

do nível de abstração do modelo para a implementação. Para isso, empregamos o Eclipse IDE, com o Eclipse Modelling Framework (EMF) instalado, de modo a executar o código ATL da transformaçãp PIT-PST.

A Figura 4.7 mostra o formato xUnit correspondente ao formato SMC da Figura 4.5. Podemos ver que a diferença principal entre elas é que uma mensagem do modelo de SMC não é diretamente mapeada para um método do modelo xUnit, mas sim para uma asserção, que é uma condição que deve verdadeiras após a execução do caso de teste. A assertiva em si contém a chamada de método e um valor esperado. Esta diferença foi observada por Javed quando realizou o mapeamento de testes, escritos como diagrama de seqüência, para um modelo xUnit.

```
<assertion name="MODEL_1_login" order="4"
 assertionType="assertTrue">
  <method name="efetuarLogin">
    <parameter type="String" setter="" name="usr"/>
    <parameter type="String" setter="" name="pwd"/>
    <ownerClass name="Login"/>
  </method>
</assertion>
```

Figura 4.7: Formato xUnit do evento de entrada login.

Transformação de modelo xUnit para o formato Selenium

O formato xUnit deve agora ser transformado em código-fonte, ou seja, uma transformação PST-código de teste ainda é necessária. Por esta razão, é preciso aplicar uma transformação de modelo-para-texto, que pode ser feita facilmente usando MOFScript (padrão da OMG).

MOFScript descreve como o modelo deve ser mapeado para o seu código de teste correspondente. Apesar deste script ter sido disponibilizado por Javed, precisamos adaptá-lo para usar as bibliotecas da Selenium.

Além disso, é nesta etapa que são adicionados os dados de teste (valores de parâmetros e resultados esperados). Estes dados são descritos em arquivos separados, de modo que, em caso de mudança nos dados, os modelos de teste não precisem ser alterados, bastando apenas gerar novamente os casos de teste executáveis. Mais ainda, mudando apenas os dados de teste, mais de um caso de teste executável pode ser gerado para o mesmo caso de teste abstrato.

Para obter o código de teste usando o script Mofscript, deve ser utilizado o Eclipse IDE, analogamente à transformação PIT-PST.

Antes de executar os casos de teste Selenium, porém, é preciso prestar atenção a um detalhe: o caso de teste executável irá realizar chamadas de método, de acordo com o diagrama de classes. No entanto, estes métodos não correspondem à implementação do

sistema, uma vez que estamos testando uma interface web, e não uma interface API-like (não estamos lidando com testes de unidade, mas com o teste caixa preta). Então é necessário adicionar detalhes da plataforma específica, a fim de mapear a aplicação de tais métodos aos comandos Selenium, que correspondem a ações que os usuários poderiam executar na interface gráfica da aplicação web.

A Figura 4.8 mostra um caso de teste abstrato e sua versão executável. A implementação do método `login` chamado no caso de teste é apresentada na Figura 4.9.

Abstract Test Case

Path = {t1: login("admin","p2011"), t8: search(), t12: searchProduct(57)}

Executable Test Case

```
package testProduct;
import selenium.utils.logging.LoggingDefaultSelenium;

import test.ProductTestCase;

public class Test_MODEL_1 extends ProductTestCase {

  public void testMODEL_1( ) {
    try {

        //Prepare constructor parameter object
        ProductTestCase testCase = this;

        //Prepare constructor parameter object
        LoggingDefaultSelenium selenium = this.selenium;
        Login login = new Login(testCase, selenium);
        MainPage mainPage = new MainPage(testCase, selenium);
        SearchProduct searchProduct = new SearchProduct(testCase, selenium);

        //Prepare parameter object
        String usr = "admin";

        //Prepare parameter object
        String pwd = "p2011";

        //Prepare parameter object
        int id = "57";

        login.login(usr, pwd);
        mainPage.search();
        searchProduct.searchProduct();
    } catch ( Exception exp ) {
                System.out.println( exp.toString() );
                fail("Exception occured during test case execution");
    }

    if (isFail()) {
                fail();
    }
  } //End of Test Method
} //End of Test Case
```

Figura 4.8: Casos de teste: abstrato X executável.

```
public class Login {

  private static String txtUsr = "//*[@id=\"USER\"]";
  private static String txtPwd = "//*[@id=\"PASSWORD\"]";
  private static String btnOk = "//p/a/img[contains(@src, 'ok')]";

  /**
      ...
  */

  public void login(String usr, String pwd) {
        selenium.open("/homologa");

        // Wait the screen to be loaded.
        while (!selenium.isElementPresent(txtUsr));

        selenium.type(txtUsr, usr);
        selenium.type(txtPwd, pwd);
        selenium.click(btnOk);

        selenium.waitForPageToLoad("30000");
  }
}
```

Figura 4.9: Implementação do método `login`.

Capítulo 5

Testes das transformações de modelos

"Transformações de modelos são cidadãos de primeira classe em Engenharia Dirigida por Modelo (MDE)" [46].

Nossa aplicação de Teste Dirigido por Modelo (MDT) para geração automática de casos de teste executáveis aplica duas transformações de modelo para obtenção do código de teste a partir do modelo de teste independente de plataforma e qualquer erro em uma delas pode comprometer o resultado final.

Na busca por garantir que uma transformação de modelos esteja correta, existem metodologias que apresentam grandes diferenças, variando desde provas formais [34] à aplicação de técnicas comuns de teste, levando-se em conta que agora os dados de entrada são modelos que têm características particulares [29], tais como entidades, atributos e relacionamentos. Alguns trabalhos encontrados na literatura não propõem como testar ou qual critério de teste deve ser utilizado, mas apenas apresentam como gerar os dados de teste (i.e. modelos de testes), dado um metamodelo que descreve o domínio de entrada da transformação de modelo [19].

Este capítulo apresenta como foi conduzido o processo de teste das transformações de modelos, que se baseou em duas metodologias: *i*) uma primeira que descreve um *framework* de alto nível para transformações de modelo, sem fornecer detalhes de implementação; e *ii*) outra, proposta por Fleurey *et al.* [29], que é uma adaptação para teste de transformações de modelo da técnica clássica de teste chamada "partição de equivalência", e que foi escolhida devido a sua independência da linguagem da transformação de modelo e a existência de uma ferramenta associada.

A organização deste capítulo se dá da seguinte maneira: a Seção 5.1 apresenta algumas técnicas existentes para teste de transformações de modelo; a Seção 5.2 descreve os passos empregados na atividade de teste; já a Seção 5.3 discute os resultados e, finalmente, a

Seção 5.4 apresenta algumas lições aprendidas.

5.1 Abordagens para teste de transformação de modelo

Segundo Lin *et al.* [44] é imprescindível assegurar que uma transformação de modelo esteja correta antes de sua aplicação. Apesar de não terem criado uma metodologia nova, eles identificaram boas práticas para mostrar a corretude de transformações de modelos.

A primeira possibilidade levantada é o uso de provas formais. Hulsbusch *et al.* [34] também sugerem esta abordagem, mais especificamente com o uso de grafos para representar os metamodelos (dos modelos de entrada e saída). Outra possibilidade é a execução de testes, já que uma transformação de modelo nada mais é do que um programa. Esta última tem várias vantagens que a tornam um método eficaz para determinar se uma tarefa (por exemplo, a transformação do modelo) foi realizada corretamente. Estas vantagens são:

1. A relativa facilidade com que muitas das atividades de teste podem ser realizadas;
2. O software a ser desenvolvido (por exemplo, as transformações de modelo) pode ser executado no ambiente de produção;
3. A maior parte do processo de teste pode ser automatizada.

"Tal como teste de software, teste de transformação de modelo só pode mostrar a presença de erros e não a sua ausência. No entanto, como uma alternativa para os métodos formais, pode ser muito eficaz em revelar tais erros" [44].

Dadas tais características, Lin *et al.* definem o teste de transformação de modelo como mostra a Definição 2.

Definição 2 *O processo que envolve a execução de transformações determinísticas com dados de teste e comparação dos resultados obtidos com os resultados esperados, que devem satisfazer aos requisitos da transformação. Se há diferenças entre os resultados obtido e esperado, a transformação de modelo precisa ser corrigida.*

Como um protótipo inicial, a Figura 5.1 mostra um framework para o teste de transformação de modelos que pode ser usado como um guia para geração, execução e documentação dos testes. Existem três componentes básicos para o framework de testes: o construtor do casos de teste, o motor de teste e analisador de teste. O construtor de casos de teste, baseado no metamodelo de entrada, produz casos de teste para a transformação a ser testada. Os casos de teste gerados são passados para o motor de teste para

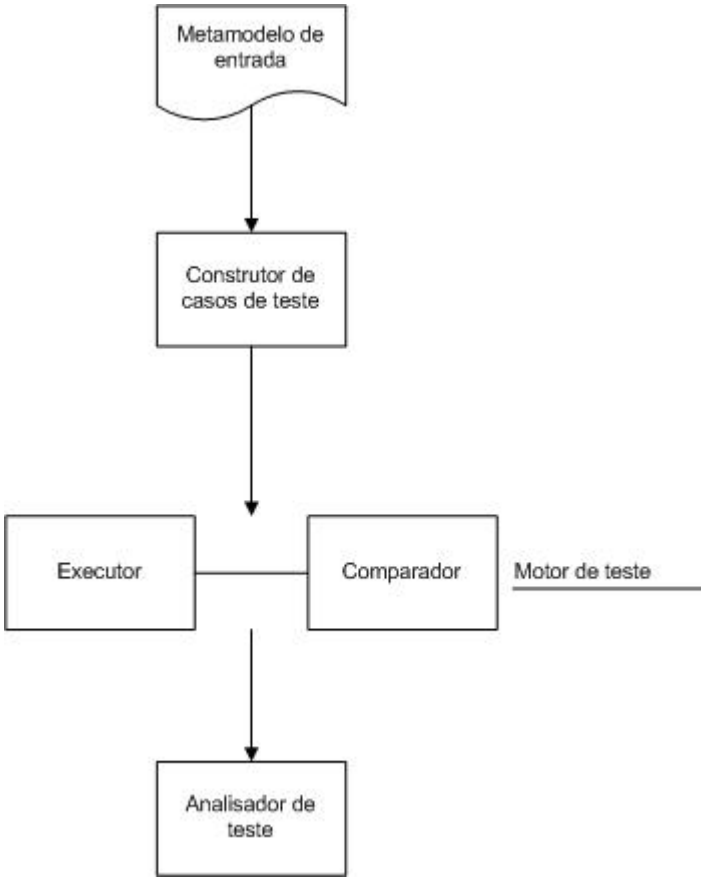


Figura 5.1: Framework para teste de transformações de modelo proposto por Lin *et al.* [44].

exercitá-los. Dentro do motor de testes, há um executor e um comparador. O executor é responsável por executar a transformação do modelo entrada; já o comparador verifica se o modelo de saída está em conformidade com o modelo esperado, recolhendo os resultados de comparação. Finalmente, o analisador de teste provê a visualização dos resultados fornecidos pelo comparador.

Tendo em mente as etapas deste framework, é possível encontrar na litetura trabalhos que almejam cobrir cada uma delas individualmente, ou um conjunto delas. Por exemplo, Brottier *et al.* [19] desenvolveram um protótipo de ferramenta, chamado OMOGEN (*Automatic Model Generator*), que se encaixa no framework como o construtor de casos de teste.

Brottier *et al.* [19] definiram um algoritmo para automatizar a geração de modelos de teste. Tal algoritmo usa um metamodelo e fragmentos de modelos como entrada e produz um conjunto de modelos de teste. Os fragmentos de modelo são ou fornecidos pelo testador ou derivados do metamodelo. Eles especificam as partes do metamodelo que devem ser

instanciadas com valores específicos que são interessantes para o teste. Então, dadas as entradas, o algoritmo consiste em combinar fragmentos de modelo e completá-los para construir instâncias válidas do metamodelo. Em seu trabalho, Brottier *et al.* apresentam algumas estratégias utilizadas para combinar e completar um modelo [19].

Apesar deste protótipo de ferramenta se encaixar perfeitamente no construtor de casos de teste do framework de Lin *et al.*, ele não foi disponibilizado pelos autores.

Também atendendo aos requisitos do contrutor de casos de teste, mesmo que de forma parcial, Fleurey *et al.* [29] apresentam uma metodologia de qualificação da relevância dos modelos de entrada para teste. Diferentemente da proposta de Brottier *et al.*, esta metodologia não gera os modelos de teste automaticamente, mas os avalia com relação à cobertura dos requisitos de teste do metamodelo de entrada.

