PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

CONJUNTO DE CARACTERÍSTICAS PARA TESTE DE DESEMPENHO: UMA VISÃO A PARTIR DE FERRAMENTAS

LEANDRO TEODORO COSTA

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Avelino Francisco Zorzo

Porto Alegre
2012

C837c Costa, Leandro Teodoro
Conjunto de características para teste de desempenho: uma visão a partir de ferramentas / Leandro Teodoro Costa. – Porto Alegre, 2012.
113 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Avelino Francisco Zorzo.

1. Informática. 2. Engenharia de Software. 3. Software – Avaliação. I. Zorzo, Avelino Francisco. II. Título.

CDD 005.1

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada ***"Conjunto de Características para Teste de Desempenho: Uma Visão a Partir de Ferramentas"***, apresentada por Leandro Teodoro Costa como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído, aprovada em 08/03/2012 pela Comissão Examinadora:

Prof. Dr. Avelino Francisco Zorzo - PPGCC/PUCRS
Orientador

Prof. Dr. Rafael Prikladnicki - PPGCC/PUCRS

Prof. Dr. Adenilso da Silva Simão - USP

Homologada em 10/07/2012, conforme Ata No. 014/2012 pela Comissão Coordenadora.

Prof. Dr. Paulo Henrique Lemelle Fernandes
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

à Deus,
meus pais Noracy e Alvino,
minha irmã Letícia e
minha namorada Bárbara.

AGRADECIMENTOS

Ao Senhor meu Deus por sempre me apoiar em todos os momentos. Sem a ajuda Dele não teria conseguido alcançar meus objetivos. Obrigado Senhor Jesus, sei que tem sido paciente comigo. Vou recompensar todas as graças que tem derramado em minha vida.

Agradeço ao apoio financeiro provido pela DELL, referente à bolsa de estudo e bolsa taxas. Também agradeço a oportunidade de desenvolver minha pesquisa como integrante junto ao projeto CoC (Center of Competence in Performance Testing) relativo ao Programa de Desenvolvimento e Tecnologia da Informação (PDTI) no convênio DELL/PUCRS.

Aos colegas e amigos do projeto CoC Anderson, Artur, Maicon, Murillo, Ana Luiza, Suzane e Valéria pelos momentos de descontração durante o período do curso. Agradeço também ao colega e grande amigo Elder de Macedo Rodrigues pelo apoio e pelas discussões, as quais contribuíram para o desenvolvimento deste trabalho.

Ao meu orientador Avelino Francisco Zorzo, pela compreensão durante os momentos conturbados que passei. Agradeço por sua paciência, pelos conselhos, pelos ensinamentos, por acreditar no meu trabalho e principalmente pelas críticas, pois me tornaram um profissional melhor capacitado.

Agradeço ao professor Flávio Moreira de Oliveira, pois sempre manifestou claramente sua confiança no trabalho desenvolvido no projeto.

Agradeço em especial ao meu pai, minha mãe e minha irmã, por sempre estarem ao meu lado em cada momento difícil. Obrigado mãe, pela magnífica pessoa que és. Sempre acreditou em mim e me apoiou de incontáveis maneiras.

Não poderia deixar de agradecer à minha amada namorada Bárbara de Vasconcellos Pinheiro, pois sempre esteve presente, me apoiando e acreditando em mim. Agradeço principalmente por sua paciência e pelas vezes que soube entender que minha ausência tinha um motivo: garantir um futuro melhor ao seu lado.

CONJUNTO DE CARACTERÍSTICAS PARA TESTE DE DESEMPENHO: UMA VISÃO A PARTIR DE FERRAMENTAS

RESUMO

Atualmente, o desenvolvimento de sistemas por parte da indústria de *software* tem aumentado. Assim como a necessidade dos clientes em automatizar seus processos, cresce também a exigência dos mesmos na melhoria da qualidade dos sistemas informatizados. Nesse sentido, o teste de *software* desempenha um papel fundamental. Apesar dos benefícios que os testes proporcionam, a maioria deles é realizada de forma manual e sem embasamento teórico e fundamentado, tornando a atividade de teste lenta e ineficaz. Uma alternativa para a solução deste problema é a utilização de ferramentas de automatização de teste. Essas ferramentas, além de agilizar o trabalho de uma equipe de testadores, provêem maior qualidade e eficácia para o processo de teste. Entretanto, ainda que essas ferramentas possam tornar o processo de teste mais rápido, a criação de casos de teste para elas é realizada manualmente. O ideal é automatizar também o processo de criação e execução de casos de teste para essas ferramentas. Com o intuito de superar esta limitação, este trabalho propõe um conjunto de características que contempla as informações necessárias para automatizar a geração e execução de casos de teste concretizados para ferramentas de automatização de teste de desempenho. A partir das informações deste conjunto, foi realizada a implementação de *plugins* para uma linha de produto de ferramentas para teste baseado em modelos (*Model-Based Testing* - MBT) denominada PLeTs (*Product Line Testing Tools*). Estes *plugins* implementam a geração e a execução automática de *scripts* e cenários de teste utilizando duas ferramentas de automatização de teste de desempenho, HP LoadRunner e Microsoft Visual Studio. Com o objetivo de demonstrar a viabilidade da proposta deste trabalho foi definido um exemplo de uso, o qual se baseia na geração e execução automatizada de casos de teste utilizando os produtos gerados pela linha de produto PLeTs.

Palavras-chave: Teste de desempenho; Ferramentas de teste; Geração e execução de casos de teste.

SET OF FEATURES FOR PERFORMANCE TESTING: A PERSPECTIVE FROM TOOLS

ABSTRACT

Currently, the development of systems in the software industry has increased. As far as the need of customers in automate their processes grows, their demand to improve the quality of computational systems also increases. In this sense, software testing has a fundamental role. Despite of the benefits provided by testing, most of it is performed manually and without a theoretical basis, making the testing activity slow and inefficacious. An alternative for solving this problem is the use of test automation tools. In addition to accelerate the work of testers, these tools provide a higher quality and efficiency for the testing process. However, even though these tools can make the testing process faster, the test case generation for them is performed manually. The ideal is to also automate the generation and execution of test cases for these tools. In order to overcome this limitation, this work proposes a set of features which includes the information necessary for the generation and execution of real test cases for performance testing tools. Based on information from this set, we developed plugins for a product line of tools for model-based testing called PLeTs (Product Line Testing Tools). These plugins implement the automatic generation and execution of test scripts and test scenarios using two performance testing tools, HP LoadRunner e Microsoft Visual Studio. In order to show the feasibility of the proposal of this work, we performed a case study, which is based on the automated generation and execution of test cases using products generated by the product line PLETs.

Keywords: Performance testing; Testing tools; Generation and execution of test cases.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

BDL	*Benchmark Description Language*
CC	*Características Comuns*
CI	*Características Individuais*
CPC	*Características Parcialmente Comuns*
DUG	*Def-Use Graph*
FSM	*Finite State Machine*
HSI	*Harmonized State Identification*
JaBUTi	*Java Bytecode Understanding and Testing*
JVM	*Java Virtual Machine*
MBT	*Model-Based Testing*
PLeTs	*Product Line Testing Tools*
PN	*Petri Net*
QTP	*HP Quick Test Professional*
RFT	*IBM Rational Functional Tester*
RPT	*IBM Rational Performance Tester*
SAN	*Stochastic Automata Networks*
SEI	*Software Engineering Institute*
SGBD	*Sistema Gerenciador de Banco de Dados*
Skills	*Workforce Planning: Skill Management Prototype Tool*
SPE	*Software Performance Engineering*
SPL	*Software Product Line*
UML	*Unified Modeling Language*
VBScript	*Visual Basic Scripting Edition*
VUGen	*Virtual User Generator*
VV&T	*Validação, Verificação e Teste*
XMI	*XML Metadata Interchange*
XML	*Extensible Markup Language*

SUMÁRIO

1. INTRODUÇÃO

Com a presente expansão tecnológica, muitas empresas estão buscando cada vez mais agilizar seus negócios através do uso de sistemas computacionais. Tendo em vista este contexto, além do interesse em automatizar seus processos de negócio, nota-se uma grande demanda dos usuários em obter confiabilidade e disponibilidade para suas aplicações. A expectativa dos usuários na qualidade dos serviços está fazendo com que cada vez mais seja necessária a utilização de técnicas que busquem avaliar, verificar e garantir a qualidade do *software*.

Atualmente, existem diversas técnicas cujo objetivo é avaliar a qualidade do *software*. Neste contexto citam-se as técnicas de teste funcional e estrutural [12]. Entretanto, estas técnicas não podem ser aplicadas apenas ao final do processo de construção do *software*, mas sim durante toda a etapa de desenvolvimento do sistema. Além destes, existem outros conceitos para avaliação de qualidade, como por exemplo, tolerância a falhas ou prevenção de falhas [13] [14]. Esses conceitos compõem um conceito mais amplo conhecido como dependabilidade (*dependability*). Para um sistema ser considerado confiável (*dependable*) ele deve possuir um conjunto de atributos, tais como, confiabilidade, disponibilidade, segurança, integridade, manutenabilidade, entre outros [15]. Desta forma, para que uma aplicação seja classificada com determinado índice de qualidade, a mesma deve passar por constantes etapas de testes e verificação. Segundo [12], a execução de processos de "Validação, Verificação e Teste" (VV&T) nunca deve ser considerada como suficiente. Logo, o processo para a geração de produtos de alta qualidade deve estar presente durante toda a etapa de construção do *software*, desde a sua concepção até o final da entrega do produto e não algo para ser pensado posteriormente.

Contudo, a evolução e incremento da complexidade dos sistemas computacionais existentes têm tornado o processo de teste destes sistemas uma atividade tão ou mais complexa que o processo de desenvolvimento em si. Para contornar os problemas decorrentes do aumento da complexidade dos sistemas, e aumentar a eficiência no processo de geração de testes das aplicações, diversas ferramentas foram criadas para automatizar os processos de avaliação e verificação de *software*. Atualmente, existem diversas ferramentas que avaliam a funcionalidade, desempenho, disponibilidade e escalabilidade das aplicações. Dentre elas citam-se Apache JMeter [16], IBM Rational Functional Tester (RFT) [17], JUnit [18], HP Quick Test Professional (QTP) [19], HP LoadRunner [20], IBM Rational Performance Tester (RPT) [21], Selenium [22], Microsoft Visual Studio [23], e JaBUTi (*Java Bytecode Understanding and Testing*) [24].

No entanto, apesar dos benefícios advindos do uso destas ferramentas para automatização da execução dos testes, ainda é necessária a execução de algumas atividades de forma manual ou semi-automatizada como, por exemplo, a etapa de elaboração dos *scripts* e cenários de teste. Esta geração manual ou semi-automatizada torna o processo de teste ineficiente e suscetível à inserção de falhas nos *scripts* até mesmo por profissionais experientes. Uma alternativa para melhorar o uso destas ferramentas é automatizar o processo de geração destes *scripts* e cenários. Todavia,

automatizar este processo pode não ser trivial, devido à complexidade de mapear e determinar para diversas ferramentas qual o conjunto de informações que elas necessitam para tornar possível a geração automatizada de *scripts* e cenários de teste.

Visando superar esta limitação, esta dissertação apresentará um conjunto de características para teste de desempenho de aplicações *web*. Este conjunto contém uma série de informações necessárias para a geração e execução de casos de teste concretizados. Estes casos de teste são instâncias de casos de teste abstratos e são definidos como os *scripts* e cenários para as ferramentas de automatização de teste de desempenho. Em outras palavras, este conjunto dispõe de um conjunto de informações, com as quais é possível automatizar a geração e execução de *scripts* e cenários para estas ferramentas. Tais informações foram extraídas a partir da análise de trabalhos que descrevem o processo de criação de *scripts* e cenários para diversas ferramentas de teste de desempenho. Esses trabalhos apresentam detalhadamente as informações referentes às diversas configurações utilizadas pelas ferramentas durante a definição de um caso de teste, tais como: quantidade de usuários virtuais para acessar a aplicação, tempo de execução do teste, contadores utilizados para monitoração como, por exemplo, tempo de resposta, percentual de utilização de CPU e quantidade de memória disponível.

Para a efetivação do trabalho adotou-se uma classificação para as características de teste das ferramentas. Para essa classificação foi utilizada como base uma abordagem similar a utilizada pela Microsoft, que descreve em seis etapas o processo de teste de desempenho para aplicações *web* [25]. Outras abordagens foram estudadas [26], [27], [28], porém devido ao fato das informações referentes às características corresponderem e se relacionarem com as informações descritas nas seis etapas do processo de teste da Microsoft, esta abordagem foi a que melhor se adaptou ao trabalho desta dissertação.

Com base nas informações definidas no conjunto de características, é possível implementar uma ferramenta que automatiza a geração e execução de *scripts* e cenários para as ferramentas de automatização de teste de desempenho existentes, tais como HP LoadRunner e Visual Studio. O objetivo é que a configuração das informações necessárias para o teste seja realizada de forma automatizada. Desta forma, o tempo gasto na remoção de falhas em *scripts* e cenários pode ser utilizado para outros fins como maior e melhor documentação do projeto de teste do sistema, entre outras atividades.

Além de definir um conjunto de características para ferramentas de teste de desempenho, este trabalho está diretamente relacionado à outra pesquisa de dissertação de mestrado [29], cujo foco também está na descrição de um conjunto de características e informações necessárias para a geração de casos de teste de desempenho. Entretanto, as informações contidas no conjunto proposto em [29] foram geradas a partir da análise de trabalhos que descrevem as características que modelos, como os definidos na Linguagem de Modelagem Unificada (*Unified Modeling Language* - UML) [30], devem conter para a geração dos casos de teste. O conjunto destas duas pesquisas faz parte de um trabalho maior e integra também uma tese de doutorado, onde está sendo desenvolvida uma linha de produto de ferramentas de teste baseado em modelos, denominada PLeTs (*Product Line*

Testing Tools) [3]. A PLeTs utiliza as técnicas de linhas de produto de *software* para automatizar processos de teste e é capaz de gerar produtos que automatizam a geração de *scripts* e cenários de teste aplicando a técnica de teste baseado em modelos (*Model Based Testing* - MBT) [31].

Para prover esta automatização foram implementados *plugins* para a PLeTs, os quais provêem a geração automatizada de *scripts* e cenários, bem como a execução de testes utilizando as ferramentas LoadRunner e Visual Studio. Estes *plugins* foram implementados por meio da análise das informações definidas no conjunto de características proposto para esta dissertação. Além disso, esta dissertação apresenta o processo utilizado para a implementação de um *plugin* para a ferramenta PLeTs que provê a geração e execução automatizada de casos de teste estruturais, utilizando a ferramenta JaBUTi [24].

Esta dissertação possui 7 capítulos e está estruturada da seguinte forma: o Capítulo 2 apresenta os princípios básicos e os principais níveis do teste de *software*. O Capítulo 3 apresenta uma descrição das funcionalidades de algumas das principais ferramentas para automatização de teste de *software* existentes no mercado. O Capítulo 4 apresenta o conjunto de características para ferramentas de teste de desempenho, os trabalhos que foram referência para a sua construção e a efetivação do conjunto utilizando como base o processo de teste de desempenho adotado pela Microsoft. O Capítulo 5 apresenta a definição de alguns casos de teste para PLeTs utilizando a aplicação Skills [32] como exemplo de uso. O Capítulo 6 apresenta as conclusões e trabalhos futuros. O final do documento apresenta ainda um apêndice com a descrição de características e funcionalidades de algumas ferramentas para automatização de teste de *software*.

2. TESTE DE *SOFTWARE*

Neste capítulo serão descritos, inicialmente, os conceitos básicos de teste, em seguida serão apresentadas as fases de teste e as principais técnicas e métodos utilizados para a geração de casos de teste. Para este capítulo o objetivo é introduzir uma base teórica fundamentada nos principais conceitos da área de teste de *software*. Para um estudo mais detalhado sugere-se a leitura dos trabalhos [1] [12], [13], [14], [15], [33] e [34].

2.1 Principios Básicos

Antes de abordar as questões referentes aos princípios de teste de *software* é importante se familiarizar com alguns conceitos básicos da área e sua nomenclatura. Atualmente, existem algumas divergências entre os autores quando o assunto se trata da definição dos conceitos de falha, erro e defeito. Portanto, com o intuito de facilitar o entendimento desses conceitos será utilizada uma nomenclatura definida por autores com trabalhos focados na área de "Tolerância a Falhas" [33]. A razão desta escolha se baseou no fato desta nomenclatura ser utilizada como referência por grande parte da comunidade científica e por estar relacionada à área de atuação dos pesquisadores do projeto que originou esta dissertação.

Durante o desenvolvimento de um sistema computacional, o mínimo que se espera é que o mesmo esteja de acordo com o que foi definido durante sua especificação. Quando o *software* não atinge essa meta diz-se que o sistema está com defeito (*failure*), ou seja, não está em conformidade com o que foi descrito e especificado para ele nas etapas iniciais do projeto [1]. Isto pode ocorrer, por exemplo, quando o sistema não atende alguma de suas especificações funcionais.

Um erro (*error*) é um estado do sistema em que um processamento a partir desse estado resulta em um defeito. Sabe-se que a execução de um serviço possui um conjunto de estados, quando algum desses estados diverge do estado correto que se espera do serviço, diz-se que o sistema está em estado de erro. Por fim, uma falha (*fault*) é definida como sendo a causa inicial, física ou algorítmica de um erro, normalmente uma consequência da ação humana [35]. A Figura 2.1 a seguir exemplifica a definição dos conceitos de falha, erro e defeito descritos anteriormente.

