Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

Geração e Execução de Testes de Aceitação de Sistemas Web

por
Eduardo Henrique da Silva Aranha

Dissertação de Mestrado

Recife, 5 de abril de 2002

Agradecimentos

Agradeço a Deus, por me inspirar e dar forças para enfrentar os desafios da vida. Ao meus pais, que sempre apoiaram meus estudos fornecendo-me a harmonia psicológica e financeira necessária. À minha irmã e conselheira, pela atenção que nunca me foi negada. À minha noiva, que sem dúvida alguma sempre apoiou meus estudos, mesmo quando isto nos privava de viajar, ou simplesmente de ficarmos por algum tempo juntos. Ao meu orientador, pela sua competência, paciência e dedicação exemplar mostrada desde a minha graduação.

Aos meus amigos Leo Galego, Leo Patinadora, Sandrelly Mau Colega e Murilo, por me apoiarem segurando para mim as dificuldades do trabalho em todo esse meu tempo de mestrado. Ao Instituto de Planejamento e Apoio ao Desenvolvimento Tecnológico e Científico – IPAD, onde me foi fornecido o tempo e apoio tão necessários para o desenvolvimento deste trabalho.

Ao professor Jorge Fernandes, por sua experiência, incentivos e reconhecimentos dados para mim durante nossa convivência.

Ao Centro de Informática, pela estrutura e o suporte técnico e acadêmico fornecidos, assim como a todos que apostaram e torceram por mim.

Obrigado.

Resumo

A fim de tornar os sistemas Web mais robustos, a construção de testes de softwares Web e sua automação vem sendo enfatizada por metodologias de desenvolvimento. A metodologia *Extreme Programming* (XP), por exemplo, tem destacado a atividade de teste (em particular, os testes de aceitação e de unidade) como uma das práticas de programação chaves para o sucesso de sua implantação, sendo a construção destes testes realizada antes mesmo da implementação do código testado.

Apesar de ter suas vantagens reconhecidas, a atividade de teste é na maioria das vezes ignorada ou apenas parcialmente realizada, principalmente quando deixado para o final do desenvolvimento. Sendo mais um esforço para o incentivo da realização de testes de sistemas Web, em particular os testes de aceitação, este trabalho tem como objetivo definir a linguagem WSat para descrição de casos de teste de aceitação com alto nível de abstração e reuso.

Para dar suporte a utilização desta linguagem, objetivamos a criação de um ambiente de execução para a mesma. Além disto, este trabalho também estuda a criação de possíveis geradores de código que possam ser utilizados para amenizar o impacto negativo na produtividade, a curto prazo, causado pela realização dos testes de aceitação. Estes geradores dão suporte a criação destes testes antes da implementação do próprio sistema, prática apresentada pela metodologia.

Para a utilização eficaz da linguagem de teste definida e das ferramentas de apoio, definimos um processo de teste. Este processo define os papéis a serem realizados, as atividades necessárias e seu fluxo de execução. Por fim, para validar a linguagem, o processo de teste e as ferramentas construídas, realizamos um experimento com um sistema real de pequeno porte.

Abstract

In order to Web systems become more robust, the construction of Web softwares tests and its automation has being emphasized by development methodologies. The Extreme Programming methodology (XP), for example, has highlighted the test activity (in particular, the unit and acceptance tests) as one of the key practices of programming for the success of its implantation, being the construction of these tests done before the implementation of the tested code.

Despite have its advantages recognized, the test activity is in general ignored or partially done, mainly when left for the end of the development. Being another effort for the incentive of the accomplishment of Web systems tests, in particular the acceptance tests, this work has the purpose to define a language for acceptance test case description with high level of abstraction and reuse.

To support the use of this language, we aim at creation of an execution environment for it. Moreover, this work also studies the creation of possible code generators that can be used to reduces the negative impact in the productivity, in a short time, caused for the accomplishment of the acceptance tests. These generators give support for the creation of these tests before the implementation of the tested system, practice presented by the methodology.

In order to achieve an efficient use of the defined test language and the auxiliares tools, we define a test process. This process defines the roles involved, the necessary activities and its execution flow. Finally, to validate the language, the test process and the constructed tools, we did an experiment with a real system of small size.

Índice

1 Introdução **1**

1.1 Testes de Aceitação em Sistemas Web 3

1.2 Organização da Dissertação 4

2 Ferramentas de Teste de Sistemas Web **5**

2.1 Critérios Utilizados 6

2.1.1 Escolha das Ferramentas 6

2.1.2 Fatores de Qualidade do Produto 7

2.1.3 Fatores de Qualidade do Desenvolvimento 8

2.1.4 Suporte Fornecido e Fatores Afetados 8

2.1.5 Metodologia de Análise Utilizada 11

2.2 QARun 13

2.2.1 A Estrutura de Um Script de Teste 15

2.2.2 Utilização de Variáveis 15

2.2.3 Comandos Para Simulação do Usuário 16

2.2.4 Comandos para a Lógica de Programação 18

2.2.5 Comandos para Verificação de Dados 19

2.2.6 Manipulação de Arquivos de Dados 20

2.2.7 Invocando Outros Scripts 20

2.2.8 Funções Definidas Pelo Usuário 21

2.2.9 Acesso a Banco de Dados 21

2.2.10 Exemplo de Implementação do Caso de Teste 21

2.3 JXWeb 23

2.3.1 Um Primeiro Script de Teste 23

2.3.2 Comandos Básicos 23

2.3.3 Comandos Para Simulação de Ações do Usuário 24

2.3.4 Objetos de Teste 26

2.3.5 Comandos de Verificação Dos Dados 27

2.3.6 Manipulação de Erros e Falhas na Execução de Scripts de Teste . 29

2.3.7 Estendendo JXWeb 29

2.3.8 Exemplo de Implementação do Caso de Teste 30

2.4 Análise das Ferramentas 30

2.4.1 Suportes Fornecidos Pela Ferramenta QARun 30

2.4.2 Suportes Fornecidos Pela Ferramenta JXWeb 32

2.4.3 Comparação entre as Ferramentas 34

2.5 Conclusões 34

3 A Linguagem WSat **36**
3.1 Introduzindo WSat 38
3.2 Detalhando WSat 39
3.2.1 Sistema de Busca na Web 40
3.2.2 Descrevendo a Estrutura da GUI do Sistema 41
3.2.3 Descrevendo o Comportamento do Sistema 53
3.3 Execução de Programas WSat 62

4 Ferramentas para a Realização de Testes Web **63**
4.1 Ambiente de Execução de WSat 64
4.1.1 Técnica de Implementação 65
4.1.2 Um *Parser* para WSat 66
4.1.3 Compilando Código WSat para Código Java 71
4.1.4 Executando os Casos de Teste 86
4.1.5 Suportes Fornecidos 87
4.2 Gerador de Código de Teste 88
4.2.1 Geração de Código de Teste ou de Telas HTML 89
4.2.2 Projeto e Implementação do Gerador 91
4.2.3 Executando o Gerador 94
4.3 Gerador de Código de Sistema Web 95
4.3.1 O Ambiente e Arquitetura de Desenvolvimento 95
4.3.2 Projeto e Implementação 96
4.3.3 Executando o Gerador de Código de Sistema 106
4.4 Modelagem do Processo de Teste com WSat 107
4.4.1 Fluxo de Atividades 107
4.4.2 Execução dos Testes WSat 107
4.4.3 Gerador de Código de Teste 108
4.4.4 Gerador de Código de Sistema 108

5 Avaliação do Uso Real de WSat e das Ferramentas Associadas **110**
5.1 Características do Experimento Realizado 111
5.1.1 O Sistema Desenvolvido no Experimento 111
5.1.2 O Experimento 112
5.1.3 Critérios Utilizados para Coleta e Avaliação dos Dados 114
5.2 Análise dos Resultados 115
5.2.1 Métricas Utilizadas 115
5.2.2 Resultados por Fatores de Qualidade de Software e de Processo . 116
5.2.3 Resultados por Fatores de Produtividade 118

6 Conclusões **122**
6.1 Conclusões 123
6.2 Trabalhos Relacionados 125
6.3 Trabalhos Futuros 125

A Gramática da Linguagem WSat **127**

Lista de Figuras

2.1 Tela inicial do sistema de busca de imóveis. 13
2.2 Tela de resposta do sistema de busca de imóveis. 14

3.1 Tela inicial do sistema de busca testado. 40
3.2 Tela de resposta do sistema de busca. 42
3.3 Tipos especiais predefinidos em WSat. 43

4.1 Classes que representam nós da árvore sintática de WSat. 67
4.2 Interface definida para os *visitors* das árvores sintáticas de WSat. 68
4.3 Topo da hierarquia de classes da árvore sintática de WSat. 71
4.4 Classes Java representando os tipos WSat predefinidos. 73
4.5 Sequência de comandos executados na tradução da declaração de uma função. 84
4.6 Sequência de comandos executados na tradução da invocação de uma função de teste. 85
4.7 Classe responsável pela geração de código WSat. 91
4.8 Hierarquia de *handlers* de processamento. 102
4.9 Hierarquia de *handlers* de apresentação. 103
4.10 Fluxo de atividades para utilização das ferramentas. 107

Lista de Tabelas

2.1 Fatores de qualidade afetados para cada suporte fornecido. 12
2.2 Símbolos utilizados para indicar o nível de suporte fornecido. 34
2.3 Comparação dos níveis de suporte fornecidos pelas ferramentas. 35

3.1 Restrições suportadas pela cláusula `validate`. 52

4.1 Comparação dos níveis de suporte fornecidos pelas ferramentas. 89

5.1 Tempo total gasto e número de linhas de código escritas em cada projeto. 118
5.2 Tempo gasto e número de linhas de código do Projeto B e a variação obtida com relação ao Projeto A. 119
5.3 Percentual de defeitos encontrados pelo seu tempo para descoberta. . . . 120
5.4 Número total de classes por projeto. 120
5.5 Tempo para finalizar primeiro protótipo usável. 121
5.6 Percentual de código gerado automaticamente. 121

Capítulo 1

Introdução

Neste capítulo introduzimos e motivamos a realização de testes de aceitação em sistemas Web. Mostramos também a necessidade de se definir novas linguagens e ferramentas que dêem suporte a realização destes testes. Por fim, é apresentada a estrutura utilizada nesta dissertação.

O uso crescente da Internet fez com que surgissem inúmeras aplicações voltadas para a Web. Uma aplicação Web é a utilização de um sistema Web (conjunto de softwares) para resolver um problema específico. Entre as aplicações de sistemas Web estão as de comércio eletrônico, sistemas de busca, B2B, entre diversas outras. Entre as características inerentes a estes sistemas estão:

- Mudança frequente de requisitos;
- Funcionamento 24h por dia, 7 dias por semana (Sistemas 24x7).

As características apresentadas mostram que o código dos softwares (programas de computador) que implementam os sistemas Web estão sendo frequentemente modificados para acompanhar as mudanças dos requisitos. Além disto, defeitos descobertos pelo seu uso por usuários em horários fora do expediente das empresas podem causar a indisponibilidade do sistema até o início do próximo expediente, ou o deslocamento de funcionários em horários fora de seu expediente normal. Supondo que o expediente encerre às 18h, erros no sistema a partir deste horário como por exemplo o de chegar a um estado inconsistente possivelmente só serão detectados e resolvidos com a chegada dos funcionários no início do próximo expediente.

A fim de tornar os sistemas mais robustos, a construção de testes de softwares (incluindo os voltados para a Web [36]) e sua automação vem sendo enfatizada por diversas metodologias de desenvolvimento. A metodologia *eXtreme Programming* (XP) [3], por exemplo, tem destacado a atividade de teste (em particular, os testes de aceitação e de unidade) como uma das práticas de programação chave para o sucesso de sua implantação, onde a construção destes testes é realizada antes mesmo da implementação do código a ser testado [12].

Quanto mais avançada é a etapa no ciclo de vida do software onde um defeito é encontrado, maior é o custo de corrigí-lo. Isto pode ser verificado na prática, onde mostra-se mais fácil, por exemplo, corrigir um defeito no código na fase de implementação do sistema do que corrigi-lo quando o sistema já está implantado em clientes. A utilização de testes aumenta as chances de detectar mais cedo os defeitos existentes no sistema, aumentando assim a produtividade geral e a qualidade dos sistemas produzidos. Além disto, com a automação dos testes, podemos verificar mais facilmente se manutenções realizadas no sistema inseriram defeitos que não existiam antes da manutenção (testes de regressão).

Apesar de ter suas vantagens reconhecidas, a atividade de teste é na maioria das vezes ignorada ou apenas parcialmente realizada, principalmente quando deixada para o final do ciclo de desenvolvimento do software. A seguir são discutidos alguns dos principais motivos para a ocorrência destes fatos.

É recional pensar que os desenvolvedores de software não gostam de testar seu próprio código, pois a descoberta de defeitos pode ferir os seus egos. Além disto, a atividade de teste pode transmitir o sentimento de perda de tempo, pois ela não acrescenta novas funcionalidades aos sistemas nem auxilia diretamente o desenvolvimento das mesmas. Este tipo de problema não é abordado neste trabalho por não ser resolvido apenas com o fornecimento de tecnologias de apoio.

Outro fator negativo é o suporte ainda inadequado das ferramentas disponíveis para construção e execução de testes. O uso de linguagens de programação convencionais

como Java [19] e Visual Basic [20] adicionam complexidade desnecessária, dificultando a programação dos testes [27]. Isto porque elas foram desenvolvidas para implementar algoritmos complexos, não provendo um mecanismo natural para a definição de testes como por exemplo os testes de aceitação. As ferramentas de apoio a testes disponíveis, como a QARun [8] e JXWeb [26], fornecem linguagens de teste de difícil programação devido ao seu baixo nível de abstração e legibilidade [33]. Este fato torna em geral a atividade de teste trabalhosa, entediante e de difícil manutenção [33].

Algumas ferramentas fornecem ainda a possibilidade de gravar os casos de teste a partir da execução do sistema testado. Este recurso, disponível apenas nas ferramentas mais avançadas como a QARun, facilita bastante a criação dos casos de teste. Entretanto, os testes gravados possuem pouca flexibilidade para eventuais manutenções nos casos de teste e no sistema testado. Pequenas mudanças nos casos de teste podem ser realizadas em casos extremos somente com uma nova gravação completa do caso de teste. Além disto, este recurso exige que o sistema testado esteja implementado, inviabilizando a criação de testes antes da implementação do código do sistema.

É racional deduzir que a atividade de teste aumente a longo prazo a corretude e produtividade geral do desenvolvimento de software, dado que defeitos no software serão descobertos mais cedo diminuindo o trabalho de sua manutenção. Entretanto, por ser trabalhosa e entediante, a atividade de teste causa, em geral, um impacto negativo na produtividade a curto prazo do desenvolvimento de software. Consequentemente, atrasos no desenvolvimento do sistema acabam sendo naturalmente amenizados através da redução ou eliminação da atividade de teste.

Sendo mais um esforço para o incentivo da realização de testes de sistemas Web, em particular os testes de aceitação, este trabalho tem como objetivo definir uma linguagem para descrição de casos de teste de aceitação com alto nível de abstração e reuso. Para dar suporte a utilização desta linguagem, objetivamos também a criação de um ambiente de execução [1] para a mesma. Além disto, este trabalho estuda a criação de possíveis geradores de código que sejam utilizados para amenizar o impacto negativo na produtividade, a curto prazo, causado pela realização dos testes de aceitação e dar suporte a criação destes testes antes da implementação do próprio sistema, prática apresentada pela metodologia *Extreme Programming*.

Para isto, realizamos em primeiro lugar a análise do suporte fornecido por duas ferramentas de teste existentes no mercado, principalmente com relação à sua linguagem. Em seguida, definimos a linguagem WSat de acordo com alguns critérios como abstração e reuso e criamos um ambiente de execução para a mesma. A partir de WSat, estudamos a criação de geradores de código de teste e de código de sistema. Por fim, validamos a linguagem definida e ferramentas construídas através da realização de um experimento.

1.1 Testes de Aceitação em Sistemas Web

Para verificar se os requisitos funcionais especificados pelo cliente são satisfeitos por um sistema Web, podemos utilizar casos de teste de aceitação. Testes de aceitação são testes construídos para verificar se o sistema testado satisfaz os requisitos funcionais especificados durante a análise dos requisitos. Isto permite que a execução correta dos testes de aceitação seja considerada como requisito básico de qualidade para a liberação de versões do sistema a clientes. Diferentemente dos testes de unidade, os testes de

aceitação não são utilizados para verificar exaustivamente as possíveis entradas e saídas do sistema.

Como os testes de aceitação indicam quais funcionalidades estão executando corretamente, o número de testes satisfeitos pode ser utilizado para acompanhamento do progresso do desenvolvimento do sistema.

Testes de aceitação interagem com a GUI (*Graphical User Interface*) do sistema, simulando as ações dos usuários e verificando o conteúdo das informações apresentadas pelo sistema. Com isto, informações sobre a estrutura e comportamento da GUI de um sistema podem ser extraídas a partir de seus casos de teste de aceitação, possibilitando a geração de trechos de código da GUI do sistema.

Este tipo de teste é comumente chamado de teste funcional, sendo neste trabalho utilizando o termo teste de aceitação por acreditarmos que o mesmo descreve melhor o seu objetivo, que é verificar se os requisitos funcionais definidos pelo cliente são satisfeitos e o sistema se encontra em um estado aceitável.

1.2 Organização da Dissertação

Este trabalho contém mais cinco capítulos. O Capítulo 2 analisa e compara duas ferramentas de teste disponíveis no mercado, de acordo com a qualidade do suporte aos testadores fornecido pelas mesmas, principalmente com relação às suas linguagens de script.

No Capítulo 3, definimos a linguagem WSat (Web System Acceptance Test) para a descrição de casos de teste de aceitação de sistemas Web. Em seguida, apresentamos no Capítulo 4 o desenvolvimento de ferramentas de apoio a realização de teste de aceitação de sistemas Web escritos em WSat. São discutidos geradores de código baseados na linguagem de teste definida e que permitem uma redução de tempo no desenvolvimento dos casos de teste e do próprio sistema.

Para validar a linguagem proposta e os geradores construídos, realizamos o experimento descrito no Capítulo 5. Nele são apresentados os resultados obtidos, em termos de fatores de qualidade e produtividade de software, pelo uso da linguagem e ferramentas desenvolvidas por este trabalho.

Por fim, no Capítulo 6, são apresentadas as conclusões e sumarizadas as principais contribuições obtidas. Por fim, são sugeridos trabalhos futuros.

Capítulo 2

Ferramentas de Teste de Sistemas Web

Neste capítulo apresentamos uma análise comparativa entre ferramentas de teste de sistemas Web. A análise destas ferramentas é focada principalmente nas suas linguagens para descrição de testes, em particular os testes de aceitação.

A atividade de testes tem um papel fundamental no desenvolvimento de software, principalmente em se tratando de sistemas Web, onde são freqüentemente realizadas mudanças em seus requisitos. Na metodologia de desenvolvimento de software *Extreme Programming* [3], por exemplo, a atividade de teste de software é considerada fundamental, onde se utiliza a prática de, para cada funcionalidade nova, criar-se o código para testá-la antes de implementá-la.

Apesar dos desenvolvedores de software saberem que devem criar testes para suas aplicações, poucos o fazem, por vários motivos:

- A atividade de teste é realizada para descobrir falhas no código do próprio desenvolvedor, podendo assim afetar seu próprio ego.
- Cronogramas de desenvolvimento fora do prazo.
- Falta de ferramentas para a implementação dos testes.

Este capítulo aborda o último motivo apresentado, tendo como objetivo orientar a escolha e desenvolvimento de ferramentas para implementação de testes de sistemas Web como, por exemplo, os testes de aceitação. Para isto, foram definidas métricas para a análise comparativa de duas ferramentas de teste disponíveis no mercado: QARun [8] e a JXWeb [26]. Foi realizado o estudo aprofundado nas linguagens para descrição de casos de teste destas ferramentas. Os resultados gerados por este estudo servem de base para o desenvolvimento de novas linguagens de teste, como é apresentado no Capítulo 3.

Na Seção 2.1, definimos métricas que facilitam a análise de ferramentas deste tipo. Após isto, nas Seções 2.2 e 2.3, são feitas introduções para duas ferramentas comerciais existentes no mercado e que foram escolhidas para serem analisadas. Em seguida, na Seção 2.4, comparamos baseado nas métricas definidas o estado atual de suporte à atividade de teste fornecída pelas duas ferramentas. Por fim, na Seção 2.5, apresentamos as conclusões obtidas sobre as ferramentas de teste analisadas neste capítulo.

2.1 Critérios Utilizados

Nesta Seção, mostramos os critérios que utilizamos para a escolha e análise das ferramentas de teste de sistemas Web. Para análise das ferramentas, levamos em conta o suporte fornecido pelas mesmas, já que este suporte pode afetar a qualidade final dos produtos bem como o desenvolvimento dos mesmos.

Em primeiro lugar, mostramos os critérios utilizados para escolha das ferramentas. Em seguida, são apresentados os principais suportes encontrados na literatura que podem ser fornecidos por ferramentas de teste de sistemas Web, assim como fatores de qualidade e produtividade afetados pelos mesmos. Por fim, apresentamos a metodologia utilizada para a análise destas ferramentas.

2.1.1 Escolha das Ferramentas

Para escolher as ferramentas a serem analisadas neste trabalho, foram procuradas aquelas que atendem às necessidades atuais do mercado. Também foram levados em consideração trabalhos acadêmicos como por exemplo o da linguagem Test Talk [28]. Para isto,

foram levadas em conta indicações encontradas em revistas on-line de Java [23, 29, 32], de teste de software [11] e de descrições das mesmas em repositórios on-line de ferramentas de teste de sistemas Web [21].

Durante esta etapa, foram encontradas algumas dificuldades. A primeira delas, foi o grande número de ferramentas de médio e pequeno porte encontradas para a realização de testes de sistemas Web, assim como a grande variação de qualidade fornecida pelas mesmas. Uma outra dificuldade foi a obtenção de licenças de avaliação para as mesmas.

As ferramentas escolhidas para análise foram a QARun [8] e a JXWeb [26]. Os fatores decisivos para a escolha da ferramenta QARun foi o seu alto conceito na comunidade de teste de software e o acesso a uma licença de avaliação. Já a escolha da ferramenta JXWeb, que ainda está em construção, se deu por possuir características interessantes ao trabalho. A JXWeb é uma extensão para Web do framework JUnit, este último largamente divulgado e utilizado pela sua indicação na metodologia *Extreme Programming*. Além disto, JXWeb é implementada em Java e XML, tendo seu código aberto ao público disponibilizado através da Internet, o qual pode ser utilizado de forma gratuita (ver respectiva licença de uso) [3].

2.1.2 Fatores de Qualidade do Produto

Esta seção apresenta os fatores de qualidade que podem ser afetados pelo uso de ferramenta de testes, entre outras coisas.

Q1. Corretude. Habilidade dos sistemas de executarem suas respectivas tarefas, de acordo com a definição em suas especificações. A corretude é um fator de qualidade fundamental, pois quando um sistema não realiza o que ele está suposto a fazer, mesmo funcionalidades adicionais acabam perdendo sua importância.

Q2. Robustez. Habilidade dos sistemas de reagir apropriadamente a condições anormais de uso. Complemento da corretude, a robustez está relacionada ao comportamento do sistema em situações não especificadas. Garantir que sistemas Web sejam robustos pode ser uma árdua tarefa, devido ao grande número de possibilidades de instalação e de uso indevido de sistemas Web.

Q3. Eficiência. Habilidade de um sistema de criar o mínimo de demanda possível em recursos de hardware, como processadores, espaço de memória ocupado e largura de banda em periféricos de comunicação. Com a Internet, pessoas de quase todo o planeta podem acessar sistemas Web. Dessa forma, a eficiência de um sistema é um fator crítico e determinante para a sua capacidade de atendimento a clientes.

Q4. Portabilidade. Facilidade de transferir produtos de software para vários ambientes de hardware e software. No caso de sistemas Web, é comum a utilização diversa destes ambientes, tanto no lado cliente como no lado servidor. Os servidores Web podem ser instalados em diversos sistemas operacionais, como por exemplo, no Linux e no Windows NT, e executados em hardwares distintos, como PCs ou máquinas AIX. O mesmo acontece no lado cliente, com várias possibilidades de navegadores Web, sistemas operacionais e hardwares intalados.

Q5. Disponibilidade. Com seus acessos realizados através da Internet, sistemas Web são conhecidos por terem seu funcionamento 24 horas por dia, sete dias por semana.

Sistemas de compras on-line, por exemplo, devem estar disponíveis 24 horas por dia para não haver possíveis prejuízos como, por exemplo, uma impossibilidade temporária de vendas.

2.1.3 Fatores de Qualidade do Desenvolvimento

A seguir, apresentamos os fatores de qualidade do processo de desenvolvimento de sistemas Web que podem ser afetados pelo uso de ferramenta de testes.

Q6. *Timeliness.* Habilidade de um sistema de software ser entregue aos seus clientes no momento, ou até antes, que eles o desejem. O advento da Internet trouxe, entre outras coisas, uma maior concorrência. Dessa forma, os produtos de software têm cada vez menos tempo para serem lançados, sendo este um fator às vezes fundamental para o sucesso do empreendimento.

Q7. Produtividade. Rendimento de uma equipe de desenvolvimento de software na realização de suas atividades. A produtividade é um fator fundamental no processo de desenvolvimento de sistemas Web.

2.1.4 Suporte Fornecido e Fatores Afetados

Nesta seção, apresentamos os suportes fornecido por ferramentas de teste de sistemas Web e como cada um destes suportes afetam os fatores de qualidade do produto e de seu desenvolvimento.

S1. Verificação da Sintaxe de Páginas HTML. Páginas HTML são em geral criadas por editores de texto específicos, abstraindo do desenvolvedor a sintaxe dos arquivos HTML. Entretanto, recursos HTML mais recentes ou até os próprios editores podem não estar disponíveis para o desenvolvedor. Desta forma, ele pode ser obrigado a editar o código fonte dos arquivos através de simples editores de texto, facilitando a inserção de erros sintáticos.

Corretude: A verificação da sintaxe das páginas HTML por ferramentas de teste de sistemas Web tornam-se importante para garantir que as páginas sejam corretamente visualizadas pelos navegadores Web.

Portabilidade: Sem o suporte à verificação da sintaxe HTML pelas ferramentas, esta verificação é feita visualizando a página através de navegadores Web. Alguns navegadores conseguem apresentar páginas HTML sintaticamente incorretas enquanto outros não. A portabilidade do sistema fica desta forma comprometida quanto ao uso de navegadores Web.

S2. Teste de *Layout* dos Componentes de Páginas HTML. Testes sobre a distribuição espacial (programação visual) dos componentes de páginas HTML são suportados por algumas ferramentas de teste.

Corretude: O posicionamento incorreto dos elementos contidos em páginas HTML podem causar o entendimento incorreto das informações.

S3. Reprodução de Ações Humanas. Para a realização de testes de aceitação em

sistemas Web, é necessária a possibilidade de simular a execução das ações dos seus usuários, reproduzindo assim todos os passos dos seus usuários dentro do sistema.

Corretude: Em uma simulação completa, as ferramentas de teste de sistemas Web testam o funcionamento correto de elementos HTML (*links*, por exemplo), códigos JavaScript, Applets, Flash, ActiveX e outros.

S3.1. Reprodução Através de Linguagem de Script. Ferramentas de teste que permitem a simulação de ações humanas podem reproduzir estas ações através de linguagens de script. Estas linguagens possuem o poder de representar sequências de ações realizadas por usuários Web.

Produtividade: Linguagens de script, além de alto nível de abstração na programação, podem conter facilidades para estender, modificar e até combinar scripts de testes.

S3.2. Reprodução Através de Gravações. As simulações de ações humanas por ferramentas de teste podem ser suportadas através da reprodução de gravações de uso por usuários do sistema.

Produtividade: Gravações do uso do sistema em geral diminuem o tempo de aprendizado da ferramenta e de implementação dos casos de teste.

S4. Simulação de Navegadores Internet. Algumas ferramentas de teste de sistemas Web possuem a capacidade de simular diferentes versões de navegadores Internet através da execução dos engenhos utilizados pelos próprios navegadores.

Corretude: A simulação dos próprios engenhos de navegadores Web permite testar com mais segurança o bom funcionamento do sistema com tais navegadores.

Portabilidade: Pode ser possível testar também o bom funcionamento do sistema com versões anteriores dos navegadores, possivelmente ainda utilizadas por parte dos usuários.

Produtividade: Desenvolvedores não precisarão mais executar um mesmo teste manualmente para cada versão de navegador Web.

S5. Verificação do Tempo de Resposta. Na Internet, o tempo de espera do usuário pode representar o sucesso ou o fracasso de sistemas Web. Agravado pelo tempo de conexão da rede, grandes tempos de resposta incentivam o usuário a cancelar a operação e possivelmente sair do site que hospeda o sistema.

Eficiência: Para garantir um tempo de resposta satisfatório para o sistema, é necessário verificar o tempo de resposta dos sistemas em teste.

S6. Acesso a Banco de Dados. Ferramentas de teste podem permitir que os dados apresentados por sistemas Web possam ser comparados com dados contidos nos bancos de dados dos sistemas testados.

Corretude: Alguns testes podem ser realizados somente com a verificação dos dados existentes no banco de dados do sistema testado.

Produtividade: Existem testes que, mesmo que não necessitem deste tipo de suporte, são mais fáceis de serem criados com o acesso a banco de dados.

S7. Verificação do Controle de acesso. Ferramentas de teste de sistemas Web podem possibilitar a verificação da segurança das conexões e do controle de acesso aos usuários, suportando então o uso de conexões HTTPS, cookies e de criptografia de dados.

Corretude: Sistemas disponibilizados através da Internet necessitam de uma atenção especial para permitir somente o acesso autorizado a seus dados.

Robustez: A execução de testes na verificação dos aspectos de segurança tornam os sistemas Web mais robustos em tentativas de invasão.

Produtividade: O suporte na verificação do controle de acesso por ferramentas de teste deixa o desenvolvedor com mais tempo para outras tarefas. Isto porque torna-se desnecessário que o desenvolvedor programe estes tipos de testes através de linguagens de programação ou realize-os manualmente.

S8. Simulação de Usuários Concorrentes. Quando em funcionamento, sistemas Web são acessados concorrentemente por possivelmente um grande número de usuários. Ferramentas de teste podem fornecer suporte para a simulação do uso concorrente de sistemas Web.

Corretude: Erros de concorrência podem ser encontrados nos sistemas Web através da simulação de usuários concorrentes.

Robustez: A simulação de usuários concorrentes gera condições anormais de uso. A execução destes tipos de testes auxilia a garantir que o sistema irá responder de maneira satisfatória em tais condições.

Eficiência: Os chamados testes de carga, simulação de um grande número de usuários concorrentes, podem medir a performance do sistema e auxiliar a detectar problemas, como por exemplo, gargalos no sistema.

S9. Escalonamento da Execução de Testes. O escalonamento da execução dos testes de software é uma característica desejável em ferramentas de teste.

Corretude: A automação dos testes permite que os mesmos sejam executados com uma maior frequência, inclusive de forma regressiva.

Produtividade: A execução dos testes necessita de mínima ou até nenhuma presença humana, liberando os desenvolvedores para outras atividades.

Timeliness: O aumento da frequência execução dos testes obtidas com o escalonamento automático permite a descoberta de erros mais cedo, possibilitando uma maior segurança em casos de antecipações no lançamento do produto.

S10. Parametrização dos Dados de Teste. A possibilidade de parametrização de testes permite que um mesmo caso de teste seja executado várias vezes, com a variação apenas dos dados utilizados no teste.

Corretude: A parametrização dos dados de teste permite um aumento na cobertura do sistema pelos testes.

Produtividade: O aumento da cobertura é realizado sem causar um grande aumento no

tempo de implementação dos casos de teste. As ferramentas também podem suportar distintas formas de parametrização, como através de bancos de dados, arquivos textos, entre outros, com um mínimo ou, se possível, a ausência de programação.

S11. Portabilidade dos Casos de Testes. Em geral, o uso do protocolo HTTP traz esta capacidade às ferramentas de teste. Entretanto, características como a capacidade de simulação de navegadores Web e de testar diretamente os dados inseridos no banco de dados da aplicação em teste podem dificultar a portabilidade dos testes. Por exemplo, o suporte pela ferramenta apenas para conexões Microsoft ODBC, simulação de navegadores Web somente no ambiente Windows, etc.

Portabilidade: A capacidade das ferramentas de executarem testes em sistemas Web instalados em diferentes ambientes de hardware e software auxilia a verificação da portabilidade destes sistemas para estes ambientes.

S12. Monitoramento do Sistema. Sistemas Web funcionam 24 horas por dia, 7 dias por semana. Desta forma, torna-se importante monitorá-los o maior tempo possível durante o seu funcionamento. Ferramentas de teste podem ser utilizadas para criar casos de teste que permitam, em caso de falhas, realizar o disparo de alarmes (como o envio de mensagens a aparelhos celulares, por exemplo) até a realização de ações restauradoras, como a re-inicialização do sistema.

Disponibilidade: As ações disparadas pelos casos de teste podem fazer com que os sistemas em estados inconsistentes voltem mais rapidamente ao seu funcionamento normal.

Produtividade: Com esta capacidade, mesmo em sistemas críticos pode não ser necessária ocupar o tempo de pessoas 24h por dia para monitorar o funcionamento dos sistemas.

S13. Depuração de Testes. Assim como as ferramentas de programação, as ferramentas de teste podem permitir a execução passo a passo dos testes, possuir arquivos de log de erro e outros artifícios que facilitem a depuração dos testes.

Corretude: A depuração pode permitir garantir que os testes criados estão testando os sistemas corretamente.

Produtividade: Através do suporte de depuração, erros podem ser encontrados mais rapidamente.

Na tabela 2.1, apresentamos os relacionamentos entre os suportes de ferramentas e os fatores de qualidade citados que podem ser impactados pelos mesmos.

2.1.5 Metodologia de Análise Utilizada

Inicialmente, foi realizada uma análise individual de cada ferramenta. Nesta análise, em primeiro lugar, buscou-se o entendimento geral da ferramenta através da implementação de casos de testes de um sistema real em funcionamento (ver Seção 2.1.5). Em seguida, foram verificados os suportes que eram fornecidos pela ferramenta de teste.

Durante o estudo de cada ferramenta, foram implementados casos de teste relativos ao sistema Web ExpoImóvel [41]. Estes casos de teste foram definidos para testar o

Suportes Fornecidos / Fatores Afetados	Q1	Q2	Q3	Q4	Q5	Q6	Q7
S1. Verificação da Sintaxe de Páginas HTML	√			√			
S2. Teste de Layout dos Componentes de Páginas HTML	√						
S3.1. Reprodução Através de Linguagem de Script	√						√
S3.2. Reprodução Através de Gravações	√						√
S4. Simulação de Navegadores Internet	√			√			√
S5. Verificação do Tempo de Resposta			√				
S6. Acesso a Banco de Dados	√						√
S7. Verificação do Controle de acesso	√	√					√
S8. Simulação de Usuários Concorrentes	√	√	√				
S9. Escalonamento da Execução de Testes	√					√	√
S10. Parametrização dos Dados de Teste	√						√
S11. Portabilidade dos Casos de Testes				√			
S12. Monitoramento do Sistema					√		√
S13. Depuração de Testes	√						√

Tabela 2.1: Fatores de qualidade afetados para cada suporte fornecido.

módulo de busca de imóveis do sistema, cuja tela inicial e tela de resposta podem ser vistas respectivamente nas Figuras 2.1 e 2.2. A seguir podemos ver um exemplo simplificado de um dos casos de teste utilizados durante a análise de cada ferramenta.

1. Acessar a URL "http://www.expoimovel.com.br". Será retornada uma página com 2 frames, um de nome superior, contendo uma página com o menu do sistema, e outro de nome principal, contendo uma página com informações do sistema.

2. Na página do frame de nome superior, clicar no *link* da imagem de nome classificados.

3. Na página retornada, conferir se existem os seguintes componentes Web:

   - 3 elementos SELECT, de nomes tipo, cidade e pretensao;
   - 2 imagens com *links* de nomes continuar e limpar.

4. Clicar na imagem de nome continuar.

5. Na página retornada, conferir se existem

   - 8 elementos SELECT, de nomes bairro, preco, areautil, aluguel, garagem, banheiro, quarto e dependenciaEmpregada;
   - 3 imagens de nomes consultar, limpar e voltar, todas com *links*.

6. Clicar no primeiro *link* da página resultante, que dá acesso ao primeiro imóvel apresentado pela busca.

7. Verificar se a resposta do pedido HTTP retorna com o código 200 (página existe e não houve erro).

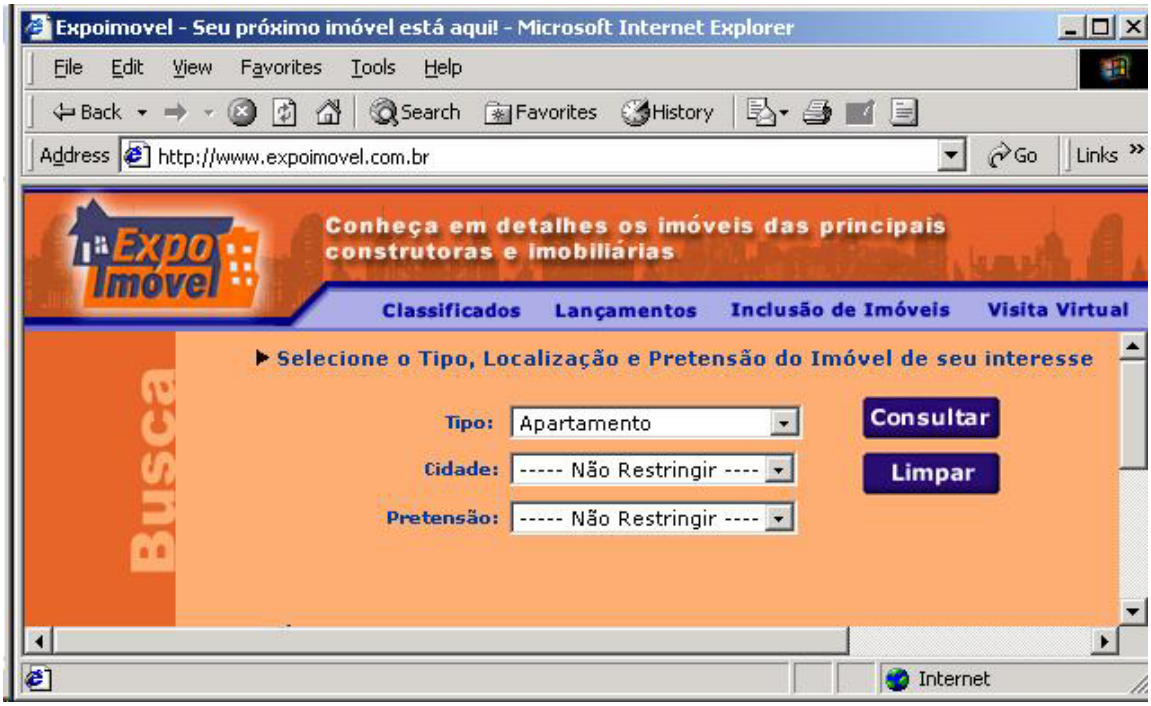

Figura 2.1: Tela inicial do sistema de busca de imóveis.

Após a análise individual de cada ferramenta, foi realizada uma análise comparativa entre as ferramentas escolhidas a partir dos suportes fornecidos pelas mesmas. Pela relevância ao trabalho, foi dada ênfase à análise das linguagens fornecidas pelas ferramentas.

2.2 QARun

Nesta seção apresentamos a QARun, ferramenta utilizada para a automação de teste de software. Rodando apenas sobre o ambiente Microsoft Windows [9], esta ferramenta é capaz de gravar e reproduzir todas as ações do usuário dentro deste ambiente. Embora a QARun possa ser utilizada para testes de software em geral, este trabalho analisa a utilização da mesma apenas para testes de sistemas Web. Os estudos de caso foram feitos então através da utilização de um navegador Web comumente utilizado.

Para a representação dos testes de software, a QARun possui uma poderosa linguagem de scripts. A construção destes scripts se dá através da gravação das ações do usuário no sistema a ser testado ou através de edição manual por usuários da QARun (testadores) mais experientes. O armazenamento dos scripts de teste é feito através de um banco de dados com tabelas criadas pela própria ferramenta.

A QARun pode executar os seguintes tipos de teste:

- De Aceitação
  Testam se o sistema Web possui as funcionalidades especificadas em seus requisitos funcionais.