Basicamente, o framework visa selecionar e qualificar os modelos de teste para a validação das transformações de modelo. O critério de adequação proposto é caixa-preta, mais especificamente por dois motivos: *i*) independência da linguagem utilizada para a transformação de modelo; e *ii*) para tirar proveito da descrição completa do domínio de entrada fornecida pelo metamodelo de entrada da transformação.

Fleurey *et al.* reforçam que é importante que a abordagem proposta seja genérica e compatível com qualquer linguagem de transformação de modelo, porque atualmente existem muitas linguagens de transformação e nenhuma delas se destacou como melhor ou mais popular. No caso de nosso trabalho, a independência de linguagem é um requisito essencial, pois lidamos com duas transformações escritas em linguagens diferentes, ATL e Mofscript.

A segunda razão pela qual foi escolhido o critério caixa-preta é aproveitar o fato do domínio de entrada para uma transformação ser definido por um metamodelo, pois este especifica completamente o conjunto de modelos de entrada possíveis para uma transformação. Neste contexto, a idéia é avaliar a adequação dos modelos de teste com relação a sua cobertura da entrada do metamodelo. Por exemplo, os modelos de teste devem instanciar cada categoria e cada relação do metamodelo de entrada pelo menos uma vez.

Neste trabalho, decidimos realizar os testes das transformações de modelo seguindo as etapas do framework proposto por Lin *et al.* [44]. Como base para o construtor de casos de teste aplicamos a proposta de Fleurey *et al.* [29] para qualificar os modelos de entrada.

Mais detalhes sobre cada uma das etapas do processo de teste das transformações de modelo podem ser vistos na Seção 5.2.

5.2 Aplicação da metodologia para teste das transformações

Nesta seção serão apresentadas cada uma das etapas da atividade de teste das transformações de modelo. Ela está dividida de acordo com os componentes do framework de teste proposto por Lin *et al.* [45], sumarizando quais as metodologias e ferramentas usadas em cada etapa.

5.2.1 Construtor de casos de teste

No teste de uma transformação do modelo, existem diferentes formas para construir casos de teste. Por exemplo, os testes podem ser construídos de acordo com critérios de cobertura ou especificações de teste [45]. Fleurey *et al.* [29] fizeram um framework, que utilizaremos neste trabalho, para definir um critério de teste para transformação de modelos. Além da definição de tais critérios, foi provida uma ferramenta, chamada *Metamodel Coverage Checker*(MMCC) que automaticamente informa qual a cobertura dos modelos de teste com relação a um determinado critério.

Os critérios de teste propostos são uma adaptação da abordagem clássica de teste chamada partição de equivalência. A idéia básica das estratégias de partição de equivalência é dividir o domínio de entrada em subdomínios e então selecionar dados de teste de cada um desses intervalos. Os intervalos para um domínio de entrada definem uma partição do domínio de entrada e, portanto, não devem se sobrepor [29]. No caso específico de transformação de modelo, o domínio de entrada é modelado pelo metamodelo de entrada da transformação. Sendo assim, a idéia é definir partições para cada propriedade[1] do metamodelo. Uma definição mais precisa de uma partição é dada pela Definição 3.

Definição 3 *Uma partição de um conjunto de elementos é uma coleção de n intervalos A*1*, ..., A*n *tais que A*1*, ..., A*n *não se sobrepõem, e a união de todos os subconjuntos forma o conjunto inicial. Esses subconjuntos são chamados intervalos.*

Como estratégias simples (por exemplo, cobrir independentemente os valores e multiplicidades de cada propriedade) não são suficientes para proporcionar um suporte satisfatório para a seleção de modelos de teste, Fleurey *et al.* [29] propuseram três elementos principais a partir dos quais definiram as combinações específicas de faixas para as propriedades que devem ser cobertas pelos modelos de teste. Tais elementos são: fragmento de modelo, fragmento de objeto e restrição de propriedade.

[1]Propriedade é a representação compacta tanto para atributos de uma classe do metamodelo como para a relação entre classes.

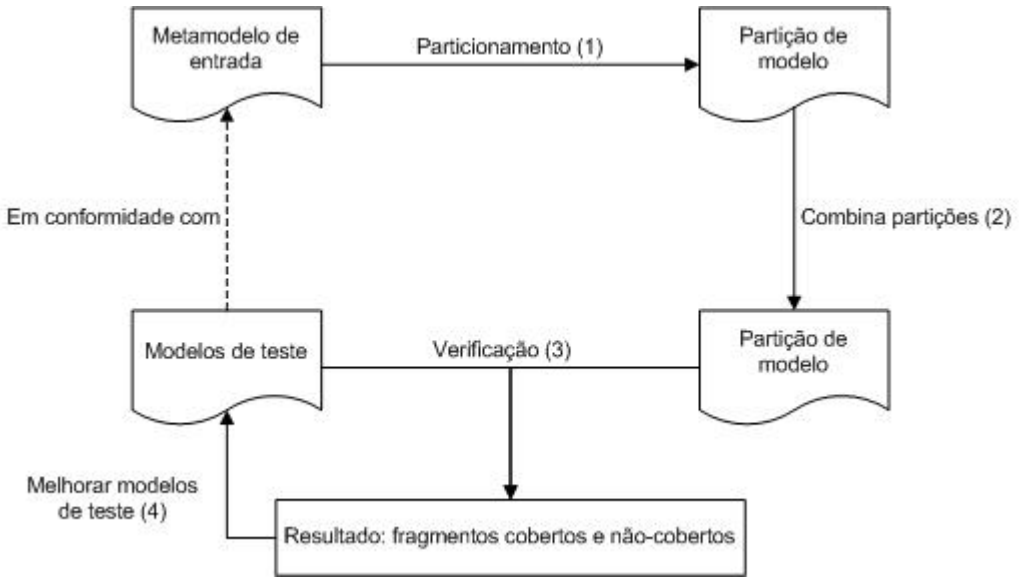


Figura 5.2: Processo para seleção de modelos de teste proposto por Fleurey *et al.* [29].

Um fragmento de modelo é composto de um conjunto de fragmentos de objeto. Um fragmento de objeto, por sua vez, é um conjunto de restrições de propriedade que especificam os intervalos que devem exercitados.

Baseados nesses conceitos, foi definido um processo iterativo para a seleção dos casos de teste (i.e. modelos de teste) que serão fornecidos como entrada para a transformação. Tal processo é dito iterativo, pois muitas vezes é necessário aumentar o conjunto de casos de teste para atingir a cobertura desejada com relação ao metamodelo de entrada. A Figura 5.2.1 mostra esse processo.

Os passos do processo são descritos a seguir:

- *Entradas:* o processo espera duas entradas: o metamodelo de entrada da transformação de modelo em teste e o conjunto de modelos de teste.

- *Passos:*

  1. A partir do metamodelo de entrada, as partições são obtidas para cada propriedade do metamodelo;
  2. Depois disso, as partições são combinadas para construir um conjunto de fragmento de modelo. Para construir tal conjunto, é necessário especificar um critério de teste que define como os fragmentos devem ser compostos;
  3. Verifica-se se há pelo menos um modelo de teste para cobrir cada fragmento de modelo;

4. Se existirem fragmentos de modelo não cobertos e a cobertura desejada não foi alcançada, novos modelos de teste devem ser adicionados com o intuito de cobrir os fragmentos remanescentes.

Este processo não só permite uma estimativa da qualidade de um conjunto de modelos de teste, mas também fornece aos testadores informações importantes para melhorar o conjunto de teste. As etapas 1, 2 e 3 são realizadas pela ferramenta MMCC.

Critérios de teste

Um critério de teste é uma regra que se utiliza para selecionar requisitos de teste e avaliar se eles foram satisfeitos. No caso da abordagem que estamos utilizando para qualificar os modelos de teste, os requisitos de teste são basicamento os fragmentos de modelo. Dessa forma, é natural que com base neles sejam derivados um ou mais critérios de teste. A ferramenta MMCC permite que seja utilizado um dos seguintes critérios: "Todos–Intervalos" e "Todas–Partições", definidos como mostram as Definições 4 e 5.

Definição 4 *Critério "Todos–Intervalos": cada intervalo de cada partição de todas as propriedades do metamodelo deve estar em pelo menos um fragmento de modelo.*

Definição 5 *Critério "Todas–Partições": requer que valores de todos os intervalos de uma partição sejam usados simultaneamente em um único modelo de teste.*

A ferramenta MMCC está disponível na forma de um projeto do Eclipse IDE que deve ser executado em dois estágios: o primeiro, de acordo com os passos 1 e 2, geram os fragmentos do metamodelo de entrada da transformação; e o segundo, de acordo com o passo 3, mede a cobertura dos modelos de teste em relação aos fragmentos de modelo, seja qual for o critério escolhido.

O primeiro estágio é implementado como uma transformação de modelo cuja entrada envolve dois parâmetros: o metamodelo de entrada e o critério de teste. A saída deste estágio é um modelo contendo um conjunto de partições e fragmentos que estão em conformidade com o critério de teste estabelecido.

Tal saída, juntamente com os modelos de teste, são as entradas para o segundo estágio que verifica se o conjunto de modelos de teste satisfaz o critério de teste. Neste caso, como saída do estágio, são apresentados os fragmentos de modelo que não foram cobertos. Se a saída for vazia, então o conjunto de modelo de teste satisfaz ao critério; caso contrário, é necessário analisar manualmente os fragmentos remanescentes para entender o motivo de eles não terem sido cobertos.

5.2.2 Motor de teste

Depois de construir o conjunto de modelos de teste, as transformações de modelos devem ser executadas e este processo é realizado pelo motor de teste.

O motor de teste é composto de um executor e de um comparador. A função do executor é executar as transformações de modelos, gerando o modelo de saída (na plataforma desejada) a partir dos modelos de entrada. Neste trabalho, nossa opção foi por usar o ambiente integrado de desenvolvimento Eclipse como executor, pois ele oferece *plugins* para executar transformações escritas em ATL e Mofscript. Dessa forma, uma grande vantagem dessa escolha é poder executar as duas transformações no mesmo ambiente, bem como facilmente visualizar os modelos.

Já o comparador é basicamente responsável por comparar o resultado esperado com o obtido e coletar os resultados desta comparação. Uma vez que o resultado esperado não pode ser automaticamente gerado, decidimos aplicar a comparação manual.

5.2.3 Analisador de teste

O analisador de teste, de acordo com o framework proposto por Lin *et al.* [44], deve permitir a visualização dos resultados providos pelo comparador, além de prover a capacidade de navegar entre as diferenças. Como nosso comparador é manual, consequentemente a análise dos testes também foi realizada manualmente, apenas usando a ferramenta Diff [1] quando necessário para comparar o modelo de teste esperado com o modelo de saída obtido da transformação. Neste caso, além do QuickDiff, pode ser usada qualquer ferramenta que permita a comparação de dois arquivos no formato XML.

5.3 Resultados dos testes

O principal objetivo deste trabalho é gerar automaticamente casos de teste executáveis a partir dos casos de teste abstratos. Para atingi-lo, usamos Teste Dirigido por Modelo (MDT). O artefato de entrada é um metamodelo que especifica como é o modelo de teste independente de plataforma (PIT); o PIT é transformado em outro modelo, também especificado por um metamodelo, mas agora específico de plataforma (PST); finalmente, é realizada a última transformação para obter o código de teste a partir do PST.

Como dito anteriormente, são utilizados dois metamodelos: SMC, como PIT, e xUnit, como PST, que respectivamente representam uma sequência de chamada de métodos e um teste de unidade independente de linguagem de programação. A transformação PIT-PST foi implementada na linguagem ATL (adaptada da implementação de Javed *et al.* em QVT), enquanto que a transformação PST-código de teste foi escrita em Mofscript.

Concretizar uma transformação de modelo, partindo de um modelo de entrada e produzindo outros modelos como saída, requer um grande entendimento de aspectos sintáticos e semânticos de ambos os modelos, de modo a ser uma atividade susceptível a erros.

Logo, para se mais ter confiança que as transformações de modelo estão corretas, elas devem ser testadas. Para esse propósito, aplicamos o processo descrito na Seção 5.2 e aqui apresentamos os resultados.

5.3.1 Resultados do construtor de casos de teste

Usando os metamodelos originais disponibilizados por Javed [36] (SMC e xUnit), os fragmentos de modelo foram obtidos com a ferramenta MMCC (*Metamodel Coverage Checker*). A Tabela 5.1 sumariza os resultados da primeira execução da ferramenta. Ela mostra quantos fragmentos de modelo foram gerados, para cada metamodelo, para cada critério de teste disponível, que são "Todos–Intervalos" e "Todas–Partições". Esses fragmentos representam os requisitos de teste para os quais tivemos que preparar casos de teste para cobri-los. Os casos de teste são modelos de teste, instâncias de seus respectivos metamodelos.