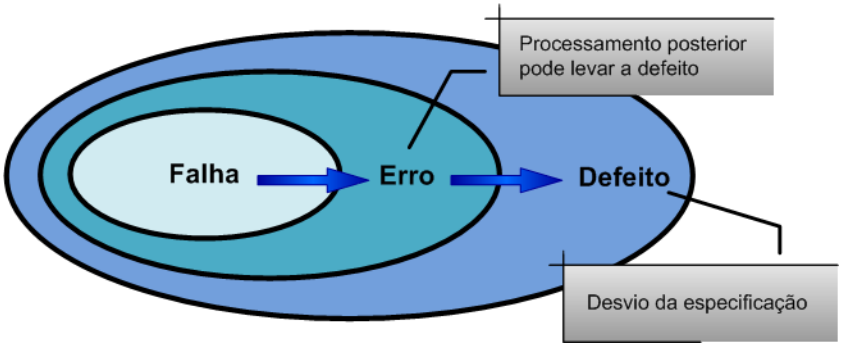

Figura 2.1: Modelo para falha, erro e defeito [1]

2.2 Princípios de Validação, Verificação e Teste

De posse do conhecimento referente aos conceitos básicos da área de teste e sua nomenclatura, as características e aplicabilidade dos princípios do teste de *software* podem ser introduzidos. Com a evolução dos sistemas computacionais existentes, o processo de desenvolvimento de sistemas computacionais torna-se uma tarefa bastante complicada. Esta situação pode agravar-se dependendo do tamanho e do grau de complexidade do *software* a ser projetado. Por este motivo, a construção de um sistema está sujeita a muitos problemas e, por consequência, pode ser gerado um produto diferente daquele especificado durante as etapas iniciais de projeto [34].

Existem inúmeros fatores que influenciam na causa desses problemas, mas a principal causa de erros ocorre devido à influência humana. Como se sabe, o desenvolvimento de sistemas computacionais depende da capacidade de interpretação dos problemas por parte de pessoas, desta forma, a existência de falhas e erros é quase que inevitável, ainda que se utilizem técnicas fundamentadas nos princípios da engenharia de *software* como auxílio na prevenção desses erros [34].

Para que esses erros não persistam e sejam identificados antes da entrega do produto final junto ao cliente, o sistema deve passar por inúmeros processos de "Validação, Verificação e Teste" (VV&T) [34]. Desta forma, é possível que o produto final esteja em conformidade com o que foi descrito durante a etapa de especificação. Processos de VV&T devem estar presentes desde a concepção do projeto e não algo para ser pensado posteriormente. Por isso, existem diversas técnicas que podem ser aplicadas ao longo do desenvolvimento do sistema, de forma a identificar falhas e com isso, garantir a consistência das funcionalidades da aplicação. Segundo [12], existem princípios que caracterizam estas técnicas, neste sentido, citam-se os princípios da sensitividade, particionamento e restrição.

Durante o desenvolvimento de um projeto de *software* a equipe de desenvolvimento inevitavelmente, em determinado momento, será responsável por alguma falha, gerando erros no sistema que podem resultar em defeitos. Um sistema com falha pode nem sempre gerar defeitos durante sua execução. Nesse sentido, é melhor que um defeito em um sistema sempre ocorra do que apenas às vezes, isso é o que afirma o princípio da sensitividade [12].

O custo na reparação de falhas de *software* pode variar de acordo com o momento em que uma falha é encontrada. O custo de uma falha encontrada durante a etapa inicial de testes através, por exemplo, da utilização de uma ferramenta que utiliza uma verificação sintática em tempo real é muito pequeno. Em contrapartida, um defeito detectado no sistema ocasionado por uma falha encontrada nas etapas finais de teste pode causar um custo enorme. Maior ainda é o prejuízo quando ocorre um defeito no sistema detectado durante a utilização do *software* por parte do usuário final. Desta forma, quanto antes se detectar um defeito, menores serão os custos envolvendo tempo para os devidos reparos.

Ainda que o ideal seja encontrar falhas nas etapas iniciais de projeto, nem sempre é trivial encontrá-las. Falhas encontradas muito tarde ou que são difíceis de encontrar têm como principal característica a não geração de defeitos. Durante a execução de testes, determinados cenários podem

ser difíceis de simular. O resultado é que nem sempre um defeito será detectado e a consequência é a persistência da falha.

Segundo [12], o ideal é fazer com que essas falhas sejam mais fáceis de serem detectadas. Desta forma, os defeitos irão aparecer mais frequentemente. O princípio da sensitividade pode ser aplicado de três maneiras, em nível de desenvolvimento, nível de teste e nível de ambiente.

- *Nível de desenvolvimento:* Verificação do tamanho e alocação de vetores em memória em tempo de execução é um exemplo concreto da aplicação da sensitividade no nível de desenvolvimento. Linguagens como Java, por exemplo, possuem suporte para aplicação do princípio neste nível;

- *Nível de teste:* Neste nível o ideal é utilização de técnicas de teste que busquem identificar falhas que geram defeitos. Em sistemas *multithreads* por exemplo, a identificação de *deadlocks* pode ou não ocorrer dependendo do tempo em que os processos são executados, ou seja, vai depender do tempo que um processo leva para utilizar determinado recurso e do momento que outro processo solicita esse mesmo recurso. Desta forma, um defeito no sistema pode ser observado em raras situações, dependendo do estado do ambiente. A execução de testes que não cobrem todas essas possibilidades pode fazer com que não sejam revelados *deadlocks*, neste caso diz-se que o teste é pouco ou insuficientemente sensitivo [12];

- *Nível de ambiente:* Fatores externos à aplicação podem afetar o resultado esperado, para contornar este problema prioriza-se a utilização de técnicas para diminuir a influência desses fatores no sistema. O ideal é que equipes e projetistas de teste busquem, de maneira eficiente, simular um ambiente mais próximo da realidade. Por exemplo, se determinada aplicação é executada em um cenário onde o acesso a rede é privado, testá-la em um ambiente onde diversas aplicações concorrem por recursos de rede pode influenciar significativamente no resultado final.

Assim como o princípio da sensitividade, o princípio do particionamento tem por objetivo auxiliar na definição de técnicas para assistir a atividade de teste. Entretanto, diferentemente do princípio da sensitividade, o particionamento consiste em dividir um problema de maior magnitude, em termos de complexidade e "tamanho", em diversos subproblemas que podem ser resolvidos de maneira independente, por este motivo, é também conhecido como "divisão e conquista". Com o intuito de facilitar a resolução de problemas, a Engenharia de *Software* utiliza muito este princípio, inicialmente nas etapas de especificação e levantamento de requisitos e durante a etapa de desenvolvimento. Como não poderia ser diferente, este princípio é também muito utilizado nas áreas de teste e análise [12].

Os níveis de processo e técnicas constituem a aplicabilidade do particionamento. Segundo [12] o particionamento pode ser aplicado em cada um dos níveis descritos a seguir:

- *Nível de processo:* Neste nível o foco é dividir um problema complexo em diversas partes independentes, tornando mais simples de executar cada uma dessas atividades. Um projeto

de teste bem elaborado é organizado em diversas fases: fase de teste de unidade, integração, validação e sistema. A ideia é focar na atividade de detectar defeitos em cada etapa e aproveitar o resultado das etapas anteriores para as etapas seguintes;

- *Nível de técnicas:* Neste nível são utilizadas técnicas de teste que tem por objetivo dividir a análise do sistema em etapas. Como exemplos dessa aplicabilidade citam-se o teste funcional, onde os processos de teste são derivados da especificação e o teste estrutural que deriva casos de teste a partir da análise da estrutura interna do programa.

Outro princípio que caracteriza abordagens e técnicas para teste e análise é definido como princípio da restrição. É utilizado, normalmente, quando o custo é muito alto ou inviável de realizar a verificação de uma propriedade. Muitas vezes, uma solução é realizar a verificação de uma propriedade diferente, mais restritiva ou limitando a verificação para uma classe de programas menor ou mais restritiva [12]. Um exemplo típico onde o problema poderia ser resolvido utilizando o conceito de restrição é a utilização de variáveis em determinado trecho do código que não foram inicializadas, ou seja, exigir a inicialização de variáveis que serão utilizadas posteriormente. Muitos compiladores como, por exemplo, o do Java faz uso do conceito de restrição, que verifica e avisa quando um problema desse gênero ocorre. Portanto, é muito importante, para o desenvolvimento de um sistema, a escolha de ferramentas e linguagens de programação que possuem esse tipo de funcionalidade.

2.3 Técnicas e Métodos de Teste

Durante o processo de teste algumas fases podem ser identificadas, tais como teste de unidade, teste de integração, teste de validação e teste de sistema [36]. Quando todas essas fases são executadas, diz-se que o objetivo de validação e verificação do *software* foi atingido. Em [36] o processo de teste é definido da seguinte forma:

- *Teste de unidade:* Também conhecido como teste unitário, ou teste de módulo, o teste de unidade tem como objetivo testar as menores unidades do *software*. Visa testar cada unidade, a fim de garantir uma correta funcionalidade dos componentes mais elementares do programa. Esses componentes podem ser funções, métodos ou classes. Normalmente o tipo de defeito que o teste de unidade busca detectar está relacionado a falhas em algoritmos, falhas de lógica e até mesmo pequenos falhas de programação. O teste de unidade é aplicado durante o processo de desenvolvimento podendo ser aplicado pelo próprio programador [34];

- *Teste de integração:* O teste de integração tem por objetivo encontrar erros derivados da integração dos módulos internos do *software*. Diferente do teste de unidade que tem por objetivo testar cada módulo do sistema individualmente, o teste de integração busca detectar defeitos quando os módulos começam a ser utilizados em conjunto, ou seja, integrados. O teste de integração também tem por objetivo verificar a compatibilidade entre os módulos,

por exemplo, cada módulo pode ter passado pelo teste de unidade e funcionar corretamente, porém, isso não garante que quando esses módulos passarem a se comunicar não haverá problemas relacionados à conectividade entre eles [34] [12];

- *Teste de validação:* O teste de validação visa verificar se o produto está de acordo com que foi descrito durante a etapa de especificação dos requisitos do *software*. Nesta fase é executado um conjunto de testes visando avaliar as funcionalidades do *software*. O objetivo é garantir que o sistema atende a todos os requisitos funcionais descritos durante a etapa de especificação;

- *Teste de sistema:* No teste de sistema um conjunto de testes com diferentes características são executados. Nesta fase o objetivo é testar o sistema do ponto de vista do usuário, onde as condições e ambiente de teste devem ser idênticos ou no mínimo estar próximo do real. Segundo [36], alguns exemplos de teste de sistema são o teste de desempenho, teste de segurança e teste de recuperação.

Também é importante esclarecer que durante cada uma das fases de projeto técnicas e métodos de teste devem ser aplicados sob o sistema com o objetivo de encontrar falhas e com isso, garantir com eficácia o cumprimento dos requisitos especificados. Entretanto, essas técnicas e métodos possuem características diferentes entre si, devendo ser aplicados em cada fase de acordo com o amadurecimento do projeto em desenvolvimento. Técnicas e métodos, tais como, teste funcional, teste estrutural, teste de regressão, teste orientado a objetos, teste baseado em defeitos e teste de desempenho servem como exemplo.

- *Teste Funcional:* O teste funcional é uma técnica que deriva casos de teste a partir da especificação do programa. Desta forma, toda a criação de casos de teste é baseada em especificações funcionais. Esta técnica busca avaliar o comportamento externo do programa e não os detalhes internos de programação, por este motivo, é chamado de teste de especificação ou teste caixa-preta (*black-box*) [12]. O teste funcional avalia o conjunto de saídas a partir das entradas, verificando se o resultado obtido corresponde ao resultado esperado [37]. O teste funcional é capaz de identificar todo e qualquer defeito em um programa ou aplicação, desde que para este se aplique todas entradas possíveis, quando isto ocorre o teste é chamado de teste exaustivo [34]. O problema é que o conjunto de entradas pode ser muito grande ou até mesmo infinito, tornando, devido ao tempo, inviável e impraticável o processo de teste. Este problema pode fazer com que não se possa afirmar ou ter certeza de que o programa esteja totalmente correto. Para contornar esse problema/limitação foram definidos critérios de teste de modo a permitir que este processo se tornasse mais sistemático. Particionamento por equivalência, análise de valor limite e Grafo Causa-Efeito são alguns dos critérios mais conhecidos para teste funcional [34];

- *Teste Estrutural:* O teste estrutural é uma técnica que tem por objetivo a geração de casos de teste a partir da análise do código fonte. Busca avaliar os detalhes internos da implementação,

tais como teste de condição, caminhos lógicos, etc. Por este motivo, é também chamado de teste orientado à lógica ou teste caixa-branca (*white-box*) [34]. Como citado anteriormente, a técnica de teste estrutural se baseia na análise da estrutura do programa para a geração dos casos de teste. Esta técnica define um conjunto de critérios estruturais, os quais se baseiam em diferentes elementos de programa para definir requisitos de teste. Estes critérios tem por objetivo a execução de componentes e partes elementares de um programa e, são classificados basicamente em [34]:

  - *Critérios baseados em fluxo de controle:* utiliza análise do fluxo de controle do programa para gerar casos de teste. Utiliza os aspectos de controle do programa, tais como, laços, desvios, condição para derivar os casos de teste;
  - *Critérios baseados em fluxo de dados:* utiliza análise do fluxo de dados do programa para gerar casos de teste. Casos de testes são derivados a partir de associações entre definições de variáveis e o uso dessas variáveis imediatamente após definidas;
  - *Critérios baseados na complexidade:* critérios baseados na complexidade fundamentam-se em informações de complexidade do programa para levantamento dos requisitos de teste. O critério de teste de caminho [38] é um dos mais conhecidos critérios de complexidade.

- *Teste de Regressão:* O teste de regressão é uma técnica de teste aplicada quando é realizada alguma alteração no sistema, por exemplo, quando se acrescenta ou retira determinada funcionalidade, quando o sistema migra para outra plataforma, ou seja, aplica-se o teste de regressão a cada nova versão do *software* [12]. Quando o *software* não mantém a corretude de suas funcionalidades em sua nova versão, diz-se que o sistema "regrediu". Neste contexto, o teste de regressão tem por objetivo contribuir para a "não regressão" das novas versões do sistema. Todo processo de teste que visa evitar problemas relacionados à regressão em sistemas é denominado teste de "não regressão", por convenção omite-se o "não" e geralmente diz-se teste de regressão [12]. A utilização desta técnica é fundamental, pois cada modificação realizada no sistema pode gerar um mau funcionamento dos módulos que em versões anteriores estavam corretos. Desta forma, os testes realizados em versões anteriores do *software* devem ser repetidos em suas novas versões, a fim de garantir que os componentes atuais funcionam corretamente e continuam válidos [34]. O ideal é que esses testes sejam automatizados, com isso, o tempo para executá-los novamente será menor. A automatização do processo de teste pode ser em muitos casos essencial, principalmente se o sistema está constantemente em processo de manutenção;

- *Teste Orientado a Objetos:* O paradigma de programação orientada a objetos surgiu trazendo um enfoque diferente do paradigma de programação procedural. Surgiu com o intuito de possibilitar uma melhor gerência e tratamento da complexidade de construção de um sistema. Possibilita uma maior abstração por focar na utilização de classes para agrupar um conjunto de objetos autônomos e de técnicas como herança e polimorfismo, que contribuem para a

reutilização de código trabalhando de forma hierárquica. Ainda que o paradigma de programação orientada a objetos tenha surgido com o intuito de facilitar a construção de *software*, as mesmas técnicas que contribuem nesse sentido podem dificultar quando o assunto diz respeito ao teste desses sistemas. Quando um defeito é detectado devido a uma falha encontrada, por exemplo, em uma classe que herda os métodos de uma classe mais abstrata (superclasse), pode ser difícil de afirmar qual das classes possui falhas. Segundo [12], as características de *softwares* com paradigma de orientação a objetos que podem impactar na testabilidade são as seguintes:

  - *Comportamento dependente do estado:* A corretude da execução de um método pode depender do estado a partir do qual ele é ativado. Projetos de teste devem considerar a possível variação do comportamento dos métodos em virtude das mudanças de estado;
  - *Encapsulamento:* A observabilidade e controlabilidade podem ser limitadas, uma vez que um programa orientado a objetos pode ter métodos e variáveis privados, dificultando o acesso a essas informações durante o teste;
  - *Herança:* Os métodos de uma classe (subclasse) podem influenciar o comportamento dos métodos da superclasse, o efeito dos métodos da subclasse sobre os métodos herdados deve ser considerado;
  - *Polimorfismo e ligação dinâmica:* Um método chamado pode ter ligações com diversos outros métodos. O projeto de teste deve ser capaz de identificar cada uma dessas ligações e exercitar todas as possibilidades de ligações possíveis;
  - *Classe abstrata:* Uma classe abstrata é uma superclasse que não possui instâncias e devem ser testadas sem considerar o modo como podem ser instanciadas;
  - *Tratamento de exceções:* Para o tratamento de exceções é importante testar o intervalo a partir do ponto de lançamento de uma exceção até o ponto onde de fato ela é tratada.