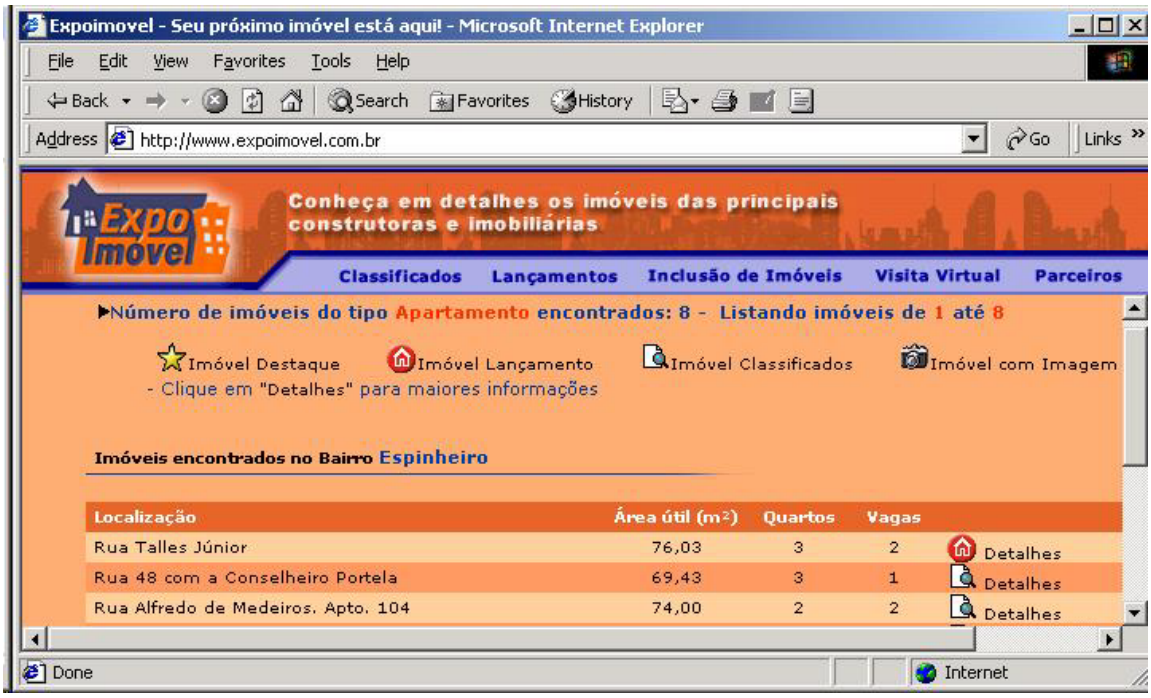

Figura 2.2: Tela de resposta do sistema de busca de imóveis.

- De regressão
  Testam se novas versões da aplicação funcionam da mesma maneira que versões anteriores.

- De performance
  Testam se as aplicações respondem em tempo de acordo com o especificado.

Uma das principais vantagens da ferramenta QARun é a gravação das ações do usuário. Esta com certeza é, na QARun, a forma mais fácil de se criar casos de teste. Com um simples clique pode-se iniciar ou finalizar a gravação das ações do usuário através do teclado ou do mouse. Após ser finalizada a gravação, um script de teste é apresentado contendo as ações realizadas pelo testador durante todo o período de gravação. A realização das gravações pode ser feita em etapas, desde que o cursor e o mouse mantenham-se na mesma posição na junção dos scripts gravados em cada etapa. Isto porque as ações dos usuários são registradas relativamente às posições iniciais do cursor e do mouse.

Ao iniciar a execução de um script, as ações do usuário representadas pelo mesmo serão reproduzidas automaticamente. No final, será apresentada ao usuário uma janela de logs onde existirá um registro para cada ação do usuário reproduzida. Além destes registros automáticos, mensagens personalizadas também podem ser inseridas pelo testador durante a criação dos scripts. A utilização de mensagens por parte do testador é vista mais adiante.

As principais informações apresentadas para cada registro contido na janela de logs são as seguintes: o nome do script, a data e hora de execução, o nome do comando executado e o resultado do teste. O resultado do teste indica se ele falhou ou obteve

sucesso.

Nas próximas seções, apresentamos os principais comandos suportados pela linguagem de script da QARun. Apesar de muitos comandos vistos poderem ser definidos através de interfaces gráficas da QARun, mostramos apenas a forma textual de se utilizar estes comandos, devido a este trabalho estar relacionada a linguagens de descrição de casos de teste de sistemas Web, mais precisamente os de aceitação.

2.2.1 A Estrutura de Um Script de Teste

Ao se criar um novo script na QARun, o seguinte código será gerado:

```
Function Main
    ; Remove the comment below to "Enable" error handling
    ; On Error Call OnErrorHandler
End Function ; Main
Function OnErrorHandler
    ; Insert your Error handling code here.
    Resume Next ; Continue On Error
End Function
```

As linhas iniciadas por "; " são comentários. Na criação de novos scripts são escritas automaticamente duas funções: `Main` e `OnErrorHandler`. A primeira função é a principal do script. Já a segunda é uma função invocada para tratar erros ocorridos durante a execução do script. Sua implementação padrão em caso de erro é continuar a execução do próximo comando, porém para habilitá-la, é necessário que se retire o comentário da segunda linha do corpo da função `Main`.

2.2.2 Utilização de Variáveis

Variáveis declaradas em scripts da QARun podem ser públicas, privadas ou locais. Variáveis públicas podem ser acessadas por todo o script e por scripts filhos, desde que os scripts filhos também as declarem como variáveis públicas. Scripts filhos são scripts invocados em outros scripts através do comando Run (ver Seção 2.2.7). Variáveis privadas são declaradas, assim como as públicas, fora da declaração de funções. Estas variáveis podem ser acessadas por todo o script na qual foi declarada, mas não por scripts filhos. Já as variáveis locais são declaradas e somente utilizadas dentro de funções. Exemplos de declaração de variáveis são vistos a seguir.

```
; Declaracao de variaveis publicas
Public varPublica, varArrayPublica[]
; Declaracao de variaveis privadas
var varPrivada, varArrayPrivada[]
Function Main
    ; Declaracao de variaveis locais
    var varLocal, varArrayLocal[]
End Function
```

Como pode ser visto nos exemplos mostrados, a linguagem da QARun não é uma linguagem tipada e é baseada na linguagem de programação Visual Basic [20], dificultando a programação dos testes.

2.2.3 Comandos Para Simulação do Usuário

Para a reprodução das ações do usuário é necessária a utilização de comandos que simulem suas ações. A seguir, apresentamos alguns dos principais comandos da linguagem de script da QARun utilizados para a simulação das ações do usuário em testes de software Web. Estes comandos manipulam componentes de janela de softwares como, por exemplo, a dos navegadores Web. Estes componentes geralmente são identificados por rótulos (nomes criados pela própria ferramenta ou índices, indicando a ordem de aparição dos mesmos na janela).

Attach. Utilizado para focar a atenção do script em uma determinada janela. No contexto de testes de sistemas Web, este comando é utilizado para focar a janela correta do navegador Web durante a simulação das ações do usuário. O exemplo a seguir, quando executado, coloca em foco a janela de um navegador Web.

```
Attach "ExpoImovel - Seu proximo imovel esta aqui!" +
       " - Microsoft Internet Explorer"
```

A identificação da janela do navegador Web é realizada através da passagem de um parâmetro ao comando `attach`. Este parâmetro é um nome identificador associado pela ferramenta à janela em questão através do ambiente operacional Windows. O operador de concatenação de strings, como podemos ver, é o operador +. Não é permitido testar e extrair informações textuais através do casamento de padrões definidos a partir de expressões regulares, por exemplo, reduzindo assim o poder de execução destes testes. A execução do comando mostrado coloca o foco na janela do navegador Web que apresenta a página do sistema ExpoImóvel.

EditClick. Simula o clique em uma caixa de edição, como por exemplo, TextBox ou ComboBox, dando-lhe o foco da janela. Este comando recebe como parâmetro o rótulo para identificar o componente, opções do uso do mouse (clique com botão esquerdo, clique simples, etc.) e as coordenadas do clique. O rótulo, se numérico, é o índice de aparição da caixa de edição dentro da janela ("∼1"para a primeira caixa, "∼2"para a segunda, etc.). Um exemplo de simulação de clique simples com o botão esquerdo na posição relativa (63, 10) da caixa de URL de um navegador Web pode ser visto logo abaixo.

```
EditClick "~1", 'Left SingleClick', 63, 10
```

ComboText. Este comando simula a digitação de um texto em uma caixa de edição do tipo `ComboBox`. O exemplo a seguir simula a digitação do texto "`http://www.expoimovel.com.br`" no primeiro `ComboBox` (rótulo "∼1") do navegador Web, ou seja, em sua caixa de edição que contém a URL da página a ser visitada.

```
ComboText "~1", "http://www.expoimovel.com.br"
```

ComboBox. Utilizado para selecionar um item de um componente do tipo ComboBox, o comando `ComboBox` retorna 1 se o item foi selecionado com sucesso e 0 caso contrário. Este comando recebe respectivamente o rótulo do ComboBox em questão, o item a ser selecionado e em seguida opções que indicam como o item será selecionado. A execução do comando abaixo, por exemplo, tentará selecionar o item `Recife` do `ComboBox` de nome cidade através de um clique simples no botão esquerdo do mouse.

```
ret = ComboBox "cidade", "Recife", 'Left SingleClick'
```

Button. Simula o pressionamento de um botão da janela. Para utilizar este comando, é necessário passar como parâmetro o rótulo do botão, as opções de como o botão é clicado e opcionalmente a posição (x, y) do clique. Exemplos de uso deste comando podem ser vistos abaixo.

```
Button "~1", "right SingleClick"
Button "&Cancelar", "DoubleClick", 10, 10
```

No primeiro exemplo, é mostrado o uso do comando para simular o clique simples com o botão direito do mouse no botão de índice 1 da janela. A ausência da posição do clique é interpretada como um clique no centro do botão. Já o segundo exemplo mostra a simulação de um clique duplo na posição (10, 10) relativa ao canto esquerdo superior do botão de nome Cancelar.

Pause. Comando utilizado para simular o tempo em que o usuário leva para executar um próximo comando. Quando executado, ele suspende a execução do script por um determinado tempo em segundos. Este comando não precisa ser utilizado para evitar que variação no tempo de resposta da rede faça com que comandos sejam executados antes das páginas serem carregadas, pois a QARun já suspende a execução do script automaticamente até que a página atual seja completamente carregada. O exemplo abaixo mostra a utilização do comando para suspender o script durante 5 segundos.

```
Pause 5
```

ImageSelect. Simula o clique em imagens criadas pela utilização do marcador HTML <IMG>. Para a utilização deste comando, é necessário respectivamente identificar a imagem através do seu rótulo ou pelo nome de seu arquivo, indicar a forma de seleção da imagem e opcionalmente a posição (x, y) onde a imagem será clicada pelo mouse. Se a posição do clique não for indicada, será considerada a realização do clique no centro da imagem. Podemos ver alguns exemplos deste comando logo a seguir.

```
ImageSelect "bot_consultar.gif", 'Left SingleClick'
ImageSelect "Mapa do Brasil", 'Left SingleClick', 100, 35
```

AnchorSelect. Através deste comando é possível simular o clique em *links* (criados com o uso do marcador HTML <A>) sobre textos e imagens, por exemplo. O exemplo abaixo mostra a simulação do clique no botão esquerdo do mouse no *link* que referencia a página "resposta.html".

```
AnchorSelect "resposta.html", 'Left SingleClick'
```

WinClose. Este comando é utilizado para fechar janelas. Pode-se fechar a janela atualmente em foco ou uma outra janela aberta. Para a segunda opção, basta passar como parâmetro o identificador da janela que se deseja fechar. Os dois exemplos abaixo mostram respectivamente o fechamento da janela atual e a de nome identificador "Expoimovel - Seu proximo imovel esta aqui! - Microsoft Internet Explorer MainWindow".

```
WinClose
WinClose "Expoimovel - Seu proximo imovel esta aqui! " +
         " - Microsoft Internet Explorer MainWindow"
```

PopupMenuSelect. Este comando seleciona um item de um pop-up menu. Utilizando em conjunto com o comando `Button`, como no exemplo visto abaixo, pode-se utilizá-lo para abrir o programa navegador Web através do menu iniciar do Windows.

```
Button "Start", 'Left SingleClick'
PopupMenuSelect "Programs~Internet Explorer"
```

2.2.4 Comandos para a Lógica de Programação

A linguagem da QARun, baseada em Visual Basic [20], possui comandos para realização de iterações, testes condicionais, etc. O exemplo abaixo executa a função `WeekDay()` para descobrir o dia da semana e executar a função definida pelo usuário (ver Seção 2.2.8) `testeDomingo`, se o dia da semana for domingo, ou executar a função `testeDiaDiferenteDomingo`, caso contrário.

```
; Invoca uma função de teste dependendo se o dia
; da semana é ou não domingo e retorna o dia da semana
ret = WeekDay( )
; Se hoje e domingo
if ret = 0
    ; realiza teste para os dias de domingo
    testeDomingo()
else
    ; realiza teste para os dias que não são domingo
    testeDiaDiferenteDomingo()
Endif
```

Já o exemplo abaixo realiza um teste para cada descrição de imóvel encontrado na página HTML retornada através das funções `retornaNumImovel()` e `testaImovel(i)` definidas pelo usuário.

```
; Pega o numero de imoveis encontrados na pagina e testa
; os dados de cada um deles
numImovel = retornaNumImovel()
For i = 1 to numImovel
    testaImovel( i )   ; Testa os dados do i-esimo imóvel
Next
```

2.2.5 Comandos para Verificação de Dados

Para garantir a corretude dos sistemas, uma verificação dos dados encontrados com os valores esperados deve ser feita durante toda a execução dos testes. Nesta seção, veremos alguns dos comandos da QARun utilizados para a tarefa de verificação dos dados.

Em caso de falha durante a execução de uma verificação de dados, o script de teste pode ser configurado para continuar ou interromper a execução. Além disto, o script também pode ser configurado para registrar ou não na janela de logs a execução das verificações de dados que são executadas. Nestes registros de log podem ser encontrados os resultados de cada verificação e, para cada caso de falha, o valor esperado e o encontrado.

Check. Através deste comando, pode-se realizar a verificação do conteúdo da interface gráfica do navegador Web, como textos e imagens, além de poder medir o tempo de execução do script de teste.

Para a utilização deste comando, se faz necessário o conhecimento do conceito de mapeamento de checks (*Check Map*), introduzida pela QARun. Este mapeamento é criado internamente pela QARun e sua função é associar nomes a funções de verificações de dados definidas de forma visual através de suas interfaces gráficas. Além do benefício da facilidade visual de definir verificações de textos, imagens e *links*, por exemplo, é possível capturar diretamente da janela a ser testada os dados esperados e reusar os testes em outras páginas Web. Supondo que são criadas as verificações do número de imóveis retornados e do tempo gasto com a execução do teste, podemos executá-las através do seguinte código:

```
Check "NumImovRet"
Check "TempoAtualExecucao"
```

Os exemplos mostrados acima executam respectivamente as verificações de dados associadas aos nomes `NumImovRet` e `TempoAtualExecucao`. A primeira linha invoca a verificação da existência do texto contendo o número de respostas esperado. Já a segunda linha invoca a verificação do tempo gasto até o momento na execução do teste ser menor do que o tempo máximo desejado. Uma desvantagem do mapeamento de checks é o comprometimento da legibilidade. Para o entendimento do script, pode-se fazer necessário o acesso às interfaces gráficas de consulta ao mapeamento da QARun.

UserCheck. Verificações podem ser definidas pelo usuário através do uso do comando UserCheck. Este comando permite ao usuário indicar a ocorrência de uma verificação dos dados. Além de permitir mostrar o resultado da verificação (1 em caso de sucesso, 0 caso contrário), este comando permite mostrar uma mensagem detalhando a verificação. Por exemplo, podemos fazer uso deste comando para verificar o número de imóveis retornados pelo sistema, como mostrado no código a seguir.

```
if numImovel > 0
     ok = 1
else
     ok = 0
UserCheck( "Verificando número de imóveis retornados.", ok,
           "Foram retornados 0 imóveis." )
```

Como podemos notar, o comando `UserCheck` é utilizado para indicar se uma verificação realizada por trechos de código anteriores ao comando foi satisfeita. Já o comando `Check` é utilizado para invocar uma verificação programada visualmente.

2.2.6 Manipulação de Arquivos de Dados

A ferramenta QARun permite o acesso e a manipulação de arquivos a partir de scripts de teste. Esta seção apresenta alguns dos comandos utilizados para este fim.

Open. Comando utilizado para abrir arquivos, permitindo criá-los quando ainda não existem e até bloqueá-los para uso exclusivo. Será retornado o valor `1` caso o comando tenha sucesso na abertura do arquivo e `0` caso contrário. Os exemplos abaixo mostram respectivamente o uso da função para criar ou apagar o conteúdo de um arquivo e para abrir um arquivo já existente,

```
Open( "c:\temp\resultados.dat", "create" )
Open( "c:\dados\teste.dat", "read" )
```

Read. Este comando é utilizado para ler um conjunto de caracteres de um arquivo. Caso o arquivo ainda não tenha sido aberto por um comando `Open()`, este comando o faz automaticamente. O exemplo abaixo lê do arquivo nove caracteres armazenando-os na variável `registroLido`.

```
Read( "c:\dados\nomes.dat", registroLido, 9 )
```

ReadLine. Comando utilizado para ler linhas de arquivos textos, este comando recebe um nome do arquivo e uma variável como parâmetros e retorna `1` caso tenha sucesso na leitura e `0` caso contrário. O conteúdo lido será armazenado na variável recebida como parâmetro. Caso o arquivo ainda não tenha sido aberto por um comando `Open()`, este comando o faz automaticamente. Um exemplo de uso pode ser visto abaixo, onde vai ser lida e armazenada na variável `proximaLinha` a próxima linha do arquivo "c:\dados\configuracao.txt".

```
ReadLine( "c:\dados\configuracao.txt",  proximaLinha)
```

As funções `Write()` e `WriteLine()`, utilizadas para a escrita de arquivos, funcionam similarmente às funções `Read()` e `ReadLine()`.

2.2.7 Invocando Outros Scripts

Scripts de teste em QARun podem invocar a execução de outros scripts através do comando Run. Este comando, quando executado, suspende a execução do script atual enquanto executa o script de nome passado como parâmetro (chamado de script filho). O exemplo abaixo mostra o uso deste comando.

```
Run( "Testa Pagina de Cadastro", "cliente" )
```

O código mostrado invoca a execucao do script de nome "`TestaPaginadeCadastro`", recebendo como parametro o argumento "`cliente`". A utilização do comando Run permite a modularização e composição de scripts de teste, bem como a parametrização dos mesmos.

2.2.8 Funções Definidas Pelo Usuário

O próprio usuário da QARun pode criar e utilizar suas próprias funções dentro de scripts de teste. Funções são definidas como mostrado no exemplo a seguir, onde checamos se um determinado codigo é válido.

```
Function codigoValido( codigo ) : var
    return codigo > 0 And codigo < 10
End Function
```

onde `Function` e `EndFunction` delimitam o corpo da função. A função acima de nome `codigoValido` possui um parâmetro chamado `codigo`. O uso da palavra reservada `var` após a declaração dos parâmetros da função indica que a mesma retorna um valor que, no caso, é um valor booleano indicando se o parâmetro passado possui valor entre `0` e `10`.

2.2.9 Acesso a Banco de Dados

A QARun possibilita ao criador de testes acessar diretamente bancos de dados, desde que sejam banco de dados Microsoft Access ou que possam ser acessados através de um driver ODBC, ambas tecnologias proprietárias da Microsoft Corporation [9]. O exemplo mostrado a seguir, quando executado, se conecta a uma fonte de dados ODBC de nome clientes, e realiza uma consulta dos nomes e do número de clientes contidos na tabela de nome clientes. Estes dados podem a partir daí serem comparados com dados retornados pelo sistema em teste.

```
01: dbConnect( "DSN=clientes" )
02: dbSelect( "SELECT NOME FROM CLIENTE" )
03: dbMoveFirst( )
04: While dbEOF = 0
05:     nomeCliente = dbGetField( "CODIGO" )
06:     ...
07:     dbMoveNext( )
08: EndWhile
09: numCliente = dbRecordCount( )
10: dbDisconnect( )
```

A linha `1` abre a conexão ODBC com a fonte de dados de nome `clientes`. Já as linhas `2` a `8` percorrem os clientes existentes no banco de dados, selecionando o seu código, entre outras coisas. Na linha `9` podemos ver a atribuição do número de registros no banco de dados, retornado pela função `dbRecordCount()`, à variável `numCliente`. Por fim, na linha `10`, fechamos a conexão com o banco de dados através do comando `dbDisconnect()`.

2.2.10 Exemplo de Implementação do Caso de Teste

Abaixo podemos ver um exemplo de um script completo implementado durante o aprendizado da QARun. Este exemplo inicia um navegador Web, acessa o sistema ExpoImóvel, fazendo então uma busca por imóveis de 2 quartos com preço de até R$

70.000,00. Em seguida, é feito o acesso ao primeiro imóvel retornado. Durante a navegação, são feitos algumas verificações como a existência de imagens e do número de imóveis retornados.

```
Function Main
    Attach "PopupWindow~1"
        Button "Start", 'Left SingleClick'
        PopupMenuSelect "Programs~Internet Explorer"
    Attach "about:blank - Microsoft Internet Explorer MainWindow"
        Maximize
        EditClick "~2", 'Left SingleClick', 49, 3
        ComboText "~1", "http://www.expoimovel.com.br"
        TypeToControl "Edit", "~2", "{Return}"
        ComboBox "cidade", "Recife", 'Left SingleClick'
        ComboBox "pretensao", "Comprar", 'Left SingleClick'
    Attach "Expoimovel - Seu proximo imovel esta aqui! "
           + "ChildWindow"
        ImageSelect "bot_continuar.gif", 'Left SingleClick'
    Attach "Expoimovel - Seu proximo imovel esta aqui! -" +
           " Microsoft Internet Explorer MainWindow"
        ComboBox "quarto", "2 Quartos", 'Left SingleClick'
        ComboBox "preco", "Até R$ 70.000", 'Left SingleClick'
    Check "NumImovRet"
    Attach "Expoimovel - Seu proximo imovel esta aqui! ChildWindow"
        ImageSelect "bot_consultar.gif", 'Left SingleClick'
        AnchorSelect "Classificado~13", 'Left SingleClick'
    Check "BtnImprimir"
    Check "BtnMail"
    Attach "Detalhes do Imovel - Microsoft " +
           + "Internet Explorer MainWindow"
        WinClose
    Attach "Expoimovel - Seu proximo imovel esta aqui! -" +
           " Microsoft Internet Explorer MainWindow"
        WinClose
End Function ; Main
```

O script mostrado foi gerado através da gravação das ações do usuário e da criação de *checks* através da GUI da QARun. Podemos notar que o código gerado durante a gravação não faz uso de variáveis nem de funções auxiliares, elementos de reuso de código bastante comuns. O *check* "`NumImovRet`", por exemplo, verifica a existência do texto indicando que pelo menos um imóvel foi encontrado. O número de imóveis retornados não é realmente verificado, pois este valor pode variar com o tempo e *checks* não podem ser parametrizados.

2.3 JXWeb

JXWeb é uma ferramenta utilizada para a realização de testes de caixa preta para sistemas Web e está sendo construída em cima de *frameworks* e ferramentas como o JUnit [15], HttpUnit [18] e a JXUnit [24]. Nesta seção, introduzimos a ferramenta JXWeb através de uma breve descrição do seu ambiente de execução. Em seguida, descrevemos sua linguagem de script através de exemplos simples porém suficientes para um bom entendimento.

JXWeb está ainda sendo definida e implementada (foi analisada uma proposta de extensão da versão beta 0.2) e alguns comandos da linguagem ainda não estão implementados, nem bem definidos. Sendo assim, comandos aqui apresentados podem não ser executáveis, por ainda não estarem disponíveis, podendo também ter sua sintaxe e/ou semântica modificados em uma versáo futura da linguagem.

JXWeb é uma ferramenta baseada em diretórios. Quando executada, a ferramenta procura no diretório corrente, e em seus subdiretórios, por arquivos com scripts de teste (arquivos de nome "`test.jxu`"). Cada script encontrado é então executado sem interferência da execução dos outros. A seguir, introduzimos a linguagem através de um exemplo simples. Por fim, descrevemos os principais comandos da linguagem.

2.3.1 Um Primeiro Script de Teste

Casos de teste em JXWeb são definidos em uma linguagem de script própria, baseada em XML. Por ser baseada em XML [31], a linguagem de script JXWeb torna-se mais fácil de ser aprendida por um grande número de pessoas e manipulada facilmente por máquinas (analisadores de texto). Arquivos XML são compostos por marcadores (*tags*), que por sua vez podem estar aninhados. O marcador raiz destes arquivos em JXWeb é o <jxw>, como visto no exemplo abaixo:

```
<jxw>
    <set name="req" value="http://www.expoimovel.com.br/" />
    <httpGet/>
    <ifEqual name="respText" file="index.html" converse="true">
        <fail>Página inicial diferente da esperada</fail>
    </ifEqual>
</jxw>
```

Comentários na linguagem são iguais a comentários XML, delimitados pelos tags " <!– –" e " – –> ". O exemplo mostrado, quando executado, realiza o *download* da página de URL "`http://www.expoimovel.com.br/`". Em seguida, ele compara o conteúdo da página retornada com uma página previamente salva no disco. Os detalhes sobre os comandos utilizados no exemplo são vistos a seguir.

2.3.2 Comandos Básicos

Nesta seção, apresentamos alguns dos comandos básicos da linguagem JXWeb.

Set. A execução de comandos na linguagem JXWeb pode necessitar de parâmetros, chamados no restante deste documento de parâmetros JXWeb, quando existirem outros tipos de parâmetros no contexto, como por exemplo parâmetros de formulários HTML.

O comando `set` é utilizado para atribuir um determinado valor a uma chave (nome do parâmetro). O valor desta chave pode a partir daí ser utilizado por outros comandos posteriormente utilizados no script. Abaixo pode ser visto um exemplo de seu uso.

```
<set name="req" value="http://www.expoimovel.com.br/" />
<set name="var" value="req" indirect="true" />
```

O primeiro exemplo de uso do comando `set` atribui ao parâmetro req o valor "`http://www.expoimovel.com.br/`". Já o segundo exemplo, quando executado, atribui o valor do parâmetro `req` ao parâmetro `var`, ou seja, atribui o valor "`http://www.expoimovel.com.br/`". O uso do atributo `indirect` com valor `true` faz com que o valor atribuído ao parâmetro não seja o conteúdo do atributo `value`, mas sim o valor atribuído ao parâmetro cujo nome está contido no atributo `value`.

Save. Este comando permite que o valor de um parâmetro seja salvo em um arquivo texto. O exemplo abaixo, quando executado, salva o valor do parâmetro `respText`, que em geral contém a última página Web recuperada pelo comando `HttpGet` (ainda a ser visto), em forma de texto, no arquivo "`h:/temp/resp.html`".

```
<save name="respText" file="h:/temp/resp.html" />
```

Uma grande utilidade deste comando é auxiliar a depuração de scripts de teste. Pode-se programar o script de teste para armazenar a página sendo visitada no momento de ocorrência das falhas, tendo assim o desenvolvedor de testes mais informações sobre situações de erro que podem ser apenas momentâneas e possivelmente difíceis de reproduzir.

Fail. Em certos momentos, é interessante poder expressar uma falha no script de teste. O comando `fail` permite indicar que o teste falhou, interrompendo a execução do mesmo. Ao se indicar uma falha através do comando `fail`, é possível passar uma mensagem descrevendo o motivo da falha, como mostrado no exemplo abaixo.

```
<fail>Código de retorno da página do imóvel diferente de 200</fail>
```

2.3.3 Comandos Para Simulação de Ações do Usuário

Para a utilização da linguagem JXWeb é necessário conhecer seus comandos para simulação de ações do usuário. O objetivo desta seção é apresentar alguns destes comandos através de uma breve descrição e exemplos de seu uso.

Vários comandos mostrados a seguir trocam informações entre si, armazenando objetos, instâncias de classes Java, em parâmetros. Estes objetos podem ser simples textos (objetos do tipo `String`), ou objetos que são instâncias de classes definidas na API HttpUnit [18], base de implementação da ferramenta JXWeb. De forma geral, os comandos utilizam nomes padrões para os parâmetros a serem utilizados, permitindo porém que o testador defina, quando desejado, outros nomes para os mesmos. A API HttpUnit possui um conjunto de classes capazes de representar conexões HTTP, requisições, retornos e componentes de páginas HTML, como *links*, formulários, tabelas, etc.

HttpGet. Outro importante comando da linguagem, HttpGet é utilizado para realizar o *download* de arquivos através de URLs, submissão de formulários e *links* de páginas HTML segundo o protocolo HTTP. Uma vez realizado o *download*, o arquivo fica disponível para manipulação por outros comandos através de parâmetros de saída. Exemplos do uso deste comando podem ser vistos abaixo.

```
<httpGet />
<httpGet request="requisicao" responseText="retorno" />
```

O primeiro exemplo mostrado realiza o *download* de um arquivo através do parâmetro `req`. Este parâmetro pode conter uma string com a URL da página, ou um objeto representando um formulário HTML a ser submetido ou um *link* HTML a ser clicado (objetos do tipo `WebForm` ou `WebLink`, respectivamente). O conteúdo do arquivo retornado é salvo como texto no parâmetro `respText` e como um objeto (do tipo `WebResponse`) que representa um arquivo HTML, no parâmetro `webResponse`.

O segundo exemplo indica o nome dos parâmetros a serem utilizados pelo comando. Ao ser executado, o comando realiza o *download* de uma página através do parâmetro `requisicao`, salvando seu conteúdo em forma de texto no parâmetro de nome `retorno`.

GetForm. Este comando é utilizado para acessar formulários HTML no objeto do tipo `WebResponse`, retornados através de comandos `HttpGet`. Formulários HTML podem ser identificados pelo nome ou por um índice. Estas duas formas de acesso são respectivamente mostradas nos exemplos abaixo.

```
<getForm form="busca" />
<getForm index="0"/>
```

O formulário acessado fica armazenado através de um objeto (do tipo `WebForm`) para uso de comandos posteriores.

GetFormParameter. Usado posteriormente a um comando `getForm`, este comando permite o acesso aos valores de parâmetros de formulários HTML. Além de informar o nome do parâmetro do formulário, como mostrado no primeiro exemplo visto abaixo, é possível informar o nome do parâmetro JXWeb que contém o formulário, como mostrado no segundo exemplo.

```
<getFormParameter name="nome" />
<getFormParameter formName="formCliente" name="nome" />
```

O nome do parâmetro JXWeb que armazenará um objeto (do tipo String) representando o valor do parâmetro do formulário HTML é igual ao nome do parâmetro do formulário em questão.

SetFormParameter. Complementar ao comando `GetFormParameter`, este comando permite atribuir valores aos parâmetros de formulários HTML. Da mesma forma que o comando `GetFormParameter`, além de informar o nome e o valor para o parâmetro do formulário, como mostrado no primeiro exemplo visto abaixo, também é possível informar o nome do próprio formulário, como mostrado no segundo exemplo.

```
<setFormParameter name="nome" value="Joao Francisco" />
<setFormParameter name="nome" value="Joao Francisco"
   form="formCliente" />
```

GetLink. Este comando permite o acesso a *links* contidos em páginas HTML. Para utilizar este comando, é necessário indicar o nome do *link*, através do atributo `linkName`, ou o texto que está sobre atuação do *link*, através do atributo `text`. As duas formas citadas são respectivamente mostradas nos exemplos abaixo.

```
<getLink linkName="cliente" />
<getLink text="Joao Francisco" />
```

Links HTML são representados internamente em JXWeb por objetos do tipo `WebLink`, da API HttpUnit.

GetImageLink. Este comando permite o acesso a *links* contidos em imagens de páginas HTML. Para utilizar este comando, é necessário indicar o nome da imagem (atributo `name` do marcador HTML <img>), através do atributo `imageName`, como mostrado no exemplo abaixo.

```
<getImageLink imageName="cliente" />
```

GetFrameLink. Este comando permite o acesso a *links* contidos em *frames* de páginas HTML. Para utilizar este comando, é necessário indicar o nome do *frame* desejado, através do atributo `frameName`, como mostrado no exemplo abaixo.

```
<getFrameLink frameName="principal" />
```

Na execução do exemplo acima, quando o *frame* existir na página HTML, o comando irá atribuir ao parâmetro `req` um objeto (do tipo `WebLink`) representando o *link* para a página referenciada pelo *frame* de nome `principal`.

GetPageCode. Permite o acesso ao código do retorno de uma página HTML. O exemplo abaixo armazena o código de retorno de uma página retornada pelo último comando `HttpGet`.

```
<getPageCode />
```

O código de retorno encontrado é atribuído ao parâmetro `pageCode`, podendo o mesmo ser acessado por comandos utilizados posteriormente a este.

2.3.4 Objetos de Teste

A ferramenta JXWeb possui incorporada as funcionalidades da ferramenta Quick [25], utilizada para ler e recuperar objetos em arquivos XML. Desta forma, um grande número de comandos da linguagem pode utilizar objetos para sua parametrização. Um exemplo destes comandos é o `TestForm`, mostrado na Seção 2.3.5, que utiliza objetos de teste que representam formulários Web. O código XML a seguir representa um formulário HTML com os campos `code`, `clientCode` e `name`, e botões para submeter ou limpar o formulário.

```
<?xml version="1.0" encoding="UTF-8"?>
<form name="form" method="POST" action="resposta.jsp">
    <input type="hidden" name="code" value=""/>
    <select name="clientCode"/>
    <input type="text" name="name" value="" size="30"/>
    <input type="submit" value="Confirmar "/>
    <input type="button" value="Limpar"/>
</form>
```

Como podemos notar, alguns objetos de teste podem utilizar parte da sintaxe de HTML para facilitar a escrita da representação dos objetos em XML. Comandos que utilizam objetos de teste definem basicamente dois arquivos chamados de

- Arquivos DTD (Document Type Definition)
  Conhecidos pela comunidade que utiliza XML, estes documentos definem a sintaxe dos arquivos que armazenam os objetos de teste.

- Arquivos Schema
  Definem o mapeamento entre os objetos de teste e os arquivos XML, fazendo automaticamente a conversão entre o código de escrito em XML e instanciação e inicialização de um objeto Java.

Como o desenvolvedor de testes só manipula diretamente os arquivos que armazenam os objetos de teste, é necessário somente o conhecimento dos arquivos DTD.

2.3.5 Comandos de Verificação Dos Dados

Nesta seção apresentamos alguns dos comandos JXWeb utilizados para verificar os valores dos dados de testes esperados com os dados de testes encontrados.

IfEqual. Este comando é utilizado para comparar valores de parâmetros JXWeb. Os valores dos parâmetros podem ser comparados com valores de outros parâmetros, com textos ou com o conteúdo de arquivos. Caso a condição do teste seja verdadeira, comandos aninhados serão executados antes da execução dos próximos comandos do script. Os exemplos mostrados abaixo comparam respectivamente se o valor do parâmetro `var` é igual a `10` e se o valor do parâmetro `var1` é igual ao valor do parâmetro `var2` (uso do atributo `indirect` com valor `true`).

```
<ifEqual name="var" value="10">
    <!-- Comandos executados se condição verdadeira-->
</ifEqual>
<ifEqual name="var1" value="var2" indirect="true">
   <!-- Comandos executados se condição verdadeira-->
</ifEqual>
```

Já os exemplos abaixo testam respectivamente se o conteúdo do parâmetro `respText` (contendo o texto da página retornada pelo comando `HttpGet`, por exemplo) é igual ao conteúdo do arquivo "`index.html`" e se o valor do parâmetro `pageCode` é diferente (uso do atributo `converse` com valor `true`) de `200`.

```
<ifEqual name="respText" file="index.html">
    <!-- Comandos executados se condição verdadeira-->
</ifEqual>
<ifEqual name="pageCode" value="200" converse="true">
    <!-- Comandos executados se condição verdadeira-->
</ifEqual>
```

IsEqual. Verifica se um dado parâmetro possui um determinado valor, interrompendo a execução do teste em caso de falha. O valor comparado com o de uma propriedade pode ser definido no próprio comando. Pode ser também o conteúdo de um arquivo ou um objeto armazenado em um arquivo (ver Seção 2.3.4). Os exemplos abaixo mostram respectivamente o teste do valor do parâmetro `nomeCliente` nas três formas possíveis.

```
<isEqual name="nomeCliente" value="Joao Bosco" />
<isEqual name="nomeCliente"
         file="c:/dadosTeste/nomeCliente.txt" />
<isEqual name="nomeCliente"
         file="c:/dadosTeste/nomeCliente.obj"
         schema="c:/dadosTeste/nomeCliente.qiml" />
```

Na primeira forma, compara-se o valor do parâmetro com a String "`JoaoBosco`". Na segunda forma mostrada, compara-se o valor do parâmetro com o valor de um arquivo texto. Por fim, na terceira forma mostrada, comparamos o valor do parâmetro com um objeto serializado em um determinado arquivo através de um determinado esquema de mapeamento.

TestText. Utilizado para verificar se um determinado texto se encontra em uma página HTML, este comando pode procurar por um texto indicado através do seu atributo `value` ou pelo texto contido em um arquivo indicado através do seu atributo `file`. Estas duas formas citadas são vistas respectivamente nos exemplos abaixo.

```
<testText value="ExpoImóvel" />
<testText file="c:/dadosTeste/textoInicial.txt" />
```

O primeiro uso do comando verifica a existência do texto "ExpoImóvel". O segundo verifica a existência do texto contido no arquivo determinado.

TestForm. Este outro comando é utilizado para testar a existência de um determinado formulário em uma página Web. Os dados do formulário esperado são obtidos a partir de um arquivo de dados XML indicado pelo atributo `objectFile` do comando. A seguir podemos ver um exemplo da utilização deste comando para verificar na página Web contida no parâmetro `webResponse` a existência do formulário representado pelo objeto de teste contido no arquivo "`formularioInicial.obj`".

```
<testForm objectFile="c:/dadosTeste/formularioInicial.obj" />
```

2.3.6 Manipulação de Erros e Falhas na Execução de Scripts de Teste

Além da possibilidade natural dos testes falharem, alguns comandos utilizados ainda partem de pressupostos que, quando não satisfeitos, podem gerar exceções em tempo de execução. O comando `getForm`, por exemplo, pressupõe a existência de um objeto (do tipo `WebResponse`) no parâmetro `webResponse`, o qual representa uma página Web que contém um formulário HTML de nome igual ao indicado no uso do comando.

Falhas em testes e suposições por comandos não satisfeitas são sinalizadas por exceções Java. As exceções são levantadas de acordo com a implementação de cada comando. Estas exceções interrompem a execução do script em execução, mas não impedem a execução de scripts contidos em outros diretórios. No final da execução dos scripts de teste, a ferramenta indicará o número de scripts que geraram exceções.

2.3.7 Estendendo JXWeb

Novos comandos podem ser introduzidos na linguagem de JXWeb. Para isto, deve-se ter um conhecimento aprofundado sobre como JXWeb foi implementado. Nesta seção, apresentamos o conhecimento mínimo de como JXWeb foi implementado, porém suficiente para entender o procedimento para estender sua linguagem.

1 - Implementando um Novo Comando. A implementação de um comando da linguagem JXWeb se inicia através da criação de uma classe em Java. Esta classe deve implementar a execução do novo comando seguindo um determinado padrão de implementação, cujos detalhes não são discutidos aqui por estarem fora do escopo deste trabalho.

2 - Executando Um Passo de Teste. Após a implementação da classe em Java, representando o novo comando em JXWeb, já podemos utilizá-la em um script de teste através do comando eval descrito a seguir.

Eval. Utilizado para instanciar e executar classes que representem comandos JXWeb, o comando `eval` recebe como parâmetro o nome da classe em questão (através do atributo `stepClass`). Estas classes devem implementar o método `eval` de acordo com a interface `JXTestStep`. O exemplo abaixo mostra a execução da classe `br.ufpe.cin.ehsa.teste.TestNotNull`.

```
<eval stepClass="br.ufpe.cin.ehsa.teste.TestNotNull" />
```

Durante a avaliação do comando `eval`, uma instância da classe `br.ufpe.cin.ehsa.teste.TestNotNull` será criada e seu método `eval` será invocado, recebendo como parâmetro os parâmetros JXWeb do script.

3 - Incorporando Um Passo de Teste À Linguagem. Por fim, é necessário que o novo comando seja incorporado à sintaxe da linguagem JXWeb. Para isto, é necessário configurar alguns arquivos utilizados pela ferramenta. Estes arquivos indicam o mapeamento entre comandos da linguagem e classes Java. Para a extensão efetiva da ferramenta, se faz necessário um maior estudo de sua documentação [26].

2.3.8 Exemplo de Implementação do Caso de Teste

Podemos ver logo abaixo um exemplo simplificado de um dos scripts de teste que foram construídos durante a análise da ferramenta JXWeb.

```
<jxw>
    <set name="req" value="http://www.expoimovel.com.br/"/>
    <httpGet/>
    <save name="respText" file="h:/temp/resp.html" />
    <getFrameLink frameName="superior" />
    <httpGet/>
    <save name="respText" file="h:/temp/resp.html" />
    <getImageLink imageName="classificados" />
    <httpGet/>
    <save name="respText" file="h:/temp/resp.html" />
    <testForm objectFile="c:/dadosTeste/formularioInicial.obj" />
    <testImage objectFile="c:/dadosTeste/imagemContinuar.obj" />
    <testImage objectFile="c:/dadosTeste/imagemLimpar.obj" />
    <getImageLink imageName="continuar" />
    <httpGet/>
    <save name="respText" file="h:/temp/resp.html" />
    <testForm objectFile="c:/dadosTeste/formBuscaClassif2.xml" />
    <testImage objectFile="c:/dadosTeste/imagemContinuar.obj" />
    <testImage objectFile="c:/dadosTeste/imagemLimpar.obj" />
    <testImage objectFile="c:/dadosTeste/imagemVoltar.obj" />

</jxw>
```

O script mostrado acessa o *frame* de nome "`superior`", clica na imagem "`classificados`" para acessar a página de busca. Em seguida, testa se existe o formulario de busca, as imagens utilizadas para submeter e limpar o formulário. Após isto, o script submete o formulário clicando na imagem continuar, testando o conteúdo da página de resposta.