Tabela 5.1: Fragmentos de modelo por metamodelo de acordo com um critério de teste.

Critério de Teste	SMC	xUnit
Todos–Intervalos	43	73
Todas–Partições	23	33

Dado tal resultado obtido, o próximo passo seria a obtenção da medida de cobertura dos modelos de teste. Entretanto, ao submeter modelos de teste que estavam de acordo com os metamodelos originais, a ferramenta MMCC não executava. Após algumas análises dos exemplos fornecidos com a ferramenta, verificou-se que o problema estava na declaração de elementos do tipo "String". Dessa forma, houve a substituição para declaração para a forma aceita pela ferramenta.

Resolvido tal problema, a ferramenta executou corretamente. Enquanto isso, no processo de melhorar a cobertura dos modelos de teste, descobrimos alguns defeitos no metamodelo SMC original. Basicamente, havia três defeitos relacionados a: *i*) multiplicidade de elemento; *ii*) elemento oposto; e *iii*) valor padrão do elemento. Abaixo são apresentados mais detalhes (vide detalhes do metamodelo SMC em 4.4.1):

1. Tanto na relação do elemento "Message" com elemento "OwnerClass" como com "ExpectedValue", a multiplicidade não estava correta;

2. No elemento "Message", atributo "parameter", não havia indicação de correspondência com o elemento "Parameter". Neste caso, o correto seria correspondê-lo ao

atributo “owner” (no elemento “Parameter”);

3. Foi retirado do elemento “Parameter” o valor *default* “no”, pois este tipo de valor se encaixa somente para atributos.

Levando-se em conta a correção dos defeitos, executamos novamente a ferramenta MMCC. Os novos resultados, em números de fragmentos de modelo, são ilustrados na Tabela 5.2.

Tabela 5.2: Fragmentos de modelo por metamodelo de acordo com um critério de teste.

		SMC	xUnit
1	Critério de Teste		
	Todos–Intervalos	54	78
	Todas–Partições	27	36
2	No. de modelos	6	7

Para cobrir tais requisitos de teste (fragmentos de modelo), foram elaborados seis modelos de teste para o metamodelo SMC e sete para o metamodelo xUnit. A Tabela 5.3 mostra qual a cobertura de fragmentos de modelo atingida ao usarmos tais modelos de teste.

Tabela 5.3: Cobertura de fragmentos de modelo por metamodelo de acordo com um critério de teste.

Critério de Teste	SMC	xUnit
Todos–Intervalos	54/54	76/78
Todas–Partições	25/27	32/36

Não foi possível alcançar 100% de cobertura dos fragmentos de teste para todos os critérios basicamente devido a uma única razão: o padrão XML (formato dos modelos de teste) não permite mais de um elemento raiz. Dessa forma, mesmo que seja possível colocar mais de uma suite de teste, por exemplo, de acordo com o metamodelo, a linguagem XML limita este tipo de teste, tornando-o infactível.

5.3.2 Resultados do motor de teste

Usando o Eclipse IDE como executor do motor de teste, basicamente foram executadas as transformações de modelos para obtenção do modelo alvo. A atividade mais importante do processo de teste das transformações de modelo é a comparação do resultado esperado com o obtido; atividade esta, feita pelo comparador do motor de teste.

O processo de comparação foi realizado manualmente, verificando-se se os elementos de saída estavam de acordo com o metamodelo de saída e com a transformação de modelo aplicada conforme o metamodelo de entrada. O resultado final do motor de teste foi: *i*) um defeito na transformação SMC-xUnit relacionado a elementos que, embora sintaticamente corretos, eram semanticamente inválidos; *ii*) seis defeitos na trasformação xUnit-código de teste relacionados a parâmetros vazios ou nulos; elementos semanticamente incorretos; e implementação incorreta da recursão de um elemento (ele ser composto por ele mesmo). Mais detalhes são apresentados abaixo:

- Defeito da transformação SMC-xUnit:

    1. Quando mais de um "ownerClass" ou "expectedValue" eram colocados em um elemento "message", eles não eram processados. Apesar de semanticamente isso ser incorreto, sintaticamente é permitido.

- Defeitos da transformação xUnit-código de teste:

    1. Elemento "ConstrutorParameter" sendo aceito com os atributos "name" ou "type" vazios;
    2. Permissão de mais de um atributo "method" por "assertion". De acordo com as regras de negócio descritas por Javed, isto não é permitido;
    3. Elemento "Method" sendo aceito com o atributo "name" vazio;
    4. Elemento "Method" com o atributo "ownerClass" vazio gerava o código "= new" sem complementação;
    5. Não é necessário tratamento para "expectedValue" no caso de *setup*;
    6. Recursão de "complexAttribute" não colocava o valor do atributo pai corretamente após dois níveis na hierarquia.

Todos os defeitos foram corrigidos antes da aplicação das transformações nos estudos de caso.

5.3.3 Resultados do analisador de teste

A Tabela 5.4 sumariza os resultados das transformações de teste em número de defeitos. Como Lin *et al.* [45] previram uma vizualização comparativa, as Figuras 5.3 e 5.4 apresentam um trecho do metamodelo SMC antes e depois do processo de teste. O defeito ilustrado está relacionado a um valor padrão incorreto que foi excluído, uma vez que ele estava fora de contexto.

Tabela 5.4: Número de defeitos por transformação de modelo.

Transformação	Número de defeitos
SMC-xUnit	1
xUnit-test code	6

```
<eClassifiers xsi:type="ecore:EClass"
 name="Parameter" eSuperTypes="//DataValue">

  <eStructuralFeatures xsi:type="ecore:EReference"
  name="owner" lowerBound="1" eType="//Message"
  defaultValueLiteral="no"
  eOpposite="//Message/parameter"/>

</eClassifiers>
```

Figura 5.3: Metamodelo SMC antes das correções.

Além dos defeitos das transformações de modelos, o processo revelou também três defeitos no metamodelo SMC, como descrito na Seção 5.3.1. Logo, no total, foram encontrados dez defeitos; um número significativo, que poderia causar grandes impactos para o objetivo deste trabalho, que é a geração automática de casos de teste executáveis.

5.4 Discussão

Nesta seção discutimos algumas lições aprendidas sobre teste de transformação de modelos.

5.4.1 Corretude da transformação de modelo x tamanho do modelo de teste

Certamente o fato de ter realizado o processo de teste com modelos de teste criados com base em critérios ajudou a obter um melhor resultado no processo de teste das transformações de modelo. Baseado nos defeitos encontrados, é possível concluir que eles não seriam alcançáveis se fossem usados casos de teste pequenos, i.e. modelos de teste que tivessem, no máximo, uma declaração de cada elemento.

5.4.2 Limitações do teste de transformações de modelo

"Uma das mais importantes limitações do teste de software é que ele pode apenas mostrar a presença de defeitos, não sua ausência. Esta é uma fundamental limitação teórica; genericamente falando, o problema de encontrar todos os defeitos é indecidível" [8].

```
<eClassifiers xsi:type="ecore:EClass"
 name="Parameter" eSuperTypes="//DataValue">

  <eStructuralFeatures xsi:type="ecore:EReference"
   name="owner" lowerBound="1" eType="//Message"
   eOpposite="//Message/parameter"/>

</eClassifiers>
```

Figura 5.4: Metamodelo SMC depois das correções.

Como uma transformação de modelo não é nada além de um programa, então não se pode dizer que, depois do processo de teste, tal transformação não tem mais defeitos. Entretanto, tem-se mais confiança que ela faz o que devem fazer e nada além disso.

Além disso, como colocam Fleurey *et al.* [29], outra limitação do teste de transformações de modelo é a dificuldade de geração de modelos que atendam a todas as restrições que se aplicam ao metamodelo em teste, ou seja modelos que estão de acordo com as regras do metamodelo e com as pré-condições da transformação. Existem algumas frentes de trabalho com relação a isso, como, por exemplo, os que usam algoritmos evolutivos, definindo operadores de mutação para construir incrementalmente novos modelos.

Capítulo 6

Estudos de Caso

Este capítulo apresenta como foi conduzida e quais foram os resultados da aplicação de nossa metodologia para geração de casos de teste executáveis a partir de abstratos em três aplicações Web. Especialmente para uma delas, chamada TestLink (muito conhecida na indústria de teste por fazer controle da documentação de requisitos e de casos de teste), será detalhado o processo completo, começando pela modelagem, seguindo com a geração dos casos de teste abstratos usando Teste Baseado em Modelo (MBT), até obter os casos de teste executáveis com nossa metodologia e executá-los. Para as demais serão ilustrados apenas os resultados obtidos, uma vez que, por motivos de confidencialidade, muitos de seus detalhes não podem ser divulgados.

Após apresentar como foi aplicada a metodologia, serão discutidos os resultados obtidos e suas consequências, como, principalmente, a factibilidade de aplicação de nossa solução para casos de teste executáveis de aplicações Web na indústria.

6.1 Descrição do TestLink

O TestLink é um sistema de gerenciamento de testes via Web, *open source*, que permite o cadastro de casos de teste que são associados a requisitos (também gerenciados pela ferramenta), permitindo que se tenha a rastreabilidade casos de teste *versus* requisitos, além da medida de cobertura dos requisitos. A Figura 6.1 mostra como esses elementos se relacionam. Uma vez especificados os casos de teste, a ferramenta permite: *i*) que sejam montados planos de teste relacionando as *releases* de um software com os casos de teste que devem ser executados para testá-las; e *ii*) que sejam armazenados os resultados das execuções.

Aqui descreveremos como foi realizada a atividades de teste do gerenciamento de requisitos provido pelo TestLink. O TestLink permite que os requisitos sejam cadastrados, editados, excluídos e versionados. A Figura 6.2 ilustra a tela de gerenciamento de requi-

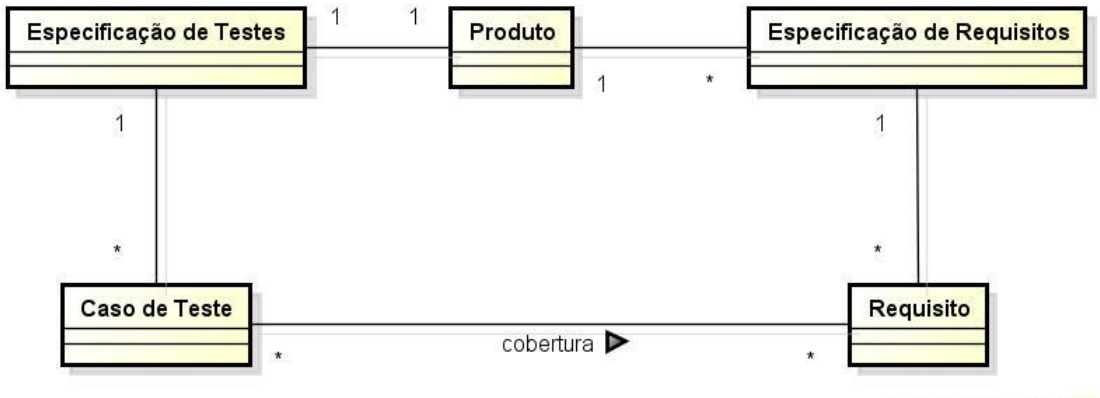


Figura 6.1: Relacionamento entre casos de teste e requisitos na ferramenta TestLink.

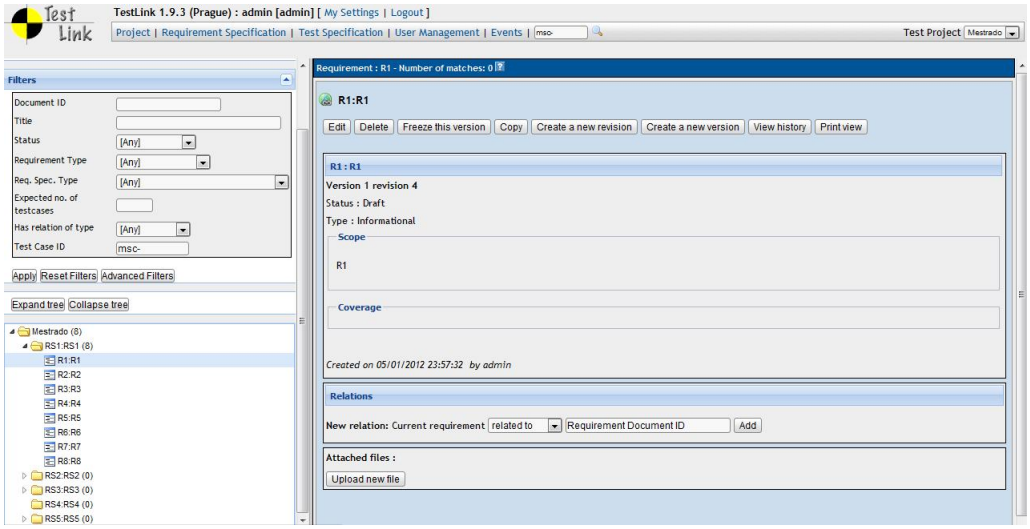


Figura 6.2: Tela de gerenciamento de requisitos da ferramenta TestLink.

sitos do TestLink.