- *Teste Baseado em Defeitos:* O teste baseado em defeitos tem por objetivo extrair a maior quantidade de informação possível dos defeitos de *software* detectados mais frequentemente. A partir dessas informações é possível gerar casos de testes com o intuito de detectar e corrigir mais rapidamente esses defeitos. Quanto maior o conhecimento sobre os defeitos de *software*, maiores são as garantias de se obter qualidade na implementação de um sistema. O teste baseado em defeitos utiliza um modelo, baseado nas informações dos defeitos que ocorrem mais frequentemente em um programa, com o intuito de determinar um conjunto de possíveis defeitos que podem ocorrer no sistema testado. Em posse das informações do modelo pode-se introduzir propositalmente no *software* um conjunto de falhas e executar o teste, a fim de verificar se quantidade de defeitos provenientes da inserção dessas falhas está coerente com a quantidade de defeitos que o teste deveria detectar [12]. Em outras palavras, o objetivo do teste baseado em defeitos é testar a qualidade do teste. A eficácia do teste baseado em

defeitos vai depender da qualidade do modelo de informações sobre os possíveis defeitos do *software*. É bem possível que os métodos de teste disponíveis para verificar a qualidade do sistema não revelem os defeitos provenientes de uma injeção de falhas nesse sistema. Para contornar este problema são utilizados alguns critérios de teste baseado em defeitos, o mais comum e conhecido desses critérios é denominado análise de mutantes [12]. Um mutante é um programa variante em relação ao original, onde essa variação é proveniente de uma pequena mudança no código (inserção de falha), como por exemplo, a substituição de um operador "$<$" por "$<=$". Diz-se que um programa mutante é válido se o resultado de sua execução é diferente do resultado do programa original;

2.4 Teste de Desempenho

Com o crescente crescimento tecnológico, os clientes não estão apenas exigindo que suas aplicações atendam os requisitos funcionais previamente acordados, mas que também atendam suficientemente os requisitos relacionados ao desempenho. Um *survey* [39] relacionado ao desempenho constatou que metade das empresas de *software* encontrou problemas relacionados ao desempenho em pelo menos 20% das aplicações por elas implantadas. Estes problemas foram evidenciados principalmente em aplicações *web*, as quais são responsáveis por um alto processamento provenientes de requisições de usuários.

Neste contexto, a modelagem de desempenho possui grande importância, pois visa identificar os problemas de desempenho existentes e soluções para estes problemas. A modelagem de desempenho, a qual é uma parte da engenharia de desempenho de *software* (*Software Performance Engineering* - SPE) [40], tem por objetivo compreender as características de desempenho de aplicações sob diferentes cargas e diferentes configurações de *hardware* e *software*. Técnicas de modelagem de desempenho são classificadas em medição, análise e simulação. As técnicas baseadas em medição tem por objetivo a realização de inúmeros testes de desempenho sob aplicação a ser testada, entretanto, estes testes somente são feitos após a aplicação estar totalmente pronta e disponível. Para contornar este problema, existem técnicas analíticas e de simulação que se baseiam na construção de modelos para estudar e prever as características de desempenho das aplicações. As técnicas analíticas utilizam modelos teóricos, enquanto técnicas de simulação emulam as funcionalidades de uma aplicação por meio de simulações computacionais, onde o desempenho pode ser observado. O problema é que técnicas analíticas e de simulação exigem um conhecimento aprofundado da aplicação, onde uma documentação detalhada das funcionalidades do sistema se faz necessária. Entretanto, em muitos casos, a maior parte das informações do sistema está presente somente no próprio código da aplicação e, por este motivo, as técnicas baseadas em medição acabam sendo mais utilizadas.

O teste de desempenho baseado em medições tem por objetivo determinar o limite do sistema submetido a uma determinada carga em um ambiente de teste específico. Ele fornece um indicador utilizado para avaliar o quanto um sistema ou o componente de um sistema é capaz de cumprir os requisitos de desempenho, tais como, tempo de resposta ou *throughput*. Além disso, o teste de

desempenho busca identificar os possíveis gargalos causadores de uma degradação de desempenho no sistema [41].

Diversos autores definem conjuntos distintos para representar os diferentes tipos de teste de desempenho existentes. Por exemplo, [42] define cinco subconjuntos de teste de desempenho, *Baseline test*, *Load test*, *Stress test*, *Soak or stability test* e *Smoke test*. Entretanto, [27] define apenas três subconjuntos, *Load test*, *Stress test*, *Endurance or Durability test*. Em [43] é definido um subconjunto para teste de desempenho muito semelhante ao utilizado em [27], o qual é formado por três tipos, *Load test*, *Stress test* e *Strength test*. Outra abordagem utilizada para representar os diferentes tipos de teste de desempenho pode ser encontrada em [44], o qual define três tipos, sendo eles *Load test*, *Stress test* e *Capacity test*. Apesar da divergência entre autores quanto à definição dos tipos de teste de desempenho existentes, os mais comuns e conhecidos são:

- *Teste de carga (Load testing)*: tem por objetivo determinar ou validar o comportamento de um sistema sob condições normais de carga. Busca verificar se o sistema cumpre os requisitos de desempenho especificados;
- *Teste de estresse (Stress testing)*: tem por objetivo determinar o comportamento de um sistema quando ele é submetido além das condições normais de carga. Busca revelar *bugs* e determinar os pontos de defeito do sistema.

Devido à importância do teste de desempenho, existem diversas ferramentas cujo objetivo é automatizar a execução de testes. Dentre estas ferramentas citam-se Apache JMeter [16], HP LoadRunner [20], IBM Rational Performance Tester [21], SilkPerformer [45] e Visual Studio [23], as quais visam mitigar os problemas de desempenho das aplicações. Para maiores detalhes acerca das funcionalidades destas ferramentas recomenda-se a leitura do apêndice A.

2.5 Considerações

Os conceitos, métodos e técnicas de teste são fundamentais para a elaboração de um projeto de teste bem estruturado e confiável. Entretanto, devido à complexidade dos sistemas computacionais, a atividade de teste está tão complexa quanto o processo de desenvolvimento. Para mitigar estes problemas e otimizar o processo de teste, diversas ferramentas de teste de *software* foram desenvolvidas.

Estas ferramentas se baseiam nos conceitos, métodos e técnicas de teste encontradas na literatura e, são utilizadas para a criação e execução de casos de teste de forma mais eficiente. No capítulo seguinte serão descritas as características e funcionalidades de algumas ferramentas de teste, desenvolvidas com o intuito de avaliar a qualidade de sistemas.

3. FERRAMENTAS DE TESTE DE *SOFTWARE*

O mercado atual tem apresentado empresas cada vez mais procurando automatizar seus processos de negócio através da utilização de sistemas computacionais. Essa necessidade tem contribuído para um aumento da produção desses sistemas por parte da indústria de *software*. Assim como a necessidade dos clientes em automatizar seus processos, cresce também a exigência dos mesmos na busca pela qualidade dos sistemas informatizados. Nesse sentido, e conforme mencionado anteriormente, o teste de *software* torna-se essencial para se atingir com êxito a qualidade de sistemas.

Apesar dos benefícios que o teste traz, a maioria deles é realizada de forma manual e sem um embasamento teórico e fundamentado, tornando a atividade de teste lenta e ineficaz. Uma alternativa que contribui na solução deste problema é a utilização de ferramentas de automatização de teste para auxiliar nesses processos. Essas ferramentas, além de agilizar o trabalho de uma equipe de testadores, contribuem no sentido de obter com eficácia a qualidade dos testes e programas, por serem criadas com base em técnicas e métodos de teste bem conhecidos, com um fundamento teórico e científico.

Neste capítulo será apresentada, inicialmente, uma análise das diferenças entre algumas ferramentas de teste[1]. A análise contempla ferramentas para diversos tipos de teste, tais como: ferramentas para automatização de teste funcional, estrutural, desempenho, unitário e regressão. Além disso, entre as diversas ferramentas de automatização de teste de *software*, optou-se por focar em ferramentas e *frameworks*, cuja principal característica é prover recursos para automatização de teste de desempenho. Desta forma, as ferramentas selecionadas para o foco do estudo foram JMeter [16], HP LoadRunner [20], IBM Rational Performance Tester [21], SilkPerformer [45] e Visual Studio [23].

3.1 Características e Funcionalidades das Ferramentas

Tendo apresentado no Capítulo 2 um estudo relacionado aos conceitos de diferentes tipos de teste de *software*, o objetivo para esta seção é apresentar uma análise de algumas ferramentas que visam automatizar os diferentes processos de teste mencionados no Capítulo 2. Portanto, pretende-se apresentar uma visão geral dos conceitos e funcionalidades destas ferramentas e os diversos tipos de teste que elas executam.

Entre as ferramentas estudadas, o JMeter [16], uma ferramenta *open source* escrita em Java [46], possui como foco a execução de testes de desempenho. Outras ferramentas que possuem a característica de automatizar e executar teste de desempenho, porém comerciais, são o LoadRunner da HP [20] e o Rational Performance Tester da IBM [21]. Ambas possuem algumas características em comum, por exemplo, as duas ferramentas utilizam a técnica *Record&Playback* para a geração de *scripts* de teste. Esta técnica consiste na gravação de todas as interações realizadas pelo usuário

[1] Uma descrição com maiores detalhes encontra-se no apêndice desta dissertação.

com determinada aplicação.

Essas duas empresas (HP e IBM) também possuem ferramentas para automatização de teste funcional e regressão: HP Quick Test Professional [19] e IBM Rational Functional Tester [17]. Além disso, as duas ferramentas têm como foco a execução de testes para aplicações *web* e possuem etapas de criação e execução de testes bem semelhantes, por exemplo, ambas geram relatórios para análise de resultados ao final de cada teste executado. Outra ferramenta que provê automatização de testes funcionais é o Selenium [22]. Esta ferramenta possui suporte para criação de *scripts* de teste em diversas linguagens, tais como, Java [46], C# [47], Perl [48], PHP [49], Python [50] e Ruby [51].

Além de analisar ferramentas que automatizam testes de desempenho e testes funcionais, foi realizado um estudo sobre uma ferramenta utilizada para auxiliar a geração de testes estruturais. Denominada JaBUTi [24], esta ferramenta *open source*, foi desenvolvida pelo grupo de engenharia de *software* da USP-São Carlos. Sua principal característica, e que a faz diferenciar-se das demais ferramentas para teste estrutural, diz respeito à possibilidade de executar testes sem que seja necessária uma análise do código fonte. Toda execução de teste é realizada a partir da análise do Java Bytecode.

Para complementar as informações referentes às diferenças entre as ferramentas descritas anteriormente é apresentado na Tabela 3.1 as principais características de cada uma delas.

Tabela 3.1: Características e funcionalidades das ferramentas de teste

Características	Ferramentas							
	JMeter	RFT	JUnit	QPT	LoadRunner	RPT	Selenium	JaBUTi
Open Source	X		X				X	X
Teste Funcional		X		X			X	
Teste Estrutural								X
Teste Unitário			X					
Teste de Desempenho	X				X	X		
Teste de Regressão		X		X				

3.2 Ferramentas para Teste de Desempenho

Tendo apresentado na seção 3.1 uma visão geral das características e funcionalidades de algumas ferramentas que automatizam diferentes tipos de teste, o objetivo desta seção é focar a análise em ferramentas que automatizam um tipo de teste específico. Neste contexto, optou-se pela seleção de ferramentas para o teste de desempenho de aplicações *web*. Desta forma, foi realizado um estudo mais aprofundado das características e funcionalidades destas ferramentas.

Para este estudo foram analisadas cinco ferramentas para teste de desempenho *web*: JMeter [16], LoadRunner [20], Rational Performance Tester [21], SilkPerformer [45] e Visual Studio [23]. O objetivo desta análise foi identificar as características de cada uma e apresentar como cada ferramenta necessita ser configurada durante o processo de criação e execução de testes. Além disso, serão

apresentadas as diferenças e semelhanças entre elas e como cada uma necessita ser configurada durante as duas principais etapas que constituem a execução automática de testes de desempenho: geração dos *scripts* e configuração do cenário.

Para a etapa de geração dos *scripts* de teste, as cinco ferramentas analisadas utilizam uma técnica denominada *Record&Playback*. Esta técnica consiste na gravação de todas as interações realizadas pelo usuário com determinada aplicação. Entretanto, antes da gravação ser iniciada, é necessária a configuração do parâmetro referente ao endereço IP da aplicação que se deseja testar, o qual ocorre por meio da interface que cada ferramenta em particular possui.

Para o JMeter, essa configuração é feita por meio da especificação do parâmetro *HTTP Request*. Uma desvantagem é que para cada *web link* da aplicação é necessário configurar um *HTTP Request*. Desta forma, quando se deseja configurar um cenário de teste com usuários requisitando 10 *web links*, por exemplo, será necessário configurar 10 *HTTP Request*. Não é o caso das demais ferramentas, para o LoadRunner, RPT, SilkPerformer e Visual Studio apenas um parâmetro necessita ser configurado. Com isso, basta editar este parâmetro somente com o endereço inicial da aplicação. Os demais *web links* são gerados automaticamente por meio da técnica *Record&Playback*, onde um navegador como o Internet Explorer, por exemplo, é carregado automaticamente com a página da aplicação definida. Neste momento toda a interação feita pelo usuário com a aplicação é gravada para gerar o *script* de teste.

Como citado anteriormente, as cinco ferramentas dispõem desta funcionalidade e o processo de gravação é muito semelhante para cada uma. Entretanto, o JMeter é a única ferramenta dentre as demais que necessita de configurações adicionais. Para habilitar esta funcionalidade o JMeter necessita adicionar um elemento denominado *HTTP Proxy Server* e alterar a configuração de *proxy* do navegador. O IP da máquina local também deve ser definido e a porta 8080 (padrão do apache) deve ser utilizada. Existem outras diferenças entre as ferramentas para esta etapa de geração de *scripts*. A principal delas diz respeito ao número de protocolos que cada uma suporta. Neste contexto, destaca-se o LoadRunner por ter suporte a uma gama muito maior de protocolos (*HTTP/HTML*, *Oracle NCA*, *SAPGUI*, etc.) se comparado com as demais ferramentas. O destaque negativo é o JMeter por suportar apenas os protocolos HTTP e HTTPS para a gravação dos *scripts* de teste.

Outra diferença entre as ferramentas está relacionada às linguagens de *scripts* que cada uma possui. O LoadRunner, por exemplo, permite a gravação de *scripts* nas linguagens C [52], Visual Basic [53] ou Java [46]. A linguagem Java também é utilizada para a gravação dos *scripts* de teste com o RPT. Por outro lado, o JMeter grava todas as informações relativas às interações dos usuários com a aplicação em um arquivo no formato *XMI* (*XML Metadata Interchange*) [54]. O SilkPerformer implementa uma linguagem específica para a gravação de seus *scripts*, os quais são gerados na linguagem denominada *Benchmark Description Language (BDL)*. Outra ferramenta que, assim como o LoadRunner, possui suporte à gravação de *scripts* em três linguagens distintas é o Visual Studio, suportando a gravação de *scripts* em Visual Basic, C# [47] e C++ [55] (ver Tabela 3.2).

A segunda etapa que constitui a geração e execução automática de casos de teste de desem-

penho refere-se à configuração do cenário de teste. Para esta etapa são configurados basicamente quatro parâmetros: quantidade de usuários, duração do teste, perfil da carga de trabalho e contadores para avaliar o desempenho do ambiente. Após a definição dos *scripts* para um cenário de teste específico as cinco ferramentas analisadas necessitam especificar cada um destes quatro parâmetros. A quantidade de usuários refere-se ao número de usuários virtuais que farão requisições ao(s) endereço(s) HTTP configurado(s). Para cada uma das ferramentas analisadas basta editar o parâmetro referente a essa informação. No JMeter essa configuração é feita por meio da edição do parâmetro *Number of Threads*, para o LoadRunner essa configuração é feita através da edição do parâmetro *VUsers*, para o RPT é feita por meio da edição do parâmetro *Quantidade de Usuários*, no SilkPerformer é feita através da edição do parâmetro *Vusers* e no Visual Studio é realizada por meio da edição do parâmetro *User Count*. Quanto à configuração do parâmetro relativo ao tempo de duração do teste, não há diferenças significativas entre as ferramentas. A distinção na verdade fica por conta dos nomes ou identificadores que cada uma possui para a configuração deste parâmetro. No JMeter, LoadRunner, RPT, SilkPerformer e Visual Studio esta informação é representada respectivamente por *Duration (seconds), Duration (Loops), Stop running the schedule after an elapsed time, Simulation time, Run Duration*.

Outro parâmetro necessário configurar diz respeito ao perfil da carga de trabalho. Entretanto, somente o LoadRunner, o SilkPerformer e o Visual Studio possuem essa funcionalidade. Refere-se ao perfil de teste que será executado. No LoadRunner essa informação é configurada em *Start VUsers*, com ela pode-se definir um perfil onde todos os *VUsers* irão iniciar o teste ao mesmo tempo ou de forma gradativa. Para o SilkPerformer essa informação é configurada em *Type of workload to run*, onde existem cinco tipos de perfis, são eles: *Increasing, Steady State, Dynamic, All Day, Queuing*. Os principais perfis são *Increasing* e *Steady State*, no perfil *Increasing* os usuários iniciam o teste de forma gradativa e no perfil *Steady State* todos os usuários iniciam o teste ao mesmo tempo. Para o Visual Studio essa informação é definida pelo parâmetro *Load Pattern*, assim como para o LoadRunner existem dois tipos de perfis de configuração para o Visual Studio. O primeiro se trata do perfil *Constant Load*, onde os usuários iniciam o teste ao mesmo tempo. O segundo é denominado *Step Load*, onde os usuários iniciam o teste de forma gradativa.