2.4 Análise das Ferramentas

Durante as seções anteriores, apresentamos os resultados da análise individual de cada uma das ferramentas escolhidas. Já nesta seção, mostramos uma comparação entre os resultados obtidos com as análises individuais de cada ferramenta, baseado nos suportes fornecidos por cada uma.

2.4.1 Suportes Fornecidos Pela Ferramenta QARun

Apresentamos nesta seção a análise da ferramenta QARun baseada nos possíveis suportes fornecidos e observados neste trabalho.

S1. Verificação da Sintaxe de Páginas HTML. Este suporte não é fornecido pela QARun. Isto porque a QARun não acessa o código HTML de páginas Web, mas sim sua apresentação realizada através da janela de navegadores Web.

S2. Teste de Layout dos Componentes de Páginas HTML. A distribuição espacial dos componentes de páginas HTML pode ser verificada através da QARun. Através da QARun, podemos comparar toda ou partes das imagens esperadas e encontradas em uma página Web apresentada pelo navegador Web. Apesar de suas limitações, principalmente quanto à manutenção do código devido ao seu baixo nível de abstração e ao fato do mapeamento de checks (*Check Map*) estar embutido na própria ferramenta, a verificação da corretude das páginas Web através da comparação das imagens gravada e atualmente apresentada pelo navegador mostrou-se eficaz.

S3.1. Reprodução Através de Linguagem de Script. A ferramenta QARun trabalha com uma linguagem de scripts baseada em Visual Basic (VB) [20]. Apesar de ter o grande poder computacional de VB, a linguagem perde em poder de abstração e alguns comandos só podem ser compreendidos com o auxílio da ferramenta visual, como o caso do comando `Check`. Além disto, a linguagem da QARun não pode ser estendida pelos seus usuários.

S3.2. Reprodução Através de Gravações. Uma das maiores facilidades da QARun é a gravação das ações dos usuários do sistema a ser testado. Outra vantagem é que, por gravar as ações através dos navegadores Web, o suporte a *Applets*, Flash, e outros componentes avançados depende somente do suporte do próprio navegador. Entretanto, o código gerado através da gravação mostrou-se de difícil compreensão.

S4. Simulação de Navegadores Internet. A ferramenta QARun permite utilizar os próprios navegadores Web nos testes. Para testar com vários navegadores, ou grava-se o mesmo teste para todos os diferentes tipos de navegadores, ou tenta-se a difícil tarefa de parametrizar os testes para executarem em diferentes navegadores, já que os mesmos podem apresentar de forma diferente um mesmo código HTML.

S5. Verificação do Tempo de Resposta. A QARun permite que o criador dos testes verifique em qualquer ponto do script, o tempo decorrido desde o seu início. Para verificar o tempo de resposta de uma página, por exemplo, torna-se necessário medir o tempo imediatamente antes e depois de se carregar a página, e então obter a diferença entre os dois.

S6. Acesso a Banco de Dados. Através da QARun, pode-se acessar diretamente bancos de dados, com a única restrição deles terem acesso através de um driver ODBC.

S7. Verificação do Controle de acesso. A QARun não tem acesso aos *cookies* utilizados pelo navegador nem tem como saber detalhes sobre a segurança das conexões, como o uso de HTTPS. Uma forma limitada de contornar este problema é verificar a existência da imagem do cadeado fechado utilizado pelos navegadores para indicar o uso de conexões seguras.

S8. Simulação de Usuários Concorrentes. A ferramenta QARun não realiza a execução de scripts concorrentemente, o que simularia o acesso concorrente de usuários no sistema. Uma ferramenta complementar chamada QALoad pode executar tais testes, porém a mesma não foi analisada neste trabalho.

S9. Escalonamento da Execução de Testes. Para executar scripts de teste na QARun, é necessário iniciar a ferramenta QARun, escolher o script e mandar executá-

lo. Ou seja, não há como automatizar a execução dos scripts de testes, o que dificulta a realização de testes de regressão.

S10. Parametrização dos Dados de Teste. Como a ferramenta QARun pode acessar o conteúdo de arquivos textos e de banco de dados, o desenvolvedor de testes pode utilizar os dados provindos destas fontes para parametrizar os seus scripts de teste. Entretanto, o código de acesso a banco de dados e arquivos textos através de linguagens de programação leva a um código mais sujo e difícil de se entender. A definição de comandos mais simples e abstratos que os das linguagens de programação seria a forma mais adequada para este tipo de código.

S11. Portabilidade dos Casos de Testes. Os scripts de testes, assim como a execução da QARun, só pode ser realizada no ambiente Windows. Não existe nenhuma restrição para o ambiente servidor que hospeda o sistema. Porém, o mesmo não acontece aos bancos de dados acessados pelos scripts, restritos a acessos através de drivers ODBC, e aos navegadores Web, que tem de ser executados na mesma máquina que contém a QARun.

S12. Monitoramento do Sistema. A utilização da QARun como ferramenta para monitoramento de sistemas Web pode ser realizada, desde que tenha a presença de um operador para disparar periodicamente os testes. Alarmes podem então ser acionados pelo próprio operador da ferramenta ou pela execução automática via QARun de uma aplicação externa.

S13. Depuração de Testes. Para depurar a execução de testes na QARun, pode-se tirar proveito da janela de logs, a qual registra a execução de cada comando e de mensagens personalizadas. Além disto, é possível realizar a execução de scripts passo a passo.

2.4.2 Suportes Fornecidos Pela Ferramenta JXWeb

Apresentamos nesta seção a análise da ferramenta JXWeb baseada nos possíveis suportes fornecidos definidos neste trabalho.

S1. Verificação da Sintaxe de Páginas HTML. JXWeb utiliza uma API para realizar a leitura dos arquivos HTML. A sintaxe destes arquivos é verificada pela API, entretanto ela não precisa estar em exata conformidade com a sintaxe do HTML padrão. Esta flexibilidade da ferramenta pode impedir que os testes detectem pequenos problemas de sintaxe.

S2. Teste de Layout dos Componentes de Páginas HTML. A realização de testes sobre a distribuição espacial dos componentes de páginas HTML ainda não é suportada por JXWeb. Podemos, entretanto, estender JXWeb com novos comandos para verificar os atributos que influenciam a apresentação dos componentes como, por exemplo, o tamanho relativo e alinhamento a outros componentes. Estes comandos possuiriam utilidade limitada, pois somente analisando as imagens apresentadas pelos navegadores Web é que podemos realmente assegurar a correta diagramação dos componentes.

S3.1. Reprodução Através de Linguagem de Script. A JXWeb possui uma

linguagem de script baseada em XML que simula as ações realizadas por usuários do sistema. Com esta linguagem, scripts de testes podem ser facilmente manipulados (alterados, compostos por outros testes, etc.). Esta linguagem possui um bom nível de abstração, podendo a mesma ser estendida pelo desenvolvedores de testes. Passos de testes também podem ser escritos em classes Java, ganhando assim todo o poder computacional de uma linguagem de programação orientada a objetos.

S3.2. Reprodução Através de Gravações. A implementação atual de JXWeb não permite realizar gravações para posteriores reproduções das ações dos usuários do sistema.

S4. Simulação de Navegadores Internet. JXWeb utiliza a API HttpUnit para a navegação dentro da Web. Esta API não representa o comportamento real que os navegadores Web possam ter. Desta forma, não existe a possibilidade de verificar o correto funcionamento do sistema executando-se os testes nas diferentes implementações de navegadores Web.

S5. Verificação do Tempo de Resposta. Em JXWeb é possível verificar o tempo de execução de um comando ou de um bloco de comandos, assim como de todo o script de teste.

S6. Acesso a Banco de Dados. A verificação direta de dados armazenados em banco de dados pode ser realizada em JXWeb somente através da implementação de passos de teste como código Java. Através de Java, pode-se acessar quaisquer dados em bancos de dados conectados por JDBC [39].

O uso de código Java como passos de teste dificulta o entendimento do script e torna necessário que o desenvolvedor manipule duas ferramentas, a JXWeb e um ambiente de desenvolvimento Java.

S7. Verificação do Controle de acesso. A JXWeb permite que o desenvolvedor de testes acesse os detalhes sobre as conexões, como *cookies* armazenados, utilização do protocolos HTTPS, etc. Entretanto, é necessário a criação de passos de testes como classes Java (com as desvantagens já citadas) e o conhecimento sobre a API HttpUnit, base de implementação de JXWeb.

S8. Simulação de Usuários Concorrentes. A linguagem de JXWeb permite que blocos de comandos ou scripts de teste sejam executados simultaneamente.

S9. Escalonamento da Execução de Testes. Pode-se programar a execução de JXWeb, automatizando assim a sua execução. Entretanto, o resultado da execução dos testes (sucesso ou falha) tem que ser extraída a partir do texto escrito na saída padrão de execução da ferramenta.

S10. Parametrização dos Dados de Teste. Por ser integrado à ferramenta Quick, utilizada para ler e recuperar objetos em arquivos XML, JXWeb permite a utilização de objetos de teste. Estes objetos ficam armazenados em arquivos XML que podem ser facilmente criados e editados. Além dos objetos de teste, arquivos textos também são utilizados para parametrizar os comandos. Para utilizar dados armazenados em banco de dados, é necessário criar classes Java representando passos de teste.

S11. Portabilidade dos Casos de Testes. Os scripts de teste podem ser executados em qualquer ambiente que forneça uma máquina virtual Java. Quaisquer bancos de dados que forneçam drivers JDBC também podem ser acessados.

S12. Monitoramento do Sistema. Scripts JXWeb podem ser utilizados para monitorar sistemas, devido a sua possibilidade de automação e a capacidade de executar ações como, por exemplo, alarmes, envio de e-mails e re-inicialização do sistema, desde que possa ser implementado através de classes Java. Não existe em JXWeb facilidades já implementadas para o monitoramento do sistema.

S13. Depuração de Testes. A execução passo a passo de scripts não é suportada pela JXWeb. Entretanto, comandos podem ser utilizados para registrar mensagens em arquivos de log.

2.4.3 Comparação entre as Ferramentas

Nesta seção, apresentamos uma tabela comparativa entre as análises individuais das ferramentas escolhidas (Tabela 2.2). Esta tabela é baseada nos resultados obtidos com a verificação do nível de cada um dos suportes fornecidos pelas ferramentas. A indicação do nível de suporte oferecido é apresentada por um dos símbolos vistos na Tabela 2.3. A ausência de símbolo indica o não fornecimento do suporte.

Símbolo	Descrição do Nível do Suporte
++	Muito bem fornecido
+	Bem fornecido
−	Fracamente fornecido
−−	Muito fracamente fornecido

Tabela 2.2: Símbolos utilizados para indicar o nível de suporte fornecido.

2.5 Conclusões

Pudemos observar que ferramentas de teste de sistemas Web que possuem linguagens de script para descrição dos casos de teste podem incorporar diretamente o poder computacional de linguagens de programação, como acontece com a linguagem da ferramenta QARun. Nestes casos, as linguagens de script têm seu nível de abstração prejudicado. Nos casos contrários, a linguagem pode manter um alto nível de abstração dos seus comandos. Entretanto, sem o poder computacional encontrado em linguagens de programação como, por exemplo, acesso a arquivos e a banco de dados e comandos para fluxo condicional de dados, as linguagens de script tornam-se bastantes limitadas.

A criação de casos de teste através de gravação mostrou ser muito mais simples do que a programação dos testes. Entretanto, seu uso não dispensa a existência da linguagem de teste, pois o testador é obrigado em certos momentos a programar diretamente no código. Exemplos disto são a definição de funções de teste, parametrização dos dados e manutenção dos casos de testes. Como trechos de código gerados pela gravação podem

Suporte Fornecido	**QARun**	**JXWeb**
S1. Verificação da Sintaxe de Páginas HTML		−
S2. Teste de Layout dos Componentes de Páginas HTML	+	
S3.1. Reprodução Através de Linguagem de Script	+	+
S3.2. Reprodução Através de Gravações	++	
S4. Simulação de Navegadores Internet	+	
S5. Verificação do Tempo de Resposta	++	++
S6. Acesso a Banco de Dados	++	+
S7. Verificação do Controle de acesso	−−	+
S8. Simulação de Usuários Concorrentes		++
S9. Escalonamento da Execução de Testes		−
S10. Parametrização dos Dados de Teste	+	+
S11. Portabilidade dos Casos de Testes	−−	++
S12. Monitoramento do Sistema	−−	+
S13. Depuração de Testes	++	+

Tabela 2.3: Comparação dos níveis de suporte fornecidos pelas ferramentas.

ser mais complicados do que se fossem gerados manualmente, é importante nestes casos a utilização de linguagens de teste com alto nível de abstração e legibilidade.

Outra característica que se mostrou interessante foi a possibilidade de estender a linguagem, adaptando-a as necessidades de cada ambiente como, por exemplo, a criação de rotinas para teste específicas para o sistema sendo testado.

Durante a escolha das ferramentas a serem analisadas por este trabalho, notou-se uma grande quantidade de ferramentas disponíveis. Estas ferramentas variam bastante quanto aos suportes oferecidos. Os critérios de análise aqui apresentadas servem de base para uma análise das vantagens e desvantagens de possíveis ferramentas a serem utilizadas em empresas desenvolvedoras de sistemas Web, devendo as mesmas realizar sua escolha baseada nas métricas consideradas mais relevantes em seus ambientes de trabalho.

Por fim, para a aquisição de ferramentas de teste de sistemas Web também é interessante que a avaliação seja feita através de exemplos de uso nos sistemas que realmente serão testados. Isto porque durante a execução dos exemplos definidos neste trabalho, foram descobertas limitações que não são documentadas nas ferramentas.

Capítulo 3

A Linguagem WSat

Este capítulo apresenta a linguagem WSat definida neste trabalho. Utilizada para a descrição de casos de teste de aceitação de sistemas Web, WSat foi definida visando possuir determinadas características, como alto nível de abstração e reuso.

A atividade de teste, como já dito anteriormente, é em geral bastante entediante e repetitiva. Este fato é válido para a realização de testes de aceitação de sistemas Web, sendo ainda agravado quando a linguagem utilizada para programar os casos de teste não é adequada (como no uso de uma linguagem de programação convencional, por exemplo).

Existem diversas características em uma linguagem de teste que podem facilitar a programação dos casos de teste de aceitação. Entre elas, estão

- Legibilidade

  A atividade de teste é uma atividade contínua que acompanha não só o desenvolvimento dos sistemas, mas também sua manutenção. Uma boa legibilidade do código fornece uma maior facilidade de criação, extensão e modificação dos programas de teste.

- Abstração

  Uma linguagem que possua tipos que representem os conceitos envolvidos nos testes de aceitação, tais como os de página Web, imagens, *links*, etc., permite ao testador programar os testes em um nível maior de abstração e consequentemente obter uma maior facilidade de programação.

- Reuso

  Como a atividade de teste é uma atividade bastante repetitiva, artifícios para reuso de código como a definição e composição de funções de teste se tornam fundamentais para o sucesso da linguagem.

- Parametrização

  A parametrização dos casos de teste permite ao testador realizar diversos testes no sistema a partir da programação de um único caso de teste implementado, variando-se apenas seus parâmetros de entrada e saída.

Além destas características, para que o objetivo deste trabalho seja alcançado, é necessário que a linguagem de teste utilizada represente bem os aspectos estruturais da GUI (*Graphical User Interface*) do sistema testado, fornecendo assim as informações necessárias para que trechos de código do sistema possam ser gerados automaticamente.

No Capítulo 2, apresentamos a análise de duas ferramentas de teste disponíveis no mercado. Podemos notar que as linguagens disponíveis nestas ferramentas para a programação dos testes de aceitação possuem características importantes, como reuso e parametrização, porém não possuem bons níveis de legibilidade e abstração. Além disto, estas linguagens não representam explicitamente os aspectos estruturais da GUI do sistema testado, inviabilizando desta maneira uma possível geração do código deste sistema. Estes mesmos problemas foram encontrados em algumas linguagens de teste acadêmicas brevemente estudadas, como por exemplo, a linguagem TestTalk [28].

Para viabilizar a geração de código de sistemas Web proposta por este trabalho, faz-se necessário a criação de uma nova linguagem para descrição de casos de teste de aceitação em sistemas Web. Chamada de WSat (*Web System Acceptance Test Language*), esta linguagem foi definida neste trabalho visando possuir um alto nível de abstração e reuso, além de expressar explicitamente aspectos relacionados à estrutura da GUI dos

sistemas testados. Também foi objetivo de WSat fornecer aos testadores os suportes de portabilidade e parametrização dos casos de teste, verificação do tempo de resposta e acesso a banco de dados, suportes estes apresentados na Seção 2.1.4.

O objetivo deste capítulo é apresentar os detalhes desta linguagem. Para o bom entendimento do mesmo, faz-se necessário o conhecimento mínimo sobre HTML [37] e a linguagem de programação Java [19].

3.1 Introduzindo WSat

As principais características que guiaram a definição de WSat foram a abstração, o reuso e a capacidade de expressar explicitamente a estrutura dos sistemas testados. Visando obter abstração, WSat permite a definição de tipos que representam os tipos de entidades a serem testadas. Estas entidades são componentes Web como, por exemplo, páginas Web, formulários HTML, imagens e *links*.

Durante esta seção, apresentamos incrementalmente um primeiro exemplo de programa WSat. Este programa tem o objetivo de transmitir a idéia geral de como são escritos programas WSat, sendo mostrados exemplos mais complexos no decorrer deste capítulo. O programa mostrado aqui verifica se a página de entrada de um determinado sistema Web contém o texto "`BemVindo!`". Para escrever este teste em WSat, nós podemos inicialmente descrever a estrutura da GUI a ser testada através da definição do tipo `PaginaBemVindo` como visto abaixo.

```
WebPage PaginaBemVindo {
}
```

Como podemos ver, tipos definidos pelo testador (programador de testes) são definidos a partir de tipos especiais predefinidos na linguagem. O tipo `PaginaBemVindo`, por exemplo, é definido a partir do tipo `WebPage` que denota o conjunto de todas as possíveis páginas Web. Tipos WSat denotam conjuntos de componentes Web que satisfazem determinadas restrições. Restrições sobre o conjunto denotado podem ser criadas, por exemplo, através da definição de propriedades. Podemos então adicionar ao tipo `PaginaBemVindo` a propriedade `textoBemVindo` como visto no código abaixo.

```
WebPage PaginaBemVindo {
    TextoBemVindo textoBemVindo;
}
Text TextoBemVindo {
    value = "Bem Vindo!";
}
```

A propriedade `textoBemVindo` indica que as páginas Web denotadas pelo tipo `PaginaBemVindo` possuem ao menos um componente pertencente ao conjunto de elementos denotado pelo tipo `TextoBemVindo`. Este tipo é subtipo de `Text`, o qual denota todos os possíveis componentes texto na Web. A propriedade `value`, herdada do tipo `Text`, indica o valor texto representado pelos componentes denotados. Desta forma, o tipo `TextoBemVindo` denota o conjunto de componentes texto que representam o texto "`BemVindo!`" em páginas Web. Podemos interpretar a definição do tipo `PaginaBemVindo`

como a representação de todas as páginas Web que contêm, entre outras coisas, o texto "BemVindo!".

Propriedades em tipos WSat podem ser herdadas ou definidas no próprio tipo. As propriedades herdadas indicam as características do componente Web representado. Já a definição de uma nova propriedade indica a existência de um outro componente dentro do componente Web representado. A definição de tipos permite descrever a estrutura da GUI que o sistema testado deve ter. Com base nesta descrição, pode-se verificar se o sistema satisfaz esta estrutura. Estes tipos podem também ser reusados para descrever estruturas mais complexas.

Após descrever a estrutura da GUI do sistema testado, através da definição de tipos, faz-se necessário descrever o seu comportamento. Isto é feito em WSat com o uso de funções e casos de teste. Abaixo podemos ver a declaração do casos de teste `testaPaginaBemVindo`.

```
testCase testaPaginaBemVindo {
    PaginaBemVindo pag =
        [PaginaBemVindo] getWebPage("http://www.bemvindo.com.br");
}
```

Casos de teste representam em WSat o ponto de partida para a execução dos testes. Eles não recebem parâmetros nem possuem retornos. Já as funções de teste, vistas adiante em exemplos mais complexos, além de poderem possuir parâmetros e retornar resultados, possibilitam uma melhor estruturação e reuso de código nos casos de teste. O caso de teste `testaPaginaBemVindo`, por exemplo, requisita a página Web de URL "`http://www.bemvindo.com.br`" (através do comando `getWebPage`) e testa se a página retornada possui as características descritas pelo tipo `PaginaBemVindo` (operação de transformação de tipo). Em caso negativo do teste, a operação de transformação de tipo interrompe a execução do caso de teste indicando a falha. Por fim, atribui um objeto do tipo `PaginaBemVindo`, página Web requisitada, à variável `pag`. Os detalhes sobre as operações citadas são vistos no decorrer deste capítulo.

Como podemos perceber no exemplo mostrado, os tipos definidos em WSat representam a estrutura da GUI dos sistemas testados. Estes tipos contêm, como mostrado no Capítulo 4, as informações necessárias para a geração de código de sistema objetivada por este trabalho. Já a definição de tipos e funções de teste nos permite obter o reuso de código desejado nos programas WSat.

No projeto de WSat, procurou-se obter uma similaridade com a linguagem de programação Java. Isto por Java ser uma linguagem largamente conhecida e utilizada para o desenvolvimento de sistemas Web, além de ser também a linguagem destino do gerador de código de sistema definido neste trabalho.

3.2 Detalhando WSat

Nesta seção, apresentamos em detalhes as estruturas e comandos utilizados na construção de programas WSat. Em primeiro lugar, apresentamos um pequeno sistema Web, o qual é utilizado nos exemplos de programas de teste mostrados. Em seguida, a linguagem WSat é apresentada incrementalmente através de exemplos para testar o

sistema Web. Para apresentar WSat, primeiro mostramos como realizar a descrição estrutural para a GUI do sistema testado. Após isto, mostramos como realizar a descrição comportamental desejada para o sistema.

3.2.1 Sistema de Busca na Web

Para a demonstração da linguagem WSat, faremos uso de um sistema de busca fictício, apresentado nesta seção. Quando acessado a partir de um navegador Web, este sistema de busca tem inicialmente a aparência vista na Figura 3.1.

Figura 3.1: Tela inicial do sistema de busca testado.

Como iremos perceber no decorrer deste capítulo, para a realização de testes de aceitação em sistemas Web se faz necessário o acesso ao código fonte de suas páginas. Isto porque é no código fonte que se encontram os detalhes sobre os componentes Web a serem testados. O mesmo pode ocorrer nos testes para sistemas não Web, onde os programas de teste podem necessitar interagir diretamente com o sistema, necessitando assim conhecer a sua interface de comunicação. Sendo assim, parte do código HTML da página inicial do sistema é mostrado abaixo.

```
<html>
    <head>
        <title>Sistema de Busca</title>
    </head>
    <body>
```

```
        <img name="logo" src="imagens/logo.jpg" align=right>
        <p><b>Buscar por:</b></p>
        <form name="formBusca" method="post"
                              action="pagResposta.jsp">
            <table width="338" border="1"> ...
                <input type="text" name="palavras" size=50> ...
                <select name="onde">
                    <option value="1">Brasil</option>
                    <option value="2">Mundo</option>
                </select> ...
                <input type="submit"
                       name="submeter" value="Submit"> ...
            </table>
        </form>
    </body>
</html>
```

Através do código HTML mostrado, podemos obter as informações sobre os componentes Web contidos na página inicial do sistema. Podemos notar, por exemplo, a presença dos marcadores HTML <`img`> e <`form`>, representando respectivamente uma imagem e um formulário HTML. Este formulário, por sua vez, possue os parâmetros `palavras` e `onde`, representados respectivamente pelos marcadores <`input`> e <`select`>. A partir desta página, o usuário pode digitar as palavras chave da busca e escolher o local onde o sistema de busca deve procurar. Em seguida ao clique do botão `submeter`, é retornada uma página de resposta ao navegador Web. A página de resposta retornada quando a palavra chave "`ufpe`", por exemplo, é digitada pode ser vista na Figura 3.2. O código HTML da página de resposta vista na Figura 3.2 é mostrado logo abaixo.

```
<html>
    <head>
        <title>Sistema de Busca</title>
    </head>
    <body>
        <p><b>Sistema de Busca</b></p>
        <p>Foram encontradas 3 respostas.</p>
        <p>1. <a name="resp1"
                 href="http://www.cin.ufpe.br">Cin-UFPE</a></p>
        <p>2. <a name="resp2" href="http://www.ufpe.br">UFPE</a></p>
        <p>3. <a name="resp3"
                 href="http://www.pernambuco.gov.br">Pernambuco</a>
        </p>
    </body>
</html>
```

3.2.2 Descrevendo a Estrutura da GUI do Sistema

Nesta seção, apresentamos como descrever a estrutura da GUI do sistema em programas WSat. Aqui mostramos conjunto de tipos existentes em WSat. A partir destes, mos-

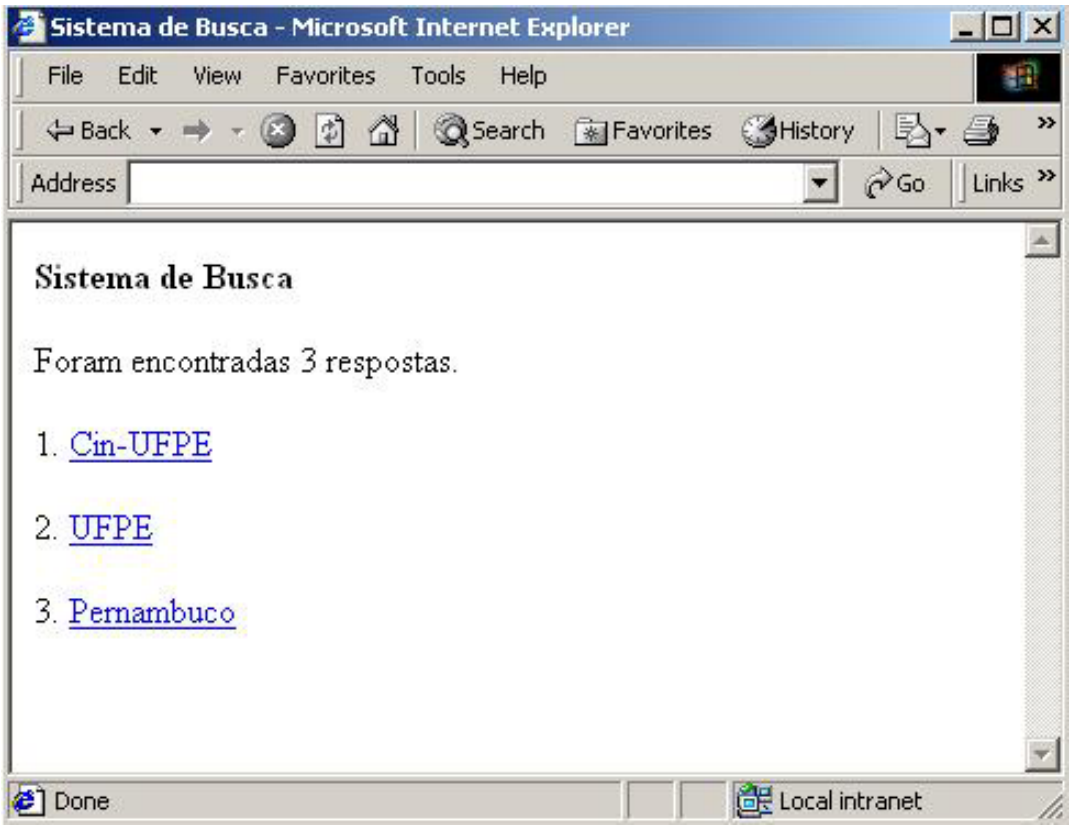

Figura 3.2: Tela de resposta do sistema de busca.

tramos como novos tipos podem ser definidos e propriedades podem ser declaradas para descrever as características que a GUI desejada para o sistema deve satisfazer. Também é mostrada a utilização de tipos anônimos na definição de propriedades, o que permite definições de tipos mais sucintas. Por fim, apresentamos descrições da estrutura da GUI do sistema que auxiliam a geração de código de sistema vista na Seção 4.3.

Por ser largamente conhecida e utilizada pela comunidade de software para modelagem de tipos, a notação UML [4] é utilizada neste trabalho para apresentar a definição dos tipos WSat. Para isso, assumiremos que a definição de um tipo a partir de outro é representada como uma especialização em UML, a definição de uma propriedade dentro de um tipo é representado como um atributo, quando o tipo da propriedade for `String` ou um dos tipo primitivo de Java, ou como uma agregação, quando o tipo da propriedade for um tipo WSat. Serviços de componentes Web como, por exemplo, a submissão de formulários, são representados em UML como métodos.

Definição de Tipos

Em WSat, a estrutura desejada da GUI de um sistema Web é descrita através da definição de tipos. Cada tipos WSat definido denota um conjunto de componentes Web que satisfazem determinadas características. Estas características são determinadas pelas propriedades dos tipos, como veremos mais adiante. Podemos então verificar se um determinado componente Web pertence ao conjunto denotado por este tipo, verificando

assim se este componente possui as características desejadas para o mesmo. Tipos também auxiliam, como mostrado mais adiante, a simular o uso do sistema. Tipos WSat são definidos a partir de tipos especiais existentes na linguagem. Na Figura 3.3 podemos ver estes tipos especiais.

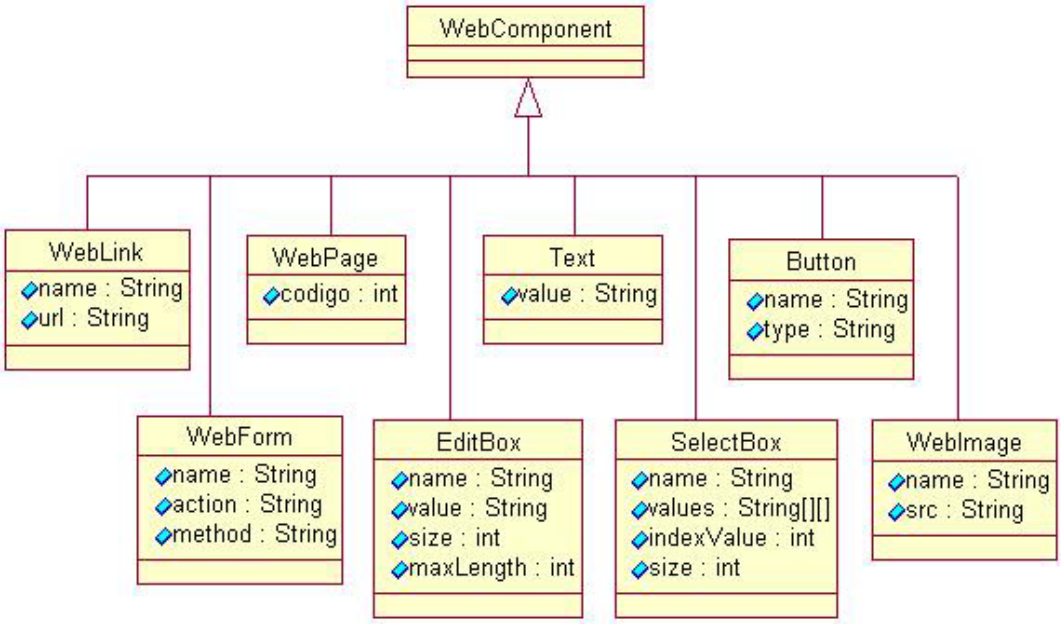

Figura 3.3: Tipos especiais predefinidos em WSat.

Como podemos ver, todos os tipos WSat são subtipos do tipo `WebComponent`, o qual denota o conjunto com todos os possíveis tipos de componentes Web. Já os outros tipos WSat vistos denotam conjuntos de componentes Web específicos. Por exemplo, os tipos `WebPage` e `WebImage` denotam respectivamente os conjuntos de todas as possíveis páginas Web e imagens possíveis.

Um dos principais componentes de sistemas Web, as páginas Web são utilizadas através dos navegadores Web para apresentar toda GUI do sistema. Para representar as características, por exemplo, que a página Web de resposta do sistema de busca mostrado na Seção 3.2.1 deve satisfazer, definimos o tipo `PagResposta` como visto abaixo.

```
WebPage PagResposta {
    codigo = 200;
}
```

Propriedades de um tipo WSat definem as restrições sobre o conjunto denotado por este tipo. Na definição do tipo `PagResposta`, por exemplo, podemos ver o uso da propriedade `codigo`, herdada do tipo `WebPage`, a qual indica o código de retorno da página Web definido pelo protocolo HTTP [7]. Este código indica, por exemplo, se a página foi retornada com sucesso (código `200`) ou não foi encontrada (código `404`) pelo servidor Web. Desta forma, o tipo definido denota o conjunto de todas as páginas Web cujo código de retorno é `200`.

Componentes Web podem naturalmente conter outros componentes. Páginas Web, por exemplo, podem conter *links*, textos, imagens, *frames*, entre outros. Em WSat,

representamos este relacionamento através da definição de novas propriedades. Podemos adicionar, por exemplo, uma nova propriedade chamada de `nomeSistema` ao tipo `PagResposta`, visto anteriormente, indicando um componente que representa o texto "`SistemadeBusca`". A definição desta nova propriedade e do componente texto são vistas a seguir.

```
WebPage PagResposta {
    codigo = 200;
    TextoNomeSistema nomeSistema;
}
Text TextoNomeSistema {
 value = "Sistema de Busca";
}
```

Como podemos ver, subtipos de `Text` representam componentes texto contidos no corpo de páginas Web. O valor deste texto é descrito pela propriedade `value`. Com esta nova definição, o tipo `PagResposta` denota o conjunto de todas as páginas Web cujo código de retorno é `200` e que possuem um componente pertecente ao conjunto denotado pelo tipo `TextoNomeSistema`, ou seja, que possuam em seu corpo o texto "`SistemadeBusca`".

Dentro de páginas Web, podemos ter também imagens (figuras). Estas imagens podem não só serem utilizadas para tornar a visualização das páginas mais agradável, como também transmitir informações através de gráficos, por exemplo. Desta forma, é importante poder representar estes componentes em WSat. Para representar imagens em WSat, definimos novos tipos a partir do tipo `WebImage`. A seguir, podemos ver a definição do tipo `PagInicial`, utilizado para a representar as características desejadas para a página inicial do sistema de busca.

```
WebPage PagInicial {
    codigo = 200;
    LogoSistBusca logoSistema;
}
WebImage LogoSistBusca {
    name = "logo";
    src = "imagens/logo.gif";
}
```

O tipo `PagInicial` possui a propriedade `logoSistema`, cujo tipo, `LogoSistBusca`, é um subtipo de `WebImage`. Podemos interpretar que o tipo `PagInicial` denota o conjunto de páginas Web que possuem um componente imagem que satisfaça as características definidas pelo tipo `LogoSistBusca`. Estas características são definidas pelas propriedades `name` e `src` herdadas do tipo `WebImage`. Estas propriedades representam respectivamente o nome e o caminho da imagem, indicados pelos atributos de mesmo nome do marcador HTML <`img`> correspondente a este componente. Desta forma, o tipo `LogoSistBusca` representa os componentes que representam imagens de nome "`logo`" e de caminho relativo "`imagens/logo.gif`".

Tipos WSat como, por exemplo, `WebImage`, `WebForm` e `WebLink`, estão relacionados a marcadores HTML, no caso, aos marcadores <`img`>, <`form`> e <`a`>. Já as suas

propriedades, em geral, estão relacionadas com atributos destes marcadores de nome igual a da propriedade. No exemplo mostrado anteriormente, as propriedades `name` e `src` do tipo `WebImage` estão relacionadas aos atributos `name` e `src` do marcador HTML <img>. O mesmo acontece com propriedades de outros tipos, podendo porém possuir nomes diferentes dos atributos do marcador HTML relacionado para não comprometer a legibilidade da linguagem.

Caso a definição de um novo tipo não atribua um valor a uma propriedade herdada de seu supertipo predefinido em WSat, esta propriedade não é utilizada para restringir os componentes que pertencem ao conjunto denotado pelo novo tipo devem satisfazer. Por exemplo, caso o tipo `LogoSistBusca` mostrado anteriormente não atribua em sua definição um valor a sua propriedade herdada `name`, apenas a restrição definida pela sua propriedade `src` é utilizada para testar se componentes imagem pertencem ao conjunto denotado pelo mesmo.

Com relação a definição de novas propriedades, subtipos de tipos como, por exemplo, `WebImage` e `Text`, não podem definir novas propriedades, pois os mesmos não podem possuir outros componentes Web. Já outros tipos como `WebForm`, por exemplo, permitem que seus subtipos defininam propriedades a partir de um conjunto limitado de tipos especiais de WSat, no caso, a partir dos tipos `EditBox`, `SelectBox` e `Button`. Componentes destes tipos representam parâmetros e botões de formulários Web.

Uma forma comum de navegação em sistemas Web é a utilização de formulários HTML. Para definir tipos em WSat que representem formulários HTML, fazemos uso do tipo `WebForm`. Para representar formulários como o do sistema de busca, por exemplo, podemos definir o tipo `FormBusca` como visto a seguir.

```
WebForm FormBusca {
    name = "formBusca";
    action = "pagResposta.jsp";
    method = "POST";
}
```

Formulários HTML possuem propriedades como, por exemplo, o nome do formulário, a URL para submissão dos seus parâmetros e o método de envio do protocolo HTTP. O método de envio significa como os valores de seus parâmetros são enviados, sendo os métodos `GET` e o `POST` os mais utilizados. Essas propriedades são representadas em WSat respectivamente pelas propriedades `name`, `action` e `method`, como visto no exemplo mostrado. Entretanto, formulários são compostos por um conjunto de parâmetros. Um dos tipos de parâmetros mais comuns em formulários HTML é o `text`, visualizado no navegador Web através de uma caixa de texto. Para definir tipos em WSat que representem campos textos de formulários Web, fazemos uso do tipo `EditBox`. Por exemplo, o espaço para preenchimento de palavras a serem procuradas pelo sistema de busca pode ser testado através do tipo `Palavras`:

```
EditBox Palavras {
    name = "palavras";
    value = "";
    size = 10;
    maxLength = 10;
}
```

As propriedades `name`, `value`, `size` e `maxLength`, vistas no exemplo mostrado, representam respectivamente o nome do parâmetro do formulário, o valor inicial deste parâmetro, o tamanho do componente e o seu número máximo de caracteres.

Alguns formulários Web possuem parâmetros cujo valor é selecionado a partir de uma lista de possíveis valores. Um exemplo disto é o parâmetro do formulário do sistema de busca com a lista de locais a serem procurados. Para representar tais parâmetros, definimos subtipos do tipo `SelectBox`. Abaixo podemos ver a definição do tipo `Onde` utilizado para testar o parâmetro como a lista de locais do sistema de busca.

```
SelectBox Onde {
    name = "onde";
    values = { {"Brasil", "1"}, {"Mundo", "2"} };
    indexValue = "1";
    size = 10;
}
```

As propriedades `name`, `indexValue`, `size` e `values`, herdada do tipo `SelectBox`, indicam respectivamente o nome do parâmetro, a lista dos possíveis valores deste campo (cada dupla possui o valor a ser visualizado no navegador Web e o seu respectivo valor a ser enviado na submissão do formulário), o índice na lista do valor selecionado e o tamanho deste componente. Como podemos ver, a lista dos possíveis valores deste campo é definida de acordo com a sintaxe de Java para *arrays*.

Para acionar a submissão dos parâmetros de um formulário HTML, o meio mais tradicional é através de botões. Para definir tipos em WSat que representem botões de formulários HTML, fazemos uso do tipo `Button`. O botão encontrado no formulário do sistema de busca, por exemplo, pode ser testado pelo tipo `Submeter` definido a seguir.

```
Button Submeter {
    name = "submeter";
    type = "submit";
}
```

As propriedades `name` e `type`, herdadas do tipo `Button`, representam respectivamente o nome e o tipo deste botão. Os tipos de botão mais comuns são o `submit` e `reset`. O primeiro tipo de botão citado submete os parâmetros do formulário. Já o segundo tipo atribui os valores iniciais aos parâmetros do formulário.

Uma vez definidos os tipos WSat que representam parâmetros e botões de formulários, podemos definir novas propriedades nos tipos que representam formulários indicando a existência destes parâmetros. O tipo `FormBusca`, definido anteriormente para testar o formulário de busca, pode ser alterado para indicar a presença de parâmetros, como visto a seguir.

```
WebForm FormBusca {
    Onde onde;
    Palavras palavras; ...
}
```

Com esta definição, o tipo `FormBusca` denota o conjunto de formulários HTML que possuem, entre outras coisas, dois parâmetros que satisfazem respectivamente as restrições impostas pelo tipo `Onde` e `Palavras`.