6.2 Processo Completo de Teste das Aplicações Web

Teste de software, mesmo automatizado, não é uma atividade simples. Ele requer um processo, ou uma série de processos, para garantir que o sistema faz o que ele foi designado a fazer e nada além disso. Mesmo se automatizados, os casos de teste devem ser de qualidade e apresentar boa cobertura.

Nesta seção é ilustrado, de acordo com a Figura 6.3, o processo de teste utilizado nas três aplicações Web usadas como estudo de caso, com enfoque no processo de geração

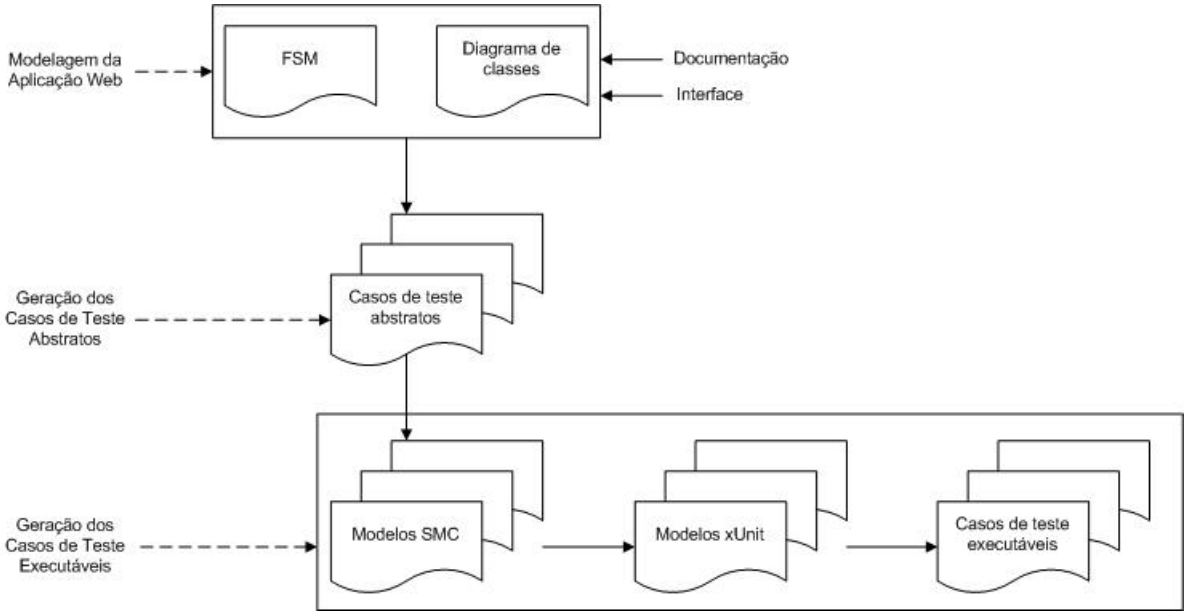


Figura 6.3: Processo aplicado no estudo de caso para geração dos casos de teste executáveis das apliçações Web.

automática dos casos de teste executáveis baseado nos casos de teste abstratos.

Resumidamente, baseado na documentação da aplicação Web e em sua interface com o usuário, são inicialmente modelados uma máquina de estados e um diagrama de classes. A primeira representa o comportamento de navegação da aplicação Web, enquanto que o segundo contém as principais operações de cada página da aplicação Web.

Em seguida, aplicando Teste Baseado em Modelo (MBT) e ferramentas desenvolvidas para automatizar o uso desta técnica, são gerados os casos de teste abstratos a partir do modelo de máquina de estados.

Então, tais casos de teste abstratos são transformados em executáveis com nossa metodologia que faz uso de Teste Dirigido por Modelo (MDT). Esta transformação significa ultrapassar a ponte entre o nível de abstração do modelo de navegação e detalhes de implementação. Primeiro, relaciona-se as interações de entrada do caminho de transições com chamadas de método, de acordo com o diagrama de classes. Em seguida, obtemos uma seqüência de chamadas de métodos, que é uma instância do metamodelo SMC, nosso modelo de teste independente de plataforma (PIT). Depois disso, os modelos SMC são mapeadas para modelos xUnit, modelo de teste dependente de plataforma (PST), por regras de transformação ATL. Finalmente, os modelos xUnit são traduzidos em comandos da linguagem de programação Java. Tais comandos devem estar de acordo com a biblioteca Selenium, uma vez que esta irá permitir a execução automática dos casos de teste.

Nas próximas seções serão detalhadas cada uma das etapas deste processo, além de serem apresentados os artefatos obtidos na aplicação Web TestLink.

6.3 Modelagem das Aplicações Web

A modelagem da navegação de uma aplicação web deve especificar, do ponto de vista do usuário, a interação com o sistema através da interface web. A interação da aplicação em teste com o ambiente é considerada como uma caixa preta e os detalhes de implementação são abstraídos. No processo que propomos, temos um modelo de máquina de estado finita estendida (MEFE), para representar o comportamento dinâmico da navegação da aplicação web, e um diagrama de classes, para representar o aspecto estático da interface.

MEFEs têm sido usadas para representar uma grande variedade de sistemas há bastante tempo [18]. Este formalismo permite a representação de controle, bem como aspectos de dados de um comportamento do sistema. A MEFE representa as entradas possíveis, resultados esperados e restrições de dados das interações do sistema. Uma MEFE pode ser definida como uma tupla (S, s_0, I, O, T, V, P) em que [11]: S é um conjunto finito de estados; $s_0 \in S$ é o estado inicial; I é um conjunto de eventos de entrada; O é um conjunto de eventos de saída; T é um conjunto finito de transições; V é um conjunto de variáveis; e P é um conjunto de parâmetros de entrada. Cada transição $t \in T$ é dada por um estado de origem $source(t) \in S$, um estado de destino $target(t) \in S$ e um rótulo na forma $i(t)[g(t)]/a(t)$ onde: $i(t) \in I$, $g(t)$ é uma expressão lógica chamada guarda e $a(t)$ é uma ação executada quando a transição é ativada. Eventos de entrada podem conter um ou mais parâmetros pertencentes P. As partes g e a do rótulo são opcionais. O modelo pode estar em somente um estado a cada momento. Para mudar o estado é necessário acionar uma transição. Para uma transição ser acionada, seu correspondente evento de entrada deve ser recebido pelo estado atual e a guarda associada deve ser satisfeita. A ação correspondente a é executada; tal ação pode conter atribuições de variáveis ou pode produzir eventos de saída.

Como exemplo, a Figura 6.6 ilustra a MEFE da aplicação Web TestLink (não estão representadas todas as transições que têm como evento "logout()", cujo próximo estado é o "Login"); as funcionalidades da aplicação, apresentadas no diagrama de caso de uso da Figura 6.4 são: *i*) login e *ii*) as operações básicas em relação aos requisitos: adicionar/editar "especificações de requisitos" (que nada mais são do que grupos de requisitos organizados por terem características comuns) e adicionar/editar requisitos.

A modelagem leva em conta não apenas a documentação, mas também a interface do sistema em teste. Nós consideramos que os estados representam as páginas do aplicativo e os eventos de entrada das transições indicam operações que podem ser realizadas em uma página web, como clicar em um botão e preencher um campo. Uma transição é disparada

quando o evento de entrada correspondente é recebido e sua guarda é satisfeita. Então, o modelo produz uma saída que corresponde a uma resposta da aplicação, tais como: um texto específico, um HTML, uma URL ou uma combinação desses elementos [53]. Portanto, um caso de teste corresponde a um caminho de navegação [42], ou seja, uma seqüência de ativações de hyperlinks através de uma aplicação web.

A modelagem proposta neste trabalho pode sofrer de um problema comum quando se fala de modelagem de aplicações Web por máquinas de estado: explosão de estados. Para solucionar este problema, existem algumas propostas na literatura. Lucca *et al.* [47], por exemplo, propõem que seja aplicada a técnica de partição de equivalência quando se tem um grande número de páginas dinamicamente geradas (que dependem das entradas dos usuários). Já Andrews *et al.* [10] apresentam uma metodologia de modelagem para aplicações Web usando hiearquias de máquinas de estados finitas, que visa solucionar exatamente este problema. Esta metodologia pode ser aplicada em conjunto com a proposta deste trabalho para obter casos de teste executáveis a partir de abstratos, desde que se tenha uma ferramenta para geração dos casos de teste abstratos correspondentes.

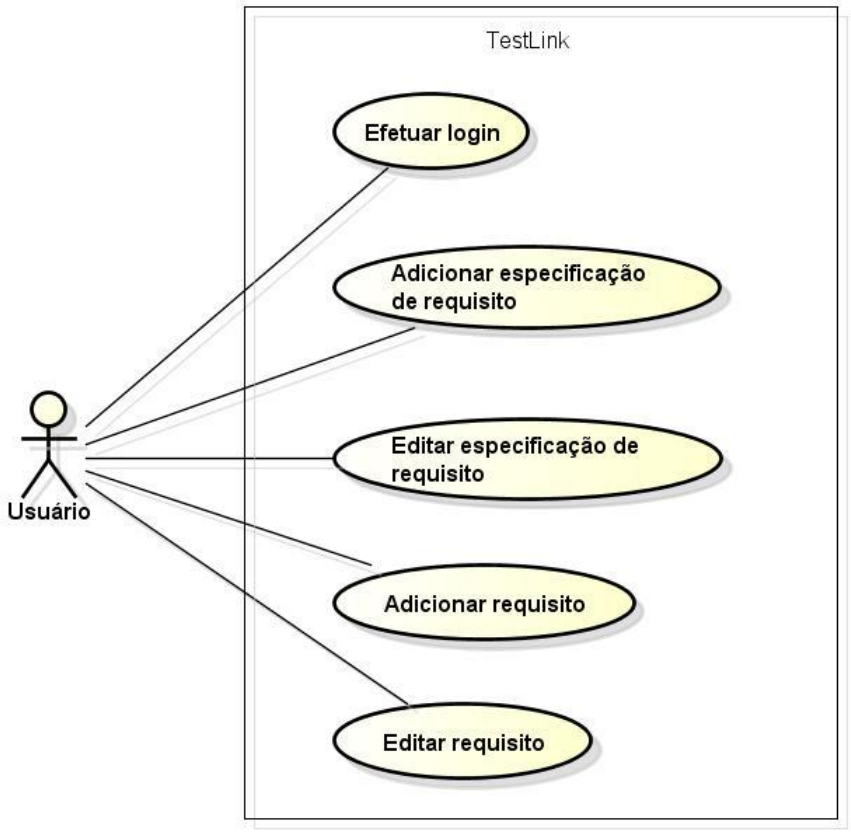


Figura 6.4: TestLink: diagrama de casos de uso.

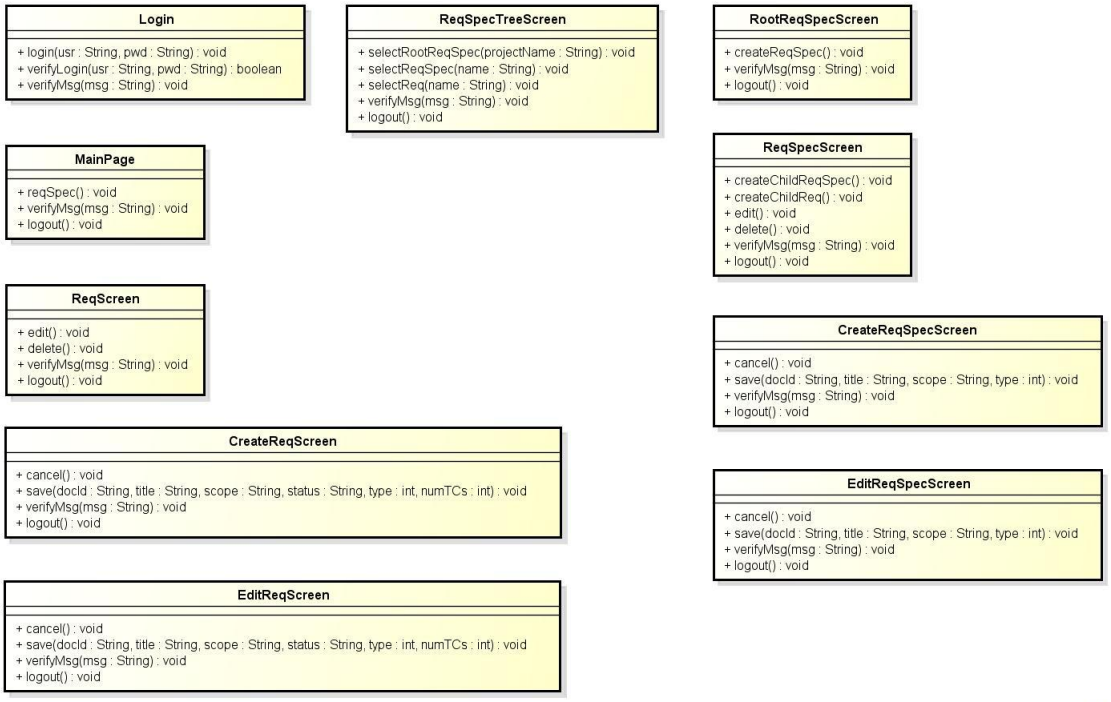


Figura 6.5: TestLink: diagrama de classes com as operações disponíveis em cada tela.