As demais ferramentas (JMeter e RPT) mesmo não permitindo definir um perfil da carga de trabalho possibilitam especificar informações de rampa de subida (*rump-up*). Este parâmetro determina a quantidade de usuários virtuais que iniciarão o teste em um intervalo de tempo pré-determinado. Inicialmente, o teste inicia com um determinado número de usuários virtuais e a cada intervalo de tempo previamente configurado a mesma quantidade de usuários definida anteriormente inicia suas interações sob a aplicação a ser testada. Este processo ocorre até que a quantidade total de usuários virtuais definidas para o teste esteja requisitando a aplicação. Para o JMeter a configuração é realizada por meio da especificação do parâmetro *Ramp-Up period* e através da especificação do parâmetro *Delay between starting each user* para o RPT. Para finalizar a configuração do cenário de teste é necessária a definição de alguns contadores, os quais são utilizados para medir informações de desempenho do ambiente de teste. Para cada uma das cinco ferramentas é possível especificar

e definir diversos tipos de contadores, tais como: percentual de utilização de CPU, quantidade de memória disponível, percentual de utilização do disco para leitura e escrita, bem como contadores referentes ao tráfego de dados na rede como, por exemplo, *throughput*.

Esta análise se baseou nas principais funcionalidades das ferramentas, as quais possibilitam a geração dos *scripts* e configuração do cenário de teste. Neste contexto, se observou que principalmente a etapa de geração dos *scripts* de teste é muito semelhante para cada uma das cinco ferramentas analisadas. Todas possuem métodos para a captura das interações do usuário com a aplicação. Entretanto, elas suportam diferentes linguagens para gravação dos *scripts* de teste e algumas permitem utilizar mais linguagens que outras, o que pode ser um diferencial no momento que se deseja optar pela aquisição desta ou daquela ferramenta de teste. Também é importante, a fim de estabelecer um processo de teste eficiente e eficaz dentro da empresa, que a equipe de teste tenha domínio das linguagens que a ferramenta de teste definida suporta. Outro fator a ser levado em consideração diz respeito à quantidade de protocolos de geração de *scripts* que cada uma possui, pois é no mínimo desejável que a ferramenta de teste tenha suporte à tecnologia utilizada pela aplicação a ser testada.

Também existem algumas diferenças entre as ferramentas do ponto de vista da configuração dos cenários de teste. Entretanto, essas diferenças são evidenciadas, por exemplo, nos identificadores ou nos nomes dos parâmetros utilizados para a configuração das informações do cenário. Visto que as cinco ferramentas analisadas possibilitam a configuração de parâmetros como quantidade de usuários, tempo de duração do teste, perfil da carga de trabalho e contadores para avaliar o desempenho do ambiente. Neste contexto, as maiores distinções são verificadas na configuração dos contadores, pois algumas ferramentas permitem mensurar uma quantidade maior de informações que outras. Também verificou-se diferenças relacionadas ao perfil da carga de trabalho, pois somente LoadRunner, SilkPerformer e Visual Studio possuem tal funcionalidade. As demais ferramentas (JMeter e RPT) ainda possibilitam a configuração de informações de rampa de subida, porém possuem características mais simples que as outras ferramentas do ponto de vista da configuração do perfil da carga de trabalho.

Um resumo das diferenças descritas entre as ferramentas é apresentado na Tabela 3.2.

Tabela 3.2: Características e funcionalidades das ferramentas de teste de desempenho

Características	Ferramentas				
	JMeter	LoadRunner	RPT	SilkPerformer	Visual Studio
Open Source	X				
Linguagem de *Script*	*XMI*	C Java Visual Basic	Java	BDL	C C++ Visual Basic
Plataforma Windows	X	X	X	X	X
Plataforma Linux	X	X			
Record & Playback	X	X	X	X	X
Contadores	X	X	X	X	X

3.3 Considerações

Este capítulo apresentou, inicialmente, uma análise referente às características e funcionalidades de algumas ferramentas que automatizam diferentes tipos de teste. Tendo realizada esta análise, a próxima etapa focou no estudo de um conjunto de ferramentas que automatizam a execução de casos de teste de desempenho. Este estudo apresentou uma análise referente às diferenças e semelhanças entre as ferramentas no que diz respeito às etapas de geração de *scripts* e configuração do cenário de teste.

Por meio desta análise pôde-se verificar que algumas informações são fundamentais para a criação e execução de casos de teste e, muitas dessas características estão presentes e são fundamentais para geração e configuração de *scripts* e cenários de teste em diversas ferramentas. No próximo capítulo será apresentado um conjunto de características para ferramentas de teste de desempenho, o qual possui informações para gerar automaticamente *scripts* e cenários de teste.

4. CONJUNTO DE CARACTERÍSTICAS

Este capítulo apresenta, detalhadamente, a descrição referente à implementação de um conjunto, o qual define, com base em diversos trabalhos, as características e informações necessárias para a geração e execução de casos de teste concretizados para aplicações *web*. Para a efetivação do trabalho adotou-se uma classificação para as informações contidas no conjunto de características implementado. Para essa classificação foi utilizada como base uma abordagem similar à utilizada pela Microsoft, que descreve em seis etapas o processo de teste de desempenho para aplicações *web* [25]. Este capítulo também apresenta alguns conceitos referentes à Linha de Produto de *Software* (*Software Product Line* - SPL) [56], bem como as etapas para implementação de *plugins* para uma ferramenta que se baseia em conceitos de SPL. Esta ferramenta é denominada PLeTs (*Product Line Testing Tools*) [3]. Ao final, este capítulo ainda apresenta as etapas para a implementação de um *plugin* para a PLeTs exclusivamente para a geração e execução de testes utilizando a ferramenta de teste estrutural denominada JaBUTi [24].

4.1 Levantamento de Informações

Inicialmente, foi realizado um estudo e uma análise de diversos trabalhos que descrevem o processo de criação de *scripts* e cenários de teste utilizando ferramentas para automatização de teste de desempenho. Esta análise se baseou, principalmente, em trabalhos que fazem referência a cinco ferramentas, sendo elas: Apache JMeter [16], HP LoadRunner [20], IBM Rational Performance Tester [21], Borland SilkPerformer [45] e Microsoft Visual Studio [23]. A seleção dessas ferramentas como base para esta análise foi motivada principalmente por uma pesquisa de mercado realizada por [11]. Como pode-se observar na Tabela 4.1, as cinco ferramentas citadas neste trabalho foram relacionadas, com destaque para LoadRunner, Rational Performance Tester e SilkPerformer que segundo a pesquisa ocupam o maior percentual da fatia de mercado.

Para esta pesquisa, verificou-se que as ferramentas comerciais são utilizadas por grande parte das empresas de desenvolvimento e teste de *software*. Por outro lado, ferramentas *open source* eram pouco utilizadas e com isso, foram fracamente evidenciadas na pesquisa. Mesmo assim, juntamente com outras ferramentas elas chegam a ocupar 3% do mercado. Tendo definido o conjunto de ferramentas, o próximo passo foi realizar a instalação de cada ferramenta e, posteriormente, executar um conjunto de testes com cada uma delas. O objetivo foi analisar o que necessita ser configurado em cada ferramenta para gerar e executar *scripts* e cenários de teste. Desta forma, após esta etapa de instalação e análise foi possível obter um conhecimento referente às características de desempenho necessárias para automatizar a geração de casos de teste para as ferramentas. De posse desse conhecimento, a próxima etapa foi realizar uma análise e uma pesquisa por trabalhos científicos que descrevessem as mesmas características de desempenho evidenciadas após a etapa de instalação e configuração das ferramentas. O objetivo foi buscar na literatura trabalhos que

comprovassem/justificassem as características de desempenho previamente definidas. A seguir, são descritos alguns dos trabalhos utilizados para a identificação das características necessárias para geração de casos de teste de desempenho concretizados.

Tabela 4.1: Pesquisa de mercado de ferramentas de teste de desempenho [11]

Principais Fornecedores		
Fornecedor	Ferramenta	Fatia de Mercado (%)
Mercury Interactive	LoadRunner	63
IBM Rational	Performance Tester	5
Compuware	QALoad	6
Segue	SilkPerformer	9
Empirix	e-TEST Suite	9
RadView	WebLoad	4
Cyrano	OpenSTA	1
Outros	Outras[1]	3

[1]Correspondem às ferramentas *Open Source* e outras ferramentas comerciais.

O trabalho realizado por [57] tem por objetivo determinar os principais fatores responsáveis pela degradação de desempenho em sistemas computacionais. Para isto, foram utilizados métodos de simulação para construir um ambiente de sistemas com diminuição de recursos, a fim de determinar os fatores referentes à estrutura do ambiente que impactam na queda de desempenho. Neste trabalho, o JMeter foi utilizado para a criação deste ambiente computacional de simulação e com isso, reproduzir este processo de diminuição de recursos. Apesar da pesquisa deste artigo focar na definição de fatores responsáveis pela degradação de desempenho em determinados sistemas computacionais, a maior contribuição deste trabalho para a pesquisa desta dissertação foi apresentar um conjunto de informações utilizadas para a execução de testes de desempenho utilizando o JMeter. Esse trabalho, o qual é apresentado na Tabela 4.2 (Jing [57]), descreve um conjunto de informações como: *HTTP request*, *Number of threads* e *Ramp-up period*. Estas informações representam, respectivamente, as informações URLs da Aplicação, Quantidade de Usuários e Tempo de Rampa de Subida, descritas na Tabela 4.3.

Em [21] são descritas as informações básicas e necessárias para a criação e execução de *scripts* e cenários de teste para o RPT. As características e informações descritas nesse documento servem como base para comprovar que é possível a geração de casos de teste utilizando um conjunto determinado de informações. O estudo é válido, uma vez que o relatório descreve e ilustra o que é necessário configurar na ferramenta para que um simples caso de teste possa ser criado e executado. Esse trabalho, ainda que apresente diversas informações relacionadas a diversas funcionalidades do RPT, sua contribuição para esta dissertação foi relevante por apresentar informações e características relativos à geração e a execução de *scripts* e cenários para o RPT. Este trabalho, o qual é apresentado na Tabela 4.2 (Chadwick [21]), descreve o seguinte conjunto de informações: *Host/URL*, *Number of users* e *Think Time*. Estas informações representam, respectivamente, as informações URLs da Aplicação, Quantidade de Usuários e Tempo de Pensamento, descritas na Tabela 4.3.

Esses e outros trabalhos, os quais podem ser observados na Tabela 4.2, apresentam detalhadamente as informações referentes às configurações de *scripts* e cenários, ou seja, descrevem um conjunto de características necessárias para a criação e execução de casos de teste concretizados. Na Tabela 4.2 são descritos os nomes dos principais autores dos trabalhos analisados referente a cada ferramenta. Como pode-se observar, um conjunto de cinco trabalhos por ferramenta foi analisado. Na última linha desta tabela são encontrados trabalhos que descrevem as características e informações de outras ferramentas, completando então um total de 30 trabalhos utilizados como referência para a implementação do conjunto de características. A seleção deste número de trabalhos, sendo a maioria 5 por ferramenta, foi motivada pelo fato de que após a pesquisa de 30 trabalhos se verificou que muitas informações referentes às características das ferramentas se repetiam. E, mesmo ao final da análise destes 30 trabalhos já não haviam características diferentes das que já tinham sido mapeadas.

Tabela 4.2: Trabalhos analisados para as ferramentas de teste de desempenho

Ferramentas	**Trabalhos Analisados**				
JMeter	Jing [57]	Keshk [58]	Wu [59]	Xiao-yun [60]	Miao [61]
LoadRunner	Jiang [62]	Zhou [63]	Chen [64]	Gaisbauer [65]	Pu [66]
Rational Performance Tester	Chadwick [21]	Schultz [67]	Ben-Yehuda [68]	Meyers [69]	Ebbers [70]
SilkPerformer	Apte [71]	Luo [72]	Ling [73]	Kim [74]	Borland [75]
Visual Studio	Arnold [23]	Levinson [76]	Subashni [77]	Bai [78]	Liu [79]
*Outras	Thakkar [80]	Krizanic [81]	Chen [82]	Hamed [83]	Romano [84]

*Outras: Correspondem às ferramentas WebLoad, The Grinder, Spirent Avalanche, Parasoft WebKing.

Com base nas informações adquiridas dos trabalhos estudados, pôde-se fazer um levantamento das características necessárias para a criação de casos de teste concretizados. E, por meio desta análise foi definido um conjunto de características para cada ferramenta. Analisando este conjunto de características verificou-se que muitas delas possuem o mesmo tipo de informação, a diferença está relacionada ao nome correspondente a cada característica. Por esse motivo, as características comuns entre as ferramentas podem ser representadas por um único nome ou identificador. Entretanto, existe um conjunto de características que não está presente em todas as ferramentas, mas sim em algumas e um outro conjunto que é particular de uma ferramenta específica. Desta forma, o conjunto de características pode ser classificado em três formas distintas:

- *Características Comuns (CC)*: conjunto das características comum a todas as ferramentas;
- *Características Parcialmente Comuns (CPC)*: conjunto das características comum a algumas ferramentas;
- *Características Individuais (CI)*: conjunto das características pertencentes a uma única ferramenta.

Tendo definido as características para cada ferramenta, o que se fez foi construir o conjunto a partir das informações das características pertencentes a CC e CPC, sendo que para o segundo

conjunto apenas as características comuns a três ferramentas optou-se por incluir no conjunto. Esta decisão foi tomada tendo como premissa o fato de que um dos objetivos para o conjunto de características é torná-lo independente de tecnologia, permitindo que o mesmo disponibilize principalmente as informações essenciais para o teste de desempenho e não informações para teste utilizando alguma ferramenta em questão. Portanto, pretende-se implementar um conjunto de características mais "limpo" e robusto. Uma alternativa para aproveitar as informações que não foram incluídas no conjunto é utilizá-las em um "template". Desta forma, sempre que alguém optar por implementar um *software* para automatizar a geração e execução de casos de teste utilizando alguma ferramenta que possui características que não foram inseridas no conjunto, poderá fazê-lo a partir da inserção dessas informações em um arquivo "template", onde o projetista poderá configurá-las da forma que achar mais conveniente.

Com base nas informações adquiridas dos trabalhos analisados e na estratégia utilizada, o conjunto de características pôde começar a ser construído. As características escolhidas são descritas a seguir e sua estrutura pode ser melhor visualizada na Tabela 4.3.

Tabela 4.3: Características das ferramentas para teste de desempenho

Características	Conjunto de Características	Ferramentas				
		JMeter	LoadRunner	RPT	SilkPerformer	Visual Studio
URLs da Aplicação	CC	X	X	X	X	X
Quantidade de Usuários	CC	X	X	X	X	X
Tempo de Duração do Teste	CC	X	X	X	X	X
Tempo de Pensamento	CC	X	X	X	X	X
Contadores	CC	X	X	X	X	X
Record & Playback	CC	X	X	X	X	X
Tempo de Rampa de Subida	CC	X	X	X	X	X
Quantidade de Usuários da Rampa de Subida	CPC		X	X	X	X
Tempo de Rampa de Descida	CPC		X	X	X	
Quantidade de Usuários da Rampa de Descida	CPC		X	X	X	
Tempo de Aquecimento	CPC			X	X	X
Perfil da Carga de Trabalho	CPC		X		X	X

- *URLs da Aplicação*: refere-se às informações de configuração dos endereços IP da aplicação para onde será gerada a carga;

- *Quantidade de Usuários*: refere-se ao número total de usuários que farão requisições ao(s) endereço(s) configurado(s);

- *Tempo de Duração do Teste*: refere-se ao tempo total de execução do teste;

- *Tempo de Pensamento*: refere-se ao tempo que cada usuário leva para realizar determinada atividade entre duas requisições consecutivas. Por exemplo, preenchendo algum formulário ou lendo algum tipo de informação referente à página *web* da aplicação;

- *Contadores*: refere-se ao conjunto de contadores que serão utilizados para medir o desempenho do ambiente. Cada uma das ferramentas analisadas possui um conjunto de contadores que pode ser configurado para o teste. Algumas ferramentas possuem ou podem utilizar um número maior de contadores em relação às outras, mas todas são capazes de mensurar informações básicas, como: utilização de CPU/Memória e *throughput*. Durante a execução do teste para cada ferramenta as informações referentes aos contadores são mostradas em tabelas e gráficos;

- *Record&Playback*: é uma técnica que consiste na gravação de todas as interações realizadas pelo usuário com uma aplicação. Para cada uma das ferramentas analisadas essa técnica é utilizada automaticamente para a geração dos *scripts* de teste. A utilização dessa técnica ocorre logo após a definição/configuração do parâmetro referente ao endereço IP da aplicação que se deseja testar. Em seguida, um navegador como o Internet Explorer, por exemplo, é carregado automaticamente com a página da aplicação definida e toda a interação feita pelo usuário com a aplicação é gravada para gerar o *script* de teste;

- *Tempo de Rampa de Subida*: define o tempo que levará para um determinado conjunto de usuários (definido em *Quantidade de Usuários da Rampa de Subida*) iniciar o teste, ou seja, realizando requisições ao endereço previamente configurado (ocorre durante o início do teste);

- *Quantidade de Usuários da Rampa de Subida*: define a quantidade de usuários que iniciarão o teste em um determinado tempo, este tempo é definido através da edição do parâmetro *Tempo de Rampa de Subida*;

- *Tempo de Rampa de Descida*: define o tempo que levará para um conjunto de usuários (definido em *Quantidade de Usuários da Rampa de Descida*) deixe de realizar requisições à aplicação (ocorre durante o final do teste). As ferramentas que possuem esta característica são LoadRunner, SilkPerformer e RPT. Para cada uma delas é muito semelhante a edição desse parâmetro. É necessário simplesmente informar o tempo (o formato dependerá da ferramenta) que levará para cada usuário ou conjunto de usuários terminar sua interação na aplicação tão logo o teste seja iniciado;

- *Quantidade de Usuários da Rampa de Descida*: define a quantidade de usuários que deixarão de fazer carga no sistema em um determinado tempo, este tempo é definido através da edição do parâmetro *Tempo de Rampa de Descida*;

- *Tempo de Aquecimento*: define um período de tempo em que a ferramenta de teste irá coletar informações referentes aos contadores configurados para o teste. Durante esse período não é realizado nenhum tipo de carga no sistema. O objetivo é verificar se não há influência de nenhum fator externo à aplicação que possa, por exemplo, estar concorrendo por recursos e com isso, afetar nos resultados do teste. As ferramentas que possuem essa funcionalidade são o Visual Studio, SilkPerformer e RPT;

- *Perfil da Carga de Trabalho*: somente o LoadRunner, o SilkPerformer e o Visual Studio possuem essa funcionalidade. Refere-se ao perfil de teste que será executado. Os usuários podem iniciar o teste gradativamente ou de forma simultânea.