Além de formulários HTML, a navegação em sistemas Web pode ser feita através de *links*. Podemos definir tipos para representar *links* de páginas Web a partir do tipo `WebLink`. O *link* da imagem vista na página inicial do sistema de busca pode ser testado, por exemplo, pelo tipo `LinkLogoSistBusca` definido a seguir.

```
WebLink LinkLogoSistBusca {
    name = "logoSistBusca";
    url = "aboutSistBusca.html";
    LogoSistBusca logoSistBusca;
}
```

Este tipo utiliza as propriedades herdadas `name` e `url` para indicar o nome e a url dos componentes *link* representados. Também é definida a propriedade `logoSistBusca`, indicando que os componentes representados por este tipo são *links* que atuam em pelo menos um componente imagem com as características especificadas pelo tipo `LogoSistBusca`. Desta forma, podemos testar não só a existência de *links* em uma página Web, como também o seu conteúdo. Outras propriedades poderiam ser adicionadas com este significado, já que um mesmo *link* pode atuar em mais de um componente Web, desde que os mesmo sejam subtipos de `Text` e `WebImage`.

Os tipos mostrados até aqui herdam diretamente dos tipos especiais predefinidos em WSat. Entretanto, WSat permite a definição de um tipo a partir de tipos definidos pelo testador, como visto no exemplo abaixo.

```
LinkLogoSistBusca LinkLogoSistBuscaEstendido {
    TextoNomeSistema textoNomeSistema;
}
```

O tipo `LinkLogoSistBuscaEstendido` denota o subconjunto dos componentes Web representados pelo tipo `LinkLogoSistBusca` que possuem um componente texto que satisfaz as restrições definidas pela propriedade `textoNomeSistema`. Esta forma de reuso de definição mostrou-se pouco utilizada na prática, sendo mais comum o reuso de definição através da declaração de propriedades. Fato semelhante ocorre no desenvolvimento da GUI dos sistemas, onde as mesmas são montadas usando-se componentes existentes ao invés de estendendo-os.

Em WSat, não é possível definir tipos mutuamente dependentes. Por exemplo, não podemos definir um tipo X que tem uma propriedade do tipo Y, tendo o tipo Y uma propriedade do tipo X. Este tipo de definição, assim como definições recursivas, não é válida em WSat por não representar corretamente componentes Web.

A definição de tipos WSat é utilizada para descrever a estrutura da GUI do sistema Web a ser testado. No entanto, esta descrição poderia ser extraída diretamente do próprio código HTML do sistema, que já representa boa parte das informações mostradas aqui, ou de programas escritos em uma linguagem de teste similar ao HTML, onde os programas poderiam ser escritos copiando-se o código fonte das páginas Web do sistema. Entretanto, como veremos mais adiante, além do alto nível de abstração e reuso fornecido

por WSat para a definição de tipos, estes tipos são utilizados para simular o uso do sistema, testando assim não só a estrutura como também o comportamento da GUI do sistema.

Tipos Anônimos

Nos exemplos mostrados até aqui, para declarar uma nova propriedade em um tipo WSat era necessário declarar também um tipo auxiliar para a mesma. Nesta seção, mostramos a declaração de propriedades através de tipos anônimos, onde não é necessária a declaração de um novo tipo WSat somente para auxiliar a definição desta propriedade. Tipos anônimos são utilizados, por exemplo, quando a definição de um tipo é feita somente para se declarar uma propriedade. A página inicial do sistema de busca, por exemplo, possui o texto "`Buscar por:`". Para testar esta característica, podemos adicionar a propriedade `textoBusca` no tipo `PagInicial`, como visto no código a seguir.

```
WebPage PagInicial {
    TextoBusca textoBusca;  ...
}
Text TextoBusca {
    value = "Buscar por:";
}
```

A definição da propriedade `textoBusca` na forma mostrada precisou da definição de um novo tipo, no caso, o tipo `TextoBusca`. A declaração de um tipo WSat permite o seu reuso na definição de outras propriedades e tipos, assim como a manipulação de componentes através de variáveis declaradas a partir deste tipo. Como podemos ver no exemplo completo de programa WSat mostrado adiante, o tipo auxiliar `TextoBusca` não é utilizado em outras definições de tipos nem tem variáveis declaradas a partir dele. Em casos como este, para diminuir o número de linhas de código e de declarações de tipos, as propriedades podem ser definidas com o uso de tipos anônimos, conforme visto no código a seguir.

```
WebPage PagInicial {
    Text {
        value = "Buscar por:";
    } textoBusca;  ...
}
```

Como podemos notar, o tipo utilizado na declaração da propriedade `textoBusca` é declarado junto com a propriedade e de forma anônima. Podemos perceber também que a definição de propriedades a partir de tipos auxiliares não anônimos, chamados daqui em diante simplesmente de tipos auxiliares, se torna mais extensa quando considerada a definição destes tipos auxiliares. Além disto, para o entendimento do tipo `PagInicial` com tipos auxiliares, por exemplo, faz-se necessário observar a definição do tipo auxiliar `TextoBusca`.

Apesar dos benefícios apresentados, a utilização de tipos anônimos nem sempre é a forma de declaração de propriedades mais apropriada mesmo que o tipo auxiliar não seja reutilizado em outras definições. Isto porque não se pode declarar variáveis em

WSat a partir de tipos anônimos. Consideremos, por exemplo, a definição mostrada anteriormente para o tipo `PaginaInicial` utilizado para testar a página Web inicial do sistema de busca. Para termos acesso direto, durante a execução dos testes, a todas as propriedades do componente que representa o formulário de busca, sejam as mesmas herdadas do tipo `WebForm` ou definidos pelo tipo `FormBusca`, basta-nos manipular este componente através de uma variável do tipo `FormBusca`, como visto no código a seguir. A notação de WSat para declaração de variáveis e acesso a propriedades é a mesma utilizada em Java para declaração de variáveis e acesso a atributos públicos.

```
01: PagInicial pag; ...
02: FormBusca form = pag.formBusca;
03: String metodo = form.method;
04: form.palavras.value = "ufpe";
```

Como podemos ver, a partir da variável `form` declarada como do tipo `FormBusca`, obtivemos acesso à propriedades herdadas do tipo `WebForm`, como a propriedade `action` (linha 3), por exemplo, e propriedades definidas no próprio tipo, como a propriedade `palavras` (linha 4). Consideremos agora a definição do tipo `PagInicial` com o uso de tipos anônimos, como mostrado a seguir.

```
WebPage PagInicial {
    WebForm formBusca {
        action = "pagResposta.jsp"; ...
    } ...
}
```

Observando a definição da propriedade `formBusca` que testa o formulário HTML de busca, podemos observar que não foi necessária a definição de um novo tipo auxiliar, no caso, o tipo `FormBusca`. A partir desta definição, para manipular diretamente o componente que representa o formulário de busca durante a execução dos testes, podemos declarar variáveis do supertipo do tipo anônimo para referenciá-los, no caso variáveis do tipo `WebForm`. A manipulação do formulário do sistema de busca através de tais variáveis é mostrado a seguir.

```
01: PagInicial pag; ...
02: WebForm form = pag.formBusca;
03: String metodo = form.method;
04: pag.formBusca.palavras.value = "ufpe";
```

De fato, através da variável `form` declarada na linha 2 temos acesso às propriedades definidas pelo tipo `WebForm`, como por exemplo, o acesso à propriedade `action` visto na linha 3. Entretanto, para termos acesso aos parâmetros do formulário, é necessário utilizar a variável `pag`, como visto na linha 4. Isto porque variáveis só fornecem acesso às propriedades definidas pelo seu tipo declarado.

A definição de propriedades a partir de tipos auxiliares torna o programa de teste mais extenso do que com a definição de propriedades a partir de tipos anônimos. Entretanto, uma vez definidos tais tipos, os mesmos podem ser utilizados na definição de outras propriedades e tipos, além de possibilitar declarar variáveis do tipo definido.

Descrevendo Informações para Geração de Código de Sistema

Para permitir que o gerador de código de sistema, apresentado na Seção 4.3, gere eficientemente uma quantidade maior de código a partir de programas WSat, é necessário introduzir informações complementares aos programas de teste. Estas informações não são utilizadas durante a execução dos testes propriamente ditos, mas somente durante a geração de código de sistema. Esta seção descreve quais informações são necessárias para a geração de código de sistema e apresenta as estruturas da linguagem WSat criadas para fornecer estas informações.

Como podemos ver no próximo capítulo, o gerador de código de sistema gera código Java para as páginas Web dinâmicas, geradas pelo servidor Web através de tecnologias como, por exemplo, Servlets [22] e *scripts* Perl [5]. Desta forma, é necessário indicar nos programas WSat quais páginas Web são estáticas, escritas em arquivos HTML simples, e quais páginas são geradas dinamicamente. Isto é feito através do uso da palavra chave `static` como visto no exemplo mostrado a seguir.

```
static WebPage PagInicial {
    ...
}
```

A utilização da palavra chave `static` pode ser usada somente na definição de subtipos de `WebPage` como, por exemplo, na definição do tipo `PagInicial`. O seu uso no exemplo mostrado indica que os componentes pertencentes ao conjunto denotado por este tipo são estáticos. No exemplo do sistema de busca, podemos deduzir a partir da nova definição do tipo `PagInicial` que a página de acesso ao sistema de busca é estática.

A declaração de um tipo com a palavra chave `static` não implica que seus subtipos também representem componentes estáticos, devendo para isto usar a palavra chave também em sua definição. No projeto da linguagem, poderiamos ter escolhido por utilizar uma palavra chave que indicasse tipos que representem componentes dinâmicos. Esta opção não foi utilizada para diminuir o código WSat a ser escrito, já que foi notado em experimentos uma maior ocorrência de páginas dinâmicas do que estáticas entre as páginas que realmente necessitavam ser testadas.

Outra informação importante que pode ser agregada na definição de subtipos de `WebPage` é o caminho do arquivo de uma página Web que satisfaz as restrições definidas pelo tipo, indicada pela propriedade herdada `template`. Estes arquivos geralmente fazem parte do protótipo de telas do sistema. A partir desta informação, podem-se criar *templates* para páginas dinâmicas e realizar a cópia das páginas estáticas para o servidor Web, como veremos no próximo capítulo. A seguir podemos ver uma nova definição dos tipos `PagInicial` e `PagResposta` que utilizam a propriedade *template*.

```
static WebPage PagInicial {
    template = "c:\projeto\wsat\exemplo\www\pagInicial.html";
    ...
}
WebPage PagResposta {
    template = "c:\projeto\wsat\exemplo\www\pagResposta.html";
    ...
}
```

Nesta nova definição, indicamos o caminho de arquivos do protótipo de telas do sistema de busca que satisfazem as restrições definidas por estes tipos.

Em sistemas Web podem existir páginas com formulários HTML utilizados para requisição de serviços. Na página inicial do sistema de busca mostrado, por exemplo, existe um formulário para requisição de um serviço de busca. Muitas vezes os parâmetros destes formulários HTML possuem restrições sobre os valores que os mesmos podem assumir. Voltando ao exemplo citado, o parâmetro `onde` do formulário de busca só pode possuir os valores `1` ou `2`, referentes a seleção dos textos "`Brasil`" e "`Mundo`", respectivamente.

De forma geral, a implementação de sistemas Web deve verificar se os valores destes parâmetros são válidos, já que os mesmos podem ser facilmente modificados indiscriminadamente no lado cliente. Como o testador está descrevendo a estrutura do sistema Web ao escrever os programas WSat, o que inclui os formulários HTML, é oportuno deixar que o mesmo indique as restrições existentes para cada um dos parâmetros existentes nestes formulários. Para isto, além de propriedades é possível utilizar na definição de tipos que representam parâmetros de formulários a cláusula `validate`, como visto no exemplo abaixo.

```
SelectBox Onde {
    values = { {"Brasil", "1"}, {"Mundo", "2"} };
    validate {
        optional = false;
        enumeration = {"1", "2"};
    } ...
}
```

As restrições apresentadas pela cláusula `validate` são utilizadas pelo gerador de código de sistema para gerar código de validação dos parâmetros de formulários HTML. As restrições que podem ser utilizadas em cláusulas `validate` são vistas na Tabela 3.1.

Um Exemplo Completo

Nesta seção, apresentamos um exemplo completo de descrição estrutural da GUI do sistema de busca em WSat que foi construído no decorrer das últimas seções.

```
WebPage PagResposta {
    codigo = 200;
    template = "c:\projeto\wsat\exemplo\www\pagResposta.html";
    Text TextoNomeSistema {
        value = "Sistema de Busca";
    }
}
static WebPage PagInicial {
    codigo = 200;
    template = "c:\projeto\wsat\exemplo\www\pagInicial.html";
    FormBusca formBusca;
    Text {
```

Nome	Descrição
`type`	Texto indicando o tipo do parâmetro. Este tipo pode ser do tipo `String` ou de um dos tipos primitivos de Java como, por exemplo, `int`, `double`. Existem também tipos especiais como, por exemplo, `date`, que representa uma data.
`enumeration`	Lista de valores permitidos para o parâmetro.
`maxExclusiveValue`	Texto indicando um valor ("3", "10/05/2000", ...) que tem que ser maior que o valor do parâmetro
`maxInclusiveValue`	Texto indicando um valor que tem que ser maior ou igual que o valor do parâmetro.
`minExclusiveValue`	Texto indicando um valor que tem que ser menor que o valor do parâmetro
`minInclusiveValue`	Texto indicando um valor que tem que ser maior ou igual que o valor do parâmetro.
`optional`	Valor booleano indicando se o valor do parâmetro pode ser omitido.

Tabela 3.1: Restrições suportadas pela cláusula `validate`.

```
            value = "Buscar por:";
        } textoBusca;
        WebLink {
            name = "logoSistBusca";
            url = "aboutSistBusca.html";
            WebImage {
                name = "logo";
                src = "imagens/logo.gif";
            } logoSistBusca;
        } linkLogoSistema;
    }
    WebForm FormBusca {
        action = "pagResposta.jsp";
        method = "POST";
        name = "formBusca";
        Button {
            name = "submeter";
            type = "submit";
        } botaoSubmeter;
        EditBox {
            name = "palavras";
            value = "";
            size = 10;
            maxLength = 10;
            validate {
                optional = false;
```

```
            type = "String";
        }
    } palavras;
    SelectBox {
        name = "onde";
        values = { {"Brasil", "1"}, {"Mundo", "2"} };
        indexValue = "1";
        size = 10;
        validate {
            optional = false;
            enumeration = {"1", "2"};
            type = "int";
        }
    } onde;
}
```

Podemos notar neste exemplo que a página de resposta do sistema de busca, testada pelo tipo `PagResposta`, foi menos explorada em sua descrição estrutural. Este é um fato que geralmente ocorre com as páginas dinâmicas, já que o seu conteúdo é gerado em tempo de execução. Em seção própria mais adiante, mostramos como utilizar WSat para testar o conteúdo dinâmico de páginas Web.

3.2.3 Descrevendo o Comportamento do Sistema

A partir da representação em WSat da estrutura desejada da GUI do sistema, é possível definir casos de teste para verificar se algum sistema satisfaz tal estrutura. Para validar esta estrutura, é necessário que os casos de teste explorem o sistema testado através da simulação da interação entre o sistema e seus usuários. Desta forma, os casos de teste acabam por validar não só a estrutura da GUI de um sistema como também o seu comportamento. Durante esta seção, são mostrados os comandos WSat que podem ser usados na definição de casos de teste.

Variáveis

Em WSat, tipos são definidos para representar os componentes de sistemas Web. Durante a execução dos programas WSat, componentes destes tipos são manipulados simulando ações executadas pelo usuário. A manipulação destes componentes se dá, em geral, pelo uso de variáveis que armazenam referências para os mesmos. Variáveis em WSat tem escopo limitado ao local onde a mesma foi declarada. Declarações de variáveis podem ser realizadas dentro do corpo da definição de casos de teste ou de funções de testes (vistas mais adiante). As variáveis podem ser declaradas como de um tipo WSat, do tipo `String` ou dos tipos primitivos suportados pela linguagem Java [19]. Podemos também ter declarações de variáveis como `arrays` dos tipos citados. Suas declarações são feitas como em Java:

```
String urlAcesso = "http://www.sistemadebusca.com.br/";
WebPage indexPage = getWebPage(urlAcesso);
int numRespostas;
```

Na própria declaração de uma variável podemos fazer a sua inicialização. No exemplo anterior, podemos ver a declaração da variável `urlAcesso` e a sua inicialização com a URL de acesso ao sistema de busca. Como em Java, os componentes Web atribuídos a uma variável devem ser do mesmo tipo ou de um subtipo do tipo da variável. O comando `getWebPage`, por exemplo, retorna um componente de mesmo tipo da variável `indexPage` declarada no exemplo mostrado.

Lógica de Programação

WSat possui os comandos de lógica de programação `if` e `for`, utilizados para manipular variáveis. Estes comandos possuem sintaxe e semântica equivalentes aos da linguagem Java. A seguir mostramos um exemplo de uso do comando `if` para verificar se não foram retornadas respostas pelo sistema de busca.

```
if (numRespostas == 0) {
    ...
}
```

Como podemos ver, o operador lógico == é utilizado para comparar o valor da variável `numRespostas` e `0`. Este operador, assim como outros, pertencem a um conjunto mínimo de WSat de operadores de lógica e aritmética que possuem a mesma sintaxe e semântica na linguagem Java.

Comando `getWebPage`

Para a realização da operação de requisição de uma página Web a partir de sua URL, podemos utilizar o comando `getWebPage`. Este comando recebe como parâmetro uma `String` representando a URL da página desejada e seu valor de retorno é um componente do tipo `WebPage` que representa a página de URL indicada. Abaixo podemos ver um exemplo de sua utilização.

```
WebPage indexPage = getWebPage("http://www.sistemadebusca.com.br/");
```

No exemplo mostrado, utilizamos o comando `getWebPage` para ter acesso ao componente Web que representa a página inicial do sistema de busca de URL "`http://www.sistema-debusca.com.br/`".

Operador de Transformação de Tipo

Para realizar, por exemplo, o teste de que a página de acesso do sistema de busca retornada pelo comando `getWebPage` possui as características definidas pelo tipo `PagInicial`, fazemos uso da operação de transformação de tipo como visto a seguir.

```
PagInicial pag = [PagInicial] getWebPage(urlAcesso);
```

No exemplo mostrado, o comando `getWebPage` retorna um componente do tipo `WebPage` que representa a página Web inicial do sistema de busca. A operação de transformação de tipo verifica se esta página Web retornada satisfaz às restrições impostas pelas propriedades definidas no tipo `PagInicial` como, por exemplo, de seu código de retorno

ser `200`. Podemos interpretar a operação de transformação de tipo de um componente Web como a verificação de que este componente pertence ao conjunto denotado pelo tipo final da transformação. Uma vez realizada a transformação de tipo, o componente testado pode ser manipulado pelo programa WSat a partir de variáveis do tipo final da transformação.

Os tipos de componentes envolvidos na operação de transformação de tipo devem ser tipos que representem componentes Web. Além disto, o tipo atual do componente a ser transformado deve ser um supertipo do tipo final da transformação. No exemplo mostrado anteriormente, por exemplo, o tipo atual do componente é `WebPage` e o tipo final é seu subtipo `PagInicial`. Durante a transformação de tipo, as propriedades definidas pelo tipo final são verificadas no componente Web operado. Caso alguma propriedade verificada não seja satisfeita por este componente, o caso de teste em execução será interrompido por motivo de falha na execução, ou seja, o teste realizado não foi satisfeito. Detalhes sobre falhas na execução de casos de teste são vistos mais adiante.

Acesso a Propriedades e Serviços de Componentes Web

Através de variáveis, podemos acessar as propriedades e serviços de seus componentes referenciados. Por exemplo, podemos acessar os parâmetros do formulário de busca da página inicial do sistema através da variável `pag`, como visto no código a seguir.

```
pag.busca.palavras.value = "ufpe";
WebPage resp = pag.busca.submit();
```

No primeiro comando, podemos ver a atribuição de um valor ao parâmetro `palavras` do formulário de busca através de uma sequência de acessos a propriedades, começando a partir da propriedade `busca`, definida no tipo `PagInicial` e que representa o formulário de busca. Por fim, podemos ver a invocação do serviço `submit` do componente que representa o formulário de busca referenciado pela propriedade `busca`, acessada através da variável `pag`. Desta forma, o código mostrado simula o preenchimento e submissão do formulário de busca.

Teste do Conteúdo Dinâmico de Componentes Web

Na seção anterior, mostramos a operação de transformação de tipo que testa um componente Web baseado a partir de um tipo WSat. Como é visto na seção sobre definição de tipos, os tipos WSat são utilizados para descrever propriedades que componentes Web devem satisfazer. Quando os componentes Web são estáticos, como páginas HTML ou imagens escritas em arquivos estáticos, é possível descrever detalhadamente o conteúdo que eles possuem em tipos WSat. No caso de componentes dinâmicos como, por exemplo, páginas Web geradas a partir de Servlets ou *scripts* Perl, partes importantes de seu conteúdo são criadas dinamicamente. De fato, uma mesma página Web dinâmica pode apresentar informações diferentes dependendo dos valores de entrada passados para a mesma. Um exemplo disto é a página de resposta do sistema de busca, que apresentam *links* para outras páginas dependendo das palavras chave utilizadas.

Podemos testar o conteúdo de páginas dinâmicas, como por exemplo, a de resposta do sistema de busca, definindo-se um tipo WSat para cada possibilidade de resposta desta página a ser testada. Se quisermos testar a página de resposta do sistema de

busca para as palavras chave "ufpe" e "pernambuco" individualmente, por exemplo, podemos criar os tipos vistos a seguir.

```
PagResposta PagRespostaPalavraChaveUFPE {
    Text numRespEncontradas {
        value = "Foram encontradas 3 respostas.";
    }
    WebLink linkUFPE {
        url = "http://www.ufpe.br";
    } ...
}
PagResposta PagRespostaPalavraChavePernambuco {
    Text numRespEncontradas {
        valur = "Foram encontradas 5 respostas.";
    }
    WebLink linkPernambuco {
        url = "http://www.pernambuco.gov.br";
    } ...
}
```

Os tipos `PagRespostaPalavraChaveUFPE` e `PagRespostaPalavraChavePernambuco` podem ser usados na operação de transformação de tipo, vista anteriormente, para validar as respostas do sistema de busca de acordo com a palavra chave utilizada. Esta forma de teste, além de se mostrar bastante trabalhosa, inviabiliza uma possível parametrização do caso de teste. Também mostramos a verificação do número de respostas de forma fixa, podendo esta porém ser feita através de casamentos de padrões formados por expressões regulares, como veremos mais adiante. Por estes motivos, foram definidos serviços especiais para testes em componentes Web dinâmicos nos tipos predefinidos em WSat. Podemos, por exemplo, verificar a existência do componente texto que representa o número de respostas encontradas na página de resposta do sistema de busca através do código mostrado a seguir.

```
Text texto =
    pagResposta.findTextByValue("Foram encontradas 3 respostas.");
```

Ao manipular um componente do tipo `WebPage`, podemos utilizar o seu serviço `findTextByValue`. Este serviço recebe como parâmetro um texto e retorna o componente que o representa na página Web manipulada. Caso não exista tal componente, este método faz com que a execução do caso de teste falhe. Existe um conjunto de serviços nos tipos WSat predefinidos que são utilizados para teste similarmente ao seviço já apresentado. Exemplos destes serviços são os serviços `findWebLinkByName`, `findWebLinkByURL` e `findImageBySrc`, que variam tanto no tipo do componente que presta o serviço como no tipo de argumento e de retorno do serviço.

Componentes Web em uma mesma página Web podem possuir as mesmas características. Por exemplo, podemos colocar na página de resposta do sistema de busca duas imagens de propaganda com *links* para o *site* de um determinado cliente, como visto a seguir.

```
...
<a href="http://www.cliente.com.br">
    <img src="imagens/banner1Cliente.jpg"></a>    ...
<a href="http://www.cliente.com.br">
    <img src="imagens/banner2Cliente.jpg"></a>    ...
```

Como podemos notar, estes *links* de mesma URL não possuem nome, definido pelo atributo `name` do seu marcador HTML. Desta forma, não podemos acessar individualmente os componentes que os representam através do serviço `findWebLinkByName`. Utilizando então o serviço `findWebLinkByURL` nesta página, este serviço só irá retornar o componente que representa o primeiro *link* encontrado. Para ter acesso a componentes em uma mesma página Web que possuem as mesmas características, podemos utilizar um conjunto de serviços semelhantes aos mostrados anteriormente, porém que retornam todos os componentes com as características indicadas. Por exemplo, o serviço `findAllWebLinkByURL` utilizado na página de resposta do sistema de busca, como visto no código a seguir, retorna o *array* com os dois componentes da página Web que representam os *links* para o *site* do cliente.

```
WebLink[] webLinks =
    pagResposta.findAllWebLinkByURL("http://www.cliente.com.br");
```

A partir do *array* retornado, podemos percorrê-lo como visto em seção mais adiante invocando o serviço `findWebImageBySrc` de cada *link* encontrado para testar se ele está realmente atuando sobre a imagem do cliente.

Em certos casos, podemos estar interessados em acessar um valor texto gerado dinamicamente pela página Web. Por exemplo, podemos querer testar o número de respostas retornadas pelo sistema de busca. Este número está representado por um componente texto. Entretanto, não podemos utilizar o serviço `findTextByValue`, pois o que queremos encontrar é justamente o valor do texto. Para estes casos, podemos utilizar o serviço `findTextByRegularExpression`, como visto a seguir.

```
String[] textoResp = resp.findTextByRegularExpression(
                        "Foram encontradas (\d{1,2}) respostas");
```

Este serviço recebe como parâmetro uma `String` contendo um padrão de texto em forma de expressão regular (*Pattern Matching*) compatível com a linguagem Perl5 [5]. O padrão definido pela expressão regular vista acima casa com textos que começam com "`Foramencontradas`", seguidos de um número de um a dois dígitos "1,2" e que terminam com "`respostas`". Os parênteses vistos são marcadores de grupos que indicam valores que queremos acessar. No exemplo, os parênteses utilizados marcam um grupo contendo o número de respostas retornado. A execução do serviço `findTextByRegularExpression` tenta casar a expressão regular indicada com textos contidos na página Web. Este serviço retorna um *array* contendo o valor casado para cada grupo definido. No caso, será retornado um *array* de uma posição contendo o número de respostas retornadas pelo sistema de busca como, por exemplo, a `String` "3" para o caso de retornar três respostas. Caso o serviço não consiga efetuar nenhum casamento de padrão, o serviço retorna um *array* vazio. Caso a página Web possua mais de uma ocorrência do padrão, podemos

utilizar o serviço `findAllTextByRegularExpression`, o qual retorna um *array* de *arrays* que contêm o valor casado de cada grupo para cada casamento de padrão efetuado.

Os tipos predefinidos em WSat, como por exemplo, `WebPage` e `WebLink`, possuem propriedades dos tipo `String` ou primitivos de Java. Podemos verificar estas propriedades através o comando `assert`. O código visto a seguir testa se uma das páginas retornadas pelo sistema de busca, referenciada pela variável `paginaEncontrada`, possui o código de retorno `200` indicando que o *link* para a mesma na página de resposta não está quebrado.

```
assert("Link de resposta quebrado.", paginaEncontrada.codigo, 200);
```

Como podemos notar, este comando recebe três parâmetros. O primeiro deles, do tipo `String`, é uma mensagem a ser utilizada no caso de falha do teste. Os parâmetros seguintes representam respectivamente o valor esperado e o valor encontrado de uma propriedade testada. Estes parâmetros devem ser do mesmo tipo, podendo eles serem dos tipos primitivos de Java ou do tipo `String`. Este comando compara o valor esperado com o valor encontrado recebidos, causando uma falha na execução do caso de teste caso estes valores sejam diferentes. Outra forma de realizar o mesmo teste é através do comando `fail`, como visto no código a seguir.

```
if (paginaEncontrada.codigo != 200) {
    fail("Link de resposta quebrado.");
}
```

O comando `fail` causa uma falha na execução do caso de teste. Este comando recebe como parâmetro uma `String` contendo o motivo da falha de execução.

Casos de Testes

A execução de programas WSat é iniciada a partir de casos de teste. Para a definição de casos de teste em WSat, fazemos uso da cláusula `testCase`. Para testar o sistema de busca apresentado na Seção 3.2.1, por exemplo, definimos o caso de teste `buscar` como visto no código a seguir.

```
testCase buscar {
    PagInicial pag = [PagInicial] getWebPage(
                              "http://www.sistemadebusca.com.br/");
    ...
}
```

O corpo da definição de um caso de teste é composto pelos comandos WSat que implementam o teste. Um mesmo programa WSat pode conter diversos casos de teste. A execução de cada um destes casos de teste é realizada de forma independente da execução dos outros, sendo indefinida a sua ordem de execução. A falha na execução de um dos casos de teste não impede nem influencia a execução dos outros casos de teste.

Funções de Testes

Visando uma melhor estruturação e reuso de código nos programas de teste, programadores WSat podem definir funções de testes. Funções de teste podem possuir parâmetros e retornar valores. Seus parâmetros são passados por referência e estas funções podem ser invocadas diversas vezes, inclusive com recursão mútua, em um mesmo ou em diferentes casos e funções de teste. Para preencher o formulário do sistema de busca mostrado anteriormente, podemos definir a função mostrada a seguir.

```
PagResposta buscarPalavras(PaginaInicial pag, String palavras,
                           int indexLugar) {
    FormBusca form = pag.formBusca;
    form.palavras.value = palavras;
    form.onde.indexValue = indexLugar;
    return [PagResposta] form.submit();
}
```

Esta função recebe respectivamente como parâmetro o componente Web que representa a página inicial do sistema de busca e o valor dos parâmetros palavras chave e lugar da busca do formulário HTML. Seu retorno é o componente que representa a página de resposta do sistema de busca para os dados parâmetros.

Funções como esta podem ser utilizadas no lugar de trechos de código utilizados em diversos locais de um programa de teste. Vejamos o exemplo a seguir.

```
PagInicial pag = getWebPage(urlAcesso);
FormBusca form = pag.formBusca;
PagResposta resp;
...
form.palavras.value = "ufpe";
form.onde.indexValue = 1;
resp = (PagResposta) form.submit();
...
form.palavras.value = "cin-ufpe";
form.onde.indexValue = 1;
resp = (PagResposta) form.submit();
...
```

De fato, utilizando a função `buscarPalavras` definida anteriormente, o código acima pode ser reescrito para o código de tamanho reduzido mostrado abaixo.

```
PagInicial pag = getWebPage(urlAcesso);
PagResposta resp;
...
resp = buscarPalavras(pag,"ufpe",1);
...
resp = buscarPalavras(pag,"cin-ufpe",1);
...
```

Desta forma, podemos notar que uso de uma funções de teste encapsulando trecho de código que se repetem promove uma melhor estruturação do programa e reuso de código.

Código Java Embutido

A definição e implementação de WSat levou em conta as limitações de tempo impostas para este trabalho. Como vemos na Seção 4.1, o ambiente de execução implementado para WSat não possui suporte direto a instruções lógicas encontradas na maioria das linguagens de programação. Também não houve tempo para definir formas simples e abstratas para parametrização dos casos de teste, realização de acesso a banco, testes de performance, entre outras funcionalidades interessantes que são suportadas pela linguagem Java, por exemplo. Para permitir uma utilização mais eficiente da linguagem, foi definida uma forma de se colocar códigos da linguagem Java embutidos em WSat.

Para parametrizar os parâmetros utilizados no formulário HTML do sistema de busca, por exemplo, podemos utilizar a classe `java.util.ResourceBundle` para ler os parâmetros de entrada `palavras` e `indexValue`, como vistos a seguir.

```
String palavras;
int indexValue;
<@
    java.util.ResourceBundle rb =
                   java.util.ResourceBundle.getBundle("sistBusca");
    palavras = rb.getString("palavras");
    indexValue  = Integer.parseInt(rb.getString("indexValue"));
@>
form.palavras.value = palavras;
form.onde.indexValue = indexValue;
```

Como pode ser visto no exemplo acima, códigos Java embutidos são delimitados pelos símbolos '<@' e '@>'. Estes códigos embutidos podem conter comandos Java válidos. Os comandos WSat e comandos Java podem se comunicar através das variáveis declaradas em WSat, como por exemplo, as variáveis `palavras` e `indexValue`, acessíveis por ambos os códigos.

Através deste tipo de código podemos construir programas bastante complexos. Entretanto, o uso excessivo de códigos Java embutidos pode comprometer consideravelmente a legibilidade do código escrito. Mesmo assim, o ganho em abstração com a utilização dos tipos e comandos WSat justifica a utilização de WSat para construção de programas de teste ao invés do uso da linguagem Java propriamente dita.

Um Exemplo Completo

Nesta seção, apresentamos um exemplo completo de caso de teste de aceitação do sistema de busca, o qual foi construído no decorrer da Seção 3.2.3 e que utiliza os tipos definidos na Seção 3.2.2. Para testar o sistema de busca, o caso de teste mostrado a seguir recebe como parâmetros a URL de acesso, a palavra chave e o valor do parâmetro que indica o local a ser utilizado na busca. O programa de teste simula o uso deste sistema a partir dos parâmetros recebido. Em seguida, testamos se o conjunto de *links* retornados na resposta está correto. Para isto, é simulado o clique em cada um destes *links*, verificando então se a página referenciada pelo *link* contém a palavra chave utilizada.

```
testCase buscar {
    String palavra;
    int indexValue;
    String urlAcesso;
    <@
        java.util.ResourceBundle rb =
                  java.util.ResourceBundle.getBundle("sistBusca");
        palavra = rb.getString("palavra");
        indexValue = Integer.parseInt(
                            rb.getString("indiceParametroOnde"));
        urlAcesso = rb.getString("urlAcesso");
    @>
    PagInicial pag = [PagInicial] getWebPage(urlAcesso);
    PagResposta resp = buscarPalavras(pag, palavra, indexValue);
    testarResposta(resp, palavra);
}
PagResposta buscarPalavras(PaginaInicial pag, String palavra,
                           int indexLugar) {
    FormBusca form = pag.formBusca;
    form.palavras.value = palavra;
    form.onde.indexValue = indexLugar;
    return [PagResposta] form.submit();
}
void testarResposta(PagResposta resp, String palavra) {
    String[] textoResp = resp.findTextByRegularExpression(
                          "Foram encontradas (\d{1,2}) respostas");
    int numResp;
    String nomeLink;
    WebLink link;
    if textoResp.length != 1) {
        fail("Não foi encontrado o texto contendo o número de " +
             "respostas encontradas.");
    }
    numResp = Integer.parseInt(textoResp[0]);
    for (int i = 0 ; i < numResp ; i++) {
        nomeLink = "resp" + i;
        link = resp.findWebLinkByName(nomeLink);
        checarLink(link, palavra);
    }
}
void checarLink(WebLink link, String palavra) {
    WebPage paginaEncontrada = link.click();
    assert("Link de resposta quebrado.", paginaEncontrada.codigo,
           200);
    Text textoPalavra = pagina.findTextByValue(palavra);
}
```

3.3 Execução de Programas WSat

A execução de casos de teste de aceitação em WSat pode resultar em sucessos, falhas ou erros. Falhas ocorrem quando os tipos definidos para representar a estrutura do sistema testado têm suas características verificadas e os seus valores encontrados são diferentes dos valores esperados. De maneira geral, falhas ocorrem quando o sistema testado se comporta de forma não esperada. Erros na execução de programas WSat são gerados quando códigos Java embutidos levantam algum tipo de exceção. Estas exceções não são tratadas pela linguagem WSat, impossibilitando a continuação da execução do programa de teste. Já a ocorrência de sucesso é obtida quando não se ocorre falha ou erro durante a execução de um caso de teste.

Em WSat, tanto a ocorrência de uma falha como de um erro em um caso de teste interrompem a sua execução. Implementações da linguagem devem apresentar relatórios contendo o resultado da execução de cada um dos casos de teste executados, indicando os motivos de falhas ou erros na ocorrência dos mesmos.

Capítulo 4

Ferramentas para a Realização de Testes Web

Neste capítulo, apresentamos as ferramentas desenvolvidas para apoiar a realização de testes de aceitação de sistemas Web. Para cada uma destas ferramentas, apresentamos a motivação para sua construção, os detalhes sobre seu projeto e implementação, além de uma descrição de como executá-las. Por fim, mostramos um processo para implementação e testes com as ferramentas apresentadas.

No Capítulo 3, apresentamos uma nova linguagem para descrição de casos de teste de aceitação de sistemas Web. Para a validação e utilização desta linguagem, é necessário a implementação de um ambiente de execução [1] para a mesma. Para implementação deste ambiente de execução, foram estudadas diversas técnicas, como por exemplo, compilação, interpretação e até a tradução para outra linguagem de teste existente. Além de mostrar o resultado destes estudos, apresentamos também o projeto e implementação do ambiente de execução a partir da técnica de implementação escolhida. Também são discutidas as ferramentas de apoio utilizadas para o desenvolvimento deste ambiente.

Além do ambiente de execução de WSat, outras ferramentas foram criadas para apoiar e incentivar o desenvolvimento dos testes. Isto foi necessário porque a atividade de teste causa um impacto negativo, a curto prazo, na produtividade do desenvolvimento de software, já que a definição e implementação de testes para o sistema pode ser bastante trabalhosa. Visando permitir um ganho de tempo, a curto prazo, no desenvolvimento dos sistemas Web, apresentamos neste capítulo o desenvolvimento de dois geradores de código. Estes geradores tem como objetivo amenizar o impacto na produtividade causado pela realização de testes em sistemas Web, em particular, os de aceitação escritos em WSat.

O primeiro gerador de código aqui apresentado trata da geração de código de teste WSat, a partir de protótipos de interfaces com o usuário em HTML. Além de possuir informações úteis para a descrição de casos de teste de aceitação, protótipos de interface em HTML possuem um baixo custo de desenvolvimento e normalmente precisam ser desenvolvidos para validar os requisitos do sistema. Este gerador de código de teste permite um ganho de tempo durante a descrição dos casos de teste, aproveitando o esforço feito para escrever o protótipo.

O segundo gerador de código apresentado tem como objetivo aumentar a produtividade com a geração de trechos de código do sistema a ser desenvolvido e testado. Para isto, foram estudados os trechos de códigos de sistemas Web que podem ser gerados automaticamente a partir de programas de teste escritos em WSat. Entre os códigos gerados, estão códigos Java [19], *templates* HTML e arquivos de configurações do sistema. Além de apresentar este estudo, mostramos também o projeto e implementação deste gerador de código de sistemas.

4.1 Ambiente de Execução de WSat

No capítulo anterior, apresentamos a linguagem WSat, utilizada para a descrição de casos de teste de aceitação. Para podermos avaliar a linguagem proposta e tornar sua utilização viável, se faz necessária a construção de um ambiente de execução para a mesma.

Esta seção apresenta inicialmente o estudo realizado para escolha da técnica de implementação deste ambiente. Em seguida, mostramos o projeto e implementação do ambiente de acordo com a técnica escolhida. Por fim, mostramos como executar o ambiente criado para a execução de programas WSat.

4.1.1 Técnica de Implementação

Antes de iniciar o desenvolvimento deste ambiente de execução, algumas técnicas de implementação para o mesmo foram analisadas. As técnicas estudadas e o resultado da análise das mesmas são vistas a seguir.

Compilação de WSat para uma outra Linguagem de Teste

Inicialmente foi pensado em traduzir programas WSat para programas escritos em uma outra linguagem de teste existente, já que esta abordagem é utilizada no desenvolvimento da linguagem de teste *TestTalk* [28]. Esta abordagem simplifica a implementação do ambiente de execução de WSat, pois não torna-se necessário implementar a execução dos comandos e expressões da linguagem. O próprio ambiente de execução da linguagem de teste destino é responsável pela execução dos programas WSat traduzidos. Outro benefício obtido com esta tradução de WSat é a possibilidade de se criar um ambiente extensível que permite a tradução de WSat para outras linguagens de teste. Isto torna WSat portável, em princípio, para qualquer ferramenta de teste existente que possua linguagem própria e seja suficientemente expressiva.

Para tornar publicamente acessível um ambiente de execução de WSat que traduza seus programas para uma outra linguagem de teste, faz-se necessário a utilização de uma linguagem destino de domínio público. Além disto, o ambiente de execução da linguagem destino deve poder interagir com o ambiente de WSat, escrito em Java, para que a tradução de WSat e a execução do programa traduzido sejam realizados, para efeito do testador, a partir de uma única ferramenta. Devido a estas restrições, nossa única opção de linguagem destino encontrada foi a linguagem de teste da ferramenta JXWeb [26], vista na Seção 2.3.

Através da implementação de um pequeno protótipo de ambiente de execução baseado nesta abordagem, pudemos notar que a linguagem de teste da ferramenta JXWeb ainda está em um estágio inicial de desenvolvimento, criando muitas limitações para a execução dos programas WSat. Desta forma, foi descartada a hipótese do desenvolvimento do ambiente de execução de WSat através de sua tradução para uma outra linguagem de teste.

Interpretação de WSat

A segunda técnica estudada para implementar um ambiente de execução para programas WSat foi a criação de um interpretador. Para implementar este interpretador, faz-se necessário implementar a execução dos comandos e expressões específicos para testes de sistemas Web, como por exemplo, a requisição de uma página Web e o teste de componentes Web, como páginas Web, formulários HTML e imagens. Também é necessário implementar a execução dos comandos e expressões normalmente encontradas em linguagens de programação, como por exemplo, declarações de variáveis, atribuições e chamadas de procedimentos. Esta abordagem foi considerada a que requer o maior esforço de implementação, sendo então descartada devido ao escopo deste trabalho que é mostrar que podemos executar WSat e implementar protótipos de ferramentas essenciais para a utilização de WSat na prática.