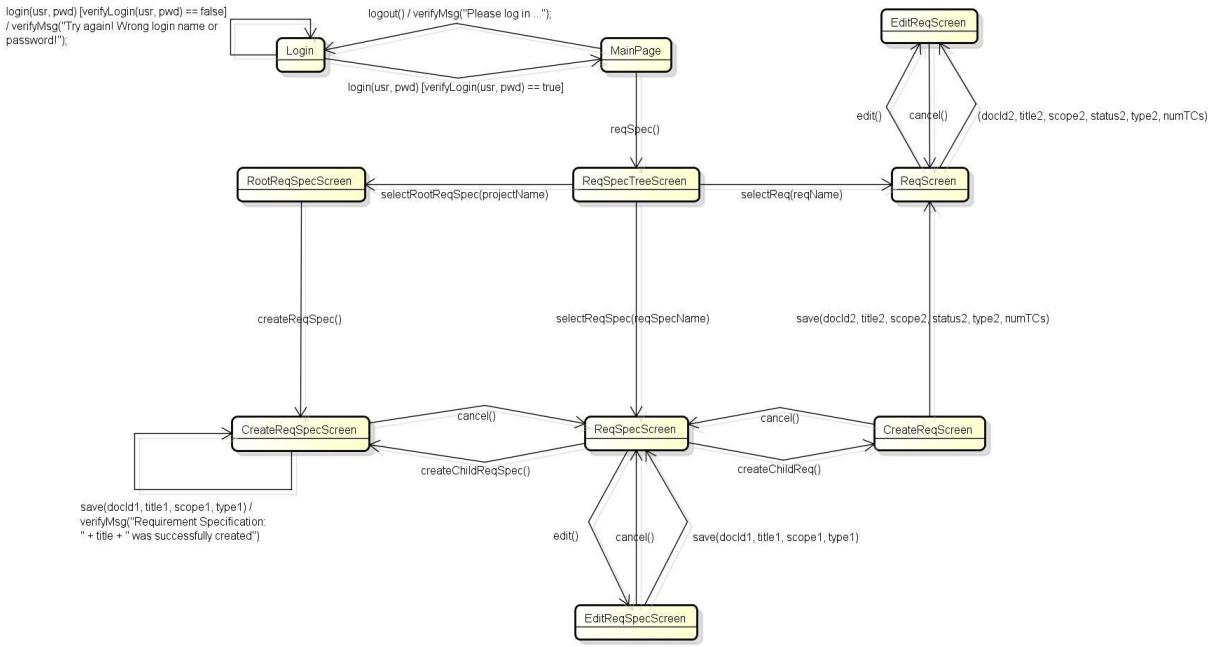


Figura 6.6: TestLink: modelo EFSM.

A Tabela 6.1 sumariza os diagramas de estados de cada uma das aplicações Web usadas como estudo de caso.

Tabela 6.1: Informações dos modelos de cada aplicação Web.

Aplicação Web	#Estados	#Transições	#Casos de Teste
TestLink	10	28	28
W_1	9	40	40
W_2	26	235	235

Já o diagrama de classes é usado para identificar as principais características da aplicação web em teste. As classes correspondem aos estados do MEFE, que são as páginas que o usuário pode visitar. Os métodos correspondem às transições, indicando uma mudança de uma página para outra. A Figura 6.5 mostra o diagrama de classe do TestLink.

6.4 Geração dos Casos de Teste Abstratos

Para geração dos casos de teste abstratos foram usadas duas abordagens de teste baseado em modelo (a ferramenta ModelJUnit e a técnica MOST), ambas aceitando MEFEs (descritas em formatos diferentes) como entrada. Está fora do escopo deste trabalho comparar a qualidade dos casos de teste. Entretanto, é importante citar que as duas têm formatos de entrada e saída diferentes; uma delas (MOST), inclusive, já diz quais os dados de entrada que devem ser usados em cada caso de teste, pois ela usa tais dados para decidir a factibilidade de um determinado caminho.

Nesta seção são descritas as duas abordagens, sendo que a ferramenta ModelJUnit foi aplicada à aplicação Web TestLink e a técnica MOST às demais. Para o primeiro caso (ModelJUnit aplicada ao modelo MEFE da ferramenta TestLink), ilustraremos os artefatos de entrada e saída.

Sumarizando os resultados obtidos, a Tabela 6.2 apresenta a quantidade de casos de teste gerados e de passos de cada um deles. Um detalhe importante é que foi selecionado apenas um caso de teste por transição, com o objetivo de aplicar a cobertura de transições.

Tabela 6.2: Geração de casos de teste abstratos: número médio de passos.

Web App.	Média de Passos	σ_p
TestLink	4.4	1.73
W_1	6.45	3.14
W_2	9.78	6.47

6.4.1 ModelJUnit

ModelJUnit é uma biblioteca Java que estende o *framework* JUnit para dar suporte a teste baseado em modelos. Ela permite a escrita de modelos MEFEs como classes Java

e a geração de testes a partir desses modelos, além de apresentar métricas de cobertura relacionadas com este tipo de modelo, como cobertura de estados e transições.

Os testes podem ser gerados a partir de um dos seguintes algoritmos:

- *Random Walk*: gera caminhos aleatórios no modelo;
- *Greedy Walk*: gera caminhos aleatórios no modelo, dando preferência para transições que não foram escolhidas;
- *Lookahead Walk*: gera caminhos avaliando N-níveis do modelo de quais ações ou transições são mais vantajosas.

Esta biblioteca foi usada para gerar os casos de teste abstratos (inclusive sem os dados de teste) da ferramenta TestLink. Para a geração de sequência de teste, escolhemos o algortimo *Greedy Walk*, pois o objetivo era cobrir todas as transições e este dá preferência para a escolha de transições ainda não escolhidas.

6.4.2 MOST

Neste caso, os casos de teste são gerados automaticamente a partir de uma Máquina de Estado Finita Estendida (MEFE) usando meta-heurísticas. A meta-heurística gera uma sequência de entrada, que consiste nas interações de entrada (por exemplo, clicar em um botão de uma aplicação web) e nos dados envolvidos nas interações (por exemplo, um número de identificação para uma consulta). Um modelo executável da MEFE recebe a sequência de entrada gerada e produz o caminho de transições desencadeada por esta sequência. O modelo executável implementa o comportamento da MEFE em uma linguagem de programação. A meta-heurística avalia o caminho de transições para determinar se o critério de teste está satisfeito. Com base nessas informações, os dados de teste são incrementalmente modificados de forma a obter soluções mais próximas para satisfazer o critério de teste, que neste caso é uma transição específica.

Uma vez que os caminhos de transições são dinamicamente obtidos durante a execução do modelo, em vez de executar uma análise estática de alcançabilidade, evita-se o problema de geração de caminhos não-executáveis, que é um dos aspectos considerados em teste com MEFE devido aos conflitos de dados ao longo de um caminho. Outra vantagem da utilizar o modelo executável é a possibilidade de validar o modelo antes da geração de casos de teste. Além disso, como os casos de teste são derivados de MEFE, é importante validar se o modelo está em conformidade com a especificação do Sistema em Teste (SeT).

Mais ainda, usando a MOST, pode-se obter casos de teste com comprimentos variáveis, resultando em comprimentos de caminhos diferentes para cobrir uma transição alvo. Nos dois estudos de caso em que aplicamos a MOST foram selecionados os casos de teste de menor comprimento para cada transição.

Figura 6.7: Processo de mapeamento de um modelo SMC para um modelo xUnit.

6.5 Conversão dos casos de teste abstratos em modelos SMC

Um caso de teste abstrato é expresso como uma seqüência de eventos de entrada de um caminho na MEFE. Estes eventos de entrada correspondem às chamadas para os métodos da classe que representam as páginas web. Considerando-se o TestLink, um caso de teste gerado pela ferramenta ModelJUnit foi: *Path = {t1 : login(user, password)}*. Ele é análogo a uma sequência de chamada de métodos.

Para mapear um caso de teste abstrato em um modelo que é instância do metamodelo SMC, estabelecemos que deve ser feita uma correspondência de um-para-um. O metamodelo SMC permite um ou mais casos de teste por modelo, mas, para uma melhor organização, colocamos apenas um caso de teste por modelo. O mapeamento foi implementado na forma de software simples que lê um arquivo contendo casos de teste e os mapeia para modelos SMC que não são nada mais que arquivos XMI (formato XML).

6.6 Transformação do modelo SMC para o modelo xUnit

A transformação entre uma instância do metamodelo SMC e uma instância do metamodelo xUnit é executada com o *framework* Eclipse IDE, conforme ilustra a Figura 6.7.

Para esta transformação foi aplicado o código desenvolvido em ATL (*Atlas Transformation Language*) para mapear o modelo de teste independente de plataforma (PIT) no modelo de teste dependente de plataforma (PST).

A Listagem 6.2 apresenta um caso de teste no formato xUnit, obtido com a aplicação da transformação de modelo no caso de teste abstrato (no formato SMC) da Listagem 6.1.

```
<?xml version="1.0" encoding="ASCII"?>
<Model>
 <interaction name="MODEL_16">
  <message name="SETUP_1" order="1">
    <OwnerClass name="Login">
      <constructorParameter name="selenium" type="LoggingDefaultSelenium
          "/>
    </OwnerClass>
  </message>
  <message name="login" order="2">
    <OwnerClass name="Login"/>
      <parameter name="user" setter="" type="String"/>
      <parameter name="senha" setter="" type="String"/>
  </message>
 </interaction>
</Model>
```

Listagem 6.1: Caso de teste no formato SMC

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<TestSuite>
 <testCase name="MODEL_16">
  <assertion order="1" assertionType="assertTrue" name="MODEL_16_SETUP_1
      ">
    <method name="SETUP_1">
      <ownerClass name="Login">
      <constructorParameter type="LoggingDefaultSelenium" name="selenium
          "/>
    </ownerClass>
    </method>
  </assertion>
  <assertion order="2" assertionType="assertTrue" name="MODEL_16_login">
  <method name="login">
    <parameter type="String" setter="" name="user"/>
    <parameter type="String" setter="" name="senha"/>
    <ownerClass name="Login"/>
  </method>
  </assertion>
 </testCase>
</TestSuite>
```

Listagem 6.2: Caso de teste no formato xUnit

6.7 Transformação do modelo xUnit em caso de teste executável

O formato xUnit deve agora ser transformado em código-fonte, ou seja, uma transformação PST-código de teste é ainda necessária. Por esta razão, é preciso aplicar uma transformação de modelo-para-texto, feita com o script Mofscript descrito na metodologia. MOFScript descreve como o modelo deve ser mapeado para o seu código de teste correspondente, com as devidas adaptações para a biblioteca do Selenium. Além disso, é nesta etapa que são adicionados os dados de teste, quando necessário.

A Listagem 6.3 ilustra o caso de teste executável correspondente ao modelo xUnit apresentado na Listagem 6.2.