4.2 Efetivação do Conjunto de Características

Para a efetivação do conjunto adotou-se uma classificação das características necessárias para a criação e execução de casos de teste de desempenho concretizados. Essa classificação se baseou no processo de teste utilizado pela Microsoft, o qual descreve em seis etapas o processo de geração de casos de teste de desempenho para aplicações *web* [25]. Entretanto, outras abordagens foram estudadas [26], [27], [28], porém devido ao fato das informações referentes às características corresponderem e se relacionarem com as informações descritas nas seis etapas do processo de teste da Microsoft, esta abordagem foi a que melhor se adaptou ao trabalho desta dissertação. A seguir são descritas cada uma das etapas deste processo:

- *Identificar os Principais Cenários*: nesta etapa são definidas as informações referentes aos cenários mais críticos para o desempenho do sistema. É onde são definidos os caminhos e atividades mais comuns de serem executados pelos usuários. Um exemplo de caminho crítico poderia ser: *Login* na aplicação, navegar a procura de um determinado produto, adicionar itens no carrinho, informar detalhes da forma de pagamento, finalizar compra, *Logout* na aplicação;

- *Identificar Carga de Trabalho*: nesta etapa são definidas as informações de carga como, por exemplo, número máximo de usuários esperado para acessar a aplicação. Esta informação é normalmente definida durante a etapa de especificação de requisitos de desempenho. Também são definidas nesta etapa informações referentes aos tipos de perfis de execução dos usuários. Em aplicações *e-commerce*, por exemplo, pode-se configurar três tipos diferentes de perfis: *Browse mix*, onde a maior parte do tempo os usuários interagem com a aplicação realizando tarefas de simples navegação; *Search mix*, onde a maior parte do tempo os usuários interagem com a aplicação realizando tarefas de busca e algumas tarefas de compra e; *Order mix*, onde a maior parte do tempo os usuários interagem com a aplicação realizando tarefas de compra [44];

- *Identificar Métricas*: nesta etapa são definidas informações referentes aos contadores e métricas que serão configuradas para o teste, a fim de avaliar o desempenho da aplicação sob teste. O objetivo é identificar os possíveis gargalos responsáveis por uma degradação de desempenho. Tempo de resposta, percentual de utilização de CPU e memória são algumas das diversas métricas utilizadas para monitorar o teste;

- *Criar Casos de Teste*: nesta etapa são definidas as características referentes à configuração de carga de trabalho realizada sob a aplicação. Por exemplo, número de usuários para o teste, percentual de usuários que irão executar os diferentes caminhos definidos na Etapa 1 do processo, tempo de duração do teste, etc.;

- *Simular Carga*: nesta etapa define-se qual ferramenta será utilizada para gerar e executar o teste;

- *Analizar Resultados*: nesta etapa é realizada a análise das informações dos dados obtidos das métricas durante o teste.

O objetivo da adoção deste processo para a construção do conjunto, o qual tem como finalidade complementar as informações referentes às características, foi de classificar as informações adquiridas baseando-se em uma abordagem estruturada, sedimentada e conhecida na comunidade científica [83] [85] [86].

Como é possível visualizar na Figura 4.1, a efetivação da construção do conjunto de características, baseando-se em alguns aspectos das etapas do processo descrito anteriormente, ocorreu da seguinte forma. Inicialmente, a Etapa 1 (Definir *script*) classifica as informações referentes às características *URLs da Aplicação* e *Tempo de Pensamento*. Nesta etapa define-se, na forma de *web links*, o conjunto de caminhos e atividades (*scripts*) a serem executados pelos usuários durante o teste e o tempo decorrido entre uma interação e outra. Esta etapa também possui todo o tipo de informação relativa à interação do usuário com a aplicação e banco de dados, como por exemplo, dados para preenchimento de formulário, dados referente à informação necessária para realizar *login* na aplicação, dados de transações bancárias como número e data do cartão de crédito.

A Etapa 2 (Definir Perfis de Execução) possui informação referente aos perfis de execução dos usuários. Esta etapa pode ser utilizada para representar, através de um identificador, cada um dos tipos de caminhos definidos na Etapa 1. Por exemplo, para um tipo de caminho que possui mais informações de navegação do que qualquer outro tipo de operação, esta etapa pode ser utilizada para representar o nome correspondente a este tipo de caminho. Neste caso pode ser identificado como *browse mix*.

A Etapa 3 (Definir Métricas e Contadores) é utilizada para representar as informações referentes à característica *Contadores*. Nesta etapa são definidas as diversas métricas e contadores que serão utilizados para monitorar os recursos e comportamento do sistema durante a execução de um teste. As métricas escolhidas para a execução de um teste podem variar de acordo com o ambiente e o sistema a ser testado, cabendo a testadores e engenheiros de teste a seleção de um conjunto

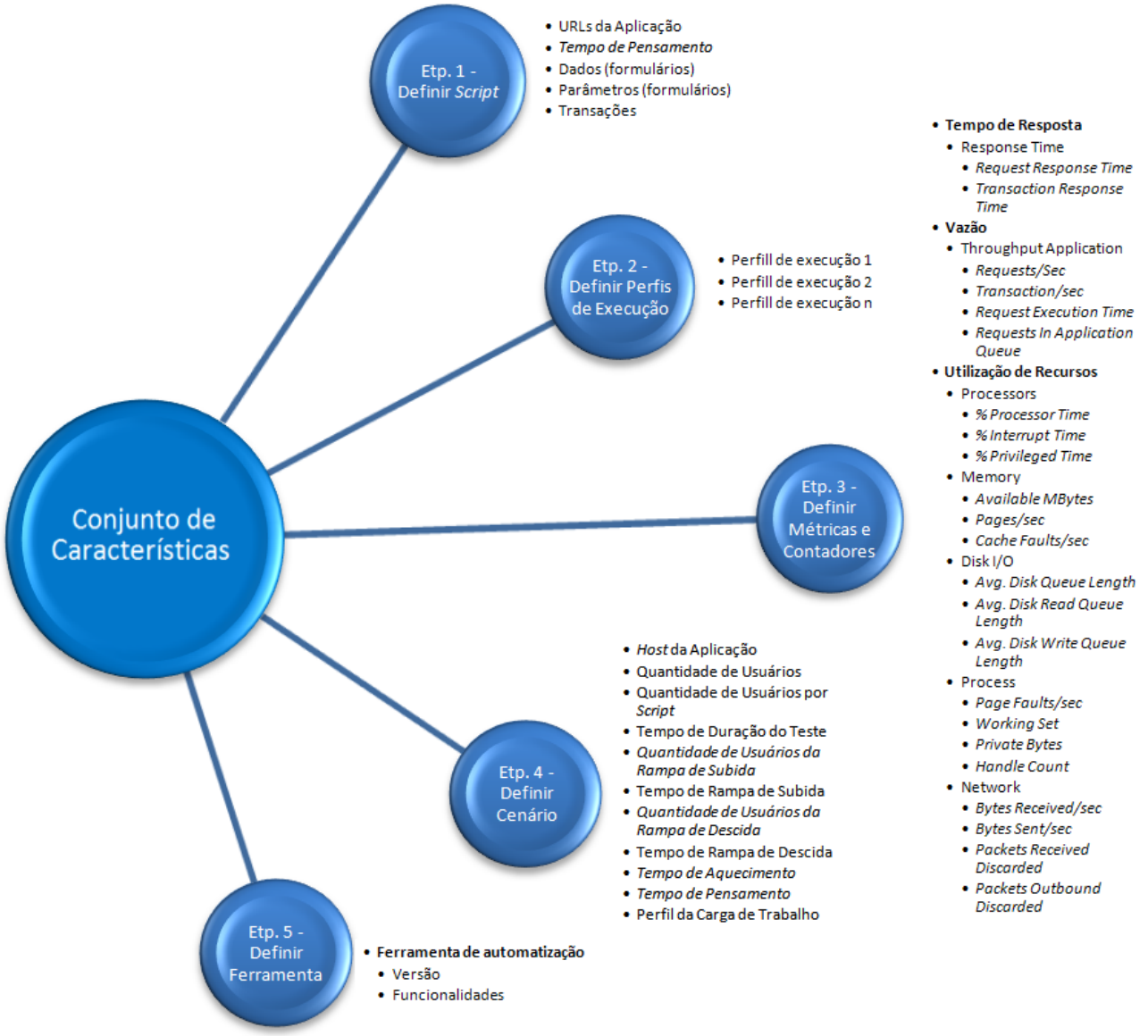

Figura 4.1: Conjunto de características

de métricas de qualidade. Entretanto, [43], [27] e [25] definem que para uma seleção de métricas de qualidade, três indicadores ou objetivos de desempenho são necessários, sendo eles: Tempo de Resposta, Vazão e Utilização de Recursos (CPU, Memória, Disco E/S, Rede E/S). A partir desses indicadores é possível derivar um conjunto de métricas e contadores.

Como é possível visualizar na Figura 4.1, o indicador Utilização de Recursos, por exemplo, possui um conjunto de cinco métricas, são elas *Processors*, *Memory*, *Disk I/O*, *Process* e *Network*. Para cada uma dessas métricas é definida uma lista de contadores, os quais servirão para monitorar o comportamento do sistema durante o teste. A mesma classificação se aplica aos indicadores Tempo de Resposta e *Throughput*. Uma vantagem na forma de organizar estes dados é que este tipo de estruturação permite uma fácil visualização das informações referentes às métricas e contadores.

Na Etapa 4 (Definir Cenário) são definidas todas as informações necessárias para a configuração da carga de trabalho para o teste. Nesta etapa são classificadas as informações referentes às

características *Quantidade de Usuários*, *Tempo de Duração do Teste*, *Tempo de Rampa de Subida*, *Quantidade de Usuários da Rampa de Subida*, *Tempo de Rampa de Descida*, *Quantidade de Usuários da Rampa de Descida*, *Tempo de Aquecimento* e *Perfil da Carga de Trabalho*. A característica *Tempo de Pensamento*, assim como na Etapa 1, é classificada nesta etapa, porém ela é utilizada somente para determinar o valor médio considerando todas as interações do usuário com a aplicação. Além dessas informações, nesta etapa define-se o percentual de usuários que irão executar os diferentes *scripts* que serão utilizados no teste.

A Etapa 5 (Definir Ferramenta) contém informação referente à escolha da ferramenta de automatização de teste que será utilizada para o teste de desempenho. A Etapa 6, como descrito anteriormente, é uma etapa de análise dos resultados realizada após o término do teste. Como o conjunto de características implementado descreve as informações necessárias para criação e execução de casos de teste concretizados, esta etapa acaba não sendo utilizada para a classificação de nenhuma característica e por este motivo, não foi incluída no conjunto.

Após a classificação de todas as características definidas, utilizando como referência o processo de teste de desempenho descrito pela Microsoft, o conjunto de características pôde ser implementado. Além de servir como uma importante base para a classificação e estruturação das características, o processo de teste descrito pela Microsoft também contribuiu para a complementação das informações destas características.

Com a definição do conjunto de características de teste de desempenho é possível desenvolver ferramentas para automatizar a geração e a execução de *scripts* e cenários de teste por meio da instanciação das informações contidas no conjunto apresentado. Em outras palavras, estas ferramentas receberiam como entrada um conjunto de informações (características) referentes aos dados da aplicação a ser testada e, posteriormente, processariam tais informações para gerar casos de teste a serem executados por ferramentas como LoadRunner ou Visual Studio. Neste contexto, uma alternativa interessante é a utilização de linhas de produto de *software* para derivar ferramentas que automatizam o processo de geração e execução de casos de teste. Na seção 4.3 será apresentada uma linha de produto capaz de derivar diversas ferramentas que automatizam a geração de casos de teste de desempenho por meio da instanciação das informações contidas no conjunto de características definido.

4.3 Ferramenta para Linha de Produto de *Software*: PLeTs

Atualmente, muitas empresas estão buscando alternativas para reaproveitar as funcionalidades dos *softwares* por elas produzidos, com o intuito de diminuir tempo e custo no desenvolvimento das novas versões destes *softwares*. Neste contexto, uma alternativa é a utilização dos conceitos de Linha de Produto de *Software* (*Software Product Line - SPL*) [56]. SPL possibilita, por meio da reutilização de componentes de *software*, criar um conjunto de sistemas similares, reduzindo assim o tempo de comercialização, custo e com isso, obter maior produtividade e melhoria da qualidade [56].

Conceitualmente, uma SPL é definida como um conjunto de *softwares* que compartilham carac-

terísticas comuns e gerenciáveis com o intuito de satisfazer as necessidades de um domínio específico, podendo este ser um segmento de mercado ou missão [56] [87]. O objetivo é explorar as semelhanças entre os sistemas visando gerenciar os aspectos relativos à variabilidade entre eles e, dessa forma, determinar uma maior reusabilidade dos componentes de *software*.

Segundo *Software Engineering Institute* (SEI) [87], a engenharia de SPL possui três conceitos principais: o primeiro é denominado desenvolvimento do núcleo de artefatos (*core assets development*), também conhecido como engenharia de domínio (*domain engineering*). O segundo conceito é chamado desenvolvimento do produto (*product development*), também conhecido como engenharia de aplicação (*application engineering*) na nomenclatura alternativa. O terceiro conceito é denominado gerenciamento da linha de produto (*management of product line*).

A parte mais importante de uma SPL diz respeito ao núcleo de artefatos, o qual forma a base de uma SPL e pode ser formado por componentes reusáveis, modelos de domínios, requisitos da SPL, casos de teste e modelo de características (*feature models*), o qual representa os aspectos relacionados à variabilidade em uma linha de produto. O modelo de características apresenta todas as características de uma linha de produto e a relação entre os componentes. Segundo [88], uma característica é uma funcionalidade importante/relevante do sistema, a qual é visível ao usuário final. Uma característica pode ser opcional (*optional*), obrigatória (*common/mandatory*) ou alternativa (*alternatives*). Uma característica opcional pode ou não estar presente no produto. Entretanto, uma característica obrigatória, necessariamente fará parte do produto. Quanto à característica alternativa, ela se trata de uma característica excludente, ou seja, a seleção de uma característica alternativa determina que as demais características alternativas pertencentes ao mesmo grupo e nível hierárquico da característica selecionada não estarão presentes no produto.

A literatura também define que uma característica pode ser do tipo "ou" (*or*), onde uma ou mais características pertencentes ao mesmo grupo e nível hierárquico podem estar presentes no produto. As características também possuem relação entre si e algumas restrições são determinadas, tais como: relação de dependência (*depends/requires*) e relação de exclusão (*excludes*).

Na Figura 4.2, a qual apresenta um modelo de características de telefone móvel, possui quatro características no primeiro nível. As características *Chamadas* e *Tela* são do tipo obrigatória, enquanto as características *GPS* e *Mídia* são opcionais. A característica *Tela* possui três subcaracterísticas alternativas: *Básico*, *Colorido* e *Alta Resolução*, implicando que estas três subcaracterísticas não podem estar presentes simultaneamente no mesmo produto. A subcaracterística *Básico* possui uma relação de exclusão com a característica *GPS*, onde a seleção de uma característica exclui a outra, ou seja, se a tela de um produto for do tipo básica, necessariamente este produto não terá a característica *GPS* e vice-versa. A característica *Mídia* possui duas características do tipo "ou": *Câmera* e *MP3*, onde a característica *Câmera* possui uma relação de dependência com a característica *Alta Resolução*. Neste contexto, caso um produto possua uma mídia do tipo *Câmera*, necessariamente a tela deverá ser *Alta Resolução*.