Compilação de WSat para Java

A terceira técnica estudada foi a compilação de programas WSat para a linguagem de programação Java. Esta técnica de implementação é, na verdade, uma composição das duas técnicas vistas anteriormente. A idéia principal desta abordagem é que os comandos e expressões de WSat que têm similares na linguagem de programação Java, como atribuições, declarações de variáveis e de tipos, sejam simplesmente traduzidos para código Java equivalente. O mesmo não ocorre com os comandos e expressões de WSat específicos para a realização de testes de sistemas Web, como por exemplo, a requisição de páginas Web e a simulação de cliques em *links* HTML, pois eles não são suportados diretamente pela linguagem Java. Entretanto, a tradução destes comandos pelo ambiente de execução de WSat pode gerar código Java que use APIs externas e que implementam estas funcionalidades.

Esta abordagem foi a escolhida neste trabalho por se mostrar a mais viável, já que o esforço necessário para implementação do ambiente de execução de WSat através desta técnica é menor do que o esforço encontrado na implementação de um interpretador. Além disto, são conhecidas APIs Java que implementam as funcionalidades necessárias para a realização de testes em sistemas Web, dando assim suporte total a tradução de programas WSat.

Para finalizar o estudo sobre a implementação do ambiente de execução de WSat, verificamos a necessidade de se criar uma árvore sintática para a linguagem. Embora a criação de uma árvore sintática torne o processo de compilação mais lento, esta foi a abordagem escolhida já que, com a criação da árvore, o *parser* WSat pode ser reutilizado nas outras ferramentas mostradas no decorrer deste capítulo, reduzindo assim o esforço de implementação das mesmas.

4.1.2 Um *Parser* para WSat

O primeiro passo realizado para a implementação do ambiente de execução de WSat foi a criação de um *parser* para a linguagem. O objetivo deste *parser* é gerar uma árvore sintática que represente um dado programa WSat. Durante esta seção, mostramos a árvore sintática definida para programas WSat e a implementação de um *parser* para geração da mesma.

A Árvore Sintática de WSat

Para criar o *parser* de WSat, foi necessário definir a árvore sintática para os programas WSat. A definição desta árvore foi baseada na gramática de WSat, vista no Apêndice A. Para cada comando ou expressão encontrada na gramática de WSat foi criada uma classe Java para representá-lo. Podemos ver parte da hierarquia de classes utilizadas para representar a árvore sintática da linguagem na Figura 4.1.

As classes `Literal`, `FunctionCall` e `JavaCode`, por exemplo, representam respectivamente expressões literais, chamadas de funções de teste e trechos de código Java embutido. Já as classes `TestCaseDeclaration`, `WebPageDeclaration` e `WebLinkDeclaration` representam respectivamente declarações de casos de teste, páginas Web e *links*.

Como podemos ver pela existência do método `accept`, a classe `TestComponent` estrutura os nós da árvore seguindo o padrão de projeto *Visitor* [16]. Este padrão de projeto

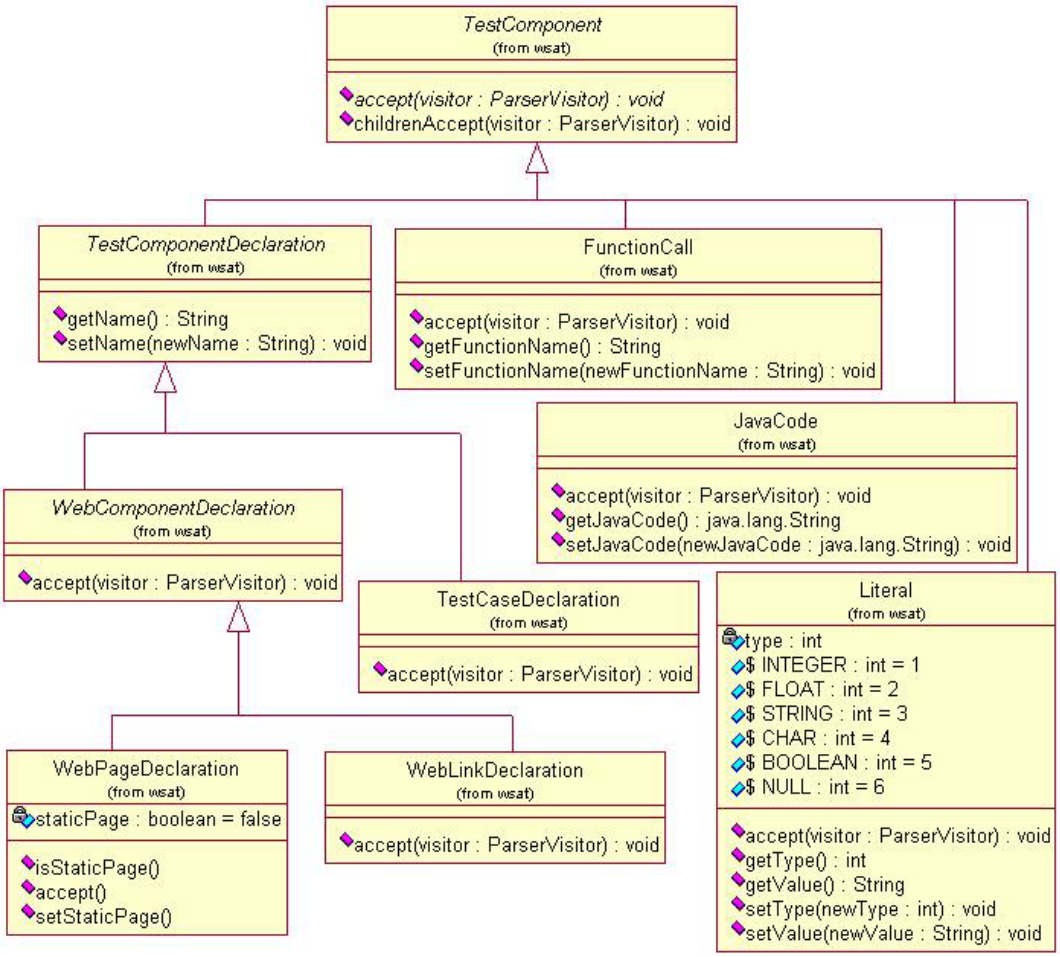

Figura 4.1: Classes que representam nós da árvore sintática de WSat.

permite separar em diferentes classes (*visitors*) os códigos das operações que manipulam (visitam) as árvores sintáticas. Nas classes da árvore em si, ficam as operações que implementam o comportamento básico de uma árvore sintática. Esta separação é muito importante neste trabalho considerando o grande número de operações definidas que atuam sobre a árvore sintática de WSat. Para a implementação deste padrão, definimos o tipo `ParserVisitor` visto na Figura 4.2 para representar o componente Visitor. Pelo grande número de métodos existentes na classe e por serem similares, apenas alguns destes métodos são mostrados.

Esta classe define um método `visit` para cada classe que representa um nó da árvore. Estes métodos são invocados pelos próprios nós da árvore sendo visitada, através do método `accept` definido de forma abstrata na classe `TestComponent`. Cada subtipo de `TestComponent` deve implementar este método de acordo com o código a seguir:

```
public void accept(ParserVisitor visitor) {
    visitor.visit(this);
}
```

Este método, como podemos ver, invoca um dos métodos `visit` declarados no *visitor*. Como o ambiente de execução de Java escolhe o método `visit` cujo parâmetro é mais

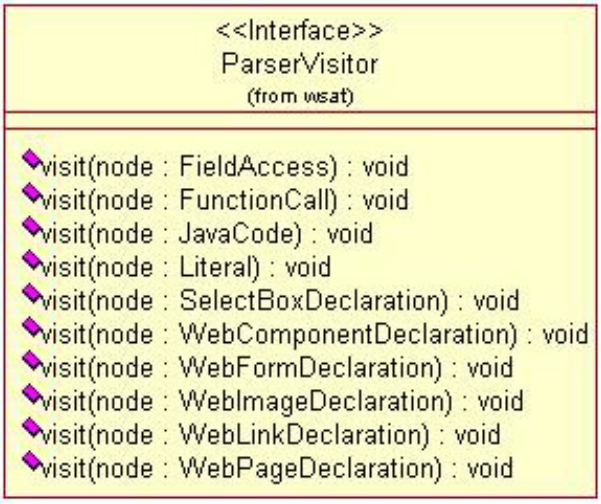

Figura 4.2: Interface definida para os *visitors* das árvores sintáticas de WSat.

próximo do tipo do argumento recebido pelo método, o método `visit` invocado será o que tem como tipo de parâmetro a classe que declarou o método `accept`. Para percorrer uma árvore através de um *visitor*, deve-se invocar o método `accept` da raiz da árvore passando como parâmetro a referência do *visitor*, como visto no exemplo a seguir:

```
noRaiz.accept(visitor);
```

O método `accept` visto irá invocar o método `visit` correspondente ao nó raiz. A partir daí, o *visitor* pode percorrer os nós filhos do filho raiz invocando-se o método `childrenAccept` definido na classe `TestComponent`. A implementação deste método é vista a seguir.

```
public void childrenAccept(ParserVisitor visitor) {
    if (children != null) {
        for (int i = 0; i < children.length; ++i) {
            ((TestComponent) children[i]).accept(visitor);
        }
    }
}
```

Este método invoca o método `accept` em cada um dos nós filho, passando como parâmetro a referência do `visitor`. Como tipo da variável `children` é *array* de `Node`, é necessária a utilização do *cast* visto acima para a invocação do método `accept`.

Como exemplo de implementação de *visitor*, definimos a classe `ParserVisitorPrettyPrinter` que, a partir de uma árvore sintática, escreve na saída *default* o programa WSat correspondente a mesma (operação *pretty-printing*). A implementação do método `visit` invocado quando o *visitor* percorre uma árvore sintática e encontra um nó do tipo `FunctionCall` é vista a seguir.

```
public class ParserVisitorPrettyPrinter implements ParserVisitor {
    public void visit(FunctionCall node) {
        System.out.println(node.getName() + "(");
        node.childrenAccept(this);
        System.out.println(")");
    }
    ...
}
```

Este método imprime o nome da função, obtido através do método `getName`, e os seus parâmetros, ao percorrer seus nós filhos através do método `childrenAccept`. O resultado da execução deste *visitor* em uma árvore sintática é a escrita de um programa WSat equivalente ao original que gerou a criação da árvore.

Implementação do *Parser*

Para a implementação do parser de WSat, foi utilizada a ferramenta para geração de compiladores JavaCC [35]. Esta ferramenta gera classes Java a partir da gramática de uma linguagem escrita em um formato próprio da ferramenta. Estas classes realizam o *parser* em arquivos textos, que devem ser escritos de forma a satisfazer a gramática declarada. Para a criação do parser de WSat, utilizamos ainda o pré-processador JJTree que acompanha a ferramenta JavaCC. Este pré-processador permite definir mais facilmente os nós da árvore sintática a serem criados para cada termo da gramática. A árvore definida não é tão otmizada, devendo ser revista no caso de um ambiente real. No código mostrado a seguir, por exemplo, podemos ver o trecho da gramática de WSat que define os literais da linguagem.

```
01:     ...
02:     < INTEGER: ["1"-"9"] (["0"-"9"])* >
03:     < STRING: ... >
04:     ...
05:     void Literal() :
06:     { Token token;  }
07:     {
08:         (     token = <STRING>
09:                     { jjtThis.setType(Literal.STRING); }
10:             |
11:               token = <INTEGER>
12:                     { jjtThis.setType(Literal.INTEGER); }
13:             | ...
14:         )
15:         { jjtThis.setValue(token.toString()); }
16:     }
```

De maneira geral, os termos da gramática têm o mesmo nome que os tipos dos nós correspondentes da árvore sintática, e suas declarações consistem de um bloco de declaração de variáveis e um bloco de declaração do termo da gramática representado. Na linha

5, podemos ver a indicação do tipo `Literal` como o nó a ser criado para representar literais. Na linha 6, vemos a declaração da variável `token` do tipo `Token`, da API de JavaCC, utilizado para referenciar o *token* casado por este termo da gramática. Nas linhas 8 e 11, vemos o uso das constantes < `STRING` > e < `INTEGER` >, definidas previamente nas linhas 2 e 3, indicando as expressões regulares que definem os termos literais. Pode-se também definir código Java para ser executado quando o *parser* encontrar um dado termo no arquivo de entrada, como visto nas linhas 9, 12 e 15.

A declaração de um termo pode ser feita através da combinação de outros termos, como na declaração do termo de atribuição da gramática de WSat:

```
void Assignment() :
{}
{
    AssignableAccess() "=" Expression()
}
```

A declaração mostrada indica que o termo de atribuição é composto pelo termo `AssignableAccess`, o qual representa acessos a variáveis e propriedades, seguido pelo símbolo " = " (sem aspas), e por uma expressão definida pelo termo `Expression`. Estas declarações compostas indicam ao *parser* gerado pela ferramenta JavaCC como criar a árvore sintática para um dado texto de entrada. Durante o casamento de atribuições em programas WSat, por exemplo, o *parser* gerado cria uma subárvore para representar cada atribuição com raiz do tipo `Assignment` e nós filhos do tipo `AssignableAccess` e `Expression`.

Ao executar as ferramentas JJTree e JavaCC, são geradas as classes que compõem o parser para a gramática indicada e os tipos `Node` e `SimpleNode`, que são utilizados como base para a definição dos tipos da árvore sintática da linguagem. Desta forma, os tipos mostrados anteriormente para representar a árvore sintática de WSat fazem parte da hierarquia de nós de árvore definida pelos tipos `Node` e `SimpleNode`, como visto na Figura 4.3.

A classe gerada para o parser possui um conjunto de métodos para auxiliar a leitura do arquivo de entrada e um método criado para cada termo da gramática. Para executar este *parser*, basta chamar um dos métodos de leitura do arquivo de entrada e o método correspondente ao termo principal da gramática, como vemos no exemplo a seguir da gramática de WSat:

```
...
ReInit(stream);
Node node = ProgramaWSat();
```

O método `ReInit` visto no exemplo é criado automaticamente pelo JavaCC e recebe como parâmetro um objeto do tipo `java.io.InputStream`, utilizado para leitura do arquivo de entrada. Já o método `ProgramaWSat`, criado a partir do termo principal da gramática de WSat, executa o *parser* no arquivo de entrada e retorna o nó raiz da árvore sintática construída para este arquivo.

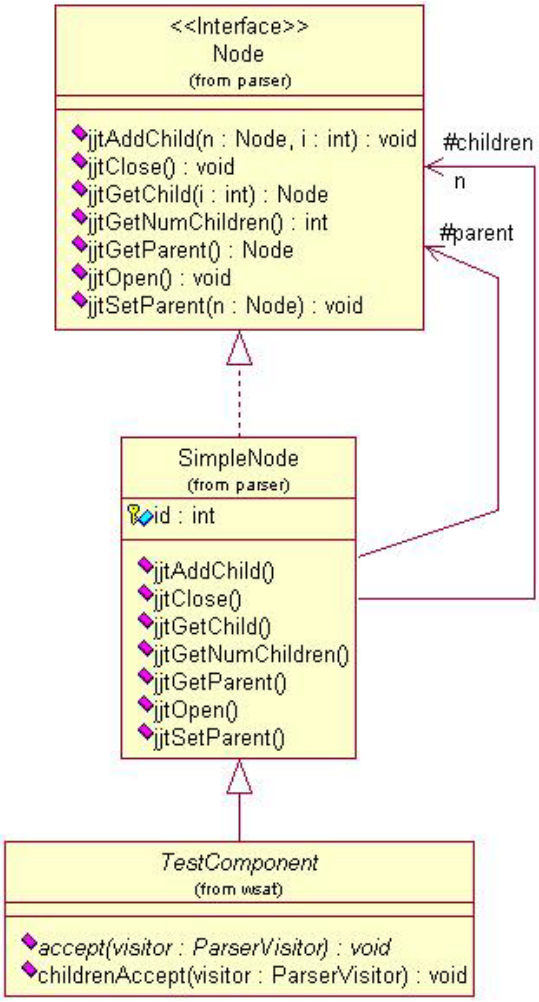

Figura 4.3: Topo da hierarquia de classes da árvore sintática de WSat.

4.1.3 Compilando Código WSat para Código Java

Para a implementação do compilador de WSat para Java, foi criado o *visitor* `WSatCompilerVisitor`. Este *visitor* percorre a árvore sintática de um programa WSat criando classes Java que implementam os casos de teste encontrados neste programa. Inicialmente são gerados os arquivos de extensão "`.java`" com código Java. Em seguida, estes arquivos são compilados pelo próprio ambiente de Java para arquivos de extensão "`.class`" com *bytecodes* executáveis [19]. Nesta seção, apresentamos incrementalmente a criação do código Java a partir de programas WSat. São utilizados, como exemplo, os tipos, casos e funções de teste definidos no Capítulo 3.

Comandos e Expressões Válidos em Java

Para a compilação de comandos WSat que possuem sintaxe e semântica equivalentes em Java, como, por exemplo, atribuições, declarações de variáveis e chamadas de métodos, podemos fazer simplesmente uma cópia do código WSat para o código Java. O mesmo é válido para código Java embutido. Por exemplo, vejamos o seguinte trecho de código

com comandos WSat:

```
codigo = 200;
String palavras = "ufpe";
int indexValue;
```

A tradução destes comandos e declarações para Java é simplesmente uma cópia dos mesmos.

Para esta forma de tradução ser válida também para códigos que manipulem tipos WSat, cada tipo WSat é representado por uma classe Java de mesmo nome. A compilação de tipos WSat é detalhada em seção adiante. A seguir mostramos declarações envolvendo os tipos `PagResposta` e `PagInicial`, definidos para testar o sistema de busca.

```
PagInicial pag;
PagResposta pagResp;
```

A tradução destas declarações para Java também é trivial, realizada através de uma simples cópia.

Além da declaração de variáveis, comandos que manipulam variáveis de tipos WSat, como, por exemplo, no acesso a propriedades e invocação de serviços, também são válidos em Java e, portanto, traduzidos diretamente. A seguir, mostramos um exemplo com a manipulação da variável `pag`:

```
FormBusca form = pag.formBusca;
form.palavras.value = "ufpe";
WebPage resp = form.submit();
```

O código mostrado é válido já que, como indicamos adiante, as propriedades e serviços de tipos WSat são representados por atributos e métodos públicos correspondentes nas classes Java que representam estes tipos.

Para compilar todos estes comandos e expressões válidos em ambas as linguagens, o *visitor* para compilação de WSat atua de forma semelhante ao *visitor* `ParserVisitor-PrettyPrinter` mostrado anteriormente.

Tipos Definidos em WSat

Como vimos na seção anterior, é necessário representar os tipos WSat através de classes Java. Desta forma, para cada definição de tipo WSat é criada uma classe Java de mesmo nome para representá-lo. O tipo predefinido `WebPage`, por exemplo, é representado pela classe Java de mesmo nome, como visto na Figura 4.4.

Como podemos notar na figura, as propriedades dos tipos WSat são representadas por atributos públicos de mesmo nome e tipo em suas respectivas classes Java. Já os serviços dos tipos WSat são representados por métodos públicos na classe Java. Isto facilita, como foi discutido, a compilação de comandos e expressões WSat equivalentes em Java. A seguir, mostramos a definição do tipo `Palavras` vista no capítulo anterior:

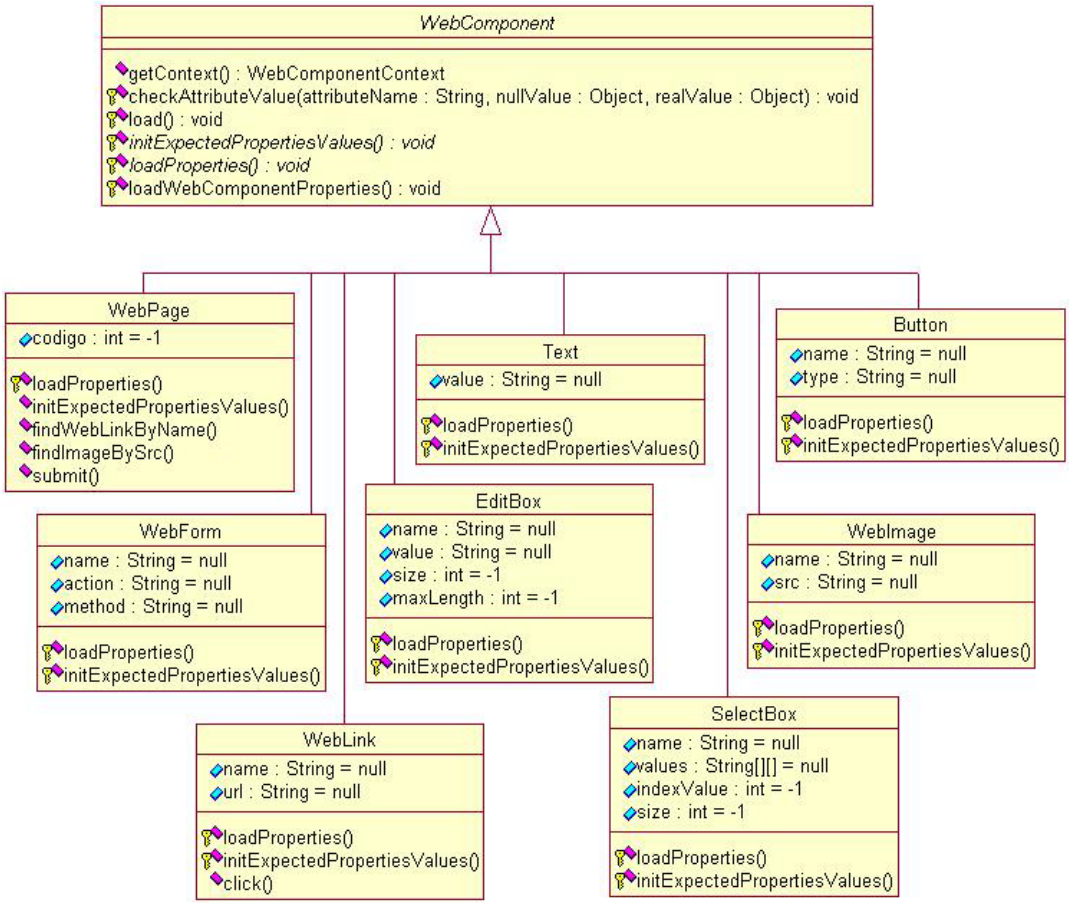

Figura 4.4: Classes Java representando os tipos WSat predefinidos.

```
EditBox Palavras {
    name = "palavras";
    value = "";
    size = 10;
    maxLength = 10;
}
```

A compilação de tipos WSat definidos pelo testador cria uma classe Java de mesmo nome que herda da classe que representa o seu supertipo WSat. A compilação do tipo WSat `Palavras` cria a classe Java `Palavras` mostrada a seguir:

```
public class Palavras extends EditBox {
    ...
}
```

Como podemos ver, a classe Java gerada herda da classe Java `EditBox` que representa o tipo WSat `EditBox`, supertipo do tipo WSat `Palavras`. Para implementar a geração desta classe Java pela compilação do tipo `Palavras`, definimos o método `generateClass` da classe `WSatCompilerVisitor`. Este método é invocado ao se visitar nós que representam a definição de componentes Web O método `generateClass` é utilizado para

escrever o código de uma classe Java e, como visto a seguir, recebe como parâmetro o nó visitado.

```
01: private void generateClass(WebComponentDeclaration node) {
02:     String superClass = node.getSuperClassName();
03:     String name = node.getName();  ...
04:     write("public class " + name);
05:     writeLine(" extends " + superClass + " {");
06:     ...
07:     writeLine("}");  ...
08: }
```

Como podemos ver, através do parâmetro `node` obtemos as informações necessárias para a geração da classe Java, como o nome do tipo definido e de seu supertipo. Os métodos `write` e `writeLine` são utilizados para escrever em memória, através de um objeto `StringBuffer`, o código Java da classe sendo gerada. São abstraídos aqui os detalhes sobre a implementação da indentação do código gerado. Além disto, os trechos denotados por "..." são mostrados mais adiante.

Já o código de inicialização das propriedades de tipos primitivos ou `String` definidos pelos tipos especiais predefinidos em WSat são compilados gerando o método `initExpectedPropertiesValues`. A seguir podemos ver a classe Java gerada para o tipo WSat `Palavras` com a definição deste método de inicialização.

```
public class Palavras extends EditBox {
    protected void initExpectedPropertiesValues() {
        name = "palavras";
        value = "";
        size = 10;
        maxLength = 10;
    } ...
}
```

Como podemos ver, o código de inicialização das propriedades herdadas do tipo WSat `EditBox` são copiados para o corpo do método `initExpectedPropertiesValues`. Este método `initExpectedPropertiesValues` é definido de forma abstrata na classe `WebComponent` e é invocado, como mostrado adiante, durante a execução do construtor desta classe.

Para gerar o código de inicialização das propriedades como mostrado, temos o seguinte trecho de código na linha `6` da definição do método `generateClass`.

```
01: writeLine("public void initExpectedPropertiesValues() {");
02: Node childNode;
03: WebAttributeInitialization attrib;
04: int childNumber = node.jjtGetNumChildren();
05: for (int i = 0; i < childNumber; i++){
06:     childNode = (Node) node.jjtGetChild(i);
07:     if (childNode instanceof WebAttributeInitialization) {
```

```
08:          attrib = (WebAttributeInitialization) childNode;
09:          write(attrib.getName() + " = ");
10:          attrib.childrenAccept(this);
11:          writeLine(";");
12:      }
13: }
14: writeLine("}");
```

O código mostrado visita os nós filhos do nó atual da árvore verificando se os mesmos são do tipo `WebAttributeInitialization` (linhas `4` a `7`), ou seja, se são inicializações de propriedades herdadas dos subtipos especiais de WSat. Em caso afirmativo, a inicialização é escrita no corpo do método `initExpectedPropertiesValues` através das linhas `8` a `11`. O método `childrenAccept` visto na linha `10` faz com que o *visitor* visite o nó que representa o valor atribuído à propriedade, escrevendo o valor do mesmo no código Java.

Durante a definição de tipos WSat, podemos definir propriedades a partir de tipos WSat. Um exemplo disto é a declaração da propriedade `palavras`, a partir do tipo `Palavras`, no tipo WSat `FormBusca`:

```
WebForm FormBusca {
    Palavras palavras;
}
```

Quando os tipos WSat definidos pelo testador possuem propriedades declaradas, elas são representadas através de atributos públicos de mesmo nome e tipo nas classes Java geradas para estes tipos. A seguir, vemos a classe Java gerada pela declaração do tipo WSat `FormBusca`:

```
public class FormBusca extends WebForm {
    Palavras palavras;
    protected void initExpectedPropertiesValues() {
    } ...
}
```

Para implementar esta compilação, temos o código mostrado a seguir também denotado pela linha `6` da definição do método `generateClass`:

```
node.childrenAccept(this);
```

Este código faz com que o *visitor* visite os nós filhos do nó que representa a declaração de tipo. Visitas a nós do tipo `WebAttributeInitialization` não realizam mais nenhum processamento, pois o código Java correspondente já foi escrito como mostrado anteriormente. Já a visita a nós do tipo `PropertyDeclaration`, o seguinte método `visit` é executado.

```
public void visit(PropertyDeclaration node) {
    writeAttribute(node.getType(), node.getName());
}
```

O método `writeAttribute` invocado recebe como parâmetro o tipo e nome de um atributo público a ser criado na classe Java sendo gerada. Como podemos ver, a execução deste método escreve a declaração de um atributo público de mesmo tipo e nome da propriedade declarada.

Além do código mostrado, cada classe Java gerada para representar um tipo WSat possui um construtor, cujos detalhes são vistos na Seção 4.1.3.

Tipos Anônimos

Ao se definir propriedades em tipos WSat, podemos definir tipos auxiliares WSat de forma anônima. Vejamos a seguir a declaração do tipo WSat `PagResposta` que define a propriedade `textoNomeSistema` representando um componente texto na página Web.

```
WebPage PagResposta {
    Text {
        value = "Sistema de Busca";
    } textoNomeSistema;
}
```

Tipos WSat anônimos também devem ser representados por classes Java para que possam ser utilizados na declaração dos atributos destas classes. Quando compilados, os tipos WSat anônimos geram classes Java com nomes iguais ao de seus supertipos acrescido de um sufixo gerado automaticamente. A classe Java que representa o tipo anônimo declarado pela propriedade `textoNomeSistema` do tipo `PagResposta` é vista a seguir:

```
public class Text_AUX_1 extends Text {
    protected void initExpectedPropertiesValues() {
        value = "Sistema de Busca";
    }
}
```

Como podemos ver, o sufixo "`_AUX_1`" foi adicionado ao nome do supertipo `Text` para formar o nome do resultado da compilação do tipo WSat anônimo. Existem diversas outras formas para se gerar automaticamente um nome aleatório para as classes que representam tipos anônimos. A abordagem mostrada foi utilizada por ser simples e satisfatória, similar à usada nas classes anônimas de Java, não houvendo entretanto um estudo sobre as outras possibilidades existentes. Durante a compilação de tipos anônimos, o seguinte método `visit` é executado:

```
public void visit(AnonymousPropertyDeclaration node) {
    String className = generateAnonymousClass(node);
    writeAttribute(className, node.getName());
}
```

Este método invoca o método `generateAnonymousClass`, cuja implementação é similar ao método `generateClass`, para gerar o código da classe que representa o tipo anônimo e retornar o nome da mesma. O método `writeAttribute` é invocado em seguida para gerar a declaração do atributo que representa a propriedade declarada de forma anônima.

Como podemos perceber, os métodos `generateClass`, `generateAnonymousClass` e `writeAttribute` adicionam código Java na classe sendo gerada. Para a execução correta do método `generateAnonymousClass`, faz-se necessária a manipulação de uma pilha de classes sendo geradas, ao invés de uma única classe. Esta pilha fornece suporte para que classes que representam tipos WSat anônimos sejam geradas durante a geração da classe que representa o tipo WSat com propriedades definidas de forma anônima. Dessa forma, a implementação do método `write` utiliza esta pilha:

```
private void write(String code) {
    StringBuffer sbuffer = (StringBuffer) sbufferStack.peek();
    sbuffer.append(code);
}
```

Este método escreve no `StringBuffer` posicionado no topo da pilha `sbufferStack`. A implementação do método `writeLine` é realizada de forma similar. Já o método `generateClass`, como visto a seguir, faze uso dos métodos `start` e `finish` para iniciar e finalizar a escrita de uma nova classe Java.

```
private void generateClass(WebComponentDeclaration node) {
    String superClass = node.getSuperClassName();
    String className = node.getName();
    start(className);
    ...
    finish();
}
```

Estes métodos são utilizados para manipular na pilha de classes o objeto do tipo `StringBuffer` que armazena o código da classe sendo gerada:

```
private void start(String className) {
    StringBuffer sbuffer = new StringBuffer();
    sbufferStack.push(sbuffer);
    classes.put(className, sbuffer);
}
private void finish() {
    sbufferStack.pop();
}
```

O método `start` é responsável por criar uma nova instância de `StringBuffer`, colocá-la no topo da pilha e inseri-la no mapeamento `classes` para posteriormente ser escrita em um arquivo Java. Já o método `finish` é responsável por desempilhar esta instância.

Comando `getWebPage`

Para compilar comandos `getWebPage` para Java, faz-se necessário a implementação em Java da funcionalidade deste comando. Para isto, utilizamos a API HttpUnit [18] que fornece a implementação necessária para a simulação das ações realizadas por usuários de sistemas Web. Por exemplo, podemos ver no código mostrado a seguir o uso do método estático `getWebConversation` da classe `TestSystem`.

```
WebConversation conv = TestSystem.getWebConversation();
```

O projeto deste método foi guiado pelo padrão de projeto *Singleton* [16] e seu objetivo é manter uma única instância da classe `WebConversation` no sistema. Esta classe, da API HttpUnit, representa a comunicação entre programas clientes e servidor Web. Instâncias desta classe são capazes, por exemplo, de se comunicar com servidores Web através do protocolo HTTP e gerenciar os *cookies* utilizados na conexão. Para requisitar uma página Web a partir de sua URL, podemos utilizar o código a seguir.

```
GetMethodWebRequest request = new GetMethodWebRequest(url);
WebResponse response = conv.getResponse(request);
WebPage pag = new WebPage(new WebComponentContext(response));
```

A classe `GetMethodWebRequest`, também de HttpUnit, representa uma requisição de página Web pelo método `GET` e recebe em seu construtor a URL da página requerida. Criada uma instância deste tipo, podemos utilizá-lo no método `getResponse` da classe `WebConversation`. Este método recebe como parâmetro uma requisição de página Web, representada pela variável `request`, realiza a comunicação com o servidor Web indicado na URL de requisição e retorna um objeto do tipo `WebResponse` que representa o conteúdo da página retornada. A partir deste objeto temos acesso a todo o conteúdo da página Web representada, como imagens, *links*, e textos existentes. Na última linha do código mostrado, podemos ver o uso do construtor da classe `WebPage` que recebe como parâmetro um objeto do tipo `WebComponentContext`. Os detalhes sobre este construtor são mostrados na próxima seção. Para estruturar o código utilizado para implementar o comando `getWebPage`, definimos a classe `getWebPage`:

```
public class getWebPage {
    public static WebPage execute(String url) {
        WebConversation conv = TestSystem.getWebConversation();
        GetMethodWebRequest request = new GetMethodWebRequest(url);
        WebResponse response = conv.getResponse(request);
        return new WebPage(new WebComponentContext(response));
    }
}
```

Através da implementação em Java apresentada para o comando `getWebPage`, invocações deste comando como `getWebPage(urlAcesso)` podem ser traduzidas para:

```
getWebPage.execute(urlAcesso)
```

Como podemos ver, invocações do comando WSat `getWebPage` são compilados simplesmente para invocações do método `execute` da classe `getWebPage`.

Transformação de Tipo em Componentes Web

Para a compilação da operação de transformação de tipo de WSat não podemos utilizar, por exemplo, a operação de *cast* de Java, pois a sua semântica é diferente. A operação

de transformação de tipo necessita criar um novo objeto de tipo mais específico, diferentemente da operação de *cast* de Java, que trabalha com o mesmo objeto mas permitindo o acesso aos atributos e métodos definidos no tipo mais específico. O objeto criado pela operação de transformação de tipo representa, de forma mais detalhada, o mesmo componente Web representado pelo objeto original. Além das diferenças mencionadas, a operação de *cast* testa se o objeto é do tipo utilizado no *cast*, enquanto a implementação da operação de transformação de tipo testa se o objeto representa um componente Web que satisfaz às características determinadas pelo tipo utilizado na operação.

Para entender a implementação adotada para a transformação de tipo, é necessário o entendimento de como os objetos Java que representam componentes Web são criados e carregados com as propriedades de seus respectivos componentes Web. As informações sobre componentes Web são representadas por objetos da API HttpUnit [18]. Para agrupar tais objetos foi criada a classe `WebComponentContext`. Esta classe representa um contexto que armazena, entre outras coisas, o código fonte da página Web onde o componente Web representado está inserido. A classe `WebComponentContext` possui um construtor que recebe como parâmetro um objeto do tipo `WebResponse` que permite o acesso às informações sobre a página Web representada.

Definimos na classe `WebComponent` um construtor que recebe como parâmetro uma instância da classe `WebComponentContext`. Objetos que representam componentes Web são criados com o uso deste construtor. A seguir vemos o construtor da classe `WebPage`.

```
public WebPage(WebComponentContext context) {
    super(context);
}
```

Este construtor invoca o construtor de seu supertipo, no caso o construtor de `WebComponent`, cujo código é visto a seguir.

```
public WebComponent(WebComponentContext context) {
    this.context = context;
    load();
}
```

Como podemos ver, este construtor armazena a referência para o contexto do componente e, em seguida, invoca o método `load`. Este método é responsável por carregar as propriedades do componente Web representado, além de testar se estas proprieades satisfazem às restrições definidas pelo tipo WSat associado. O código do método `load` pode ser visto a seguir:

```
private void load() {
    initExpectedPropertiesValues();
    loadAndCheckPropertiesValues();
    loadWebComponentProperties();
}
```

Como podemos ver, o método `load` é composto pela invocação sequencial dos métodos `initExpectedPropertiesValues`, `loadAndCheckPropertiesValues` e `loadWebComponentProperties`. O método `initExpectedPropertiesValues`, como visto anteriormente, é responsável por carregar os atributos do objeto com os valores determinados pelas propriedades do tipo WSat associado. O método `loadAndCheckPropertiesValues`,

por sua vez, é responsável por carregar os atributos do objeto com os valores encontrados no componente Web representado, verificando se estes valores encontrados satisfazem os seus valores esperados. Já o método `loadWebComponentProperties` carrega e testa as propriedades definidas pelo testador.

O método `loadAndCheckPropertiesValues` é declarado como abstrato na classe `WebComponent`, sendo implementado pelos subtipos especias predefinidos em WSat. Para facilitar o desenvolvimento destes códigos de teste, foi definido o método `checkAttributeValue` na classe `WebComponent` como visto a seguir.

```
01: protected void checkAttributeValue(String attributeName,
02:                          Object defaultValue, Object realValue) {
03:     try {
04:         Field attrib = this.getClass().getField(attributeName);
05:         Object value = attrib.get(this);
06:         boolean valueEqualsDefaultValue = false;
07:         if (defaultValue == null) {
08:             valueEqualsDefaultValue = (value == null);
09:         } else {
10:             valueEqualsDefaultValue =
11;                                     defaultValue.equals(value);
12:         }
13:         if (valueEqualsDefaultValue) {
14:             attrib.set(this, realValue);
15:         } else {
16:             if (!value.equals(realValue)) {
17:                 throw new DataTestValidationException(...);
18:             }
19:         }
20:     } catch (Exception e) {
21:         throw new TestExecutionException(e);
22:     }
23: }
```

Este método utiliza as classes de Java contidas no pacote `java.lang.reflect` [19] para manipular atributos de objetos. Através do nome do atributo, indicado pelo parâmetro `attributeName`, podemos acessar o valor deste atributo (linhas `4` e `5`) e compará-lo a seu valor *default*, indicado pelo parâmetro `defaultValue`. Caso o valor do atributo seja igual ao valor *default*, simplesmente atribuimos o valor encontrado (parâmetro `realValue`) a este atributo (linha `14`). Caso contrário, fazemos a comparação do valor esperado com o valor encontrado (linha `16`). Se estes valores não forem iguais, o teste não é satisfeito e a exceção `DataTestValidationException` é levantada (linha `18`). Os detalhes sobre tratamento de exceções durante a execução dos testes são vistos mais adiante.

Com a definição do método `checkAttributeValue`, podemos implementar o método `loadAndCheckPropertiesValues` de `WebPage` da forma vista a seguir:

```
protected void loadAndCheckPropertiesValues() {
    WebResponse response = getContext().getWebPageInformation();
    int codeFound = response.getResponseCode();
    checkAttributeValue("code", new Integer(-1),
                        new Integer(codeFound));
}
```

A implementação do método `checkAttributeValue` facilita o desenvolvimento dos métodos `loadAndCheckPropertiesValues` de outras classes, permitindo também uma maior facilidade para a inserção de novas propriedades em tipos WSat.

A carga das propriedades de tipos WSat definidas pelo testador é realizada pelo método `loadWebComponentProperties` da classe `WebComponent`. Este método é invocado pelo método `load` visto anteriormente e também utiliza o pacote `java.lang.reflect` de Java, como mostrado a seguir.

```
01: private void loadWebComponentProperties() {
02:     Class[] paramTypes = {WebComponentContext.class};
03:     Object[] args = {getContext()};
04:     try {
05:         Field[] attribs = this.getClass().getFields();
06:         WebComponent comp, realComp;
07:         Constructor constr;
08:         Class componentClass;
09:         for (int i = 0 ; i < attribs.length ; i++) {
10:             componentClass = attribs[i].getType();
11:             if ((WebComponent.class).isAssignableFrom(
12:                                             componentClass)) {
13:                 constr = componentClass.getConstructor(
14:                                             paramTypes);
15:                 realComp = (WebComponent) constr.newInstance(
16:                                             args);
17:                 attribs[i].set(this, realComp);
18:             }
19:         }
20:     } catch (Exception e) {
21:         throw new TestExecutionException(e);
22:     }
23: }
```

Este método obtém a lista de atributos públicos definidos na classe que representa o tipo WSat ou em suas superclasses (linha `5`) e a percorre (linha `9`) verificando, para cada atributo da lista, se o seu tipo é subtipo de `WebComponent` (linhas `10` e `11`). Em caso afirmativo, uma instância do tipo do atributo é criada a partir do contexto do componente Web (linhas `13` e `15`). Ao ser criada, esta instância verifica a existência do componente Web que ela representa dentro do contexto recebido. Por fim, o objeto criado é atribuído ao atributo analisado (linha `15`).