```
1  package testcases;

3  import reference.Login;
   import basic.MyTestCase;
5
   public class Transition16 extends MyTestCase {
7    public void testMODEL_16( ) {
       try {
9        //Prepare constructor parameter object
         LoggingDefaultSelenium selenium = this.selenium;
11       Login login = new Login(selenium);

13       //Prepare parameter object
         String user = "admin";
15       //Prepare parameter object
         String senha = "admin";
17       login.login(user, senha);
       } catch ( Exception exp ) {
19       System.out.println( exp.toString() );
         fail("Exception occured during test case execution");
21     }

23     if (isFail()) {
         fail();
25     }
     } //End of Test Method
27 } //End of TestCase
```

Listagem 6.3: Caso de teste executável

Antes da execução dos casos de teste da aplicação Web, é preciso prestar atenção a um detalhe: o caso de teste executável irá realizar chamadas de método, de acordo com o diagrama de classes. No entanto, estes métodos não correspondem à implementação do

sistema, uma vez que estamos testando uma interface web, e não uma interface do tipo API. Então é necessário adicionar detalhes específicos de plataforma, a fim de mapear a aplicação de tais métodos aos comandos de Selenium que correspondem a ações que os usuários poderiam executar na página da aplicação Web.

Este processo em que a implementação dos métodos chamados nos casos de teste executáveis é separada do teste em si foi adaptada da técnica *behaviour-driven development* (BDD) [9], uma técnica ágil para desenvolvimento de software. Nesta técnica, os analistas de negócio escrevem os requisitos em formato de texto (usando ações padronizadas que seguem o seguinte modelo: *ator* + *ação* + *parâmetros* + *resultado esperado*, como se fosse um teste); os desenvolvedores implementam tais ações padronizadas; e, finalizada a implementação, o analista de negócio pode executar seu requisito (usando um *framework* de apoio) na aplicação que foi feita. Assim, os requisitos e os respectivos testes ficam prontos antes da implementação do código fonte do software. Dessa forma, durante o desenvolvimento e após sua finalização, os testes podem ser facilmente executados. BDD é uma adaptação de *test-driven development* (TDD), sendo que neste último os testes são diretamente escritos na linguagem de programação utilizada. Os benefícios destas técnicas são [51]: *i*) facilidade no entendimento dos requisitos e diminuição dos bugs, pois escrever o teste antes do código fonte ajuda o programador saber o que realmente deve ser feito; *ii*) desenvolvimento de um código fonte testável; e *iii*) facilidade para aplicar testes de regressão, pois os testes são automatizados.

Para casos de teste executáveis, o processo é semelhante: o analista de teste monta os modelos de teste (MEFE e diagrama de classe); ele mesmo, ou com a ajuda de um desenvolvedor, implementa os métodos do diagrama de classe; e, com os casos de teste executáveis gerados, estes podem ser executados acionando os métodos implementados que correspondem às ações de um usuário.

6.8 Discussões

6.8.1 Suceptibilidade a Erros e Redução de Tempo

É possível observar que com a aplicação da nossa metodologia, além da redução no tempo para transformar o caso de teste abstrato em executável (trabalho que seria feito por alguém com habilidades de programação), também temos menos susceptibilidade a erros nesse processo, uma vez que ele é feito automaticamente por transformações de modelo que foram submetidas ao processo de teste.

6.8.2 Reusabilidade da Metodologia

A metodologia pode ser reutilizada em qualquer aplicação Web, inclusive aquelas que usam a tecnologia Ajax[1], pois a ferramenta Selenium já possui alternativas para lidar com elementos Ajax. Além disso, o problema de explosão de estados pode ser tratado por uma das propostas existentes na literatura e que citamos na Seção 6.3.

Vale observar que, para cada aplicação Web, deverão ser implementados os métodos do diagrama de classes, pois eles são unicos para cada Sistema em Teste (SeT). Lembrando que os métodos do diagrama de classes representam as funcionalidades de uma tela, enquanto a implementação desses métodos reproduz os passos do usuário para executar a funcionalidade; logo, esta parte da metodologia deve ser feita individualmente por SeT.

6.8.3 Mudança na Interface Gráfica de Aplicações Web

Outro ponto de discussão quando se trata de aplicações web e automação de teste é a estabilidade da interface gráfica. Ferramentas de captura e reprodução, em geral, usam coordenadas da tela para localizar os itens a serem clicados, ou elas criam e usam referências a objetos janela. No primeiro caso, quando as localizações do objeto em uma página mudam de uma versão para o outra, os casos de teste não são mais aplicáveis. No segundo caso, é necessário contar com um formato proprietário de mapeamento de objetos, onde todos os objetos da página web devem ser capturados e salvos antes da escrita dos scripts de teste.

Uma vez que nossa metodologia utiliza MDT, os casos de teste são independentes do layout da aplicação web, porque eles fazem uso das funcionalidades de uma tela e não de seus itens diretamente. Ou seja, se o layout da aplicação Web for alterado, isso não implicará em alterações nos casos de teste em si. Apenas o último passo, que mapeia as chamadas de métodos para elementos da interface real, deve mudar de acordo.

A biblioteca do Selenium aceita alguns tipos de localizadores de elementos da tela [2]:

- *ID:* Seleciona o elemento cujo atributo “id” for igual ao especificado.
- *DOM:* É especificado através de uma expressão em Javascript. O Selenium procura o elemento avaliando o valor especificado. Isso permite atravessar o Modelo de Objetos do Documento HTML usando Javascript.
- *XPath:* Localiza um elemento com expressões XPath. XPath é uma linguagem para criar consultas a elementos em um documento XML.

[1] Ajax (*Asynchronous Javascript and XML*) é o uso metodológico de tecnologias como Javascript e XML, por navegadores, para tornar páginas Web mais interativas, utilizando-se de solicitações assíncronas de informações.

Se for usada a localização por posição (usualmente com XPath) do Selenium, modificações nos métodos deverão ser feitas. Em caso de mapeamento com ID (que é bastante complicado para aplicações web que são desenvolvidos com *frameworks* atuais que geram IDs dinâmicos), pode não ser necessário qualquer retrabalho.

Capítulo 7

Trabalhos relacionados

Este capítulo apresenta os principais trabalhos relacionados com Engenharia Dirigida por Modelo (MDE), mais especificamente tratando do ramo voltado para teste de software, que é Teste Dirigido por Modelo (MDT), e com teste de aplicações web. Estes dois temas se destacam neste trabalho, pois nosso objetivo é gerar casos de teste executáveis para aplicações web usando MDT.

7.1 Engenharia Dirigida por Modelo (MDE)

Não são poucos os trabalhos que fazem menção à Engenharia Dirigida por Modelo (MDE) a partir do ano 2000. O uso de MDE começou nos anos 80 com as ferramentas CASE (*Computer-Aided Software Engineering*), mas a consolidação desse conceito se deve à aprovação da UML (*Unified Modeling Language*), em 1997, como padrão da OMG (*Object Management Group*), um consórcio internacional de empresas que define e ratifica padrões na área de Orientação a Objetos.

Esta seção mostra o uso de MDE em duas partes: na Seção 7.1.1, são apresentadas algumas propostas encontradas na literatura, enquanto que na Seção 7.1.2 são destacadas aplicações reais.

7.1.1 Abordagens envolvendo MDE

Javed *et al.* [36] propuseram um método, com a respectiva implementação de uma ferramenta, que utiliza uma das abordagens de transformação de modelos, chamada Arquitetura Dirigida por Modelo (MDA), para gerar testes de unidade a partir de modelos independentes de plataforma. No primeiro passo, tais modelos são obtidos dos diagramas de sequência do sistema e, em seguida, eles são transformados em modelos de teste específicos de plataforma, que, no caso, são instância do modelo *xUnit*, uma abstração para

os casos de teste executáveis usados em testes de unidade.

Mesmo gerando o modelo de teste específico de plataforma no formato *xUnit*, Javed *et al.* [36] ainda apresentam duas versões de casos de teste executáveis para JUnit (teste de unidade na linguagem de programação Java) e SUnit (teste de unidade na linguagem de programação Smalltalk). Esta abordagem se torna bastante aplicável e expansível por dois motivos: *i*) o fato de trabalhar com *xUnit* como modelo independente de plataforma facilita bastante caso se queira gerar o código fonte do teste em diferentes linguagens de programação; e *ii*) por ter fornecido a opção de código de teste em formato JUnit, propiciou um ponto de partida para adaptar esta abordagem para aplicações web, uma vez que os casos de teste do Selenium são um especialização de casos de teste JUnit.

Em uma continuação de seu trabalho, Javed expande o framework proposto para teste de componentes. Um componente que é desenvolvido e usado pela primeira vez deve ser testado por completo, principalmente porque ele será reusado em outras aplicações. A proposta de Javed é, como anteriormente, modelar os componentes em diagramas de casos de uso e diagramas de interação (i.e. diagramas de sequência) para derivar os casos de teste executáveis.

Além disso, ele compara os casos de teste obtidos com esta metodologia com os casos de teste que foram executados pelo desenvolvedor do componente (durante o processo de teste) e anexados ao mesmo como metadado. Por fim, ele executa o conjunto mais adequado para o componente.

Ainda usando a abordagem MDA na atividade de teste, Dai [23] introduz uma metodologia de como aplicar os conceitos do perfil de teste da UML (*Unified Modeling Language*), chamado U2TP (*UML Testing Profile*), para transformar os modelos dos sistemas de software, que estão no formato UML, em testes. Isto é feito de acordo com a abordagem MDA, especificando regras de transformação no formato QVT (*Query/Views/Transformations*). Para isso, a partir dos modelos existentes, são listados os modelos obrigatórios e opcionais para o uso de U2TP, e possíveis formas de derivá-los. Cabe observar, porém, que nem todas as regras de transformação foram implementadas e provadas. Além disso, diferente do nosso trabalho em que um dos requisitos especificados foi a independência dos modelos de teste dos modelos de desenvolvimento, o objetivo de Dai é gerar os testes a partir dos modelos de desenvolvimento.

Unindo as duas principais vertentes de MDE, Desenvolvimento Dirigido por Modelo (MDD) e Teste Dirigido por Modelo (MDT), Alves *et al.* [7] propõem um *framework* tendo em vista os seguintes benefícios: máxima automatização no desenvolvimento do software e dos testes; e maior controle do software para evolução.

Com essa finalidade, são definidos quais os tipos de modelos e as tecnologias a serem empregados tanto para modelagem do sistema, bem como para a geração, respectivamente, do código fonte e dos casos de teste. Para a geração do código fonte, os autores estabelecem

que os modelos independentes de plataforma devem ser representados em UML e OCL, enquanto que para o modelo dependente de plataforma opta-se por Java. Com relação às transformações entre os modelos sugere-se a utilização de ATL (*Atlas Transformation Language*).

Em relação ao processo MDT, os modelos independentes de plataforma devem ser obtidos, através de regras de transformação, dos modelos independentes de plataforma do MDD. Tais modelos no MDT devem ser condizentes com a especificação do perfil de teste da UML, o U2TP. Além disso, assim como no processo de desenvolvimento, o *framework* também utiliza a linguagem Java como plataforma, seguindo o *framework* de teste de unidade JUnit.

Neste trabalho, Alves *et al.* apresentam uma aplicação prática do *framework* (um estudo de caso que descreve o sistema de controle de empréstimos em uma biblioteca), mas, infelizmente, não mostram os resultados finais alcançados com a proposta e nem disponibilizam os artefatos desenvolvidos.

7.1.2 Aplicações reais de MDE

No ramo de MDE voltado para desenvolvimento de software, chamado Desenvolvimento Dirigido por Modelo (MDD), um exemplo de caso de sucesso é a aplicação na Motorola, uma empresa americana especializada em eletrônica e telecomunicações. Weigert *et al.* [66] dizem que a Motorola tem usado MDE por mais de 15 anos com o objetivo de desenvolvimento de sistemas confiáveis no ramo de telecomunicações em larga escala. Os principais benefícios com essa prática foram aumento na qualidade e melhora na produtividade.

Entretanto, o caminho para o sucesso do MDE na empresa não foi um processo simples, principalmente devido à falta de capacidade para modelagem dos engenheiros e a processos que não foram bem definidos. Outro detalhe importante dessa aplicação prática de MDE foi que ele nem sempre era aplicado em todos os projetos; no começo de cada projeto era feito um julgamento se seria possível utilizá-lo. A dificuldade era maior principalmente quando se tratava de projetos legados.

Em outro relato sobre a experiência com MDE na Motorola, Baker *et al.* [12] destacaram que, ao utilizaram MDE para especificação do sistema, design e teste, foi possível aumentar a aderência a padrões da área de telecomunicações, como o ITU-T (*International Telecommunications Union*) e ETSI (*European Telecommunication Standards Institute*).