O modelo de características de uma SPL é responsável por representar os aspectos relacionados à variabilidade, a qual pode estar vinculada a diferentes níveis de *abstração* como código fonte e

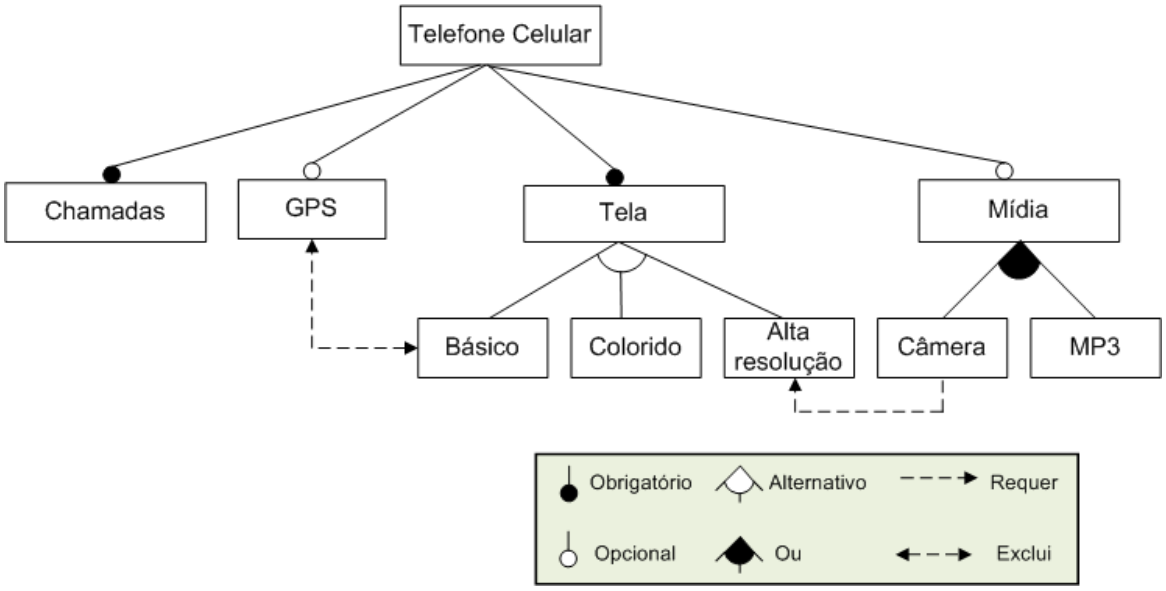

Figura 4.2: Exemplo de um modelo de características para telefone móvel [2]

documentação. As variabilidades são representadas por pontos de variabilidades (*Variation points*) e variantes (*Variants*), onde um ponto de variabilidade pode conter uma ou mais variantes. Em uma linha de produto de telefones móveis, por exemplo, um ponto variabilidade poderia ser o protocolo de comunicação e as variantes deste ponto de variabilidade poderiam ser GSM, UMTS.

Devido à capacidade de modelar e representar variabilidades e, com isso, obter maior reuso das características, funcionalidades e componentes, a utilização de SPLs pode trazer inúmeras vantagens e benefícios a clientes e consumidores. Segundo [89], depois da chegada das linguagens de programação de alto nível, as SPLs podem representar a mais "empolgante" e significativa mudança no paradigma de desenvolvimento, devido à facilidade e eficiência em desenvolver sistemas com a utilização de SPLs. O autor ainda enfatiza que, em nenhuma outra área da engenharia de *software*, são evidenciadas melhorias como as que a SPL provê. O fator mais preponderante relativo à afirmação do autor, diz respeito aos benefícios advindos com a utilização de SPL. Neste sentido, citam-se a qualidade dos produtos, menor tempo para o lançamento no mercado de um novo produto da família e produtividade no desenvolvimento dos produtos. Além dessas vantagens, muitas empresas, dentre elas citam-se Philips, Nokya, têm descoberto que, quando bem implementada, uma estratégia para utilização de linhas de produto pode trazer diversas outras melhorias, tais como [87]: ganho de produtividade em larga escala; aumento da qualidade do produto; maior satisfação dos clientes; maior facilidade na construção de produtos em massa; maior eficiência no uso de recursos humanos; maior facilidade da empresa se manter no mercado; menor tempo para o produto chegar ao mercado; redução de custo; habilidade para migrar, em meses, para novos mercados (não em anos).

Atualmente, está em processo de desenvolvimento no Centro de Pesquisa em Engenharia de Sistemas da PUCRS uma linha de produto denominada PLeTs (*Product Line Testing Tools*) [3]. Esta linha de produto faz parte do trabalho de uma tese de doutorado e utiliza técnicas de SPL para

automatizar processos de teste. A PLeTs é uma SPL que busca facilitar a derivação de ferramentas de teste baseado em modelos (*Model Based Testing* - MBT) [31], com os quais é possível criar e executar casos de teste de forma automatizada. O objetivo desta SPL não é apenas gerenciar a reutilização de artefatos e componentes de *software*, mas também tornar mais fácil e rápido o desenvolvimento de uma nova ferramenta e, como citado há pouco, otimizar a criação e execução de casos de teste. Ela está sendo desenvolvida com o intuito de ser utilizada por engenheiros de *software*, programadores e engenheiros de teste, auxiliando no processo de planejamento e execução de casos de teste e *scripts* de teste. Para melhor entendimento, é apresentado na Figura 4.3 o modelo de características atual da SPL PLeTs.

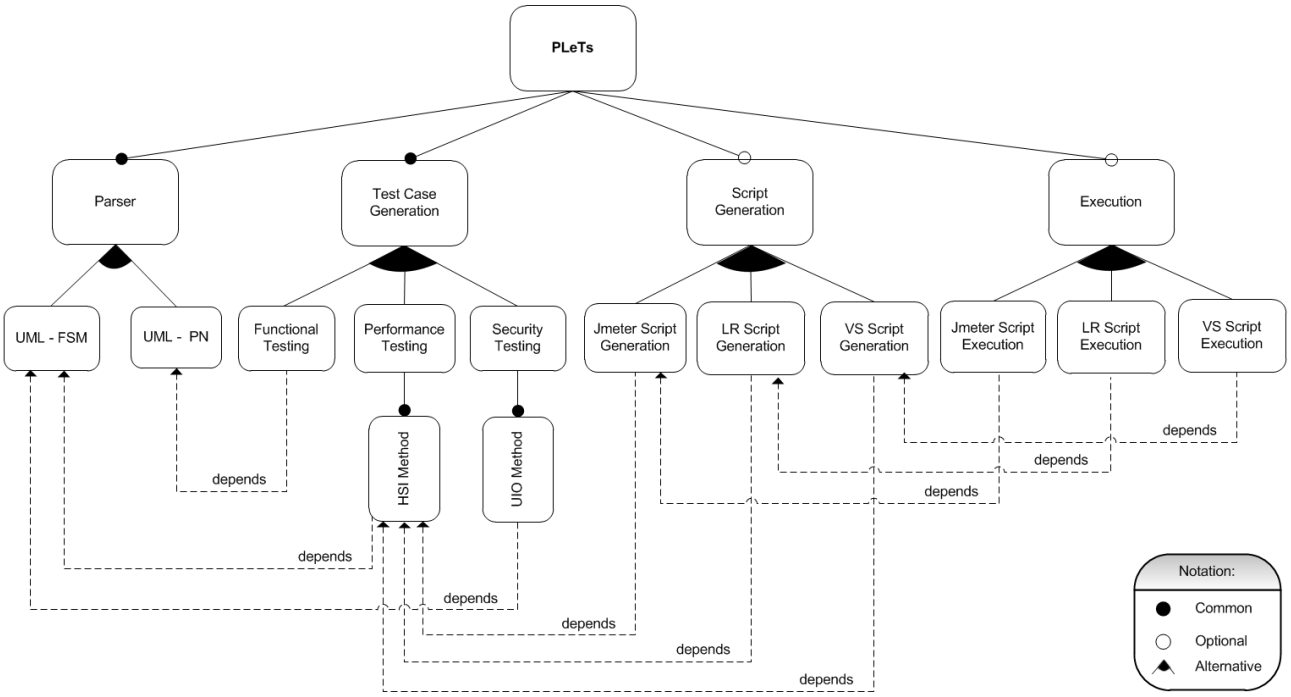

Figura 4.3: Modelo de características da PLeTs [3]

Atualmente, o modelo de características da PLeTs é constituída de quatro características em seu primeiro nível, *Parser*, *Test Case Generation*, *Script Generation* e *Execution*.

- *Parser*: esta é uma característica ou ponto de variabilidade obrigatório e seu objetivo é extrair as informações contidas, por exemplo, em um arquivo UML para então gerar um modelo formal na característica *Test Case Generation*. Este modelo, atualmente pode ser representado por uma Máquina de Estados Finitos (*Finite State Machine* - FSM) ou uma Rede de Petri (*Petri Net* - PN). Atualmente ela possui duas variantes mutuamente excludentes: *UML - FSM* e *UML - PN*, onde apenas uma das duas variantes é selecionada;

- *Test Case Generation*: também é uma característica obrigatória. É responsável por gerar sequências de teste a partir das informações contidas no modelo formal (FSM ou PN). Podem ser geradas sequências para teste funcional, desempenho ou segurança. Este ponto de variabilidade possui três variantes (*Functional Testing*, *Performance Testing* e *Security Testing*) e

obrigatoriamente, assim como a característica *Parser*, apenas uma delas pode ser selecionada. Como pode-se observar na Figura 4.3, as três variantes possuem uma relação de dependência (*depends*) com as variantes do *parser*. Indicando a seleção de uma variante é dependente de outra. Por exemplo, caso a variante *Functional Testing* seja escolhida, obrigatoriamente a variante *UML - PN* deve ser selecionada;

- *Script Generation*: consiste na geração de *scripts* de teste para ferramentas de automação de teste. Também possui três variantes, sendo elas: *JMeter Script Generation*, *LR Script Generation* e *VS Script Generation*, as quais representam a implementação de *scripts* e cenários de teste, respectivamente, para as ferramentas LoadRunner, JMeter e Visual Studio;

- *Execution*: representa a execução da ferramenta e também a execução do teste sob a aplicação a ser testada, utilizando uma determinada ferramenta com os *scripts* e cenário de teste gerados na etapa anterior. Possui três variantes e cada uma possui uma relação de dependência com as variantes pertencentes ao ponto de variabilidade *Script Generation*.

É importante de salientar que, o modelo de características atual pode evoluir e ser incrementado. Podem ser adicionados novos pontos de variabilidade ou até mesmo outras variantes aos pontos de variabilidade existentes. É totalmente factível que uma nova variante representando a geração e execução de teste com outra ferramenta de automatização de teste seja incluída. Também poderia ser adicionada uma nova variante ao *Parser*. Por exemplo, as informações do teste que, atualmente, provêm de diagramas UML poderiam ser expressas utilizando outra estrutura, como arquivos de texto, *logs*, XML (*Extensible Markup Language*) [90], entre outras.

4.4 Geração de Casos de Teste

O conjunto de características para ferramentas de teste de desempenho, o qual é o foco desta dissertação, serviu como referência para a implementação de *plugins* para a PLeTs. Estes *plugins* são responsáveis pela geração de *scripts* e cenários para os produtos derivados da PLeTs. Neste contexto, serão detalhadas as funcionalidades de cada *plugin* implementado. Entretanto, para realizar a geração destes *scripts* e cenários, foi necessária a extração de informações provenientes de outra pesquisa de mestrado [29]. A referida pesquisa possui como foco a análise de características de modelos formais ou semi-formais a serem utilizadas para descrição de requisitos de teste. Ela se utiliza de modelos como Linguagem de Modelagem Unificada (*Unified Modeling Language* - UML) [30], Redes de Autômatos Estocásticos (*Stochastic Automata Networks* - SAN) [91], Redes de Petri (*Petri Nets* - PN) [92], Máquina de Estados Finitos (*Finite State Machine* - FSM) [93]. A dissertação de mestrado oriunda da pesquisa [29] tem como objetivo a extração de características de teste que os modelos possuem para posterior geração de *scripts* e/ou cenários de teste.

Nesse contexto, o objetivo foi verificar se as informações necessárias para a execução de casos de teste para as ferramentas coincidem com as informações presentes nos modelos. Durante o início da pesquisa três hipóteses foram levantadas, a primeira hipótese e também a menos desejável

era que todas as características presentes nos modelos fossem diferentes das características para a geração de casos de teste para as ferramentas (ver Figura 4.4-a). Outra hipótese levantada era de que o conjunto das características presentes nos modelos fosse igual ao conjunto das características definidas para as ferramentas, entretanto, era pouco provável que esta hipótese fosse verdadeira (ver Figura 4.4-b). A terceira hipótese e também a mais coerente era que houvesse uma intersecção entre as características presentes nos modelos e as características definidas para as ferramentas, ou seja, poderia ocorrer que algumas características presentes nos modelos não fossem necessárias para a geração de *scripts* e cenários para as ferramentas e que algumas características definidas para as ferramentas não estivessem presentes nos modelos (ver Figura 4.4-c).

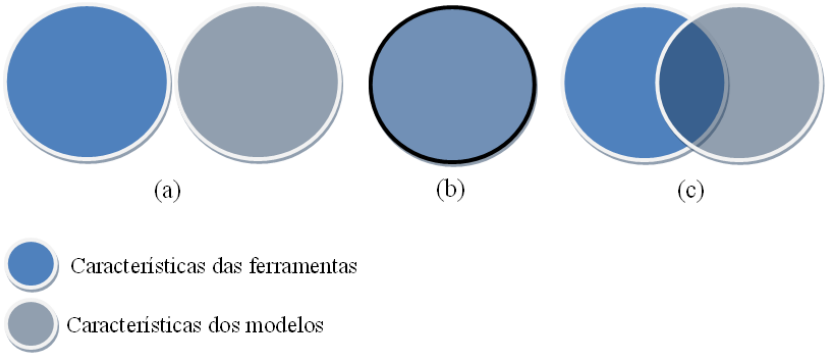

Figura 4.4: Relação entre as características das ferramentas e dos modelos

Ao final, foi evidenciado que a última hipótese foi de fato verdadeira, pois a maioria das características definidas para as ferramentas coincidem com as informações referentes às características presentes nos modelos. Como é possível visualizar na Tabela 4.4, existe uma série de características comuns a ambos os conjuntos, ainda que possuam nomes ou identificadores distintos representam na verdade o mesmo tipo de informação. Isto é o que está descrito na quinta coluna da Tabela 4.4, a qual apresenta a relação dos identificadores das características coincidentes entre os dois conjuntos. Por exemplo, a característica SUT com ID 1 é coincidente com a característica *Host* da Aplicação que também possui ID igual a 1. As demais colunas (2 e 4) desta tabela apresentam respectivamente as características presentes nos modelos e as características definidas para as ferramentas.

As pesquisas que definiram o conjunto de características presentes nos modelos e o conjunto das características para as ferramentas foram utilizadas para a implementação de *plugins* para as quatro etapas do modelo de características da PLeTs. O conjunto de características presentes nos modelos foi utilizado para a implementação de *plugins* para as etapas *Parser* e *Test Case Generation*, enquanto, o conjunto de características para as ferramentas foi utilizado como referência na implementação de *plugins* para as duas etapas subsequentes, *Script Generation* e *Execution*. Até o presente momento a PLeTs é constituída de seis *plugins*:

- *Parser UML*: é responsável por extrair informações do teste de um arquivo no formato *XMI*, o qual foi gerado a partir de diagramas de caso de uso e diagramas de atividades, e armazená-las em uma estrutura de dados chamada "estrutura intermediária". As informações referentes ao

Tabela 4.4: Características coincidentes

ID	Características definidas para os modelos	ID	Características definidas para as ferramentas	Características coincidentes entre os conjuntos (ID–ID)
1	SUT	1	*Host* da Aplicação	1–1
2	Usuários Virtuais	2	Quantidade de Usuários	2–2
3	Probabilidade	3	Quantidade de Usuários por Script	3–3
4	Tempo de Execução	4	Tempo de Duração do Teste	4–4
5	Tempo de Inicialização	5	Tempo de Rampa de Subida	5–5
6	Usuários de Inicialização	6	Quantidade de Usuários da Rampa de Subida	6–6
7	Tempo de Finalização	7	Tempo de Rampa de Descida	7–7
8	Usuários de Finalização	8	Quantidade de Usuários da Rampa de Descida	8–8
9	Tempo de Espera	9	Tempo de Pensamento	9–9
10	Requisição	10	URLs da Aplicação	10–10
11	Parâmetro	11	Parâmetros	11–11
12	Dados	12	Dados	12–12
13	Transação	13	Transações	13–13
14	Transações por Segundo	14	Contadores	14–14
	Tempo de Resposta			
	Requisições por Segundo			
	Vazão			
	Utilização de Recursos			
		15	Tempo de Aquecimento	
		16	Perfil da Carga de Trabalho	

conjunto de características descritas na pesquisa de [29] serviram como referência para popular os diagramas UML com os dados do teste. Os diagramas de caso de uso possuem informações do cenário de teste, tais como: número de usuários, tempo de duração do teste, etc. Por outro lado, os diagramas de atividades apresentam informações para posterior geração dos *scripts* de teste, ou seja, descreve as informações referentes às ações dos usuários para a realização de uma determinada tarefa na aplicação a ser testada;

- *FSM Test Case Generation*: extrai as informações do teste (apenas as informações referentes às ações dos usuários na aplicação), as quais foram armazenadas na estrutura intermediária pelo *plugin Parser UML*, para gerar uma FSM. Após gerar a FSM é aplicado o método *Harmonized State Identification* (HSI) [94] para geração de sequências de teste e em seguida, as informações do teste são armazenadas em outra estrutura de dados, a qual chamamos de

"estrutura de sequência de teste" (a Seção 4.5 explica com maiores detalhes os componentes desta estrutura). A estrutura de sequência de teste também é populada com as informações referentes ao cenário de teste, as quais foram armazenadas na estrutura intermediária, entretanto, estas informações são diretamente repassadas de uma estrutura a outra. As informações contidas na estrutura de sequência de teste são utilizadas para a geração do cenário e *scripts* de teste para as ferramentas LoadRunner e Visual Studio;

- *LoadRunner Script Generation*: extrai as informações armazenadas na estrutura de sequência de teste para gerar o cenário e os *scripts* para a ferramenta LoadRunner. Entretanto, antes da geração dos cenários e *scripts* de teste, uma interface é apresentada ao usuário que por meio dela tem a opção de selecionar o cenário de teste a ser executado e os *scripts* vinculados a este cenário;

- *Visual Studio Script Generation*: extrai as informações armazenadas na estrutura de sequência de teste para gerar o cenário e os *scripts* para a ferramenta de teste do Visual Studio. Entretanto, antes da geração dos cenários e *scripts* de teste, uma interface é apresentada ao usuário que por meio dela tem a opção de selecionar o cenário de teste a ser executado e os *scripts* vinculados a este cenário;

- *LoadRunner Test Execution*: executa o teste com a ferramenta LoadRunner utilizando o cenário e os *scripts* gerados pelo *plugin LoadRunner Script Generation*;

- *Visual Studio Test Execution*: executa o teste com a ferramenta de teste do Visual Studio utilizando o cenário e os *scripts* gerados pelo *plugin Visual Studio Script Generation*.