Com a definição dos métodos apresentados, a compilação da operação de transformação de tipo pode ser realizada pela criação do novo objeto a partir do contexto do

objeto já existente. A seguir podemos ver o uso do operador de transformação de tipo para testar se a página Web retornada pela URL armazenada na variável `urlAcesso` satisfaz às características da página inicial do sistema de busca, definidas pelo tipo `PagInicial`.

```
WebPage webPage = getWebPage(urlAcesso);
PagInicial pag = [PagInicial] webPage;
```

A implementação para a compilação desta transformação gera o seguinte código Java.

```
WebPage webPage = getWebPage.execute(urlAcesso);
PagInicial pag = new PagInicial(webPage.getContext());
```

A instanciação da classe `PagInicial` através do contexto utilizado pela variável `webPage` testa se a página Web deste contexto satisfaz às restrições impostas pelo tipo `PagInicial`, levantando uma exceção caso as restrições não sejam satisfeitas. Além da geração deste código, o compilador deveria realizar análises semânticas como, por exemplo, a verificação de que a transformação de tipo só pode ser realizada de um tipo para um de seus subtipos. Por serem bem conhecidas, estas análises não foram implementadas.

Teste do Conteúdo Dinâmico de Componentes Web

Além da transformação de tipos, podemos utilizar comandos como `assert` e `fail` para testar componentes Web. A seguir podemos ver o uso do comando `fail`.

```
fail("Núm. de resposta não encontrado.");
```

O comando `fail` mostrado é compilado para o levantamento da exceção `TestFailedException`, como visto no código Java a seguir.

```
throw new TestFailedException("Núm. de resposta não encontrado.");
```

A exceção `TestFailedException` recebe em seu construtor a mensagem de falha do teste.

O comando `assert` é compilado similarmente ao comando `fail`, verificando entretanto o valor de seus parâmetros para depois levantar a exceção, caso os mesmos sejam diferentes. A seguir, mostramos o uso deste comando para verificar o código de retorno de uma página Web acessada a partir do sistema de busca:

```
assert("Link de Resposta Quebrado.", paginaEncontrada.code, 200);
```

Este comando WSat é compilado para o seguinte código Java:

```
if (paginaEncontrada.code != 200) {
    throw new TestFailedException(
                    "O número de resposta não foi encontrado.");
}
```

Como podemos notar, a exceção `TestFailedException` é levantada apenas quando os parâmetros recebidos pelo comando são diferentes. Caso os valores comparados não sejam de tipos primitivos, o método `equals` e o operador de negação ! são utilizados no lugar do operador !=.

O teste de conteúdo dinâmico de componentes Web também é verificado por serviços como `findTextByValue` e `findAllWebLinkByURL`. Estes serviços de componentes Web poderiam ser representados em Java por métodos de mesma assinatura. Entretanto, eles não fazem parte desta implementação do ambiente de execução de WSat devido ao curto espaço de tempo disponível e por não serem necessários para validar as principais idéias de WSat, como por exemplo, a representação e manipulação de componentes Web através da definição de tipos. Durante a realização de experimentos, o teste dos conteúdos dinâmicos foram realizados de forma estática, criando-se um tipo para cada possibilidade de resposta testada. Na prática, este procedimento é inviável devido ao grande número de tipos necessários para cobrir cada possibilidade de resposta de sistemas reais. A implementação destes comandos pode ser feita com o auxílio da API HttpUnit, sem apresentar maiores problemas.

Declaração e Invocação de Funções de Teste

Programas WSat podem declarar e utilizar diversas funções de teste. A função `buscar` definida na Seção 3.2.3, por exemplo, é vista a seguir:

```
PagResposta buscarPalavras(PaginaInicial pag, String palavras,
                           int indexLugar) {
    FormBusca form = pag.formBusca;
    form.palavras.value = palavras;   ...
}
```

A declaração de uma função de teste em WSat é compilada para Java através da geração de um método estático, de assinatura idêntica a da função WSat correspondente, na classe `TestFunctions` como mostrado a seguir:

```
public class TestFunctions {
    public static PagResposta buscarPalavras(PaginaInicial pag,
                                             String palavras,
                                             int indexLugar) {
        FormBusca form = pag.formBusca;
        form.palavras.value = palavras;   ...
    } ...
}
```

Para a tradução de declarações de funções de teste, são executados os comandos vistos no diagrama de sequência da Figura 4.5. Os métodos `start` e `finish` são utilizados respectivamente para indicar o início e o fim da escrita de uma nova classe Java, cujo nome é indicado como parâmetro do primeiro método. O método `getParameters` retorna a lista de nós que representam os parâmetros formais declarados, os quais são traduzidos ao serem visitados pelo método `accept`. Já o método `getBodyStatements` retorna o

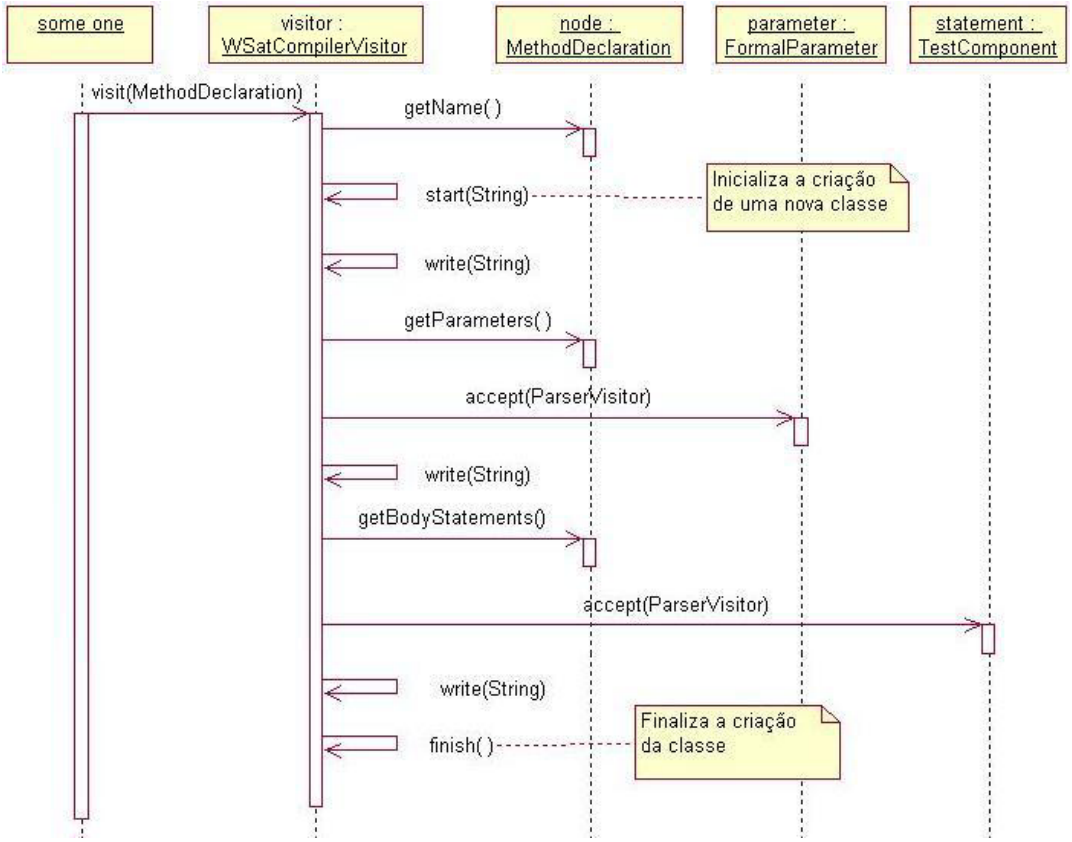

Figura 4.5: Sequência de comandos executados na tradução da declaração de uma função.

nó que representa o corpo da função declarada. Este corpo de função é traduzido pelo *visitor* através do método `accept`.

Já para compilar as invocações de funções de teste, podemos traduzi-las como a chamada do método estático gerado para representar a função de teste invocada. Abaixo podemos ver o código Java gerado para a invocação da função de teste `buscarPalavras(pag,"ufpe",1)`.

```
buscarPalavras.execute(pag, "ufpe", 1)
```

A tradução da invocação de uma função de teste, cujo diagrama de sequência é visto na Figura 4.6, é realizada pelo método `visit` que recebe um parâmetro do tipo `FunctionCall`. O método `getName()` da classe `FunctionCall` retorna o nome da função invocada. Já o método `getArguments()` retorna o nó que representa a lista dos parâmetros passados para a função. A invocação de seu método `childrenAccept` faz com que os nós que representam os parâmetros da função sejam visitados e traduzidos.

Casos de Teste

Além de funções de teste, é necessário traduzir os casos de testes de programas WSat. As declarações de casos de teste são traduzidas de forma similar à tradução das declarações de funções de teste. A seguir podemos ver o caso de teste `buscar` escrito em WSat.

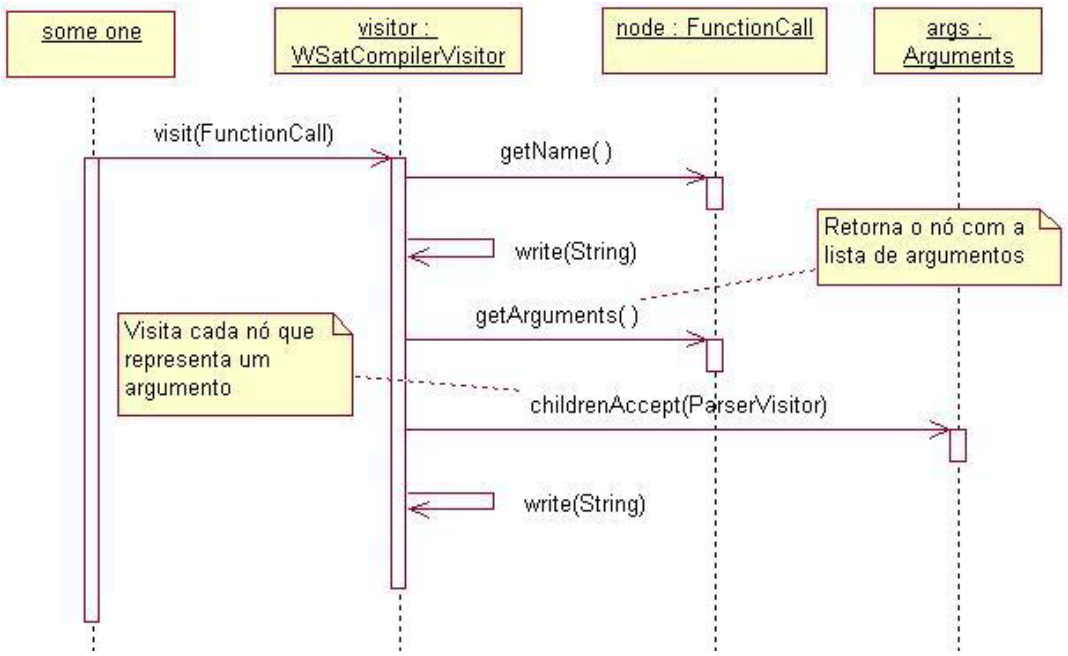

Figura 4.6: Sequência de comandos executados na tradução da invocação de uma função de teste.

```
testCase buscar {
    ...
    PagResposta resp = buscarPalavras(pag, palavra, indexValue); ...
}
```

A tradução de um caso de teste para Java, assim como ocorre com a tradução de uma função de teste, gera uma classe Java de nome igual a do caso de teste. A classe gerada para a definição do caso de teste `buscar` é mostrada a seguir.

```
public class buscar {
    public static void main(String[] args) {
        ...
        PagResposta resp = buscarPalavras(pag,
                                          palavra, indexValue); ...
    }
}
```

Como podemos ver, as classes geradas a partir de definições de casos de testes possuem o método `main`. Este método, com a assinatura vista, é utilizado em Java para iniciar a execução de um programa Java. Desta forma, podemos iniciar a execução dos casos de teste de forma independente, inclusive, se desejado, com relação a máquina virtual Java utilizada. Ao compilar casos de teste, os nomes das classes geradas são armazenados em um arquivo texto, como visto mais adiante.

4.1.4 Executando os Casos de Teste

Para a execução de casos de teste WSat, primeiramente é necessário a sua compilação para código Java. Isto é feito executando-se o método `main` da classe `WSatJavaCompiler` a partir da linha de comando do ambiente Java. Na linha de comando, passamos como parâmetro o nome do arquivo com o programa WSat a ser compilado. O método `main` desta classe invoca o método `execute`, visto abaixo, passando o parâmetro recebido.

```
public static void execute(String testFileName) throws Exception {
    FileInputStream file = new FileInputStream(testFileName);
    try {
        Parser parser = new Parser();
        TestComponent no = parser.parse(file);
        WSatCompilerVisitor compiler =
                  new WSatCompilerVisitor("c:\temp\wsat",
                                          "br.ufpe.cin.wsat.tests");
        no.accept(compiler);
        compiler.writeFiles();
    } finally {
        file.close();
    }
}
```

Este método realiza o *parser* no programa WSat criando a árvore sintática do programa WSat indicado. Em seguida, é instanciado um *visitor* `WSatCompilerVisitor` passando como parâmetro o diretório temporário a ser utilizado para saída do código Java gerado e o nome do pacote que as classes geradas irão pertencer. Este *visitor* visita a árvore sintática do programa lido gerando o código Java em memória. Após isto, o *visitor* escreve os arquivos no diretório de saída através da invocação de seu método `writeFiles`. Um ambiente de execução otimizado deve permitir a carga de diversos arquivos contendo, por exemplo, declarações de tipos e de funções de teste, permitindo assim uma melhor estruturação dos programas WSat.

Após serem traduzidos para código Java, os programas WSat ainda necessitam ser compilados para *bytecodes* para serem executados pelo ambiente Java. Para isto, utilizamos o próprio compilador de Java. O compilador de Java foi executado neste trabalho manualmente, utilizando-se as ferramentas encontradas no ambiente de programação da equipe de desenvolvimento. Este processo pode ser executado automaticamente após a tradução de WSat para código Java, porém não foi incorporado ao compilador de WSat por simplicidade e limitações de tempo. Para adicionar esta funcionalidade, podemos utilizar classes encontradas no próprio ambiente de execução de Java [34].

Uma vez realizada a compilação do código Java gerado a partir de programas WSat, podemos executar os casos de teste. Para isto foi definida a classe `WSatExecution`, executada a partir da linha de comando do ambiente Java. Na linha de comando, indicamos o diretório que contém os *bytecodes* gerados. Os casos de teste de WSat são representados por classes Java e o nome destas classes encontram-se escritos no arquivo texto de configuração. O arquivo de configuração contido neste diretório é então lido e o seguinte método é executado para cada um dos nomes de classes lidos.

```
01: public void executeTestCase(String testCaseClassName)
02:                                                 throws Exception {
03:     Class testCaseClass = Class.forName(testCaseClassName);
04:     TestCase testCase = (TestCase) testCaseClass.newInstance();
05:     String[] args = {};
06:     testCase.main(args);
07: }
```

Como podemos ver, a linha `3` realiza a carga da classe Java no ambiente de execução. Este método pode levantar a exceção `java.lang.ClassNotFoundException`, quando a classe não for encontrada. Já na linha `4`, uma instância desta classe é criada. Por fim, na linha `6`, o método `main` é invocado, iniciando assim a execução do caso de teste.

4.1.5 Suportes Fornecidos

Apresentamos nesta seção a análise do ambiente de execução de WSat baseada nos possíveis suportes fornecidos definidos neste trabalho.

S1. Verificação da Sintaxe de Páginas HTML. A implementação realizada para o ambiente de execução de WSat, como mostrado, utiliza a API HttpUnit para realizar a leitura dos arquivos HTML. A sintaxe destes arquivos é verificada pela API, entretanto ela não precisa estar em exata conformidade com a sintaxe do HTML padrão. Esta flexibilidade da ferramenta pode impedir que os testes detectem pequenos problemas de sintaxe.

S2. Teste de Layout dos Componentes de Páginas HTML. WSat não foi projetada com intenção de realizar testes sobre a distribuição espacial dos componentes de páginas HTML. Desta forma, este tipo de teste ainda não é suportado.

S3.1. Reprodução Através de Linguagem de Script. Em WSat, podemos reproduzir as ações dos usuários através dos serviços fornecidos pelos tipos WSat, como por exemplo, os serviços `click` e `submit`, respectivamente para o clique em um *link* e a submissão de um formulário HTML.

S3.2. Reprodução Através de Gravações. Este tipo de suporte está fora do escopo deste trabalho, não existindo assim ferramenta WSat para tal suporte.

S4. Simulação de Navegadores Internet. Com o uso da API HttpUnit para a navegação dentro da Web, não existe a possibilidade de se utilizar e testar engenhos de navegadores Web. Isto porque esta API não representa o comportamento real que os navegadores Web possam ter. Não existe a possibilidade nesta implementação de verificar o correto funcionamento do sistema executando-se os testes, por exemplo, nas diferentes implementações de navegadores Web.

S5. Verificação do Tempo de Resposta. Podemos utilizar código Java embutido para verificar o tempo de execução de trechos e de todo o programa. A utilização deste tipo de código, porém, não obtem o mesmo nível de abstração dos outros comandos de WSat.

S6. Acesso a Banco de Dados. A utilização de código Java nos dá a possibilidade de acessar diretamente dados armazenados em banco de dados, desde que conectados por JDBC [39].

S7. Verificação do Controle de acesso. WSat não foi projetada nesta primeira versão para acessar os detalhes sobre as conexões, como *cookies* armazenados, utilização do protocolos HTTPS, etc.

S8. Simulação de Usuários Concorrentes. Através de código Java embutido podemos fazer com que blocos de comandos sejam executados concorrentemente.

S9. Escalonamento da Execução de Testes. Programas WSat criados podem ser programados para executar de forma automatizada. Entretanto, como não existem ferramentas amigáveis para sua utilização, o resultado da execução dos testes (sucesso ou falha) tem que ser extraída a partir do texto escrito na saída padrão de execução da ferramenta.

S10. Parametrização dos Dados de Teste. Atualmente, a parametrização dos testes pode ser obtida através de código Java embutido. Isto fornece uma enorme gama de formas de parametriação, comprometendo porém o nível de abstração e legibilidade do programa.

S11. Portabilidade dos Casos de Testes. Os programas WSat podem ser executados em qualquer ambiente que forneça uma máquina virtual Java. Bancos de dados que forneçam drivers JDBC também podem ser acessados.

S12. Monitoramento do Sistema. Programas WSat podem ser utilizados para monitorar sistemas, pois os mesmos podem, devido a utilização de código Java embutido, representar alarmes, enviar e-mails e re-inicializar do sistema, desde que implementado através de código Java. Ainda não existem facilidades implementadas para o monitoramento do sistema.

S13. Depuração de Testes. Podemos utilizar os códigos Java embutidos para para registrar mensagens em arquivos de log, registrando assim todas as ações do teste.

A seguir, apresentamos na Tabela 4.1 um resumo sobre a análise de WSat a partir de seus suportes fornecidos. A notação utilizada para indicar o nível do suporte é a mostrada através da Tabela 2.2. Os dados nos mostram o bom desempenho de WSat semelhantemente aos resultados obtidos com as ferramentas analisadas, indicando que WSat consegue fornecer praticamente os mesmos suportes porém com um nível maior de abstração e reuso. Além disto, WSat pode ser estendida para satisfazer os suportes ainda inexistentes ou deficientes que são necessários para a sua utilização eficaz na prática.

4.2 Gerador de Código de Teste

Durante a análise de requisitos de sistemas Web, é comum a utilização de protótipos de tela HTML construídos por programadores visuais (*Web Designers*). Estes protótipos possuem um baixo custo de desenvolvimento e permitem um melhor entendimento e validação dos requisitos através de simulações de uso do sistema próximas do seu fun-

Suporte Fornecido	WSat
S1. Verificação da Sintaxe de Páginas HTML	–
S2. Teste de Layout dos Componentes de Páginas HTML	
S3.1. Reprodução Através de Linguagem de Script	++
S3.2. Reprodução Através de Gravações	
S4. Simulação de Navegadores Internet	
S5. Verificação do Tempo de Resposta	+
S6. Acesso a Banco de Dados	+
S7. Verificação do Controle de acesso	
S8. Simulação de Usuários Concorrentes	+
S9. Escalonamento da Execução de Testes	–
S10. Parametrização dos Dados de Teste	+
S11. Portabilidade dos Casos de Testes	++
S12. Monitoramento do Sistema	+
S13. Depuração de Testes	+

Tabela 4.1: Comparação dos níveis de suporte fornecidos pelas ferramentas.

cionamento real.

Ao se criar protótipos de tela HTML e programas WSat para um mesmo sistema, é possível notar semelhanças entre as informações escritas no código fonte dos protótipos e dos programas de teste WSat. Ambas realizam a descrição estrutural da GUI do sistema, sejam através de marcadores HTML no caso dos protótipos ou através dos tipos WSat representando os componentes Web da GUI. Por exemplo, informações como a declaração de formulários HTML, imagens e links, podem ser vistas tanto nos arquivos HTML dos protótipos de telas como nos programas WSat. O alto nível de abstração de WSat chega a ser, em casos extremos, a única diferença entre trechos de código destes arquivos. Isto sugere que um gerador de código poderia ser usado para reduzir o esforço necessário para criar o protóripo e os testes.

Esta seção apresenta o estudo realizado sobre a possibilidade de geração de código HTML ou WSat, visando eliminar esforços duplicados e aumentar a produtividade. Com isto, esperamos reduzir a rejeição natural contra o esforço necessário para se criar e implementar casos de teste. Além disto, discutimos a implementação de um gerador de código de teste.

4.2.1 Geração de Código de Teste ou de Telas HTML

Foi estudada qual seria a melhor possibilidade entre a de se gerar parte do código HTML do protótipo de tela a partir de descrições de casos de teste ou de se gerar código WSat a partir dos protótipos de tela. Este estudo, visto a seguir, foi feito através de uma simulação do funcionamento de tais geradores de código em cenários específicos.

Geração de Código HTML

Foi realizada primeiramente a simulação de geração de código do protótipo de tela a partir de casos de teste escritos em WSat. Um pequeno experimento foi realizado

como discutido a seguir. Em primeiro lugar, fizemos no papel um esboço da GUI do sistema para guiar o desenvolvimento dos programas WSat. Em seguida, escrevemos os programas WSat de acordo com o esboço definido. Por último, simulamos manualmente uma geração de código criando, a partir dos programas WSat escritos, um conjunto de arquivos HTML que formam a estrutura inicial do protótipo de tela.

A estrutura inicial gerada para o protótipo de tela foi apresentada para dois programadores visuais. Estes profissionais verificaram que o conteúdo gerado para o protótipo é facilmente criado através das suas ferramentas de trabalho, sendo qualificado os aspectos estéticos como os mais difíceis de serem feitos. De fato, os programadores visuais mostraram-se mais dispostos a começar o desenvolvimento do protótipo de telas a partir de arquivos novos, tomando os arquivos gerados apenas como referência.

A abordagem de gerar código de um protótipo de tela a partir de programas de teste WSat foi então descartada. Isto porque o esboço do sistema também guia perfeitamente os programadores visuais no desenvolvimento do protótipo de telas. Um outro problema notado nesta abordagem é o fato de que criar os programas de teste antes da criação do protótipo de telas é muito arriscado, porque é comum ocorrer mudanças nos requisitos do programa durante a validação destes protótipos. Isto traria a necessidade de modificar os programas WSat para que os mesmos testem os novos requisitos. Mesmo metodologias como XP, por exemplo, não indicam a criação de testes antes da validação dos requisitos.

Geração de Código WSat

Para analisar a viabilidade da geração de código de programas WSat a partir de um protótipo de tela HTML, foi realizado um segundo experimento. Este experimento iniciou-se com o desenvolvimento, por parte do programador visual, do protótipo de telas HTML do sistema. Este protótipo foi construído de acordo com o mesmo esboço de sistema utilizado no experimento anterior. A partir deste protótipo de tela, foi pesquisado que tipos de informações podem servir para a criação de programas WSat. Em seguida, simulamos manualmente a geração de código WSat a partir destas informações. O resultado deste pequeno experimento foi a geração em WSat de descrições estruturais simplificadas do sistema. Por fim, completamos manualmente o programa de teste gerado.

Com o programa WSat completo, comparamos este programa com o programa escrito no primeiro experimento, onde os testes foram escritos antes do protótipo. Pudemos notar um aumento no número de linhas do programa WSat do segundo experimento. Isto porque a geração a partir do protótipo trouxe informações de apresentação do sistema para o código. Exemplos disto são as imagens e links colocados pelo programador visual durante o desenvolvimento do protótipo. Além disto, algumas destas informações vieram duplicadas e com uma pobre nomenclatura, problemas detalhados mais adiante. Entretanto, o esforço de criação dos testes foi reduzido no segundo experimento. De forma geral, o segundo experimento provou ser satisfatória a geração de código WSat a partir de protótipos de tela HTML, reduzindo o esforço da criação de teste, evitando a criação de código antes de uma validação mínima de requisitos e suportando o uso da técnica indicada pela metodologia XP onde a criação de testes é realizada antes da implementação do código a ser testado.

4.2.2 Projeto e Implementação do Gerador

Esta seção descreve o projeto e implementação do gerador de código WSat a partir de protótipos de telas. O gerador implementado possui algumas limitações, como mostrado durante esta seção, utilizadas para simplificar sua implementação e que podem ser removidas em futuras versões. Durante os exemplos mostrados, utilizamos o protótipo de telas do sistema de busca apresentado no capítulo anterior.

Lendo Arquivos HTML

O primeiro passo realizado para implementar o gerador de código WSat foi a definição da classe `WSatGenerator`, vista na Figura 4.7.

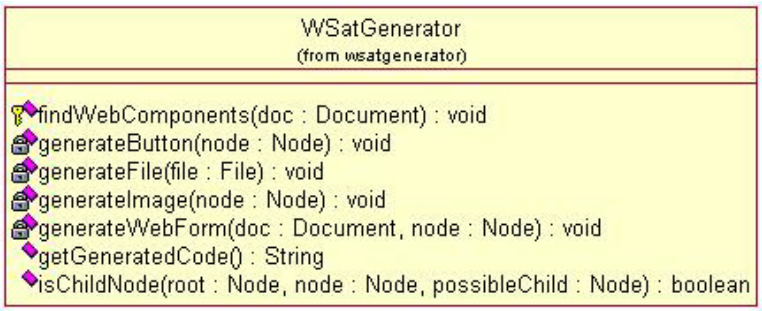

Figura 4.7: Classe responsável pela geração de código WSat.

Para gerar o código WSat em um arquivo HTML, invocamos o método `generateFile` desta classe. Ao receber o nome completo de um arquivo HTML, o método `generateFile` faz uso da API JTidy [38] para realizar o *parser* deste arquivo. Esta API é utilizarda para realizar as operações de *parser* e escrita de arquivos HTML. A operação de *parser* resulta na geração de uma árvore que representa o arquivo lido. O código para realizar o parser de um arquivo HTML utilizando esta API é mostrado abaixo.

```
01: public void generateFile(File file) {
02:     Tidy td = new Tidy();
03:     InputStream input = new FileInputStream(file);
04:     Document doc = td.parseDOM(input, null);
05: }
```

O código mostrado realiza o parser no arquivo HTML representado pelo parâmetro `file`. A linha `2` inicializa a API e a linha `3` abre para leitura o arquivo HTML. Já na linha `4`, podemos ver a utilização do método `parseDOM` para a realização do *parser* no arquivo. Este método recebe como parâmetro os *streams* do arquivo de entrada e do de saída (o valor `null` é utilizado quando queremos só a leitura do arquivo). Já o valor retornado, do tipo `Document`, representa através de uma árvore DOM [6] o documento lido. Os detalhes sobre esta árvore são mostrados sempre que necessário.

O método `generateFile` também pode receber o nome de um diretório contendo os arquivos HTML do protótipo. Neste caso, o diretório é percorrido e todos os seus arquivos e subdiretórios são analisados. Este método é invocado recursivamente para

cada arquivo com extensão `.html` ou `.htm` encontrado. O código que verifica se o arquivo recebido é um diretório foi abstraído para simplificar o entendimento do código.

Páginas Web

Inicialmente, para cada arquivo lido é gerado um novo tipo WSat, mais precisamente um subtipo de WebPage. O arquivo `paginaInicial.html`, que representa a página inicial do sistema de busca, por exemplo, resulta na definição do tipo de nome igual ao arquivo, sem sua extensão, e com a primeira letra maiúscula, como mostrado abaixo:

```
WebPage PaginaInicial {
    ...
}
```

O fato do nome do tipo gerado ser igual ao nome da página cria a limitação de não poder existir dois arquivos HTML de mesmo nome, mesmo que em diretórios diferentes. Esta limitação pode ser trivialmente eliminada, por exemplo, através da inclusão do nome do diretório no nome do tipo gerado para a página ou pela extensão da linguagem WSat através da definição de pacotes, como em Java.

Propriedades de Páginas Web

Ao se percorrer um arquivo HTML, nós podemos encontrar várias informações úteis para a geração de código WSat como, por exemplo, links, imagens e formulários HTML. Estas informações geram propriedades no tipo WSat gerado a partir do arquivo HTML lido. Consideremos a seguinte declaração de imagem em código HTML no arquivo da página inicial do sistema de busca:

```
<img name="logo" src="imagens/logo.jpg" align=right>
```

Para este código HTML, o gerador de código de teste cria, como visto a seguir, a propriedade `logo` no tipo WSat que representa o arquivo HTML lido.

```
WebPage PaginaInicial {
    WebImage {
        name = "logo";
        src = "imagens/logo.gif";
    } logo;
}
```

Como podemos ver, para marcadores HTML `<img>` são criadas propriedades do tipo `WebImage`. Estas propriedades possuem o mesmo nome do atributo `name` do seu marcador HTML, não devendo o arquivo possuir duas aparições de marcadores HTML quaisquer com o mesmo valor para este atributo, fato possível na linguagem HTML. Neste caso, por exemplo, são gerados dois tipos WSat com o mesmo nome, ocasionando erro durante a compilação do programa WSat gerado. Caso o valor deste atributo esteja indefinido, um nome identificador único é gerado semelhante, por exemplo, ao nome `img001`. Deve-se também ter o cuidado de não colocar nomes no protótipo neste

formato para evitar conflitos. Estas restrições devem ser seguidas pelos programadores visuais durante a criação dos protótipos de tela para possibilitar o uso eficaz do gerador.

As propriedades definidas no tipo `WebImage` são inicializadas de acordo com os atributos do marcador HTML <img>. Também podemos observar que as propriedades são definidas de forma anônima, diminuindo o número de tipos WSat gerados e facilitando o entendimento deste código. A geração de propriedades para outros componentes Web como *links* e formulários HTML, por exemplo, é realizada da mesma forma.

Para o gerador de código percorrer a árvore que representa o arquivo HTML em busca de componentes Web úteis para a geração de código WSat, foi definido o método `findWebComponents`:

```
01: protected void findWebComponents(Document doc) {
02:     NodeList nodeList;
03:     Node node;
04:     nodeList = doc.getElementsByTagName("img");
05:     for (int i = 0 ; i < nodeList.getLength() ; i++) {
06:         node = nodeList.item(i);
07:         generateWebImage(node);
08:     }
09:     nodeList = doc.getElementsByTagName("form");
10:     for (int i = 0 ; i < nodeList.getLength() ; i++) {
11:         node = nodeList.item(i);
12:         generateWebForm(doc, node);
13:     }
14:     ...
15: }
```

Este método recebe como parâmetro um objeto do tipo `Document` representando o arquivo lido. A partir deste objeto, invocamos o seu método `getElementsByTagName` que, dado o nome de um marcador HTML, retorna uma lista de nós que representam aparições deste marcador. Na linha `4`, por exemplo, podemos ver o uso deste método para encontrar aparições do marcador <img>. Para cada nó que representa o marcador HTML procurado, é invocado um método para gerar uma propriedade que o represente. No caso dos marcadores <img>, isto pode ser visto na linha `7` através do método `generateWebImage`. Este processo é repetido para cada componente Web, como por exemplo para imagens, da linha `4` a `8`, e para formulários Web, da linha `9` a `13`. A invocação do método `findWebComponents` é feita no método `generateFile` visto anteriormente, uma linha de código antes de fechar o bloco da definição do tipo que representa o arquivo HTML lido.

Os métodos para geração de propriedades como, por exemplo, os métodos `generateWebImage` e `generateWebForm` recebem como parâmetro um objeto do tipo `Node`, representando o nó criado para o marcador HTML a ser analisado. Estes métodos escrevem a definição dos tipos WSat sendo gerados, incluindo a inicialiação de suas propriedades de acordo com os seus respectivos valores definidos no código HTML.

Informações Complementares

Existem informações utilizadas pelo gerador de código de sistema visto na Seção 4.3 que podem ser geradas automaticamente pelo gerador de código de teste. Por exemplo, indicamos, através da palavra chave `static`, as páginas do sistema que são inicialmente consideradas estáticas. Além disto, também utilizamos a propriedade `template`, como visto a seguir, para indicar o caminho do arquivo HTML que originou a definição deste tipo.

```
static WebPage PaginaInicial {
    template = "c:\wsat\exemplo\www\paginaInicial.html";
}
```

Com isto, o testador deve remover a palavra chave `static` da definição de subtipos de `WebPage` para indicar as páginas que, na verdade, são geradas dinamicamente no sistema real. Foi considerado aqui que, em geral, o esforço de identificar as páginas Web estáticas do sistema é maior do que a identificação das páginas dinâmicas. Em casos onde isto não é verdade, a ferramenta pode ser configurada para não gerar esta informação, ficando o testador responsável agora pela identificação das páginas estáticas do sistema. Na verdade, este gerador pode ser configurado para não escrever o código a ser utilizado apenas pelo gerador de código de sistema e não durante a execução dos testes propriamente ditos.

4.2.3 Executando o Gerador

Para executar o gerador de código de teste, basta invocar o método `generateFile` passando o caminho do diretório que contém o protótipo de telas. Desta forma, criamos na classe `WSatGenerator` o seguinte método:

```
public static void main(String[] args) {
    try {
        File file = new File(args[0]);
        WSatGenerator generator = new WSatGenerator();
        generator.generateFile(file);
        FileWriter writer = new FileWriter(args[1], false);
        try {
            writer.write(generator.getGeneratedCode());
        } finally {
            writer.close();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

O método `main` pode ser executado a partir da linha de comando Java. Ele recebe um *array* de `String` contendo o diretório do protótipo de telas e o caminho completo do arquivo a ser criado com o programa WSat gerado. O programa WSat resultante vai

sendo escrito incrementalmente em memória, sendo em seguida escrito definitivamente em um arquivo texto.

Uma vez gerado o código WSat, o testador pode livremente alterá-lo para que ele descreva com mais detalhes a estrutura da GUI do sistema ou para aumentar sua legibilidade. Por exemplo, podemos alterar o nome dos tipos criados nomes mais intuitivos, facilitando assim o entendimento deste tipo. Além disto, podemos também descrever o comportamento do sistema através da definição de funções e casos de teste.

4.3 Gerador de Código de Sistema Web

Devido às características especiais de WSat como a descrição estrutural explícita da GUI do sistema através da definição de tipos que representem componentes Web, trechos de código de sistema podem ser gerados a partir de casos de teste escritos em WSat. Nesta seção, mostramos como trechos de código de sistemas Web podem ser gerados automaticamente a partir de programas WSat escritos para testar tais sitemas. Em particular, apresentamos o projeto e implementação de um gerador de código que tem este objetivo. Em primeiro lugar, apresentamos o ambiente de desenvolvimento de sistemas Web considerado e que será configurado por trechos de código gerados automaticamente. Em seguida, apresentamos que tipos de trechos de código podem gerados e o projeto e implementação do gerador. Por fim, mostramos como este gerador de código é executado.

4.3.1 O Ambiente e Arquitetura de Desenvolvimento

A geração de código de sistemas Web é, em parte, dependente do ambiente utilizado para o desenvolvimento destes sistemas. Um exemplo disto é a linguagem de programação utilizada. Desta forma, para criar um gerador de código do qual seus usuário possam tirar o máximo proveito, faz-se necessário determinar o ambiente de desenvolvimento dos sistemas Web considerados.

Para escolher qual seria o ambiente de desenvolvimento destino do gerador de código de sistema, foram estudados alguns padrões de projeto e ferramentas utilizadas no desenvolvimento de tais sistemas. Este estudo foi realizado com o intuito de descobrir que tipos de código podem ser gerados automaticamente a partir de programas WSat, porém sem analisar a eficácia da utilização de tais padrões e ferramentas para não fugir do escopo do trabalho. O ambiente escolhido é, na verdade, uma composição de padrões de projeto e ferramentas utilizados em diversas empresas. Os detalhes sobre este ambiente são vistos a seguir.

A linguagem Java [19] é utilizada por um grande número de empresas locais no desenvolvimento de sistemas Web. Por este motivo, ela foi escolhida como a linguagem de programação do ambiente de desenvolvimento. Para o desenvolvimento de sistemas Web em Java, o uso de Servlets é bastante comum. Entretanto, diversas técnicas de projeto para a estruturação dos Servlets podem ser aplicadas. Entre as encontradas, o padrão *Web Handler* [2] foi escolhido por fornecer benefícios interessantes como uma boa modularização do código e por ter seu maior impacto negativo, o grande número de classes necessárias para sua implementação, amenizado pelo uso deste gerador.

Sistemas Web necessitam, em geral, validar os parâmetros recebidos em suas requisições. Para validar estes parâmetros, a técnica mais interessante encontrada foi o uso de regras de validação escritas em arquivos XML [30]. Estes arquivos XML possuem informações sobre regras para validação dos parâmetros recebidos em uma requisição como, por exemplo, os valores permitidos para cada um destes parâmetros.

Outro fator a ser considerado no desenvolvimento de sistemas Web é a separação entre códigos Java e HTML. Para isto, foi notado em várias empresas o uso da API FreeMarker [17]. O funcionamento desta API é baseado em *templates* HTML. *Templates* HTML são arquivos textos que armazenam códigos HTML e variáveis que são substituídas por informações calculadas em tempo de execução. Esta API foi escolhida pelo seu grande uso no mercado e pela experiência de uso do autor deste trabalho.

Para a realização de testes unitários no ambiente escolhido, utilizamos o *framework* JUnit [15]. Este *framework* define como classes de teste devem ser criadas e utilizadas na realização de testes unitários e é indicado por metodologias como a *Extreme Programming* [3].

A execução de sistemas Web são realizadas através de servidores Web. Entre os servidores utilizados, o servidor Jakarta Tomcat [14] foi escolhido por ser um servidor de código aberto largamente utilizado pela comunidade.

Como podemos ver na próxima seção, a utilização do padrão de projeto *Visitor* permite que o código do gerador implementado seja facilmente alterado para gerar código de sistemas desenvolvidos em ambientes diferentes do apresentado.

4.3.2 Projeto e Implementação

Similarmente ao compilador de WSat para Java, foram criados diversos classes que implementam *visitors* que percorrem a árvore sintática de WSat para a geração de código de sistema. Cada um destes *visitors* possui uma funcionalidade específica como, por exemplo, gerar o código dos *handlers* de requisições Web ou de seus arquivos de configuração. No decorrer desta seção, mostramos a funcionalidade, projeto e trechos de código da implementação de cada *visitor* criado.

Web Handlers

De forma geral, com a utilização deste padrão, cada página dinâmica do sistema é associada a um par de *handlers*. Sendo assim, este gerador gera um par de *handlers* para cada tipo WSat que representa uma página Web dinâmica, como a do tipo `PagResposta` vista a seguir.

```
WebPage PagResposta {   ...
}
```

Para cada um destes tipos que não são declarados como estático, ou seja, que é declarado sem a cláusula WSat `static`, um par de *handlers* é gerado para implementar a página Web do sistema que satisfaz as características definidas por este tipo. O primeiro destes *handlers* é o de processamento, responsável pela execução da funcionalidade requerida a uma dada página do sistema. O trecho de código do *handler* de processamento gerado para o tipo `PagResposta` é visto a seguir:

```
public class PagRespostaHandlerProcessamento
                                extends HandlerProcessamento {
    public void processar(HttpServletRequest request,
                          HttpServletResponse response)
                                throws ProcessamentoException {
    }
}
```

Este *handler* herda da classe `HandlerProcessamento` definida pelo padrão *Web Handlers*. Esta classe define também o método `validar` utilizado na validação dos parâmetros de requisições de serviços Web, como visto mais adiante. A classe `HandlerProcessamento` faz parte do padrão *Web Handlers* e possui o método abstrato `processar`. Este método é responsável por executar a funcionalidade do sistema requerida e é implementado pelos *handlers* do sistema como, por exemplo, o *handler* `PagRespostaHandlerProcessamento` apresentado.

O segundo *handler* gerado é o *handler* de apresentação, responsável por montar a apresentação do resultado obtido pela execução do serviço requisitado através de uma página Web. Para o tipo WSat `PagResposta`, o seguinte *handler* de apresentação é gerado:

```
public class PagRespostaHandlerApresentacao
                                extends HandlerApresentacao {
    public void apresentar(HttpServletRequest request,
                           HttpServletResponse response)
                                 throws ApresentacaoException {
    }
}
```

Como podemos ver, o *handler* gerado herda da classe `HandlerApresentacao` definida no padrão *Web Handlers*. Esta classe possui o método abstrato `apresentar`, utilizado por suas subclasses para montar a página de resposta para as requisições de serviços Web. O código gerado para os *handlers* não é substancial, mas como o número de *handlers* necessários para implementar o sistema pode ser grande, esta geração de código se torna bastante interessante.

A escrita dos arquivos Java dos *handlers* gerados são realizados a partir de um arquivo *template* com o auxílio da API FreeMarker. Para a escrita destes arquivos Java, são passados para o FreeMarker, o nome do *template* e o valor de variáveis como, por exemplo, a variável `${name}` do *template* mostrado a seguir, variável que é substituída em tempo de execução pelo nome do *handler* sendo gerado. A seguir podemos ver o *template* utilizado para geração do *handler* de processamento.