Mais especificamente falando sobre a forma de testar os sistemas desenvolvidos, eles estenderam a linguagem MSC (*Message Sequence Charts*), usada para especificação de requisitos na forma de cenários, de modo a acrescentar dados e possibilitar a geração automática das especificações de teste em TTCN-2.

O resultado final após a implantação do MDE na Motorola Europe foi a redução em

2,3X no esforço de uso de simulação, geração automática de código e testes com modelos.

Em outro exemplo de aplicação real, feito na Faculdade de Engenharia da Universidade de Auckland [48], MDE é aplicado com cunho social. Em uma das pesquisas, foi desenvolvida uma aplicação (para ser executada em um robô) para cuidar de pessoas idosas com doenças. Esta aplicação, baseando-se no histórico ou em ações tomadas por pessoas idosas, verifica se ela deve fazer algo para manter sua saúde. Um exemplo prático é o controle de horário dos remédios e atividades de pessoas idosas.

No total, além do projeto com o controle da saúde de idosos, a Faculdade de Engenharia apresentou mais quatro projetos na linha de MDE relacionado com robótica. Dentre eles, o mais interessante para a área de computação foi a criação uma linguagem visual, chamada Ruru, para a programação de software com elementos visuais.

Já Ali *et al.* [4] introduzem uma nova abordagem para desenvolver ferramentas para Teste Baseado em Modelo (MBT) usando transformações de modelos. A motivação para isso é que, mesmo existindo um grande número de ferramentas para suportar MBT, elas não são flexíveis o suficiente para que sejam adaptadas para vários contextos de aplicação. Além disso, os autores valorizam a importância da configurabilidade para que se possa escolher entre diferentes configurações de teste, como, por exemplo, se configurar qual o critério de cobertura que deve ser aplicado.

Ali *et al.* então desenvolveram uma ferramenta MBT baseada em transformações de modelos e em modelos de teste UML (no caso, diagramas de estado). O processo básico de uso consiste na modelagem do Sistema em Teste (SeT), derivação de casos de teste abstratos de acordo com uma estratégia de teste, e, finalmente, obtenção os casos de teste executáveis a partir dos abstratos e de dados de entrada.

Tal ferramenta é aplicada em dois estudos de caso reais para demonstrar sua configurabilidade e extensibilidade. Além disso, eles discutem os desafios e como esta abordagem pode reduzir custos comparado ao teste manual, dado que os resultado obtidos foram bastante satisfatórios com relação a esse aspecto. Entretanto, com relação à MDT, devemos destacar um ponto negativo: o fato dos autores não terem aplicado sistematicamente testes nas transformações de modelo.

Além da redução de esforço, Hailpern e Tarr [32] ressaltam ainda que o desenvolvimento de software se torna mais disciplinado com MDE. Anteriormente, os programadores chegavam nos seus postos de trabalho, abriam seu editor de texto preferido e escreviam o software, importando-se principalmente com o fato que este precisaria compilar.

Com o início da cultura dos testes de unidade, em meados dos anos 90, com a introdução de *Extreme Programming* (XP), este cenário foi ligeiramente modificado, pois não bastava apenas o código ser compilado, ele ainda deveria ser submetido aos testes. Outro avanço da combinação entre XP e teste de unidade é o fato que o teste era considerado uma especificação.

Usando MDE, a principal forma de documentação são os modelos, de forma que não são necessários diferentes tipos de documento que podem se perder ao longo do tempo.

7.2 Teste de aplicações Web

Segundo Yang *et al.* [68], usualmente a arquitetura de teste de software pode ser dividida em cinco subsistemas: *i*) desenvolvimento de teste, *ii*) medição de teste, *iii*) execução de teste, *iv*) análise dos defeitos, e *v*) gerenciamento. No caso de aplicações Web, eles consideram que a arquitetura de teste é uma especialização da arquitetura para o teste de software tradicional. Em função disso, eles acrescentam um novo subsistema, chamado "Análise de Documentos-Fonte", devido aos problemas introduzidos pelas novas linguagens de programação (que podem ser a combinação de mais de uma e devem ser interpretadas em uma ou mais camadas, i.e. servidor, banco de dados, etc.).

Neste novo subsistema são analisados cada um dos documentos envolvidos nos componentes de uma aplicação web, como, por exemplo, o lado cliente ou o lado servidor. A partir de tais análises é possível estabelecer um fluxo de controle e, com base neles, gerar casos de teste (subsistema de desenvolvimento de teste). Então, tais testes podem ser executados manual ou automaticamente, gerando relatórios de defeitos baseados no oráculo de teste (que determina se o resultado do teste está correto ou errado). Além da análise de defeitos, o subsistema de gerenciamento prevê medidas de cobertura e adequação a um critério de teste.

Os autores desta proposta utilizaram-na em uma aplicação web de demonstração, mas não mostraram a escalabilidade para aplicações web reais, o que pode ser bastante difícil, principalmente devido ao tamanho das aplicações web estarem crescendo a cada dia.

Numa abordagem mais recente, Offutt and Wu [55] analisam aplicações web mais complexas, que trazem novos desafios para programadores e testadores, uma vez que possuem interfaces gráficas sofisticadas e vários componentes de software que rodam em *back-end*. Dessa forma, para os testadores, não basta apenas verificar links estáticos dado que grande parte do desenvolvimento de uma aplicação web não é visível aos seus consumidores e se encontra armazenada nos servidores. Além de ter se tornado mais difícil, o teste deste tipo de software se tornou mais importante, pois qualquer problema em uma aplicação web pode rapidamente afetar milhões de usuários e causar muitos prejuízos.

Mais ainda, os autores ressaltam que o controle de fluxo desse tipo de aplicação é muito variável e difícil de ser determinado estaticamente, requerindo que novas formas de modelagem sejam criadas para atender aplicações web.

Os sistemas web são compostos de componentes heterogêneos, que interagem através de troca de mensagens. Muitas são as abordagens para se garantir que a troca de mensagens esteja sintaticamente correta; entretanto, isto não é suficiente, pois podem existir

erros semânticos, que podem levar o programa a produzir saídas incorretas. Visando uma solução para este problema, Lee e Offutt [41] apresentam uma técnica, baseada em mutações[1], para verificar a corretude semântica das interações entre componentes web que usam XML.

Neste caso, eles definem um modelo semântico para a interação entre os componente e aplicam mutações no modelo semântico, usadas para criar instâncias das mutações em forma de mensagens XML. É interessante observar que, diferente da análise de mutação comum que introduz alterações no código fonte, neste caso as alterações são nos dados (mutação na semântica do XML) que são passados, uma vez que nem sempre se tem acesso ao código fonte dos componentes.

Dessa forma, compara-se as saídas esperadas quando se tem um caso de teste correto (i.e. uma mensagem XML correta) com um caso de teste mutante. Se elas forem diferentes, o mutante foi morto.

Esta abordagem deve ser aplicada quando se tem conhecimento dos componentes internos de uma aplicação web e não no caso que ela é considerada uma caixa-preta. Além disso, este trabalho pode ser inclusive considerado uma forma de Teste Dirigido por Modelo, pois são aplicadas alterações no modelo de teste original, mantendo-o no mesmo nível de especificidade de plataforma, apenas refinando o modelo de teste com transformações de modelo.

Recentemente, com o desenvolvimento da tecnologia Ajax a forma de se testar uma aplicação web se tornou mais difícil, pois as requisições cliente-servidor, que antes eram síncronas, passaram a ser assíncronas, realizando atualizações dinâmicas nas páginas, como, por exemplo, validação de campos de valores assim que eles são preenchidos.

Dado que os principais elementos de aplicações que usam Ajax são as requisições assíncronas e as manipulações das informações das páginas são feitas com DOM (*Document Object Model*), é esperado que os principais defeitos estejam relacionados a isso. Pensando assim, Marchetto, Tonella e Ricca [50] propuseram que tais aplicações fossem representadas por uma máquina de estados finita (FSM), na qual os estados seriam instâncias de DOM e as transições representariam os efeitos das requisições. Os casos de teste são gerados pela aplicação de um dos critérios de teste associados a FSM, como cobertura de estados ou transições.

Sprenkle *et al.* [63] investigam como o modelo de navegação de uma aplicação web pode afetar a representação em modelo do comportamento do usuário, o tamanho do modelo e os casos de teste gerados. Os modelos de navegação usados para gerar casos de

[1] A análise de mutantes utiliza um conjunto de programas ligeiramente modificados (mutantes) obtidos a partir da aplicação de operadores de mutação em determinado programa P para avaliar o quanto um conjunto de casos de teste T é adequado para o teste de P. O objetivo é determinar um conjunto de casos de teste que consiga revelar, através da execução de P, as diferenças de comportamento existentes entre P e seus mutantes.

teste são criados com base em registros de interações do usuário com a aplicação Web.

Entretanto, há muitas questões em aberto nesta área, tais como: *i*) como construir o modelo de navegação; *ii*) quanta informação é necessária para criar um modelo que melhor descreve a realidade; *iii*) como representar as requisições; e *iv*) como gerar casos de teste. O estudo analisa a influência de alguns fatores no modelo e nos casos de teste abstratos. Estes fatores são a variação de "n" do modelo de navegação e a representação de requisições (i.e. utilizando os valores dos parâmetros ou não).

As conclusões, baseadas em dados reais de cinco aplicações web e mais de 3500 históricos de sessões de usuário, foram que com um pequeno número de sessões, é possível construir o modelo e gerar casos de teste, mas se os objetivos do testador exigem uma porcentagem de cobertura alta, é possível utilizar mais informações e também considerar a adição de valores de parâmetros na representação da requisição.

7.3 Aplicação de MDT em teste de aplicações Web

Testar aplicações web é um grande desafio, ainda mais devido ao seu grande crescimento recente e a seu caráter heterogêneo (podendo ser executadas em diversos tipos de hardware, sistemas operacionais e web browsers, sem contar com a diversidade de linguagens de programação para desenvolvê-las). Algumas propostas têm surgido para tentar atender esta demanda. Dentre elas o trabalho de Andrews *et al.* [10] se destaca e tem algumas similaridades com nosso trabalho.

Em seu trabalho, os autores explicam os problemas em testes de aplicações web causados por dinamicidade e interatividade.

Para resolver o problema de teste caixa-preta de aplicações web, é proposta uma abordagem na qual os testes são criados compondo subtestes derivados de máquinas de estado finitas. O nome da abordagem é FSMWeb e consiste na agregação hierárquica de FSMs, onde as de mais baixo nível são páginas web e a de mais alto nível representa a aplicação web. Os testes completos são obtidos pela combinação das sequências de teste das FSM de mais baixo nível.

Diferentemente de nosso trabalho, este não gera casos de teste executáveis, mas somente casos de teste abstratos que ainda precisam ser executados manualmente ou automatizados de alguma outra forma. Entretanto, não podemos deixar de imaginar que a abordagem proposta por Andrews *et al.* poderia ser usada para derivar os casos de teste abstratos que depois transformamos em executáveis.

Outro *framework* MDT encontrado na literatura é proposto por Li *et al.* [43]. Esse *framework*, nomeado MDWATP, foi desenvolvido como um *plug-in* do ambiente integrado de desenvolvimento Eclipse. Ele trabalha de forma semi-automática e é composto por um modelador e um testador. A função principal do modelador é que seja feito um modelo

do Sistema em Teste (SeT). Este modelo deve representar a navegação da aplicação Web e é denominado *Web Application Navigation Model* (WANM). Já o testador, a partir do modelo WANM gera os casos de teste e os executa. Além disso, após a execução dos testes, os resultados são coletados e apresentados na forma de um relatório.

Contudo, segundo os próprios autores, o MDWATP ainda precisa ser melhorado para que possa ser empregado na maioria das aplicações Web reais. Isso inclui também a geração de dados de teste e o acréscimo de regras para a geração dos casos de teste.

Capítulo 8

Conclusão e Trabalhos Futuros

O principal objetivo deste trabalho foi transpor a lacuna entre os casos de teste abstratos e os casos de teste executáveis, mais conhecidos como scripts de teste, automaticamente. Tal lacuna advém do fato dos casos de teste abstratos serem obtidos de artefatos que estão no mesmo nível da especificação do Sistema em Teste (SeT), enquanto que os scripts de teste devem estar de acordo com a implementação do SeT.

Apesar deste problema se aplicar a muitos tipos de sistemas de software, este trabalho foi focado em aplicações Web, uma vez que estas são as aplicações que mais crescem em termos de importância, uso e complexidade atualmente. Sendo assim, um processo de teste estruturado é de extrema importância, pois um defeito é capaz de causar impactos em nível global.