Pelo fato de sua arquitetura ser baseada em *plugins*, a PLeTs permite a seleção de diversos *plugins* e a integração dos *plugins* selecionados possibilita gerar até dois produtos para teste de desempenho. Sendo que um produto é capaz de automatizar a geração e execução de *scripts* e cenários de teste com o LoadRunner e o outro com o Visual Studio. Outra característica importante de salientar diz respeito à arquitetura dos novos produtos gerados, a qual é integralmente baseada nas funcionalidades dos *plugins* selecionados durante a execução da PLeTs.

Quando um novo produto (ferramenta de teste de desempenho) é gerado pela PLeTs por meio da combinação dos *plugins* descritos recentemente, a geração e execução de casos de teste pode ser iniciada. Entretanto, é necessária, em um primeiro momento, a geração de um modelo de teste. Este modelo é composto por diagramas de caso de uso, os quais possuem informações referentes ao cenário de teste como número de usuários, tempo de duração do teste, entre outras. Cada diagrama de caso de uso é decomposto em um diagrama de atividades, o qual possui informações referentes às ações dos usuários sob a aplicação. As informações de teste descritas nos diagramas são representadas em um arquivo no formato *XMI* e, durante a execução da ferramenta de teste, este arquivo é submetido a um *parser* com a finalidade de popular a estrutura intermediária. Em seguida, são geradas sequências de teste quando aplicado o método HSI sob uma FSM gerada a partir das informações inseridas na estrutura intermediária.

Estas sequências de teste, bem como as informações referentes ao cenário armazenadas na estrutura de sequência, se tratam de casos de teste não instanciados e por este motivo são chamados de "casos de teste abstratos". A próxima etapa diz respeito à instanciação dos casos de teste abstratos, a qual consiste na geração do cenário e *scripts* de teste para uma determinada tecnologia (LoadRunner ou Visual Studio). Estes casos de teste quando instanciados são chamados de casos de teste concretizados. A etapa final consiste na execução automática do teste utilizando determinada tecnologia.

4.5 Implementação do Conjunto de Características

Como citado no Capítulo 4.3, a PLeTs é uma SPL que utiliza técnicas de Linhas de Produto para gerar ferramentas que automatizam processos de teste. Utilizando os produtos por ela gerados é possível, por meio da extração de informações de modelos como UML, automatizar o processo de geração e execução de casos de teste. Para tornar possível este processo de automatização, foram implementados *plugins* para as etapas *Script Generation* e *Execution* da PLeTs, os quais utilizam as informações do conjunto de características apresentado anteriormente como referência. Tendo como base as informações do conjunto, estes *plugins* implementam a geração automática de *scripts* e cenários para as ferramentas de automatização de teste de desempenho. A seguir, é apresentado o processo de criação destes *plugins* para a PLeTs, utilizando duas ferramentas para execução automática de testes de desempenho, HP LoadRunner e Microsoft Visual Studio.

A implementação dos *plugins* para teste de desempenho para as ferramentas LoadRunner e Visual Studio abrangeu duas etapas: a primeira foi realizar uma busca por arquivos de configuração para as duas tecnologias que representassem as informações definidas no conjunto de características; a segunda etapa consistiu na implementação de uma estrutura de dados, a qual foi utilizada para representar as informações definidas no conjunto de características para as ferramentas. Esta estrutura de dados foi chamada de "estrutura de sequência de teste" e as informações contidas nesta estrutura foram utilizadas para a geração do cenário e *scripts* de teste para as ferramentas LoadRunner e Visual Studio.

4.5.1 LoadRunner

Para a ferramenta LoadRunner, inicialmente, foi realizada uma pesquisa por arquivos de configuração que pudessem conter o conjunto de informações definidas para a Etapa 1 do conjunto de características. Foi identificado que as informações especificadas na primeira etapa do conjunto deveria estar presente em arquivos (*scripts*) que possuíssem todo o tipo de informação referente à interação do usuário com a aplicação. Como é possível visualizar na Figura 4.5, informações definidas na Etapa 1 do conjunto, como *URLs* da aplicação, *Tempo de Pensamento* e informações referentes aos dados de autenticação de usuário (*username e password*) foram necessárias incluir no *script* LoadRunner. Validando portanto, a primeira etapa do conjunto de características implementado.

O próximo passo, foi analisar as informações contidas na Etapa 2 do conjunto e verificar onde seria

```
Action()
{
web_url("Search",
"URL=http://192.168.1.26/skillsApp/mainIE.jsp",
"Resource=0",
"RecContentType=text/html",
"Referer=",
"Mode=HTML",
LAST);

lr_think_time(10);
web_submit_form("Loging.jsp",
ITEMDATA,
"Name=username", "Value=admin", ENDITEM,
"Name=password", "Value=123456", ENDITEM,
LAST);
```

Figura 4.5: *Script* LoadRunner gerado pelo produto derivado PLeTs

necessária a inclusão destas informações. Para o LoadRunner esta informação não foi mandatória, uma vez que para a ferramenta de teste da HP, os dados desta etapa foram utilizados apenas para representar o nome do *script*. Portanto, a informação contida nesta etapa não pode ser substancialmente aproveitada.

Entretanto, as informações contidas nas duas etapas subsequentes do conjunto de características foram fundamentais e imprescindíveis para o processo de geração automática de casos de teste concretizados para o LoadRunner. Isto se deve ao fato de que as etapas 3 e 4 do conjunto foram utilizadas para configurar todo o tipo de informação referente ao arquivo que o LoadRunner usa para armazenar os dados do cenário de teste. Dados de configuração, tais como: *Quantidade de Usuários*, *Quantidade de Usuários* que irão executar as atividades de determinado *script*, *Tempo de Rampa de Subida*, *Tempo de Rampa de Descida*, *Perfil da Carga de Trabalho* e todas as informações de métricas e contadores de desempenho são definidas neste arquivo.

Na Figura 4.6, são apresentadas algumas das informações utilizadas para a configuração do cenário de teste para o LoadRunner. Nesta figura estão destacados sete tipos de informação: *Vusers*, que corresponde ao número total de usuários configurados para executar o teste; *RunFor*, refere-se ao tempo total de execução do teste em milissegundos; *TotalVusersNumber*, que corresponde à quantidade de usuários que irá executar as atividades descritas em determinado *script*; *Count*, que corresponde ao número de usuários que iniciarão o teste (*Quantidade de Usuários da Rampa de Subida*) ou irão deixar de realizar suas requisições (*Quantidade de Usuários da Rampa de Descida*) e *Interval*, que define o tempo que levará para um conjunto de usuários iniciar o teste (*Tempo de Rampa de Subida*) ou terminá-lo (*Tempo de Rampa de Descida*).

Para o produto derivado da PLeTs, aquele arquivo foi utilizado como *template*, pois nele existem diversas informações, como por exemplo, a versão do LoadRunner, onde a modificação é desnecessária, pois estas informações não são parametrizáveis e, por este motivo, não variam de um teste para outro. Portanto, somente as informações referentes à configuração do cenário de teste são consideradas parametrizáveis e portanto, é obrigatória a informação destes parâmetros.

A Etapa 5 do conjunto de características, define as informações da ferramenta de teste a ser

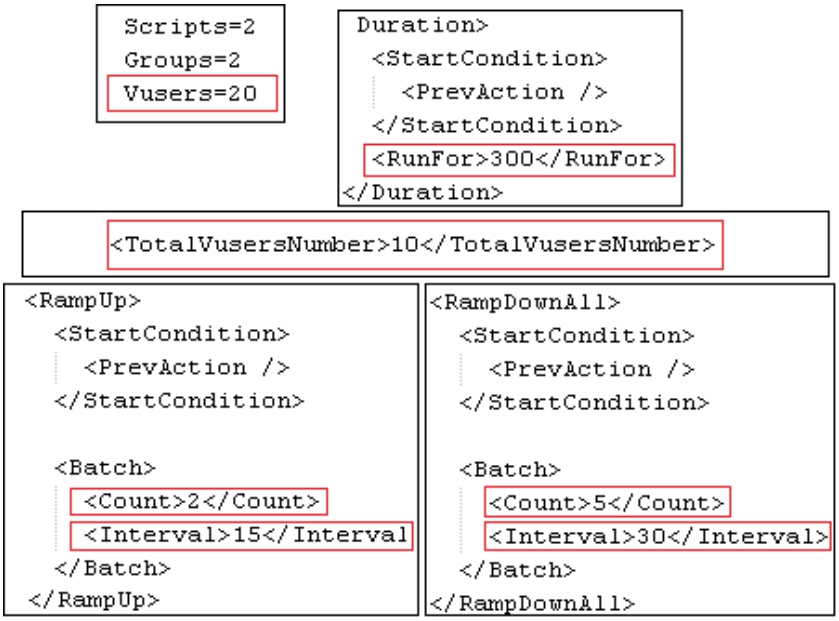

Figura 4.6: Cenário LoadRunner

utilizada. Nesta etapa são definidas, portanto, diversas informações referentes ao LoadRunner como, por exemplo, versão da ferramenta e funcionalidades.

4.5.2 Visual Studio

Para o Visual Studio, os mesmos passos utilizados para a geração e execução automática de casos de teste concretizados com o LoadRunner foram necessários. Portanto, a mesma pesquisa por arquivos de configuração que foi realizada para o LoadRunner, foi realizada para a ferramenta da Microsoft. Entretanto, diferentemente da ferramenta de teste da HP, para o Visual Studio os arquivos que contém as informações de configuração do teste possuem dados que estão estruturados em um formato XML. As informações referentes à interação do usuário com a aplicação, definidas na Etapa 1 do conjunto de características, estão em um arquivo cuja extensão é "*webtest*". Na Figura 4.7, estão destacadas algumas das informações que foram configuradas para este arquivo, baseando-se nas informações da primeira etapa do conjunto. Como é possível visualizar, nesta figura foram destacadas informações referentes a *URLs da Aplicação*, *Tempo de Pensamento* e dados de autenticação de usuário para efetuar *login* na aplicação.

As informações referentes à definição dos perfis de execução dos *scripts* de teste, especificados na Etapa 2 do conjunto de características, são configuradas para o Visual Studio no arquivo de cenário "*loadtest*". Esta informação é descrita neste arquivo pela *tag TestProfile Name* e o valor desta *tag* possui o nome dado ao perfil de execução e referencia o arquivo *webtest* (*script de teste*) que será utilizado para a configuração do cenário de teste.

As informações referentes às etapas 3 e 4 do conjunto, assim como na Etapa 2, são utilizadas para mapear as informações referentes à configuração do arquivo de cenário do Visual Studio (*loadtest*). Como é possível visualizar na Figura 4.8, são definidas diversas informações de configuração do

```
<?xml version="1.0" encoding="utf-8" ?>
<WebTest Name="Teste" Id="9335d430-921d-4710-a600-b0ec7506103c"
  Owner="" Priority="2147483647" Enabled="True" CssProjectStructure=""
  CssIteration="" Timeout="0" WorkItemIds=""
  xmlns="http://microsoft.com/schemas/VisualStudio/TeamTest/2010"
  Description="" CredentialUserName="" CredentialPassword=""
  PreAuthenticate="True" Proxy="" StopOnError="False" RecordedResultFile="">
- <Items>
    <Request Method="GET" Version="1.1"
      Url="http://localhost/skillsApp/mainIE.jsp" ThinkTime="1"
      Timeout="300" ParseDependentRequests="True"
      FollowRedirects="True" RecordResult="True" Cache="False"
      ResponseTimeGoal="0" Encoding="utf-8"
      ExpectedHttpStatusCode="0" ExpectedResponseUrl=""
      ReportingName="" />
  - <Request Method="POST" Version="1.1"
      Url="http://localhost/skillsApp/Loging.jsp" ThinkTime="3"
      Timeout="300" ParseDependentRequests="True"
      FollowRedirects="True" RecordResult="True" Cache="False"
      ResponseTimeGoal="0" Encoding="utf-8"
      ExpectedHttpStatusCode="0"
      ExpectedResponseUrl="http://localhost/skillsApp/mainHome.jsp?
      sessionId=11234249546279206" ReportingName="">
    - <FormPostHttpBody>
        <FormPostParameter Name="nome" Value="admin"
          RecordedValue="admin" CorrelationBinding=""
          UrlEncode="True" />
        <FormPostParameter Name="pass" Value="admin"
          RecordedValue="admin" CorrelationBinding=""
          UrlEncode="True" />
        <FormPostParameter Name="submit" Value="Login"
          RecordedValue="Login"
          CorrelationBinding="{{FormPostParam1.submit}}"
          UrlEncode="True" />
      </FormPostHttpBody>
    </Request>
      ⋮
  </Items>
+ <ValidationRules>
</WebTest>
```

Figura 4.7: Arquivo *WebTest* gerado para o Visual Studio

cenário de teste. Por exemplo, a *tag LoadProfile Pattern* que possui o valor "*Constant*", define para este caso que a quantidade de usuários executando o teste não irá aumentar em nenhum momento durante o teste. Em outras palavras, não serão utilizadas as informações referentes à *Tempo de Rampa de Subida* (*Ramp-up*) neste exemplo. Esta figura também destaca outros tipos de informações que foram necessárias configurar com base no conjunto de características, tais como: tempo de duração do teste em milissegundos, definida pela *tag RunDuration*; contadores de monitoramento, definidos na *tag CounterCategory Name* e; quantidade de usuários para o teste, definida pela *tag InitialUsers*.

Assim como no LoadRunner, a Etapa 5 do conjunto de características é utilizada para definir as informações da ferramenta de teste a ser utilizada. Nesta etapa são definidas, portanto, diversas informações referentes ao Visual Studio, como por exemplo, versão da ferramenta e funcionalidades. Após o mapeamento de cada informação definida no conjunto de características nos diferentes

Figura 4.8: Arquivo *LoadTest* gerado para o Visual Studio

arquivos de configuração pertencentes às duas ferramentas, foi implementada uma estrutura de dados para representar as informações definidas no conjunto de características. Esta estrutura, a qual foi chamada de estrutura de sequência de teste, é composta pelas informações definidas no conjunto de características para ferramentas de teste de desempenho.

O objetivo de criar esta estrutura de dados, a qual serviu como referência para a geração do cenário e *scripts* de teste, foi definir uma estrutura genérica pela qual as informações necessárias para a criação dos casos de teste concretizados (cenário e *scripts*) não fosse condicionada ao método ou formalismo onde tais informações foram, inicialmente, geradas. Por exemplo, não é desejável que a geração dos *scripts* e cenários de teste seja totalmente dependente das informações do teste geradas a partir de diagramas UML, como é o caso da abordagem utilizada com as ferramentas ou produtos originados da PLeTs. Visto que o objetivo do conjunto de características é automatizar a geração e execução de cenários e *scripts* independentemente de onde as informações necessárias para o teste foram geradas, seja a partir de diagramas UML, seja de *logs* com informações das interações do usuário com a aplicação ou qualquer outra abordagem.

As informações contidas no conjunto de características, foram utilizadas para a implementação de *plugins* para a PLeTs, neste caso foram implementados *plugins* para as ferramentas LoadRunner e Visual Studio. Estes *plugins* implementam a geração automática dos *scripts* e cenários de teste para estas ferramentas de teste de desempenho. Durante sua execução, a ferramenta de teste gerada pela PLeTs cria e configura estes arquivos e em seguida executa o *software* de automatização de teste selecionado com os arquivos de *script* e cenário criados.

4.6 Conjunto de Características para Uma Ferramenta de Teste Estrutural

Conforme apresentado anteriormente, o foco da pesquisa desta dissertação se baseou em um estudo onde diversas ferramentas foram analisadas a fim de determinar um conjunto de características

necessário para automatizar a geração e a execução de *scripts* e cenários para ferramentas de teste de desempenho. Com o objetivo de expandir o escopo da pesquisa foi realizado outro estudo com a finalidade de verificar se aplicando a mesma abordagem utilizada para teste de desempenho seria possível definir um conjunto de características para automatizar a geração e a execução de casos de teste para uma ferramenta de teste estrutural, a JaBUTi [24]. O objetivo deste estudo foi verificar quais arquivos de configuração a JaBUTi utiliza para armazenar seus dados de execução e análise de teste e com isso, determinar um conjunto de características (informações) necessário para a geração e execução de casos de teste estrutural. Para essa pesquisa um conjunto de trabalhos foi analisado como, por exemplo, dissertações, artigos e relatórios técnicos [24] [95] [96] [97].