```
public class ${name} extends HandlerProcessamento {
    public void processar(HttpServletRequest request,
                          HttpServletResponse response)
                                throws ProcessamentoException {
    }
}
```

O uso de *templates* facilita a modificação do código gerado no caso, por exemplo, da geração de extensões do padrão `WebHandlers`. Para isto, em muitos casos é suficiente apenas a alteração do arquivo de *template* sem a necessidade de compilação das classes Java do gerador de código. A declaração de pacote dos *handlers* e a importação de classes são também geradas, mas abstraídas aqui visando a simplificação do código a ser mostrado, já que estas informações não têm relevância neste trabalho. Para a geração dos *handlers* através do *template* mostrado, foi criada a classe `JavaCodeGeneratorVisitor`. Esta classe implementa um *visitor* que percorre a árvore sintática de WSat em busca de declarações de subtipos de `WebPage`. O método `visit` utilizado para visitar nós da árvore sintática que representam tais declarações é visto a seguir:

```
public void visit(WebPageDeclaration node) {
    if (!node.isStaticPage()) {
        String name = node.getName();
        writeHandler(name + "HandlerProcessamento",
                     PROCESS_HANDLER_TEMPLATE);     ...
    }
}
```

Caso o nó não represente a declaração de um tipo WSat estático, verificado através do método `isStaticPage`, o método `writeHandler` é invocado para gerar cada elemento do par de *handlers*. Este método recebe como parâmetro o nome do *handler* a ser gerado, formado pela concatenação do nome do tipo WSat com sufixo "`HandlerProcessamento`" ou "`HandlerApresentacao`", e o nome do *template* a ser utilizado na geração do *handler*.

Arquivo de Configuração de Serviços

Para o funcionamento correto do sistema, é necessário indicar para cada serviço Web fornecido o par de *handlers* que o implementa. Isto é feito através do arquivo XML de configuração de serviços. O arquivo de configuração para o par de *handlers* mostrados anteriormente, por exemplo, é visto a seguir.

```
<servidor>
 <tratamento_erro default="handler.ExecutionExceptionHandler">
 </tratamento_erro>
    <servicos>
        <servico nome="PagResposta">
            <processamento
                nome="handler.PagRespostaHandlerProcessamento" />
            <apresentacao
                default="handler.PagRespostaHandlerApresentacao" />
        </servico>
    </servicos>
</servidor>
```

Para cada página Web, além de um par de *handlers* é também gerado um serviço com o seu nome . No exemplo mostrado, podemos ver a definição do serviço `PagResposta`,

onde é indicado o nome das classes dos *handlers* de processamento e de apresentação que implementam o serviço.

Para gerar este arquivo automaticamente, foi criada a classe `ServiceConfiguratorVisitor`. Esta classe percorre a árvore sintática de programas WSat em busca de nós que representam declarações de páginas Web estáticas. O método utilizado para visitar nós do tipo `WebPageDeclaration` é visto a seguir:

```
01: public void visit(WebPageDeclaration node) {
02:     if (!node.isStaticPage()) {
03:         String name = node.getName();
04:         String processHandlerName = packageName + "."
05:                                     + name + hpSufix;
06:         String presentationHandlerName = packageName + "."
07:                                          + name + haSufix;
08:         SimpleHash serv = new SimpleHash();
09:         serv.put("name", name);
10:         serv.put("processHandlerName", processHandlerName);
11:         serv.put("presentationHandlerName",
12:                  presentationHandlerName);
13:         services.add(serv);
14:     }
15: }
```

Caso o nó represente a declaração de um tipo estático, as linhas 3 a 12 montam a estrutura do FreeMarker contendo três variáveis, `name`, `processHandlerName` e `presentationHandlerName`. Estas variáveis representam respectivamente o nome do serviço e dos *handlers* de processamento e de apresentação. Como um sistema Web pode possuir diversos serviços, a estrutura montada com as variáveis citadas é adicionada, como visto na linha 13, na lista de serviços a serem gerados referenciada pela variável `services`. Esta lista é inserida na estrutura do FreeMarker como visto a seguir.

```
root.put("services", services);
```

Desta forma, esta lista pode ser referenciada no *template* do FreeMarker através do uso do marcador <list>:

```
01: <servidor>
02:     <tratamento_erro
03:             default="handler.ExecutionExceptionHandler" />
04:     <servicos>
05: <list services as service>
06:         <servico nome="${service.name}">
07:             <processamento
08:                 nome="${service.processHandlerName}" />
09:             <apresentacao
10:                 default="${service.presentationHandlerName}" />
11:         </servico>
```

```
12: </list>
13:     </servicos>
14: </servidor>
```

O marcador <list> representa para o FreeMarker uma lista de estruturas. Através da linha 5 do exemplo mostrado, este marcador indica que o nome da lista é `services` e seus elementos são referenciados pelo nome `service`. O código delimitado por este marcador, linhas 6 a 11, é repetido para cada elemento da lista. No caso, cada elemento da lista é uma estrutura que contém os valores para três variáveis. Estes valores podem ser acessados colocando-se o nome utilizado para referenciar os elementos da lista como, por exemplo, `service.name` para a variável `name`.

Ao percorrer todos os nós da árvore sintática de WSat, o gerador de código de sistema invoca o método `writeResult` do *visitor* para escrever, através da API do FreeMarker, o arquivo de configurações de serviços.

Arquivos de Validação de Dados

Os *handlers* de requisições de sistemas Web recebem como parâmetro um objeto do tipo `HttpServletRequest` que contém, entre outras coisas, os parâmetros da requisição. Para validar os parâmetros recebidos, podemos utilizar arquivos JavaScript e XML que descrevem regras de validação de parâmetros. JavaScript é utilizado para validar os dados no próprio navegador Web do cliente, melhorando assim a performance do sistema. Já os arquivos XML são utilizados para validar no servidor Web os dados enviados. A validação dos dados pelo servidor é essencial, já que o código de validação em JavaScript enviado ao cliente pode ser facilmente desativado. Como a geração dos arquivos JavaScript e XML é feita de forma similar, apenas a geração dos arquivos XML foi abordada na implementação deste gerador.

Um exemplo de arquivo XML para validação dos parâmetros é o utilizado para validar os parâmetros do formulário HTML do sistema de busca, visto a seguir:

```
<?xml version="1.0"?>
<schema>
    <attribute name="palavras">
        <simpleType baseType="string" optional="false">
        </simpleType>
    </attribute>
    <attribute name="onde">
        <simpleType baseType="int" optional="false">
            <enumeration value="1" />
            <enumeration value="2" />
        </simpleType>
    </attribute>
</schema>
```

Como podemos notar, este arquivo contém informações, como o tipo do parâmetro e seus valores válidos, encontradas também nas cláusulas `validate` de WSat. Cada formulário HTML deve possuir um arquivo de validação correspondente. Desta forma, o

visitor `XMLValidationVisitor` foi definido para visitar os nós de árvores sintáticas de programas WSat em busca de formulários HTML. Para cada formulário encontrado, um arquivo XML de validação de seus parâmetros é escrito de acordo com as informações encontradas nas cláusulas `validate` de seus parâmetros. Parâmetros que não utilizam esta cláusula são representados nos arquivos XML como parâmetros sem nenhuma restrição de validação.

Para facilitar o manuseio dos arquivos XML gerados, cada um destes arquivos possui o nome do tipo WSat que testa o formulário HTML. Para o tipo `FormBusca`, por exemplo, é gerado o arquivo `FormBusca.xml`. No caso de declarações anônimas, utilizamos o nome da propriedade que define de forma anônima o formulário HTML com prefixo igual ao nome do tipo WSat que contém esta propriedade. Por exemplo, caso o tipo `FormBusca` fosse definido de forma anônima através da propriedade `formBusca` do tipo `PagInicial`, seria gerado o arquivo XML de nome `PagInicial_formBusca.xml`. Para que os *handlers* validem os parâmetros recebidos em uma requisição de serviço Web, utilizamos o seguinte trecho de código:

```
File schemaFile = new File(xmlValidationPath,
                           xmlFormValidationFile);
Validator validator = Validator.getInstance(schemaFile.toURL());
validator.validate(request);
```

Na primeira linha, podemos ver o uso da variável `schemaFile` para representar o caminho completo do arquivo XML. Este caminho é montado a partir do nome do arquivo XML (igual ao nome do formulário HTML) e do diretório de arquivos XML, armazenados respectivamente pelas variáveis `xmlFormValidationFile` e `xmlValidationPath`. A segunda linha cria um objeto do tipo `Validator`, da API de validação de parâmetros [30], para o arquivo indicado para então invocar o método `validate` deste objeto. Este método valida os parâmetros recebidos na requisição (representados pela variável `request`) de acordo com o arquivo XML lido.

Para que o código mostrado não seja repetido para cada *handler* de processamento, foi criado o *handler* abstrato `HandlerProcessamentoGenerico`. Este *handler* define o método `validar` que recebe como parâmetro o objeto do tipo `HttpServletRequest` e o nome do arquivo XML a ser utilizado e realiza a validação dos parâmetros recebidos. O diretório dos arquivos XML, referenciado pela variável `xmlValidationPath` no código mostrado, é inicializado através dos parâmetros de inicialização dos *handlers*, como visto mais adiante. A geração das classes Java para os *handlers* de processamento foi então modificada para que as mesmas herdem da classe `HandlerProcessamentoGenerico`, conforme visto na Figura 4.8. Especializamos desta forma o padrão *Web Handlers* para o uso de validação de parâmetros através de arquivos XML.

Arquivos HTML e *Templates* para *Handlers* de Apresentação

A API FreeMarker é utilizada pelos *handlers* para construção de páginas Web dinâmicas, permitindo a separação entre código Java e código HTML. Para a sua utilização, os *handlers* de apresentação são associados a *templates* HTML. Estes *templates* são arquivos HTML com variáveis que serão substituídas por valores através da API FreeMarker. Observando o desenvolvimento de sistemas Web por várias empresas, pudemos perceber

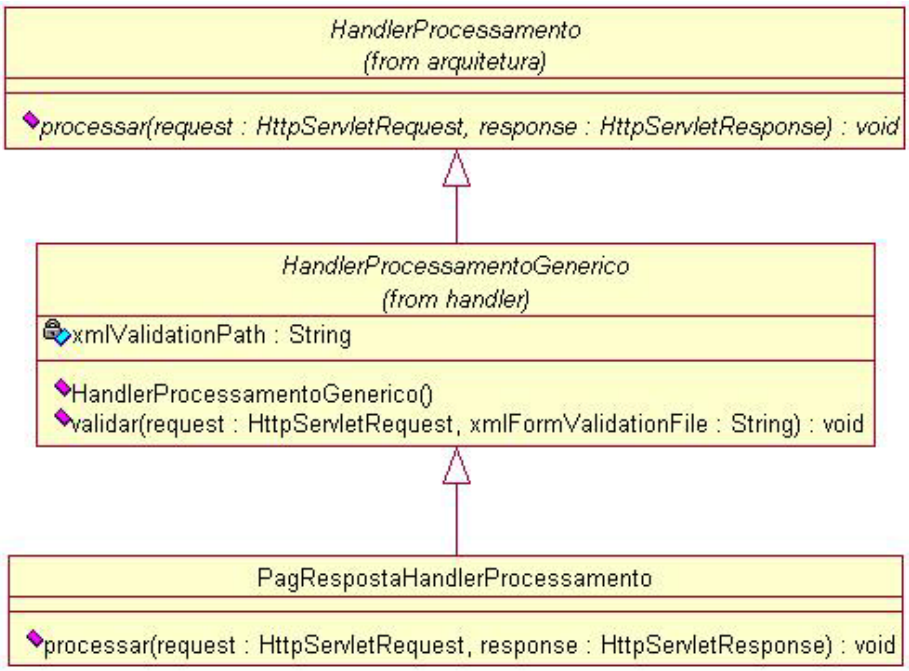

Figura 4.8: Hierarquia de *handlers* de processamento.

que estes *templates* são criados, em geral, a partir dos arquivos que formam os protótipos de tela HTML criados pelos programadores visuais. Quando os programas WSat são gerados a partir destes protótipos de telas, referências para os arquivos HTML do protótipo são escritas nos tipos que representam páginas Web através da propriedade `template` do supertipo `WebPage`. Desta forma, a partir dos programas WSat, podemos copiar os arquivos HTML do protótipo de tela e utilizá-los como base para o desenvolvimento de páginas estáticas ou dinâmicas (*templates* dos *handlers*).

A classe `HTMLGeneratorVisitor` é um *visitor* que, ao percorrer a árvore sintática de WSat, gera um arquivo HTML para cada subtipo `WebPage` encontrado. Caso o tipo encontrado seja declarado de forma estática, o arquivo é gerado na área Web definida para o projeto. No caso de tipos que representem páginas dinâmicas, o arquivo é gerado na área de *templates* dos *handlers* do sistema Web. Caso o tipo não tenha um valor atribuído pelo programador à sua propriedade `template`, um arquivo contendo uma página Web *default* é gerado para o mesmo. Desta forma, o programador pode escolher entre manter, editar ou substituir o arquivo gerado por um outro já existente.

Para realizar a cópia dos arquivos, utilizamos a API do FreeMarker. Para isto, consideramos o arquivo do protótipo de telas como um *template* sem variáveis. A própria API de Java também pode ser utilizada para realizar a cópia dos arquivos. Após esta cópia, deve-se alterar o arquivo inserindo variáveis FreeMarker nos locais onde o conteúdo é gerado dinamicamente.

Os *handlers* de apresentação montam uma estrutura da API do FreeMarker para que esta API gere as suas páginas de respostas. Visando um melhor reuso de código foi definido o *handler* abstrato `HandlerApresentacaoGenerico`. Este *handler* define a seguinte implementação para o método `apresentar`:

```
public void apresentar(HttpServletRequest request,
                       HttpServletResponse response)
                                    throws ApresentacaoException {
    try {
        response.setContentType("text/html");
        Writer out = response.getWriter();
        TemplateModelRoot root = criarModelRoot(request, response);
        Template template = new Template(templatePath + "/" +
                                         templateName);
        template.process(root, new PrintWriter(out));
    } catch (IOException e) {
        throw new ApresentacaoException(e);
    }
}
```

Este método monta a página de resposta com o FreeMarker a partir do objeto retornado pelo método abstrato `criarModelRoot`. Este objeto, da API do FreeMarker, é criado por cada um dos *handlers* de apresentação do sistema Web. Desta forma, subtipos de `HandlerApresentacaoGenerico` necessitam implementar apenas o método `criarModelRoot`, ao invés do método `apresentar` (ver Figura 4.9). O *template* a ser utilizado para montar a resposta do *handler* gerado é definido pelos atributos `templatePath` e `templateName`, representando o diretório de *templates* e o nome do *template* a ser utilizado. Os dois atributos desta classe são inicializados por parâmetros de mesmo nome contidos no arquivo de inicialização dos *handlers*. Este arquivo de inicialização é gerado automaticamente por este gerador, como visto mais adiante. O seguinte código, por exemplo, é gerado para o *handler* associado ao tipo WSat `PagResposta` visto.

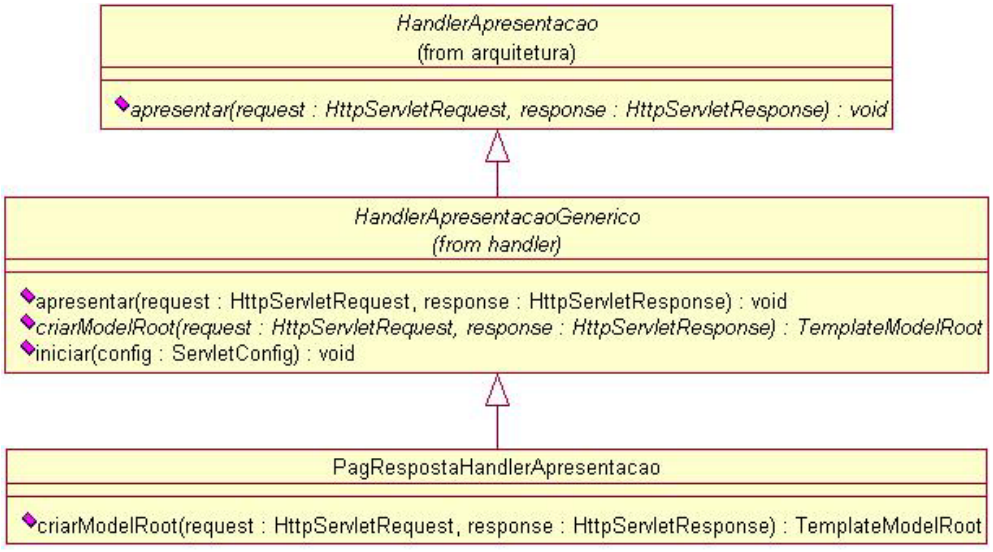

Figura 4.9: Hierarquia de *handlers* de apresentação.

```
public class PagRespostaHandlerApresentacao
                              extends HandlerApresentacaoGenerico {
    public TemplateModelRoot criarModelRoot(
                                        HttpServletRequest req,
                                        HttpServletResponse resp)
                                    throws ApresentacaoException {
        SimpleHash root = new SimpleHash();
        return root;
    }
}
```

Como podemos notar, o código gerado para este *handler* possui uma implementação básica do método `criarModelRoot`, o que já possibilita aos desenvolvedores uma compilação e execução preliminar do sistema Web gerado. Desta forma, especializamos o padrão *Web Handlers* não só para validação de parâmetros através de arquivos XML, como visto anteriormente, como também para o uso da API FreeMarker para separação entre códigos Java e HTML.

Arquivo de Parâmetros de *handlers*

Para o funcionamento correto dos *handlers* é necessário configurá-los adequadamente através de parâmetros de configuração. A implementação utilizada aqui [2] faz com que os próprios *handlers* sejam Servlets. Sendo assim, seus parâmetros de inicialização são passados através dos arquivos de parâmetros dos Servlets. Como o servidor Web alvo deste gerador é o Jakarta Tomcat, o arquivo com a configuração dos Servlets contém, para cada *handler* gerado, uma instância do marcador XML <servlet>. A seguir podemos ver o uso deste marcador para o *handler* `PagRespostaHandlerProcessamento`:

```
<servlet>
    <servlet-name>handler.PagRespostaHandlerProcessamento
    </servlet-name>
    <servlet-class>handler.PagRespostaHandlerProcessamento
    </servlet-class>
    <load-on-startup>1</load-on-startup>
    <init-param>
        <param-name>exampleParamName</param-name>
        <param-value>exampleParamValue</param-value>
    </init-param>
</servlet>
```

No código XML mostrado, indicamos o nome e a classe do Servlet através dos marcadores XML <servlet − name> e <servlet − class>. Como este Servlet é um *handler*, a implementação dos *handlers* necessita que o nome do Servlet seja igual ao nome da própria classe. Também indicamos, através do marcador XML <load − on − startup> que o *handler* deve ser carregado no momento da inicialização do servidor Web. Podemos observar também a existência de um parâmetro utilizado como exemplo, de nome `exampleParamName` e valor `exampleParamValue`.

O arquivo de configuração de Servlets é gerado similarmente a geração do arquivo de configuração de serviços. Além deste arquivo de configuração, a especialização do padrão de projeto *Web Handlers* com o uso de FreeMarker e validação de parâmetros através de arquivos XML faz com que estes *handlers* necessitem de mais um arquivo de configuração. Este arquivo, mostrado a seguir, é um arquivo de propriedades lido pelos *handlers* para encontrar os arquivos de *templates* e de validação de parâmetros:

```
xmlFileDirectory  = D:/projeto/wsat/exemplo/xmlFiles
templateDirectory = D:/projeto/wsat/exemplo/templates
```

Este arquivo possui os valores dos parâmetros `xmlFileDirectory` e `templateDirectory` que representam respectivamente o diretório onde se encontram os arquivos XML para validação de parâmetros e os *templates* dos *handlers* de apresentação. O valor dos parâmetros deste arquivo são os diretórios de saída utilizados na geração dos próprios arquivos de validação de parâmetros e de *templates*.

Testes Unitários de *Handlers*

A realização de testes unitários é indicada para todos os tipos de classes, inclusive para as classes que fazem papel de *handlers* de requisições de serviços Web. Utilizando-se o *framework* JUnit, cada *handler* gerado deve ser testado por uma classe de teste. A seguir mostramos o esqueleto da classe gerada para realizar testes unitários na classe `PagRespostaHandlerProcessamento` mostrada:

```
public class TestPagRespostaHandlerProcessamento extends TestCase {
    public TestPagRespostaHandlerProcessamento(String name) {
        super(name);
    }
    public void testProcessar() {
    }
}
```

O código de teste de unidade a ser executado para o *handler* deve ser escrito pelo testador no método `testProcessar` da classe de teste apresentada. Como podemos notar, o esqueleto destas classes de teste podem ser gerados automaticamente com o uso do *template* FreeMarker visto a seguir.

```
public class Test${name} extends TestCase {
    public Test${name}(String name) {
        super(name);
    }
    public void testProcessar() {
    }
}
```

Como podemos ver, este *template* utiliza apenas a variável `${name}` representando o nome da classe de teste. Por causa da limitação de tempo para desenvolvimento deste trabalho, não foi realizada a implementação de um *visitor* para gerar o código de testes mostrado. A implementação deste *visitor* pode ser feita de maneira semelhante ao do *visitor* `JavaCodeGeneratorVisitor`.

4.3.3 Executando o Gerador de Código de Sistema

A execução do gerador de código é realizada através da linha de comando do ambiente de execução de Java. Podemos, por exemplo, executar a seguinte linha de comando para gerar o código de sistema no diretório "`c : /temp/wsat/sistema`" a partir do programa de teste encontrado no arquivo "`c : /temp/wsat/exemplo.wsat`":

```
java br.ufpe.cin.wsat.SystemCodeGenerator
        "c:\temp\wsat\exemplo.wsat" "c:\temp\wsat\sistema"
```

Esta linha de comando faz com que o ambiente de execução de Java invoque o método `main`, mostrado a seguir, definido na classe `SystemCodeGenerator` passando como argumento o nome do arquivo com o programa de teste WSat e o diretório de saída do código gerado para o sistema.