Entretanto, os gerentes e desenvolvedores de software estão sendo solicitados a produzir seus produtos dentro de um tempo cada vez menor e com recursos mínimos. Colocar um produto no mercado o mais cedo possível significa a diferença entre a sobrevivência do produto ou sua morte - e, portanto, sobrevivência ou morte da companhia. Na tentativa de fazer mais com menos, as organizações querem testar seus sistemas de software de forma adequada, mas dentro de cronograma mínimo. Para isso, eles usam testes automatizados [25], que nada mais são do que scripts de teste.

Escrever os scripts de teste manualmente a partir dos casos de teste abstratos é uma atividade que dispende muito tempo e é susceptível a erros. Por isso, neste trabalho propusemos uma metodologia para mapear automaticamente os casos de teste abstratos para casos de teste executáveis.

A metodologia proposta faz uso de Teste Dirigido por Modelo (MDT), estabelecendo quais são os modelos de teste independente (PIT) e específico (PST) de plataforma, bem como as regras de transformação PIT-PST e PST-código de teste. Tais regras foram, inclusive, submetidas a um processo de teste exclusivamente proposto para testar este tipo de artefato.

Além disso, como ilustrado nos três estudos de caso reais (três aplicações Web reais, duas aplicadas em escala nacional e uma em escala internacional), descrevemos um processo de teste completo, desde a modelagem, passando pela geração de casos de teste abstratos (com ferramentas de Teste Baseado em Modelo (MBT)), até a obtenção dos casos de teste executáveis com nossa metodologia. Dessa forma, vemos que outro objetivo alcançado neste trabalho é o fato da metodologia proposta ser aplicável em aplicações Web reais de grande porte.

8.1 Trabalhos futuros

8.1.1 Expansão dos tipos de sistema de software

São muitos os tipos de sistema de software em que casos de teste executáveis podem ser aplicados. Sendo assim, nosso objetivo é expandir a proposta desta metodologia para outros tipos de sistema. O primeiro tipo escolhido foi sistemas embarcados, considerados muitas vezes críticos, pois envolvem a segurança de pessoas e cuja qualidade é imprescindível, mas que também sofrem do problema clássico do pouco tempo para desenvolvimento com poucos recursos.

8.1.2 Modelagem de aplicações Web

Desejamos avaliar em trabalhos futuros se a forma de modelar as aplicações Web que utilizamos neste trabalho é a que melhor se adequa a nossas necessidades de ter um método genérico para gerar casos de teste executáveis para qualquer tipo de aplicação Web, inclusive aquelas que usam as tecnologias de desenvolvimento mais novas, como Ajax.

8.1.3 MOST-WEB

Nosso objetivo aqui é criar uma ferramenta para teste de aplicações Web que, a partir dos modelos de teste, derive diretamente os casos de teste executáveis. Tal ferramenta faria a união da técnica MOST, apresentada na Seção 6.4.2, com a metodologia deste trabalho.

Referências Bibliográficas

[1] Quickdiff. http://www.quickdiff.com/.

[2] Selenium introduction. http://seleniumhq.org/docs/, 2009.

[3] Jan Aagedal and Ida Solheim. New roles in model-driven development. In *Proceedings of the Second European Workshop on Model Driven Architecture (MDA)*, pages 109–115, 2004.

[4] S. Ali, H. Hemmati, N. E. Holt, E. Arisholm, and L. C. Briand. Model transformations as a strategy to automate model-based testing – a tool and industrial case studies. Technical report, Simula Research Laboratory and University of Oslo, 2010.

[5] Freddy Allilaire, Jean Bézivin, Frédéric Jouault, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 2008.

[6] Everton Alves. Usando mda e mdt para modelagem e geração automática de arquiteturas de teste para sistemas de tempo real. Master's thesis, Universidade Federal de Campina Grande, 2011.

[7] Everton L. G. Alves, Patrícia D. L. Machado, and Franklin Ramalho. Uma abordagem integrada para desenvolvimento e teste dirigido por modelos. In *2nd Brazilian Workshop on Systematic and Automated Software Testing (SAST)*, pages 74–83, 2008.

[8] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[9] Jennitta Andrea. Envisioning the next generation of functional testing tools. *IEEE Softw.*, 2007.

[10] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with fsms. *Software and Systems Modeling*, 2005.

[11] K. Androutsopoulos, N. Gold, M. Harman, Z. Li, and L. Tratt. A theoretical and empirical study of EFSM dependence. In *Proc. ICSM'09: 25th IEEE Int. Conf. on Software Maintenance*, pages 287–296, 2009.

[12] Paul Baker, Shiou Loh, and Frank Weil. *Model–Driven Engineering in a Large Industrial Context – Motorola Case Study*, pages 476–491. Springer Berlin – Heidelberg, 2005.

[13] Krishnakumar Balasubramanian, Aniruddha Gokhale, Gabor Karsai, Janos Sztipanovits, and Sandeep Neema. Developing applications using model-driven design environments. *Article Summaries*, 2006.

[14] Daniel Lima Barbosa. Um método automático de teste funcional para a verificação de componentes. Master's thesis, Universidade Federal de Campina Grande, 2005.

[15] Kent Beck and Erich Gamma. Junit. Technical report, JUnit.org, 2005.

[16] Carlo Bellettini, Alessandro Marchetto, and Andrea Trentini. Testuml: user-metrics driven web applications testing. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 1694–1698, 2005.

[17] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[18] G. v. Bochmann and A. Petrenko. Protocol testing: Review of methods and relevance for software testing. In *International Symposium on Software Testing and Analysis (ISSTA'94)*, pages 109–124, 1994.

[19] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *ISSRE '06 Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 85–94, 2006.

[20] Jean Bézivin. In search of a basic principle for model driven engineering. *UPGRADE*, 2004.

[21] Jim Conallen. Modeling web application architectures with uml. *Commun. ACM*, 1999.

[22] Bruno C. da Silva, Rita S. P. Maciel, and Leandro A. Mascarenhas. Transforms: Uma ferramenta mda/edoc para desenvolvimento de serviços específicos de middleware. In *Simpósio Brasileiro de Engenharia de Software (SBES)*, pages 19–24, 2006.

[23] Zhen Ru Dai. Model-driven testing with uml 2.0. In *Proceedings of the Second European Workshop on Model Driven Architecture*, pages 179–187, 2004.

[24] Camila de Luna Maciel. Uma abordagem dirigida por modelos para a geração automática de casos de teste de integração usando padrões de teste. Master's thesis, Universidade Federal de Campina Grande, 2010.

[25] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley, 1999.

[26] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering*, pages 49–59, 2003.

[27] Marcelo Fantinato, Adriano Cunha, Sindo Dias, Sueli Mizuno, and Cleida Cunha. Autotest - um framework reutilizável para a automação de teste funcional de software. In *Proceedings of Simpósio Brasileiro de Qualidade de Software*, page 15, 2004.

[28] Alain-Georges Vouffo Feudjio. *A Methodology For Pattern-Oriented Model-Driven Testing of Reactive Software Systems*. PhD thesis, Technical University of Berlin, 2011.

[29] F. Fleurey, B. Baudry, P. Muller, and Y. Traon. Qualifying input test data for model transformations. *Software and Systems Modeling*, 8(2):185–203, 2009.

[30] Juergen Grossmann, Ines Fey, Mirko Conrad, and Wolfgang Mueller. Testml- a test exchange language for model-based testing of embedded software, 2008.

[31] Object Management Group. Omg's metaobject facility. http://www.omg.org/mof/.

[32] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *Ibm Systems Journal*, 2006.

[33] Reiko Heckela and Marc Lohmann. Towards model-driven testing. *Electronic Notes in Theoretical Computer Science*, 2003.

[34] M. Hülsbusch, B. König, A. Rensink, M. Semenyak, C. Soltenborn, and H. Wehrheim. Full semantics preservation in model transformation - a comparison of proof techniques. Technical report, Centre for Telematics and Information Technology University of Twente, 2010.

[35] Zentrum Informatik and Dennis Neumann. Test case generation using model transformations, 2009.

[36] A. Javed, P. Strooper, and G. Watson. Automated generation of test cases using model-driven architecture. In *AST '07 Proceedings of the Second International Workshop on Automation of Software Test*, pages 3–9, 2007.

[37] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of atl and qvt. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1188–1195, 2006.

[38] Sheena Judson, Robert France, and Doris Carver. Specifying model transformations at the metamodel level. In *Proceedings of Workshop in Software Model Engineering associated to UML'03*, 2003.

[39] Stuart Kent. Model driven engineering. In *Third International Conference on Integrated Formal Methods*, pages 286–298, 2002.

[40] David C. Kung, Chien-Hung Liu, and Pei Hsia. An object-oriented web test model for testing web applications. In *Proceedings of the The First Asia-Pacific Conference on Quality Software (APAQS'00)*, pages 111–, 2000.

[41] Suet Lee and Jeff Offutt. Generating test cases for xml-based web component interactions using mutation analysis. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 200–, 2001.

[42] K. R. P. H. Leung, L. C. K. Hui, S. M. Yiu, and R. W. M. Tang. Modeling web navigation by statechart. In *COMPSAC '00 Proceedings of the 24th International Computer Software and Applications Conference*, pages 41–47, 2000.

[43] Nuo Li, Qin qin Ma, Ji Wu, Mao zhong Jin, and Chao Liu. A framework of model-driven web application testing. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC)*, pages 157–162, 2006.

[44] Y. Lin, J. Zhang, and J. Gray. Model comparison: A key challenge for transformation testing and version control in model driven software development. In *Object Oriented Programing, Systems, Languages and Application*, 2004.

[45] Y. Lin, J. Zhang, and J. Gray. A testing framework for model transformations, 2005.

[46] Yuehua Lin, Jing Zhang, and Jeff Gray. Model comparison: a foundation for model composition and model transformation testing. In *Proceedings of the 2006 International Workshop on Global Integrated Model Management*, pages 13–20, 2006.

[47] Giuseppe Di Lucca and Anna Fasolino. Testing web-based applications: The state of the art and future trends. *Inf. Softw. Technol.*, pages 1172–1186, 2006.

[48] B. MacDonald, P. Roop, T. Abbas, C. Jayawardena, C. Datta, J. Diprose, J. Hosking, and Z. Bhatti. Case studies for model driven engineering in mobile robotics, 2011.

[49] Natanael E. N. Maia. Odyssey-mda: Uma abordagem para transformação de modelos. Master's thesis, Universidade Federal do Rio de Janeiro, 2006.

[50] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. State-based testing of ajax web applications. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 121–130, 2008.

[51] E. Michael Maximilien and Laurie Williams. Assessing test-driven development at ibm. In *Proceedings of the 25th International Conference on Software Engineering*, pages 564–569, 2003.

[52] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Model-driven development. *IEEE software*, 2003.

[53] G. Morales. *A Test Methodology for the Validation of Web Applications*. PhD thesis, Institut Telecom SudParis, Paris, France, 2010.

[54] Glenford J. Myers, Tom Badgett, Todd M. Thomas, and Corey Sandler. *The Art of Software Testing*. John Wiley and Sons, 2004.

[55] Jeff Offutt and Ye Wu. Modeling presentation layers of web applications for testing. *Springer's Software and Systems Modeling*, 2010.

[56] Ivan Rodolfo Duran Cruz Perez. Geração automática de cenários de teste a partir de modelos da especificação de sistemas. Master's thesis, Universidade Estadual de Campinas, 2008.

[57] Giovanni Piemontese and Guido Diodato. Model-driven architecture, the revolution of software engineering. http://www.idt.mdh.se/kurser/ct3340/archives/ht08/papersRM08/31.pdf, 2008.

[58] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math, 2001.

[59] Gustavo Rossi, Daniel Schwabe, and Fernando Lyardet. Web application models are more than conceptual models. In *Proceedings of the Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling*, pages 239–252, 1999.

[60] Ismayle Santos and Pedro Neto. Automação de testes funcionais com o selenium. http://www.ufpi.br/ercemapi/arquivos/file/minicurso/smc2.pdf, 2009.

[61] Douglas Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 2006.

[62] Manuel Serrano, Mario Piattini, Jose-Norberto Mazón, and Juan Trujillo. Applying mda to the development of data warehouses. In *Proceedings of the 8th ACM international workshop on Data warehousing and OLAP*, pages 57–66, 2005.

[63] Sara Sprenkle, Lori Pollock, and Lucy Simko. A study of usage-based navigation models and generated abstract test cases for web applications. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 230–239, 2011.

[64] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing, 2006.

[65] Dolores R. Wallace and Roger U. Fujii. Software verification and validation: An overview. *IEEE Softw.*, 1989.

[66] Thomas Weigert, Frank Weil, Kevin Marth, Paul Baker, Clive Jervis, Paul Dietz, Yexuan Gui, Aswin Van Den Berg, Kim Fleer, David Nelson, Michael Wells, and Brian Mastenbrook. Experiences in deploying model-driven engineering. In *Proceedings of the 13th international SDL Forum conference on Design for dependable systems*, pages 35–53, 2007.

[67] Marco Winckler, Christelle Farenc, Philippe Palanque, and Rémi Bastide. Designing navigation for web interfaces. In *Proceedings of the 15th Conférence Annuel du Group HCI Group*, 2001.

[68] Ji-Tzay Yang, Jiun-Long Huang, Feng-Jian Wang, and William C. Chu. An object-oriented architecture supporting web application testing. In *23rd International Computer Software and Applications Conference*, pages 122–, 1999.