```
public static void main(String[] args) {
    String testFileName = args[0];
    String dirGeneratedCode = args[1];
    String dirTemplate = "h:/projeto/wsat/templates";
    ...
    try {
        loadFile(testFileName);
        generateJavaCode(dirTemplate, dirGeneratedCode,
                         packageName);
        generateTemplates(dirTemplate, dirGeneratedCode,
            packageName, dirGeneratedPages, dirGeneratedTemplates);
        generateServiceConfig(packageName, fileOutput,
                              serviceConfiguratorTemplateFile);
        generateHandlerConfig(packageName, fileOutput,
          templateDirectory, xmlValidationDirectory, templateFile);
    } catch (Exception e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
    }
}
```

Como podemos ver, o método `main` desta classe invoca os métodos `generateJavaCode`, `generateTemplates`, `generateServiceConfig` e `generateHandlerConfig` passando como parâmetro, além dos parâmetros recebidos, constantes definidas no início do método. Cada um destes métodos utiliza um dos *visitors* deste gerador de código de sistema para visitar a árvore sintática do programa lido pelo método `loadFile`. Cada um destes *visitors*, como mostrado, realiza a geração de um trecho de código de sistema. Para que o gerador gere outros tipos de código, como por exemplo, os de validação ou de testes de unidade, basta criar métodos, como por exemplo, `generateXMLValidation` ou `generateJUnitTest`, e adicionar sua invocação ao código mostrado anteriormente.

4.4 Modelagem do Processo de Teste com WSat

Para utilizar de forma eficaz as ferramentas construídas neste trabalho, é necessário rever e possivelmente modificar o processo de desenvolvimento de sistemas Web. Esta seção apresenta algumas considerações sobre o desenvolvimento de sistemas Web com as ferramentas apresentadas neste capítulo.

4.4.1 Fluxo de Atividades

O fluxo de atividades necessário para a utilização das ferramentas desenvolvidas por este trabalho é visto na Figura 4.10. Para o uso eficaz destas ferramentas, é necessário a existência de pessoas executando os papéis de programador visual, testador e programador. O programador visual é responsável por criar o protótipo de telas HTML do sistema. Este protótipo de telas é utilizado pelo testador para gerar o esqueleto do programa de teste WSat. Em seguida, o testador completa a descrição estrutural do sistema Web no programa WSat gerado. A partir daí, o código de sistema já pode ser gerado pelo programador. O testador ainda escreve os casos de teste do sistema Web e, por fim, os executa no sistema Web desenvolvido.

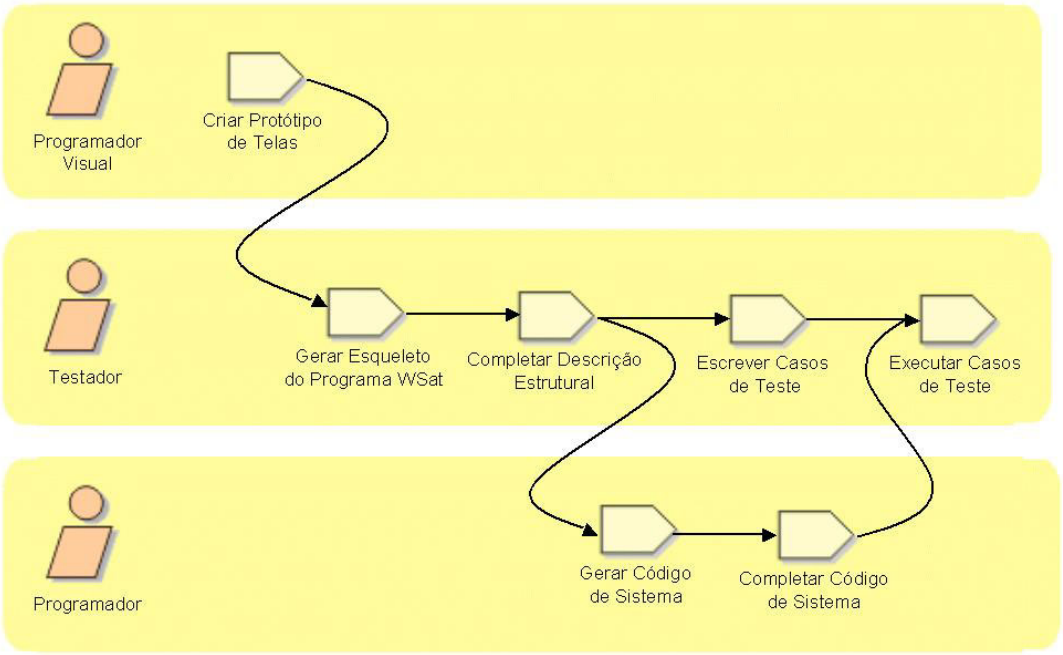

Figura 4.10: Fluxo de atividades para utilização das ferramentas.

Outras atividades no desenvolvimento de sistemas Web foram aqui omitidas por simplicidade, já que as mesmas não são afetadas por este trabalho.

4.4.2 Execução dos Testes WSat

Casos de teste de aceitação são utilizados para verificar se um dado sistema satisfaz os seus requisitos. Desta forma, é fortemente indicado que a execução satisfatória dos

casos de teste no momento da liberação de novas versões do sistema seja considerada como um requisito necessário de qualidade.

Além disto, podemos executar os casos de teste no final de cada ciclo de desenvolvimento de uma versão do sistema. Com isto, é possível utilizar o número percentual de casos de teste executados corretamente para medir o progresso do desenvolvimento do sistema. Por exemplo, se ao final de um ciclo de desenvolvimento temos dois de oito casos de uso cujos testes estão executando corretamente, podemos considerar que cerca de 25% dos requisitos do sistema estão implementados. Para analisar o progresso com relação ao tempo de desenvolvimento, devemos considerar também o grau de complexidade de implementação dos requisitos testados.

Podemos também executar os casos de teste do sistema com uma frequência ainda maior, como por exemplo, uma frequência diária. A execução diária dos casos de teste pode auxiliar a descoberta antecipada de defeitos inseridos durante o desenvolvimento de novas funcionalidades ou durante a manutenção do sistema, onde é comum a inserção de defeitos em outras partes do sistema que já estavam funcionando corretamente.

Após a liberação de uma versão do sistema, os casos de teste ainda podem ser utilizados para monitorar o funcionamento correto do sistema. Podemos executar periodicamente, por exemplo, casos de teste simples que não sobrecarregam o sistema que está atendendo às requisições dos seus usuários, mas que conseguem testar a infra-estrutura necessária para o funcionamento correto do sistema, como servidores Web e servidores de banco de dados.

Como podemos notar, quanto antes os casos de teste são criados e executados, maiores são as possibilidades de benefícios obtidos. Entretanto, deve-se observar que a antecipação da programação dos casos de testes causa um impacto negativo, a curto prazo, na produtividade.

Não foram realizados neste trabalho, estudos sobre que momentos são mais adequados para a programação dos casos de teste. Entretanto, as principais metodologias utilizadas no mercado já definem as fases para a realização dos testes. A metodologia *Extreme Programming* [3], por exemplo, indica a programação dos testes no início de cada ciclo de desenvolvimento.

4.4.3 Gerador de Código de Teste

Para a utilização eficiente do gerador de código de teste, faz-se necessário o desenvolvimento de protótipos de telas HTML. Estes protótipos, desenvolvidos geralmente por programadores visuais, devem ser criados com alguns cuidados como, por exemplo, a nomeclatura utilizada para os arquivos HTML e componentes Web contidos nos mesmos. Estes cuidados tornam os programas WSat gerados mais legíveis. Além disto, limitações da ferramenta devem ser consideradas como, por exemplo, a de não poder existir dois arquivos de mesmo nome no protótipo de telas, mesmo que em diretórios diferentes.

4.4.4 Gerador de Código de Sistema

A geração de código de sistema pode ser realizada quando os programas de teste possuem ao menos a descrição estrutural do sistema completa. Os nomes utilizados nos tipos WSat definidos são utilizados para compor o nome de classes Java, por exemplo. Com

isto, estes nomes devem ser identificadores Java válidos que satisfaçam o padrão de nomeclatura de código utilizado no desenvolvimento de sistemas Web.

Capítulo 5

Avaliação do Uso Real de WSat e das Ferramentas Associadas

Neste capítulo descrevemos o experimento realizado para avaliação da linguagem WSat e das ferramentas usadas durante o desenvolvimento de sistemas Web. Em primeiro lugar, relatamos as condições existentes para a realização do experimento. Em seguida, apresentamos os resultados obtidos com a realização do experimento assim como algumas conclusões obtidas através da análise destes resultados.

Este trabalho incentiva a realização de testes de aceitação durante o desenvolvimento de sistemas Web. Em particular, ele incentiva a criação dos testes antes mesmo da implementação do sistema, motivo suficiente para vários questionamentos pela comunidade tecnológica e por organizações de software. Entre estes questionamentos, os principais são quanto a sua viabilidade quando utilizado na prática durante o desenvolvimento de sistemas reais em ambientes reais.

Para que este trabalho seja viável é necessário que o mesmo traga benefícios reais para a qualidade dos sistemas Web desenvolvidos sem grandes impactos negativos na produtividade do desenvolvimento destes sistemas. Assumindo que é válida a utilização de testes de aceitação no desenvolvimento de sistemas Web, faz-se necessário validar a linguagem e ferramentas produzidas por este trabalho. Esta validação pode ser feita disponibilizando-se a linguagem definida e as ferramentas produzidas para a comunidade de software em geral e observando-se os resultados de sua utilização. Esta abordagem não foi possível devido ao curto espaço de tempo disponível para o trabalho.

Uma outra alternativa é a realização de um estudo empírico, utilizado na engenharia de software para estudar questões que necessitem uma validação através de resultados práticos. Estudos empíricos podem ser feitos através de estudos de caso ou experimentos. Em estudos de caso, o pesquisador possui pouco controle sobre o ambiente de desenvolvimento. Já em experimentos, o pesquisador possui condições de modificar características do ambiente que se relacionam diretamente com hipóteses (definidas no início do estudo) a serem validadas. Por ser mais sistemática e controlável e pelo curto espaço de tempo disponível, a utilização de um experimento foi a abordagem escolhida para a validação deste trabalho.

A utilização de WSat e das ferramentas produzidas por este trabalho podem trazer benefícios na corretude do sistema sem causar, no entanto, grandes impactos na produtividade do desenvolvimento, mesmo que a curto prazo. Nas próximas seções descrevemos o experimento realizado para validação da linguagem WSat e das ferramentas utilizadas no desenvolvimento de sistemas Web.

5.1 Características do Experimento Realizado

Como o aumento na corretude do sistema com a utilização de testes é esperado, este experimento avalia principalmente o impacto na produtividade obtido com a realização dos testes com WSat. Desta forma, o foco do experimento é maior nos geradores de código produzidos do que nos benefícios propriamente ditos com a utilização de WSat. Nesta seção, apresentamos as condições utilizadas para a realização do experimento. Primeiramente apresentamos o sistema escolhido para o experimento e sua forma de desenvolvimento. Após isto, mostramos os critérios utilizados para coleta e avaliação dos resultados.

5.1.1 O Sistema Desenvolvido no Experimento

Para a realização do experimento foi escolhido um sistema Web que está sendo desenvolvido por uma empresa real, cujo nome não é aqui divulgado por motivos estratégicos da empresa. Este sistema visa dar suporte a tomadas de decisão e é formado basicamente por um módulo de carga e um módulo de consulta. Apenas o módulo de consulta

(composto por relatórios de análise manual e mineração de dados) foi considerado neste experimento. Isto porque o módulo de carga não possui uma interface Web.

Um dos motivos da escolha deste sistema foi o uso de tecnologias utilizadas por diversas empresas e diretamente relacionadas aos geradores de código construídos neste trabalho. Entre elas, as principais são as seguintes:

- Java [19]: a linguagem de programação Java é requisito fundamental para a utilização do gerador de código Java apresentado neste trabalho.
- HTML [37]: linguagem utilizada para a visualização do sistema por navegadores Web e para a realização dos testes de aceitação na linguagem WSat.
- Web Handlers [2]: padrão para estruturação do atendimento a requisições de serviços Web, parte do código do mesmo pode ser gerada pelo gerador de código de sistema desenvolvido.
- Freemarker [17]: ferramenta para separação de código Java e código HTML; o sistema sendo desenvolvido tem parte de seu código de utilização do Freemarker escrita pelo gerador de código de sistema deste trabalho.
- Junit [15]: utilizado para a realização de testes de unidade, este framework pode ser utilizado para a realização de testes de unidade nos handlers do sistema. Foi verificado que parte do código que usa JUnit também poderia ser gerado pelas ferramentas desenvolvidas neste trabalho, apesar disto ainda não ter sido implementado.
- Tomcat [14]: servidor Web que tem parte de sua configuração para o sistema gerada automaticamente pelo gerador de código de sistema desenvolvido.

Além de sua aplicabilidade no trabalho, este sistema também foi escolhido devido à participação do autor na equipe de desenvolvimento do sistema Web referido, equipe composta ainda por mais dois programadores. O sistema escolhido também não estava sobre pressão com relação aos prazos de desenvolvimento, o que facilitou a realização do experimento.

5.1.2 O Experimento

Através deste experimento, esperamos validar as hipóteses de que a utilização dos testes de aceitação expressos na linguagem definida neste trabalho traz benefícios a fatores de qualidade do sistema, como por exemplo a sua corretude. Também queremos validar que a utilização dos geradores de código aqui definidos diminui o impacto na produtividade, a curto prazo, causado pela criação de testes de aceitação durante o desenvolvimento dos sistemas Web. Este experimento tem como objetivo apenas indicar a ordem de magnitude da perda de produtividade com a realização de testes de aceitação de sistemas Web. Para obter resultados significantes do ponto de vista estatístico, faz-se necessário a realização de um número maior de experimentos.

Para realizar o experimento, decidimos implementar o sistema Web escolhido de duas formas:

- Utilizando a metodologia convencional da empresa, sem a utilização de testes de aceitação e dos geradores de código aqui desenvolvidos.
- Utilizando a metodologia convencional da empresa, mas adicionando a utilização de testes de aceitação escritos em WSat e dos geradores de código aqui desenvolvidos.

Este experimento foi então realizado em duas etapas. A primeira delas foi feita utilizando a primeira opção de desenvolvimento mostrada acima. Na segunda etapa, o sistema Web escolhido foi desenvolvido de acordo com a segunda opção mostrada.

Primeira Etapa: Projeto A

Para o desenvolvimento do projeto A, foi utilizado a metodologia, ferramentas e APIs atualmente utilizadas pela equipe de desenvolvimento de sistemas Web. Esta metodologia é superficialmente descrita a seguir.

Em primeiro lugar, um protótipo de telas do sistema é criado para análise dos requisitos. Esta etapa envolve, além da própria equipe, o cliente do sistema e um programador visual.

A partir do protótipo de tela, o sistema é então desenvolvido. Os aspectos relativos a este desenvolvimento são aqui omitidos por não terem influência direta no trabalho. Entretanto, vale salientar que neste projeto o código para validação dos parâmetros recebidos pelos *handlers* é codificado em Java, sem o uso de arquivos XML para validação destes parâmetros [30].

Segunda Etapa: Projeto B

O desenvolvimento do projeto B foi realizado similarmente ao projeto A, inclusive com a participação do autor e da mesma equipe de desenvolvimento do projeto A. Por limitações de tempo e de programadores, foi realizado sempre que possível o reuso do código desenvolvido no projeto A. Os resultados finais obitidos com este reuso do código não são impactados, já que o desenvolvimento dos códigos reutilizados não são influenciados pela utilização de WSat e das ferramentas desenvolvidas neste trabalho. Detalhes sobre o desenvolvimento do projeto B são vistos a seguir.

O protótipo de tela foi reutilizado do projeto A. A partir deste protótipo, gera-se parte do código WSat para realização de testes de aceitação. Este código é alterado então pelo programador, o qual efetua os seguintes ajustes:

1. Retirada de possíveis trechos de código WSat que não interessam ser executados.
2. Indicação de quais páginas Web não são estáticas no sistema.
3. Retirada de componentes Web duplicados, como uma mesma imagem que aparece em mais de um lugar em uma mesma página.
4. Reutilizar definições de componentes Web, renomeando-as quando necessário.

Após estes ajustes, criam-se os casos de teste de aceitação para cada funcionalidade do sistema sendo desenvolvido. O desenvolvimento destes casos de teste são baseados nos

requisitos do sistema, expressos neste experimento, por exemplo, através de casos de uso.

Para iniciar o desenvolvimento do sistema foi utilizado o gerador de código de sistema. Esta ferramenta gera trechos de código Java para o sistema a partir do programa WSat escrito. Ela também gera parte da configuração do sistema no servidor Web. Neste momento, configura-se o servidor Web para que os *handlers* e páginas geradas possam funcionar semelhantemente ao protótipo de telas. Para isto também são feitos ajustes em links que possam por ventura ter sido quebrados. Ao funcionar de acordo com o protótipo de telas, o sistema mostra-se estar corretamente configurado com relação ao seu servidor Web, restando então apenas o preenchimento dos *handlers* com a implementação das funcionalidades do sistema.

Deste momento em diante, como o desenvolvimento do sistema não sofre mais influência deste trabalho, ele é executado do mesmo modo que no projeto A, sendo que são executados com frequência diária os casos de teste no sistema. Isto permite uma antecipação da descoberta de defeitos no sistema.

5.1.3 Critérios Utilizados para Coleta e Avaliação dos Dados

Durante o desenvolvimento dos Projetos A e B foram coletados dados para serem analisados posteriormente. Os tipos de dados coletados foram definidos previamente, de acordo com cada atividade do programador e do objetivo do experimento (hipóteses a serem validadas). As principais informações registradas durante as atividades foram:

- Nome da atividade
- Data de realização
- Tempo gasto
- Linhas de código escritas e geradas automaticamente
- Número de defeitos inseridos e o tempo para a sua descoberta

Todos estes dados foram então coletados e armazenados em planilhas. Com o reuso de código do projeto A no projeto B, algumas das planilhas do projeto A foram reutilizadas no projeto B. Este fato não é um problema para o que queremos analisar, já que estes dados não sofrem influência do uso de WSat e das ferramentas aqui desenvolvidas.

5.2 Análise dos Resultados

A partir das planilhas com os dados coletados durante o desenvolvimento dos dois projetos envolvidos no experimento, realizamos uma análise comparativa. Para realizar esta análise, foram utilizadas métricas de qualidade e produtividade selecionadas para que pudéssemos retirar conclusões importantes acerca da utilização prática de WSat e das ferramentas descritas aqui.

Em primeiro lugar, apresentamos as métricas utilizadas na análise comparativa. Em seguida, são mostrados os resultados desta análise.

5.2.1 Métricas Utilizadas

Para guiar a análise comparativa dos resultados obtidos, foram selecionadas métricas de qualidade e produtividade. As métricas para medir fatores de qualidade de software indicam os benefícios e os problemas obtidos nos sistemas Web desenvolvidos com a utilização de WSat e das ferramentas aqui construídas. Já as métricas utilizadas para medir a produtividade indicam o impacto na produtividade obtido no desenvolvimento de tais sistemas.

As métricas utilizadas para medir fatores de qualidade são vistas a seguir.

Facilidade de leitura e entendimento do código. Impressão dos programadores sobre a legibilidade do código de teste desenvolvido e do código de sistema gerado.

Esforço para testar novas funcionalidades. Quantidade necessária de tempo e código de teste a ser escrito para testar novas funcionalidades.

Número de defeitos inseridos durante o desenvolvimento do sistema. Número de defeitos total que foram inseridos e detectados durante o desenvolvimento do sistema.

Esforço para acompanhamento do progresso do sistema. Impressão do gerente de projeto sobre a dificuldade de acompanhar e relatar o progresso do desenvolvimento do sistema.

Já as métricas utilizadas para medir a produtividade são vistas abaixo.

Esforço de codificação. Tempo gasto e número de linhas de código escritas para todo o sistema e para cada uma de suas camadas.

Tempo para descoberta de defeitos no sistema. Tempo, a partir do início do desenvolvimento do sistema, no qual os seus defeitos foram descobertos.

Número de classes. Número de classes resultantes do desenvolvimento do projeto.

Esforço necessário para finalizar um primeiro protótipo usável. Tempo utilizado e linhas de código escritas até que o primeiro protótipo usável seja desenvolvido.

Frações de código de sistema escritas e geradas automaticamente. Valor percentual de linhas de código do sistema que foram geradas automaticamente.

5.2.2 Resultados por Fatores de Qualidade de Software e de Processo

Alguns fatores de qualidade foram selecionados, como por exemplo, legibilidade de código e testabilidade, e a eles foram associadas as métricas de qualidade utilizadas. Isto permite uma análise abrangente sobre as hipóteses definidas sobre este trabalho. Os resultados da análise comparativa por cada uma das métricas, com seu fator de qualidade associado, são mostrados a seguir.

Legibilidade: Facilidade de leitura e entendimento do código

Para avaliar o grau de facilidade da linguagem para descrição de casos de teste de aceitação e do código de sistema gerado, foi analisada de forma qualitativa a impressão dos programadores sobre estes códigos no Projeto B. Não foi possível obter dados quantitativos quanto a esta métrica, pois a leitura e o entendimento do código são ações ligadas às características psicológicas do programador, difíceis de serem abstraídas através de números.

Foram escutadas as impressões dos programadores envolvidos no desenvolvimento do sistema, incluindo o próprio autor deste trabalho, que também teve participação neste desenvolvimento. Os programadores que criaram os testes de aceitação através de WSat, de maneira geral, não mostraram dificuldade em entender o código, mesmo o escrito por outros programadores. Isto foi enfatizado principalmente quando se tratando da descrição estrutural da GUI do sistema Web. Na descrição comportamental do sistema, foi notada uma pequena dificuldade em se entender o código quando o mesmo utiliza código Java embutido.

A impressão mostrada pelos programadores com relação à geração de código também foi boa. O código Java gerado, por exemplo, mostrou-se semelhante ao escrito no Projeto A. Com relação ao código WSat gerado a partir do protótipo da interface com o usuário, foram encontradas algumas dificuldades para o seu entendimento. Estas dificuldades se deram pelo fato da utilização de uma pobre nomenclatura nos componentes Web do protótipo (esta nomenclatura é reutilizada em WSat pelo gerador de código), dificultando assim a identificação destes componentes no código WSat.

Para evitar estes problemas, podemos definir padrões de nomeclatura para serem utilizados pelos programadores visuais durante a criação dos protótipos de interface. Com isto, a lebigilidade de códigos WSat gerados tornam-se equivalentes a códigos escritos manualmente.

Testabilidade: Esforço para testar novas funcionalidades

Esta métrica foi utilizada para verificar, no Projeto B, o esforço em termos de tempo e código necessário para se testar novas funcionalidades. Nos casos de teste escritos foi notado um padrão: o número de linhas da descrição comportamental do caso de teste ficou entre 15% e 30% do número de linhas utilizados para a descrição estrutural da GUI do sistema, o que representa algo entre 13% e 24% do número de linhas total de código de teste. Como parte da descrição estrutural da GUI do sistema é parcialmente gerada a partir de protótipos de telas, uma boa parte do código WSat é automaticamente escrito, diminuindo consideravelmente o esforço para codificação dos testes. Caso a programação dos testes fosse realizada diretamente em Java, com o uso da API HttpUnit, estimamos

que haveria um acréscimo de pelo menos 50% no número de linhas de código de teste.

Entretanto, não foi encontrado nenhum padrão com relação ao tempo de criação destes testes. Isto porque o tempo de desenvolvimento mostrou-se depender dos tipos e não do número de componentes Web envolvidos. Componentes Web tipo imagens e *links* são automaticamente testados em sua descrição estrutural, diferentemente de tabelas que em geral não são representadas na descrição estrutural da GUI do sistema por poder ter tamanho e conteúdo diferentes em cada resposta.

Novas funcionalidades podem ser testadas rapidamente caso a descrição da parte estrutural do sistema já estiver realizada. Caso contrário, pode-se utilizar o gerador de código de teste para gerar boa parte do código WSat necessário.

Corretude: Número de defeitos inseridos durante o desenvolvimento do sistema

Para identificar o benefício na corretude do sistema obtido pelo uso dos geradores de código, foi contabilizado o número total de defeitos que foram inseridos durante o desenvolvimento do sistema. O número absoluto de erros não é divulgado por este ser um sistema real em desenvolvimento. Entretanto, podemos dizer que o número de erros encontrados no Projeto B foi 11,53% menor que o número de erros no Projeto A.

Foi verificado que esta diferença foi praticamente resultante da eliminação, no caso total, dos erros com validação de dados. Praticamente todo o código de validação de dados pode ser gerado sem erros a partir da sua descrição em WSat. Também foram eliminados pela geração de código alguns defeitos na configuração dos Handlers.

A divisão de tarefas realizadas pela equipe de desenvolvimento foi feita de forma a evitar que um mesmo código fosse escrito por um programador, o que beneficiaria a corretude do segundo projeto pela experiência do programador obtida durante a implementação do primeiro projeto.

Facilidade de Gerenciamento: Esforço para acompanhamento do progresso do sistema

Este métrica foi utilizada para verificar se os casos de testes de aceitação auxiliam o acompanhamento do progresso do desenvolvimento do sistema. Para isto, foi analisada a impressão do gerente dos projetos A e B.

Para indicar o progresso no Projeto A, o gerente de projeto apresentou um percentual do código já escrito sobre o código imaginado como necessário para a total da implementação do sistema. Como não tem-se como saber previamente qual a quantidade exata de código para se implementar um sistema, este número não é calculado com muita segurança pelo gerente, gerando questionamentos em certos casos.

Já no Projeto B, o gerente de projeto indicou o progresso do projeto através do número percentual de casos de teste de aceitação que executam corretamente, abordagem utilizada por XP [3]. Foi verificada que esta abordagem permitiu uma maior segurança no cálculo do progresso do projeto e a possibilidade de comprovação quando questionado. A primeira abordagem também pode ser utilizanda aqui, entretanto, para para medir o progresso de funcionalidades cujos casos de testes ainda não são executados corretamente. Isto faz com que os números que descrevem o progresso do desenvolvimento do sistema não aumentem bruscamente, já que vão considerar também as funcionalidades que ainda não estão 100% prontas.

Entretanto, alguns casos de teste testavam funcionalidades com maior facilidade de implementação que outros. De fato, não analisar no cálculo do progresso a dificuldade de implementação da funcionalidade testada pode causar um efeito colateral. Por exemplo, dizer que 80% do sistema foi implementado com 8 semanas de desenvolvimento não quer dizer que os outros 20% serão implementados em 2 semanas, pois isto vai depender da dificuldade das funcionalidades restantes a ser implementada, e não do número de casos de testes restantes. Uma solução comumente utilizada para este problema é o uso de pesos que indicam a dificuldade da funcionalidade testada por cada caso de teste.

5.2.3 Resultados por Fatores de Produtividade

O desenvolvimento dos geradores de código é uma tentativa de amenizar o impacto negativo na produtividade, a curto prazo, causado pela criação dos testes de aceitação. Para avaliar o impacto causado pelo uso da WSat e das ferramentas criadas, foram utilizadas as métricas para medir produtividade. Os resultados da análise comparativa por cada uma das métricas de produtividade são mostrados a seguir.

Esforço de codificação

Esta métrica foi utilizada na análise dos dados obtidos em cada projeto com o intuito de medir o tempo gasto e número de linhas de código escritas para todo o sistema, incluindo os testes, e para determinados tipos de código. Esta métrica pode identificar o valor percentual do esforço gasto em atividades executadas pelos geradores de código desenvolvidos.

A Tabela 5.1 apresenta a comparação entre o total do tempo gasto e do número de linhas de código escritas em ambos os projetos. O número de linhas de código escritas no protótipo não foi considerado por este não ser escrito diretamente pelos programadores, mas sim por web designers através de editores de HTML.

Projeto	Tempo (h)	Número de Linhas de Código
A	220	7583
B	208	7614

Tabela 5.1: Tempo total gasto e número de linhas de código escritas em cada projeto.

Como pode ser visto, houve uma redução de 5,45% em relação ao tempo de desenvolvimento e um aumento de 1,73% do número de linhas de código escritas. Estes dados podem ser melhor analisados através de valores percentuais por tipos de código, conforme ilustrado na Tabela 5.2. Esta tabela mostra a divisão percentual do tempo e número de linhas de código do Projeto B, de acordo com o tipo de código produzido. Também é apresentada a variação dos valores obtidos no Projeto B com relação ao Projeto A. Em outras palavras, o percentual de aumento ou redução dos valores com relação ao Projeto A .

Pela variação de tempo mostrada na tabela 5.2, podemos ver que o tempo de codificação de teste não existiu no Projeto A. Já o código para configuração do servidor Web, por exemplo, foi totalmente gerado pelo gerador de código de sistema. O ganho com a geração de trecho de códigos de sistema mostrou-se suficiente para a programação

Tipo de Código	Tempo (%)		Número de Linhas de Código (%)	
	Proj B	Variação	Proj B	Variação
Protótipo de Tela	11,54	0,10	-	-
Testes de Aceitação	2,40	100,00	2,68	100,00
Handler de Apresentação	2,71	-37,45	4,03	-0,09
Validação de Dados	1,92	-60,19	0,13	-95,24
Configuração do Servidor Web	0,48	-75,06	0,33	0,41
Configuração dos Handlers	0,00	-100,00	0,97	-0,62
Handler de Processamento	4,02	-39,34	6,05	-0,13
Fachada do Sistema e sub-módulos	76,93	0,00	85,68	0,00

Tabela 5.2: Tempo gasto e número de linhas de código do Projeto B e a variação obtida com relação ao Projeto A.

dos testes de aceitação. Entretanto, se forem criados um número grande de casos de teste, o ganho de tempo com a geração de código não deve ser suficiente para compensar totalmente o tempo gasto com a definição dos testes.

Podemos notar também a redução de mais de 95% do número de linhas de código utilizados para a validação dos dados no Projeto A. Isto se deve ao superior nível de abstração de WSat com relação a validação de dados através de linguagens de programação. Entretanto, a utilização direta de APIs de validação de dados como, por exemplo, a de arquivos XML, podem obter resultados similares. Desta forma, o aumento do número de linhas de código escritas no Projeto B só não foi mais expressivo devido a quase eliminação total de código Java escrito para a validação de dados.

Geradores de código de sistema a partir de código HTML podem ser construídos e utilizados diretamente a partir dos protótipos de tela. Entretanto, as informações adicionais encontradas em WSat como, por exemplo, a indicação das páginas estáticas e dinâmicas e dos valores válidos para parâmetros de formulários, permitem a geração de uma maior quantidade e qualidade de código.

Tempo para descoberta de defeitos no sistema

Esta métrica foi utilizada em ambos os projetos para identificar os tempos entre a inserção e a detecção de um defeito nos sistemas produzidos. O tempo de inserção do erro é calculado baseando-se na data em que o trecho de código no qual ele se encontra foi escrito. A detecção de defeitos no Projeto A foi feita pelo execução manual e diária das funcionalidades do sistema desenvolvidas no dia. Já a detecção de defeitos no Projeto B foi feita pelo execução diária dos testes escritos em WSat que cobrem todas as funcionalidade do sistema Como boa parte do código desenvolvido no Projeto A foi reutilizado no Projeto B, os erros detectados nos dois projetos foram praticamente os mesmos. Somente alguns dos erros encontrados no Projeto A foram eliminados no Projeto B com a utilização do gerador de código. A Tabela 5.3 mostra uma análise sobre este tempo.

Podemos concluir pelos dados mostrados que a utilização de testes de aceitação reduz o tempo de descoberta de alguns defeitos. Isto porque, com a programação dos testes,

Tempo para Descoberta	Projeto A (%)	Projeto B (%)
1 dia	38,46	46,16
2 dias	42,31	38,46
3 a 7 dias	19,23	15,38

Tabela 5.3: Percentual de defeitos encontrados pelo seu tempo para descoberta.

podemos executá-los com uma maior frequência, inclusive de forma regressiva, cobrindo uma maior área do sistema. Em sistemas de médio e grande porte, a programação dos testes se torna mais importante, pois é mais difícil a realização de testes de forma manual devido a grande quantidade de funcionalidades a ser testada. No experimento, foram descobertos principalmente defeitos de concorrência e de cálculos errados quando com valores de consulta extremos.

Número de classes

Nesta métrica, fizemos a contagem do número resultante de classes criadas no desenvolvimento dos projetos A e B. A contagem foi dividida entre classes escritas pelo programador e classes geradas pelos geradores de código utilizados. O resultado desta contagem pode ser visto na Tabela 5.4.

Projeto	Número de Classes		
	Escritas	Geradas	Total
A	48	0	48
B	44	4	48

Tabela 5.4: Número total de classes por projeto.

O número equivalente de classes resultantes em cada projeto era de certa forma esperado, já que o uso das ferramentas aqui construídas não modifica a metodologia de desenvolvimento. As ferramentas apenas geram parte do código que já ia ser construído.

Entretanto, 2 Handlers de apresentação gerados automaticamente no Projeto B foram removidos. Isto se deu ao fato de que, com o uso do padrão Web Handlers, pôde-se reutilizar um mesmo Handler de apresentação para diferentes serviços. O gerador de código construído neste trabalho não tem como identificar estes casos de reuso, gerando assim classes desnecessárias que podem ser posteriormente removidas do projeto.

Por fim, podemos dizer que a utilização do gerador de código no Projeto B resultou na geração automática de 8,33% do número final de classes. O número excessivo de classes desnecessárias geradas pode se tornar problemático no caso de sistemas de médio e grande porte.

Esforço necessário para finalizar um primeiro protótipo usável

O tempo e linhas de código necessárias para se obter um primeiro protótipo usável nos dois projetos foram analisados. Um protótipo ser usável significa aqui ter algumas funcionalidades prontas, porém sem nenhuma garantia de robustez, como validação de dados, por exemplo. O resultado desta análise pode ser visto na Tabela 5.5.

Projeto	Tempo (h)
A	102
B	99

Tabela 5.5: Tempo para finalizar primeiro protótipo usável.

De acordo com estes dados, podemos ver que a utilização dos geradores de código resultou em uma redução de tempo gasto no desenvolvimento do primeiro protótipo usável. Entretanto, o ganho de tempo mostrado é pequeno, tornando-se significativo somente se considerarmos que a maior parte do trabalho com a validação dos dados recebidos pelos Handlers já foi realizado neste ponto no Projeto B, pois o mesmo é feito durante a definição dos tipos em WSat.

Frações de código de sistema geradas automaticamente

Esta métrica, utilizada apenas no Projeto B, indica o percentual de linhas de código do sistema que foram geradas automaticamente por tipos de código. O resultado deste cálculo pode ser visto na Tabela 5.6.

Tipo de Código	Linhas de Código Geradas (%)
Handler de Apresentação	23,80
Validação de Dados	0,00
Configuração do Servidor	100,00
Configuração dos Handlers	100,00
Handler de Processamento	16,95
Outros	0,00

Tabela 5.6: Percentual de código gerado automaticamente.

Os dados apresentados nos mostram que a utilização dos geradores de código trouxe uma geração de 4,52% do total das linhas de código. Este valor se torna bastante significativo quando se observa o tipo dos códigos que foram gerados. Mesmo que sem grandes complexidades, estes códigos fornecem em geral um trabalho repetitivo e entediante.

Capítulo 6

Conclusões

Neste capítulo apresentamos conclusões obtidas sobre a definição da linguagem WSat, assim como sobre as ferramentas de apoio desenvolvidas. Trabahos relacionados são comentados, assim como são sugeridos trabalhos futuros que permitam melhorar os resultados alcançados.

Neste capítulo apresentamos nossas conclusões, indicando primeiramente as contribuições fornecidas com a definição da linguagem WSat, assim como a construção das suas ferramentas de apoio. Em seguida, discutimos sobre trabalhos relacionados a este. Por fim, apresentamos sugestões de trabalhos futuros para melhorar os resultados obtidos.

6.1 Conclusões

A principal contribuição deste trabalho é a definição da linguagem WSat, utilizada para descrição de casos de teste de aceitação de sistemas Web. Esta linguagem possui características que lhe permitem possuir um alto nível de abstração e reuso de código. Em WSat, os testes são estruturados em duas partes. A primeira delas é a descrição estrutural da GUI do sistema, onde são definidos tipos para representar os componentes Web presentes na GUI. Nestes tipos são definidas propriedades que indicam as características que os componentes Web testados devem satisfazer. Também é possível definir tipos de forma anônima, tornando a descrição estrutural mais concisa.

A segunda parte dos testes é a descrição comportamental desta GUI, onde os tipos definidos são utilizados para manipular os componentes Web simulando a execução do sistema. Através dos serviços fornecidos pelos tipos especiais previamente definidos em WSat, podemos simular ações dos usuários como, por exemplo, cliques em *links* e o preenchimento e submissão de formulários. Serviços também podem ser utilizados para verificar o conteúdo dinâmico gerado por páginas Web.

Para não perder abstração, WSat permite um conjunto limitado de expressões e operadores lógicos. Entretanto, não foi definida para WSat uma forma abstrata de parametrização dos dados de teste e acessos a banco de dados, assim como a verificação de performance e concorrência. Atualmente, estas tarefas são realizadas com a utilização de código Java [19] embutido, o que pode comprometer a abstração do código.

Outra contribuição importante deste trabalho foi a criação de ferramentas de apoio a realização dos testes WSat, como um ambiente de execução para WSat e geradores de código. O ambiente de execução foi utilizado para validar e viabilizar o uso de WSat. Para sua implementação, utilizamos o ambiente de execução de Java e APIs externas, como por exemplo, a API HttpUnit [18]. Com isto, conseguimos reduzir boa parte do esforço necessário para a sua implementação

Para reduzir o impacto causado na produtividade, a curto prazo, pela criação dos casos de teste, foram definidos geradores de código. Isto só foi possível devido à característica de WSat descrever de forma explícita a GUI do sistema testado a partir dos tipos WSat definidos. A construção de geradores de código a partir de outras linguagens de teste que não WSat são praticamente inviáveis, devido à mistura das descrições estruturais e comportamentais da GUI do sistema .

Estes geradores utilizam o *parser* de WSat construído no ambiente de execução de WSat. Isto porque a árvore sintática de WSat foi estruturada de acordo com o padrão *Visitor*, desacoplando o código da árvore sintática do código que manipula a mesma.

O primeiro dos geradores criados é um gerador de código de teste, capaz de gerar parte do código que descreve a estrutura da GUI do sistema testado a partir de protótipos de interface em HTML. O segundo é um gerador de código de sistema Web, onde é possível gerar código Java seguindo padrões de projeto como, por exemplo, o padrão

Web Handler [2], assim como códigos de validação de parâmetros e configuração do sistema.

A alteração ou adição de tipos de código de sistema gerados é facilmente realizada, devido a utilização de *templates* de código e da facilidade de implementação de novos *visitors*. Para se tornar mais poderoso, este gerador faz uso de informações adicionais que podem ser adicionadas em programas WSat, através da palavra chave `static` e da cláusula `validate`, por exemplo.

Outra contribuição fornecida foi a especialização do padrão de projeto *Web Handler* para a utilização da API de validação de parâmetros através de arquivos XML e da API FreeMarker, [17] para separação entre códigos Java e HTML.

Foi realizado um experimento com um sistema Web real, de pequeno porte, para validar e analisar a viabilidade da utilização de WSat e das ferramentas de apoio desenvolvidas. Este experimento provou que a utilização de WSat e dos geradores pode trazer um ganho na corretude dos sistemas sem comprometer a produtividade do seu desenvolvimento. De fato, o ganho com a geração de trecho de códigos de sistema mostrou ter sido suficiente para a programação dos programas WSat. Entretanto, se a cobertura do sistema pelos testes fosse ampliada com um número grande de casos de teste, o ganho de tempo com a geração de código não seria suficiente para compensar totalmente o tempo gasto com a definição dos casos de teste.

Para provar os benefícios gerados pela utilização do processo de teste e ferramentas de apoio definidos, faz-se necessária a realização de um número maior de experimentos. Entretanto, através do experimento realizado pudemos verificar a ordem de magnitude da perda de produtividade, a curto prazo, obtida com a utilização do processo de teste definido. O fato de obtermos uma redução no tempo de desenvolvimento ao invés do acréscimo que era esperado sugere que, mesmo obtendo resultados inferiores com a realização de outros experimentos, não devemos verificar impactos negativos significantes na produtividade com a realização dos testes.

Verificamos também, através da impressão dos programadores participantes do experimento, um maior nível de abstração e facilidade de entendimento do que programas escritos em outras linguagens de teste. Com isto, os programadores aparentaram estar mais dispostos para a realização de atividades de teste.

Inicialmente WSat foi projetada para visando a realização de testes de aceitação em sistemas Web. Entretanto, com a realização do experimento, verificamos também a utilidade de WSat para a realização de outros tipos de teste de sistemas Web que não os de aceitação, como os testes de performance, concorrência e carga.

Estre trabalho mostra que testes de aceitação verificam a corretude do sistema, podendo também ser utilizados para acompanhar o progresso dos sistemas em desenvolvimentos, assim como para o monitoramento dos sistemas produzidos e em operação. Além disto, eles provêem informações úteis para possíveis gerações de código. Esperamos com isto incentivar a utilização deste tipo de teste no desenvolvimento de sistemas Web.

Na análise de duas ferramentas de teste realizada no Capítulo 2, verificamos o suporte inadequado de suas linguagens de teste. Mesmo poderosas, estas linguagens não apresentam um nível de abstração satisfatório. Além disto, verificamos que, embora o modo mais fácil para criação de casos de teste é através de ferramentas que gravem as ações de usuários durante a execução do sistema, o papel das linguagens de teste con-

tinua importante. Isto porque para a realização de testes mais elaborados, é necessário recorrer para recursos contidos nas linguagens de teste que os casos de teste são gravados. Este mesmo fato acontece durante a manutenções dos casos de teste existentes, assim como na utilização da prática *Test First Design* [12], indicada pela metodologia *Extreme Programming* [3], onde os testes são criados antes da implementação do código testado.

Para analisar ferramentas e linguagens de teste disponíveis no mercado, definimos critérios de análise que levam em consideração os suportes fornecidos por estas ferramentas e o impacto dos mesmos na qualidade do processo de desenvolvimento e dos produtos desenvolvidos. Estes critérios identificam as principais características que estas ferramentas e linguagens devem possuir, podendo ser utilizados como base para a análise ou desenvolvimento de outras ferramentas e linguagens de teste.

6.2 Trabalhos Relacionados

A linguagem TestTalk [28], utilizada para descrição de casos de testes, inclusive testes de aceitação, foi definida com o objetivo de ser uma linguagem compreensiva e portável. Programas escritos em TestTalk, para serem executados, necessitam ser compilados para uma outra linguagem de teste. Desta forma, programas TestTalk são, a princípio, independente de plataforma e de ferramentas de teste.

TestTalk utiliza termos específicos do domínio que podem ser definidos pelo testador. Para cada termo definido, uma regra de transformação tem que ser definida para que este termo seja transformado em código executável na linguagem da ferramenta de teste destindo. Os programas TestTalk são estruturados de forma a separar os dados de entrada, a execução e a verificação do resultado de cada passo do teste. Entretanto, só esta estruturação dos programas e a utilização de termos do domínio não conseguem produzir o nível de abstração conseguido por WSat com a representação de tipos que representam componentes Web.

Ferramentas como a QARun, mostrada no Capítulo 2, possuem o poder de programação encontrados nas linguagens convencionais. Este poder, no entanto, traz complexidades desnecessárias ao código de teste, dificultando a sua criação e manutenção. Facilidades como a gravação de casos de teste reduzem bastante este problema durante a criação dos testes, sem poder auxiliar significantemente nas manutenções.

Outras ferramentas desenvolvidas e disponíveis são baseadas em XML. Exemplos disto são a JXWeb, também mostrada no Capítulo 2, e a Canoo WebTest [10], lançada no mercado recentemente. Estas ferramentas fornecem a facilidade do aprendizado, já que a linguagem XML é bastante difundida no mercado. Embora de fácil aprendizado, XML não foi construída com foco na sua programação por pessoas, mas sim por programas durante as suas comunicações com outros programas. Desta forma, o nível de abstração destas linguagens também fica comprometido.

6.3 Trabalhos Futuros

Com relação à linguagem WSat, sugerimos como trabalho futuro a definição de sua semântica formal. Além disto, para uma melhor estruturação dos programas WSat, a

linguagem pode ser estendida através da criação de módulos. Isto facilita a manipulação do código de teste, principalmente quando se tratando de testes em sistemas Web de médio e grande porte. Podemos também definir formas abstratas de parametrização dos dados de teste, acessos a banco de dados, entre outros, com o intuito de reduzir e até eliminar a utilização de código Java embutido, garantindo a abstração do código.

As associações entre páginas Web representadas por como *links* e ações de formulários HTML não foram exploradas neste trabalho. Estudando mais profundamente estas associações, podemos verificar se as mesmas podem permitir que os geradores de código gerem de outros tipos de código que não os já implementados.

O ambiente de execução de WSat pode ser incrementado com verificações semânticas como, por exemplo, a verificação dos tipos. Além disto, podemos implementar a execução dos serviços fornecidos por tipos WSat que são utilizados para a verificação do conteúdo dinâmico de páginas Web.

Para o gerador de código de teste, mesmo sem ser atividades de pesquisa, é interessante sua extensão através da criação de uma ferramenta para gravação de casos de teste de aceitação de sistemas Web na linguagem WSat, visando fornecer uma alternativa mais simples para a criação dos casos de teste. A implementação deste gravador pode se utilizar do gerador de código de teste, capaz de extrair a definição de tipos WSat a partir de arquivos HTML.

O gerador de código de sistema pode ser estendido, por exemplo, para gerar código de validação de parâmetros através de JavaScript [13]. O gerador pode realizar o carregamento de *visitors* dinamicamente, facilitando assim a sua extensão para geração de novos tipos de código. Otmizações podem ser realizadas no gerador, como por exemplo, a geração de código de sistema percorrendo-se apenas uma vez a árvore sintática dos programas WSat ao invés da utilização de um *visitor* para cada tipo de código gerado, que é mais simples porém menos eficiente.

Limitações das ferramentas podem ser removidas e funcionalidades novas podem ser adicionadas. A implementação atual do gerador de código de teste, por exemplo, não permite o uso de arquivos HTML de mesmo nome no protótipo de telas, mesmo que em diretórios diferentes. Para a utilização eficaz das ferramentas produzidas, também faz-se necessário a implementação de interfaces de usuários que facilitem sua execução.

Por fim, sugerimos também a realização de um estudo de caso utilizando também um sistema real, porém de médio ou grande porte, visando identificar possíveis problemas, como por exemplo, um grande número de linhas de código de teste. Também é interessante realizar a geração de código de sistema em outros padrões e tecnologias, como a geração de código C++ [40], por exemplo.

Apêndice A

Gramática da Linguagem WSat

A seguir apresentamos a documentação da gramática implementada pelo *parser* [1] desenvolvido para a linguagem WSat. Esta documentação é gerada pela ferramenta jjDoc que acompanha a ferramenta JavaCC [35].

```
TOKEN :
{
    < INTEGER_LITERAL:
          <DECIMAL_LITERAL> (["l","L"])?
        | <HEX_LITERAL> (["l","L"])?
        | <OCTAL_LITERAL> (["l","L"])?
    >
  |
    < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
  |
    < #HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ >
  |
    < #OCTAL_LITERAL: "0" (["0"-"7"])* >
  |
    < FLOATING_POINT_LITERAL:
          (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)?
                                              (["f","F","d","D"])?
        | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
        | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
        | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
    >
  |
    < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
  |
    < CHARACTER_LITERAL:
        "'"
        (   (~["'","\\","\n","\r"])
          | ("\\"
              ( ["n","t","b","r","f","\\","'","\""]
              | ["0"-"7"] ( ["0"-"7"] )?
```

```
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )
      "'"
  >
|
  < STATIC: "static" >
|
  < NULL: "null" >
|
  < STRING_LITERAL:
      "\""
      (   (~["\"","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )*
      "\""
  >
|
  < BOOLEAN: "true" | "false" >
|
  < WEB_PAGE: "WebPage" >
|
  < WEB_FORM: "WebForm" >
|
  < WEB_LINK: "WebLink" >
|
  < WEB_IMAGE: "WebImage" >
|
  < TEXT: "Text" >
|
  < SELECT_BOX: "SelectBox" >
|
  < EDIT_BOX: "EditBox" >
|
  < BUTTON: "Button" >
|
  < ACTION : "action">
|
  < INDEX_VALUE : "indexValue">
|
```

```
  < MAX_LENGTH : "maxLength">
|
  < METHOD : "method">
|
  < NAME : "name">
|
  < SIZE : "size" >
|
  < SRC : "src" >
|
  < VALUE : "value">
|
  < URL : "url">
|
  < TYPE : "type">
|
  < ENUMERATION : "enumeration">
|
  < MODULE : "module">
|
  < TEMPLATE : "template">
|
  < EQUALS : "=">
|
  < LBRACE: "{" >
|
  < MAX_EXCLUSIVE_VALUE : "maxExclusiveValue">
|
  < MAX_INCLUSIVE_VALUE : "maxInclusiveValue">
|
  < MIN_EXCLUSIVE_VALUE : "minExclusiveValue">
|
  < MIN_INCLUSIVE_VALUE : "minInclusiveValue">
|
  < NEW : "new">
|
  < OPTIONAL : "optional">
|
  < RBRACE: "}" >
|
  < RETURN : "return" >
|
  < SEMICOLLON : ";" >
|
  < TESTCASE : "testCase">
|
```

```
    < VALIDATE : "validate">
  |
    < JAVA_CODE : "<@" ( "\\@>" | ~[ "@" ] | "@" ~[ ">" ] )* "@>" >
  |
    < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
  |
    < #LETTER: ["_","a"-"z","A"-"Z"] >
  |
    < NOT_NEGATIVE_NUMBER: ["1"-"9"] (<DIGIT>)* >
  |
    < NUMBER: ("-")? <DECIMAL_LITERAL> >
  |
    < #DIGIT: ["0"-"9"] >
}

Type := ( <WEB_PAGE> | <WEB_LINK> | <WEB_IMAGE> | <WEB_FORM> |
    <TEXT> | <SELECT_BOX> | <EDIT_BOX> | <BUTTON> | <IDENTIFIER> )
Start := ( WebComponentDeclaration )+
WebComponentDeclaration :=
         MethodDeclaration
    |    WebPageDeclaration
    |    WebPageComponentDeclaration
    |    WebFormComponentDeclaration
    |    TestCaseDeclaration
WebPageComponentDeclaration :=
         WebFormDeclaration
    |    WebLinkDeclaration
    |    WebImageDeclaration
    |    TextDeclaration
WebFormComponentDeclaration :=
         SelectBoxDeclaration
    |    EditBoxDeclaration
    |    ButtonDeclaration
WebLinkComponentDeclaration :=
         WebImageDeclaration
    |    TextDeclaration
WebPageDeclaration := ( ( <STATIC> )? <WEB_PAGE> <IDENTIFIER>
    <LBRACE> ( <TEMPLATE> <EQUALS> <STRING_LITERAL> <SEMICOLLON> |
               <MODULE> <EQUALS> <STRING_LITERAL> <SEMICOLLON> |
               WebPageComponentDeclaration |
               PropertyDeclaration )* <RBRACE> )
WebAttributeDeclaration := ( Name | <TYPE> ) <EQUALS>
    ( Literal | MultiArrayStringDeclaration ) <SEMICOLLON>
WebFormDeclaration := ( <WEB_FORM> <IDENTIFIER> <LBRACE>
    ( PropertyDeclaration | WebAttributeDeclaration |
      WebFormComponentDeclaration )* <RBRACE> )
```

```
WebLinkDeclaration := ( <WEB_LINK> <IDENTIFIER> <LBRACE>
    ( PropertyDeclaration | WebAttributeDeclaration |
      WebLinkComponentDeclaration )* <RBRACE> )
WebImageDeclaration := ( <WEB_IMAGE> <IDENTIFIER> <LBRACE>
    ( WebAttributeDeclaration )* <RBRACE> )
TextDeclaration := ( <TEXT> <IDENTIFIER> <LBRACE>
    ( WebAttributeDeclaration )* <RBRACE> )
SelectBoxDeclaration := ( <SELECT_BOX> <IDENTIFIER> <LBRACE>
    ( WebAttributeDeclaration | ValidateDeclaration )* <RBRACE> )
EditBoxDeclaration := ( <EDIT_BOX> <IDENTIFIER> <LBRACE>
    ( WebAttributeDeclaration | ValidateDeclaration )* <RBRACE> )
ButtonDeclaration := ( <BUTTON> <IDENTIFIER> <LBRACE>
    ( WebAttributeDeclaration )* <RBRACE> )
ValidateDeclaration := <VALIDATE> <LBRACE>
    ( <TYPE> <EQUALS> <STRING_LITERAL> <SEMICOLLON> |
      <MAX_INCLUSIVE_VALUE> <EQUALS> <STRING_LITERAL> <SEMICOLLON> |
      <MAX_EXCLUSIVE_VALUE> <EQUALS> <STRING_LITERAL> <SEMICOLLON> |
      <MIN_INCLUSIVE_VALUE> <EQUALS> <STRING_LITERAL> <SEMICOLLON> |
      <MIN_EXCLUSIVE_VALUE> <EQUALS> <STRING_LITERAL> <SEMICOLLON> |
      <OPTIONAL> <EQUALS> <BOOLEAN> <SEMICOLLON> |
      <ENUMERATION> <EQUALS> ArrayString <SEMICOLLON> )* <RBRACE>
ArrayString := ( <LBRACE> ( <STRING_LITERAL>
    ( "," <STRING_LITERAL> )* )? <RBRACE> )
MultiArrayStringDeclaration := ( <LBRACE> ( <LBRACE> ArrayValue ","
    ArrayValue <RBRACE> ( "," <LBRACE> ArrayValue ","
    ArrayValue <RBRACE> )* )? <RBRACE> )
ArrayValue := ( <STRING_LITERAL> | <NULL> )
PropertyDeclaration := ( <IDENTIFIER> <IDENTIFIER> <SEMICOLLON> )
TestCaseDeclaration := <TESTCASE> <IDENTIFIER> Block
Block := "{" ( BlockStatement )* "}"
JavaCode := <JAVA_CODE>
BlockStatement := LocalVariableDeclaration
    |    Statement
Statement := Block
    |    EmptyStatement
    |    StatementExpression ";"
    |    ReturnStatement
    |    JavaCode
Assignment := AssignbleAccess <EQUALS> Expression
ReturnStatement := <RETURN> ( Expression )? ";"
EmptyStatement := ";"
StatementExpression := Assignment
    |    MethodCall
LocalVariableDeclaration := ( Type VariableDeclarator
    ( "," VariableDeclarator )* <SEMICOLLON> )
VariableDeclarator := ( Name ( "=" Expression )? )
```

```
Expression := ComposedExpression
    |    SimpleExpression
ComposedExpression := CastExpression
    |    "(" Expression ")"
SimpleExpression := Literal
    |    Access
    |    JavaCode
Access := ( SystemFunctionCall | AssignbleAccess )
AssignbleAccess := VariableAccess ( ObjectMemberAccess )*
VariableAccess := ( Name )
ObjectMemberAccess := ( ObjectMethodCall | FieldAccess )
FieldAccess := "." Name
ObjectMethodCall := "." FunctionCall
MethodCall := ( SystemFunctionCall | VariableAccess
    ( ObjectMemberAccess )* )
CastExpression := "(" Type ")" Expression
AllocationExpression := <NEW> Name ( Arguments )
SystemFunctionCall := <IDENTIFIER> Arguments
FunctionCall := <IDENTIFIER> Arguments
Arguments := "(" ( ArgumentList )? ")"
ArgumentList := Expression ( "," Expression )*
Literal := ( <INTEGER_LITERAL> | <FLOATING_POINT_LITERAL> |
     <CHARACTER_LITERAL> | <STRING_LITERAL> |
     BooleanLiteral | NullLiteral )
BooleanLiteral := ( <BOOLEAN> )
NullLiteral := <NULL>
Name := <IDENTIFIER>
MethodDeclaration := Type Name FormalParameterList Block
FormalParameterList := "(" ( FormalParameter
    ( "," FormalParameter )* )? ")"
FormalParameter := Type Name
```

Bibliografia

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, January 1986.

[2] Gibeon Aquino and Paulo Borba. Web Handlers. In *First Latin American Conference on Pattern Languages of Programming, Sugarloaf PLoP*, Rio de Janeiro, Brazil, 3th–5th October 2001.

[3] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[4] Grady Booch, Ivar Jacobson, and James Rumbaugh. *Unified Modeling Language – User's Guide*. Addison–Wesley, 1999.

[5] Tom Christiansen and Nathan Torkington. *Perl Cookbook*. O'Reilly & Associates, August 1998.

[6] World Wide Web Consortium. Document Object Model. http://www.w3.org/DOM.

[7] World Wide Web Consortium. HTTP - Hypertext Transfer Protocol, 2000. http://www.w3.org/Protocols.

[8] Compuware Corporation. QARun. http://www.compuware.com.

[9] Microsoft Corporation. Microsoft Corporate Web Site, 2002. http://www.microsoft.com.

[10] Canoo Engineering. Canoo WebTest, 2002. http://webtest.canoo.com.

[11] Software Quality Engineering. The software testing & quality engineering magazine. http://www.stqemagazine.com.

[12] Michael Feathers. Test First Design, 2000. http://www.xprogramming.com/xpmag/test_first_ intro.htm.

[13] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, 4th edition, December 2001.

[14] Apache Software Foundation. Jakarta Tomcat. http://jakarta.apache.org/tomcat.

[15] Erich Gamma and Kent Beck. JUnit. http://www.junit.org.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, October 1994.

[17] Benjamin Geer and Mike Bayer. FreeMarker. http://freemarker.sourceforge.net.

[18] Russell Gold. HttpUnit, 2000. http://httpunit.sourceforge.net.

[19] Bill Goslin and Guy Steele. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[20] Michael Halvorson. *Microsoft Visual Basic .NET Step by Step*. Microsoft Press, February 2002.

[21] Rick Hower. Web site test tools and site management tools. http://www.softwareqatest.com/qatweb1.html.

[22] Jason Hunter and William Crawford. *Java Servlet Programming*. O'Reilly & Associates, second edition, April 2001.

[23] IDG.net. JavaWorld. http://www.javaworld.com.

[24] Bill la Forge. JXUnit. http://jxunit.sourceforge.net.

[25] Bill la Forge. Quick: XML Made Easy. http://jxquick.sourceforge.net/quick3.

[26] Bill la Forge. JXWeb, 2002. http://sourceforge.net/projects/jxweb.

[27] Chang Liu and Debra J. Richardson. Programming Languages Considered Harmful in Writing Automated Software Tests. Technical Report 99-09, Information & Computer Science, University of California, Irvine, February 1999.

[28] Chang Liu and Debra J. Richardson. TestTalk: A Comprehensive Testing Language. In *the 14th IEEE International Conference on Automated Software Engineering, Doctoral Symposium*, Cocoa Beach, Florida, USA, October 1999.

[29] SD Magazine. Software development magazine. http://www.sdmagazine.com.

[30] Brett McLaughlin. Validation with Java and XML Schema. http://www.javaworld.com/javaworld/jw-09-2000/jw-0908-validation.html.

[31] Brett McLaughlin. *Java and XML*. O'Reilly & Associates, 2000.

[32] SYS-CON Media. Java Developers Journal. http://www.javadevelopersjournal.com.

[33] Bertrand Meyer. *Object Oriented Software Construction*. Prentice Hall, second edition, 1997.

[34] Sun Microsystems. Java 2 Runtime Environment, Standard Edition. http://java.sun.com/j2se.

[35] Sun Microsystems. Java Compiler Compiler (JavaCC) - The Java Parser Generator, 2000. http://www.webgain.com/products/java_cc.

[36] Nguyen. *Testing Applications on the Web: Planning for Internet Based Systems.* John Wiley, 2000.

[37] Thomas Powell. *HTML: The Complete Reference.* McGraw Hill, third edition, 2000.

[38] Andy Quick. JTidy. http://sourceforge.net/projects/jtidy.

[39] George Reese. *Database Programming with JDBC & Java.* O'Reilly & Associates, second edition, August 2000.

[40] Herbert Schildt. *C++: The Complete Reference.* McGraw Hill, 3th edition, June 1998.

[41] ExpoWorks Technology. Expoimóvel - Sistema de Busca de Imóveis. http://www.expoimovel.com.br.