Por meio da análise desses trabalhos e também da própria ferramenta, considerações importantes puderam ser realizadas a respeito de seu funcionamento interno. Para a execução de casos de teste com a JaBUTi, é necessário criar um arquivo de projeto, cuja extensão é "*.jbt*". Todas as informações referentes ao *bytecode* das classes que serão testadas e o caminho dessas classes são armazenadas neste arquivo. Neste arquivo também são descritos os caminhos referentes a todas as bibliotecas que pertencem à aplicação a ser testada, bem como o *bytecode* da classe "TestDriver", esta que contém informações que serão utilizadas para testar as classes da aplicação sob teste. Com base nas informações do *bytecode* das classes referenciadas neste arquivo, a JaBUTi constrói o grafo definição-uso (*Def-Use Graph* - DUG) para cada uma dessas classes, baseando-se nos critérios de análise de fluxo de dados e análise de fluxo de controle.

Após a criação do arquivo de projeto, a JaBUTi realiza a instrumentação das classes a serem testadas. Ela executa os casos de teste, descritos no arquivo *TestDriver*, por meio da chamada do método *probe.DefaultProber.probe* que armazena as informações do programa que será testado. Em seguida, outro método é chamado (*probe.DefaultProber.dump*) e todos os dados coletados na chamada do método anterior são armazenados em um arquivo de rastro ("*.trc*"). Os dados gravados no arquivo de rastro correspondem aos caminhos percorridos pelo programa durante a execução do teste. A JaBUTi extrai as informações deste arquivo de rastro, atualiza os dados do teste e recalcula as informações de cobertura.

De posse desse conhecimento, foi possível criar um conjunto de características contendo as informações necessárias para a automatização de teste estrutural utilizando a ferramenta JaBUTi. Como é possível visualizar na Figura 4.9, o conjunto de características necessário para a geração e execução automática de casos de teste para a JaBUTi é dividido em três etapas. Na primeira etapa são especificadas as informações referente ao conjunto de classes que se deseja testar; o conjunto de métodos que serão chamados por essas classes; e o conjunto de parâmetros que será utilizado por esses métodos. As informações da Etapa 1 são utilizadas para a criação automática da classe *TestDriver*. Na Etapa 2 são descritas todas as informações necessárias para a criação do arquivo de projeto da JaBUTi, como por exemplo, o caminho referente ao *bytecode* das classes que serão testadas. Na Etapa 3 são definidas as informações necessárias para a execução dos casos de teste e consequentemente a geração do arquivo de rastro, utilizado para recalcular as informações de cobertura.

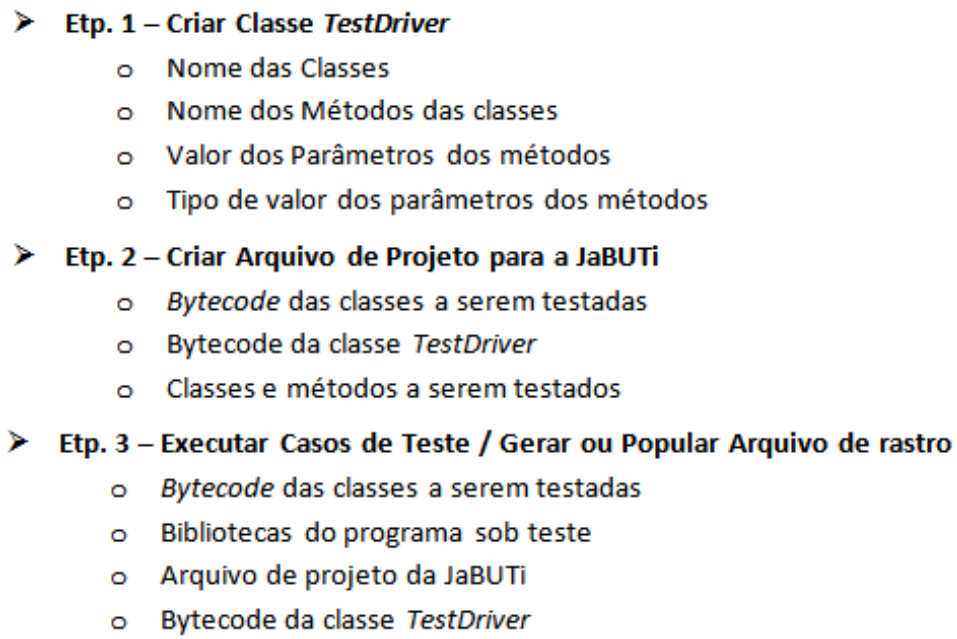

Figura 4.9: Conjunto de características para a JaBUTi

Com base nessas informações foi possível implementar um outro *plugin* para a PLeTs, com o qual é possível gerar um produto que possibilita a geração e execução automática de casos de teste estrutural utilizando a ferramenta JaBUTi. Para tornar possível a implementação deste *plugin*, inicialmente, foram utilizadas as informações descritas na Etapa 1 do conjunto para a geração automática da classe *TestDriver*. Após a geração deste arquivo é criado um processo do compilador Java (Javac) passando por parâmetro a classe *TestDriver* gerada. Em seguida, as informações descritas na Etapa 2 do conjunto são utilizadas para a geração automática do arquivo de projeto da JaBUTi. Na Etapa 3 são utilizadas as informações para automatizar a execução de um caso de teste e gerar o arquivo de rastro. Ao final, um processo Java é criado para executar a JaBUTi contendo as informações de cobertura já atualizadas. Nas Figuras 4.10 e 4.11 é possível visualizar respectivamente algumas informações do arquivo de projeto da JaBUTi gerado pelo produto derivado da PLeTs e a interface da JaBUTi com as informações de cobertura atualizadas.

4.7 Considerações

Este capítulo apresentou as etapas para a implementação de um conjunto de características para ferramentas de teste de desempenho. Com base nas informações descritas no conjunto implementado, foram criados quatro *plugins* para a PLeTs. Também foi implementado um *plugin* para automatizar a atividade de teste utilizando a ferramenta JaBUTi. No próximo capítulo será apresentado um exemplo de uso utilizando uma aplicação *web* desenvolvida no Centro de Pesquisa em Engenharia de Sistemas da PUCRS, onde as ferramentas derivadas da PLeTs serão utilizadas para automatizar a geração e execução de *scripts* e cenários de teste para essa aplicação.

```
<JABUTI>
    <PROJECT name="C:\Jabuti\JabutiProjects\Skills\PRJCT_SKILLS.jbt" type="research" mobility="N" (
        <BASE_CLASS name="TestDriver"/>
        <CLASSPATH path=". C?\Jabuti\bin C?\Temp\Workspace\CmTool_SkillsTest\web\WEB-INF\classes C
        <JUNIT_SRC_DIR dir=""/>
        <JUNIT_BIN_DIR dir=""/>
        <JUNIT_TEST_SET name=""/>
        <JUNIT_JAR name=""/>
        <AVOIDED_PACKAGES>
        </AVOIDED_PACKAGES>
        <CLASS name="TestDriver" size="0000" checksum="00000000">
            <EXTEND name="java.lang.Object" level="1"/>
        </CLASS>
        <INST_CLASS name="servlets.ServletPassword" size="4711" checksum="50-0">
            <EXTEND name="javax.servlet.http.HttpServlet" level="1"/>
            <SOURCE name=""/>
            <METHOD id="6" name="getServletInfo()Ljava/lang/String;">
                <All-Nodes-ei id="0" totreq="1" act="1" inf="0">
                    <NODE id="0" label="0" active="Y" covered="N" infeasible="N" effectivetcs=""/>
                </All-Nodes-ei>
                <All-Nodes-ed id="1" totreq="0" act="0" inf="0">
```

Figura 4.10: Arquivo de projeto da JaBUTi gerado pelo produto derivado da PLeTs

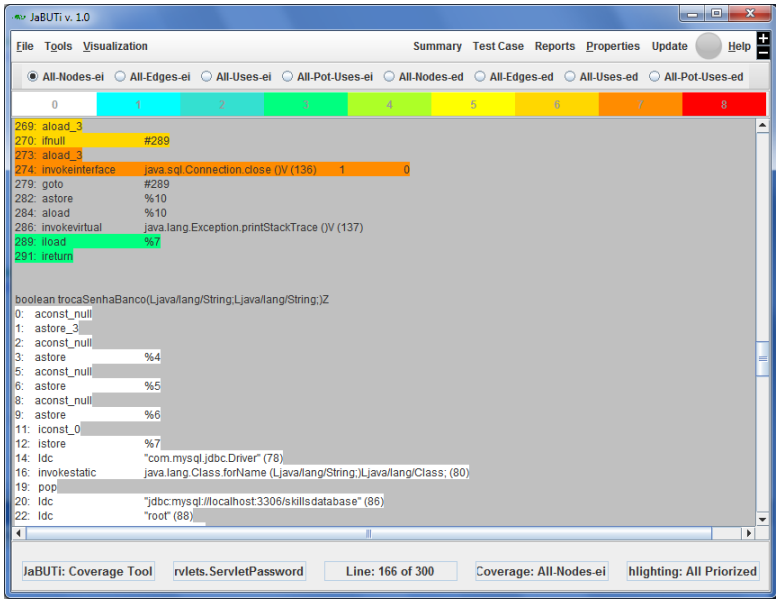

Figura 4.11: Interface da JaBUTi com informações de cobertura atualizadas

5. EXEMPLO DE USO

Neste capítulo será apresentada a definição de um conjunto de casos de teste para serem gerados e executados a partir dos produtos gerados pela ferramenta PLeTs. Para isto, será utilizada como exemplo de uso uma aplicação denominada Skills, a qual tem por objetivo a gerência de perfis profissionais de funcionários de uma empresa. O objetivo é verificar e validar os aspectos funcionais do conjunto de características apresentado no capítulo anterior, bem como, os *plugins* implementados para a PLeTs. Ao final, pretende-se realizar uma análise, apresentando as vantagens e desvantagens em automatizar casos de teste utilizando como referência o conjunto de características implementado.

5.1 Ferramenta a Ser Testada: Skills

O projeto Skills (*Workforce Planning: Skill Management Prototype Tool*) [32] consiste de uma aplicação que tem por objetivo gerenciar os perfis profissionais de funcionários de uma dada empresa. Esta aplicação foi desenvolvida por um grupo de pesquisa da PUCRS em colaboração com uma empresa de tecnologia e tem como principal funcionalidade o gerenciamento do cadastro de habilidades, certificações e experiências de funcionários. Este *software* utiliza o Sistema Gerenciador de Banco de Dados (SGBD) MySQL [98] para a persistência de dados e o TomCat [99] como servidor de aplicação. Para melhor compreensão, é apresentada na Figura 5.1 um exemplo referente ao cadastro de uma habilidade utilizando a interface da aplicação.

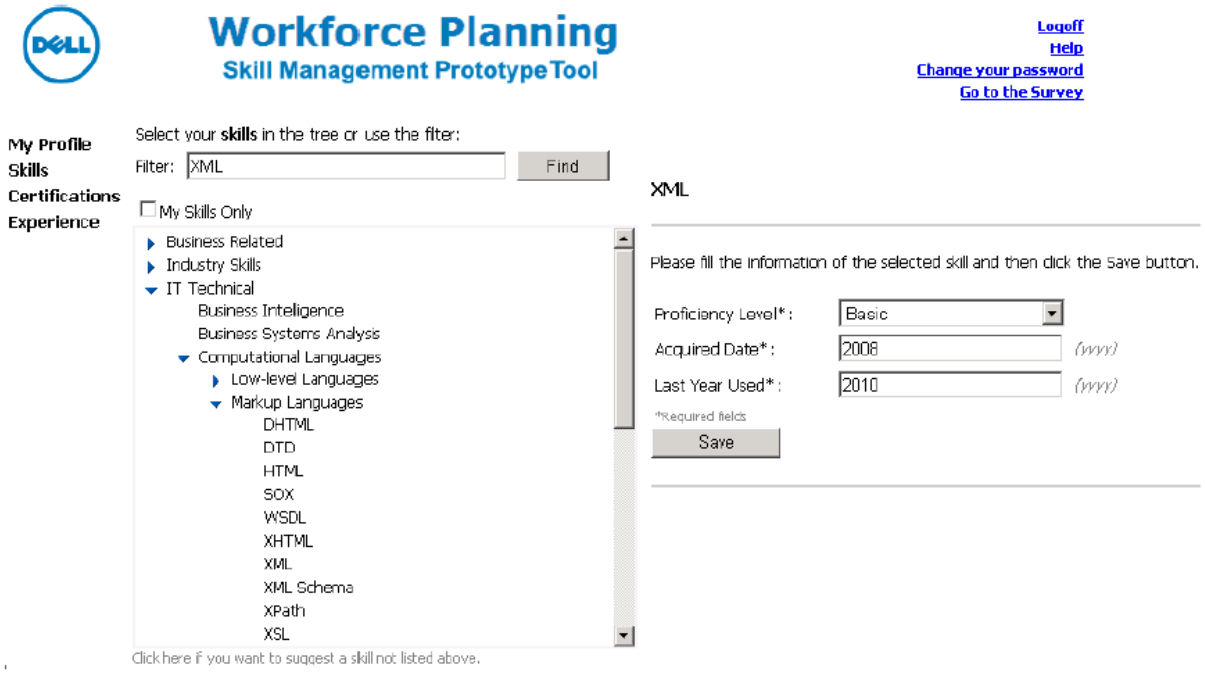

Figura 5.1: *Layout* referente à *interface* da aplicação Skills

No exemplo descrito na Figura 5.1, o usuário utiliza o campo "filter", localizado no canto superior esquerdo da interface da aplicação, para procurar a habilidade que deseja cadastrar, neste caso optou-se pelo cadastro da habilidade XML. Após encontrá-la, as informações referentes ao nível de proficiência e as datas de aquisição e última oportunidade de uso de tal habilidade são informadas no formulário localizado no canto direito da mesma interface. Em seguida, a habilidade é cadastrada e as informações são persistidas na base de dados do usuário.

Neste exemplo, o cadastro da habilidade XML foi realizado através da utilização do campo "filter", porém qualquer habilidade ou certificação pode ser cadastrada ou editada navegando-se nos itens da árvore até encontrar a habilidade/certificação desejada. O usuário ainda pode visualizar todas as habilidades, certificações ou experiências por ele cadastradas clicando no *link* "My Profile", localizado no canto superior esquerdo da interface da aplicação. O usuário também é capaz de trocar a sua senha de "login" e obter informações de ajuda, quando possuir alguma dificuldade durante a interação com as funcionalidades da aplicação.

5.2 Definição dos Casos de Teste Concretizados

Com o intuito de verificar os aspectos funcionais do conjunto de características e dos *plugins* implementados para a PLeTs, foram definidos alguns casos de teste para serem executados pelas duas ferramentas por ela geradas. Para isso, foram criados casos de teste utilizando a aplicação Skills como exemplo de uso. Neste contexto, foram definidos casos de teste de desempenho idênticos para serem executados pelas ferramentas LoadRunner e Visual Studio. O objetivo é mostrar o funcionamento dos *plugins* desenvolvidos para a PLeTs e o que muda de uma ferramenta para outra, durante a criação e configuração automática dos *scripts* e cenários de teste.

Os casos de teste definidos para os produtos descrevem o processo de cadastro e edição das informações de habilidades, certificações e experiências de um determinado funcionário de uma empresa. As informações referentes aos casos de teste são representados por diagramas UML (caso de uso e atividades). Na Figura 5.2 é apresentado um diagrama de casos de uso, o qual descreve as informações referentes a dois cenários de teste. Este diagrama apresenta dois atores e cada um define informações de um cenário de teste específico. O cenário representado pelo ator *Gerente RH* define, no comentário a ele conectado, um conjunto de informações necessárias para o teste, tais como:

- Quantidade de usuários (*TDpopulation*) = 50
- *Host* da aplicação (*TDhost*) = *localhost*
- Tempo de duração do teste (*TDtime*) = 02:00:00
- Tempo de rampa de subida (*TDrampUpTime*) = 00:01:00
- Quantidade de usuários da rampa de subida (*TDrampUpUser*) = 10

- Tempo de rampa de descida (*TDrampDownTime*) = 00:01:00
- Quantidade de usuários da rampa de descida (*TDrampDownUser*) = 10

Da mesma forma que o ator *Gerente RH* descreve um conjunto de informações configuradas para o cenário de teste, o mesmo se aplica ao ator *Empregado* o qual define as seguintes informações para o teste:

- Quantidade de usuários (*TDpopulation*) = 1000
- *Host* da aplicação (*TDhost*) = *localhost*
- Tempo de duração do teste (*TDtime*) = 04:00:00
- Tempo de rampa de subida (*TDrampUpTime*) = 00:10:00
- Quantidade de usuários da rampa de subida (*TDrampUpUser*) = 100
- Tempo de rampa de descida (*TDrampDownTime*) = 00:10:00
- Quantidade de usuários da rampa de descida (*TDrampDownUser*) = 100

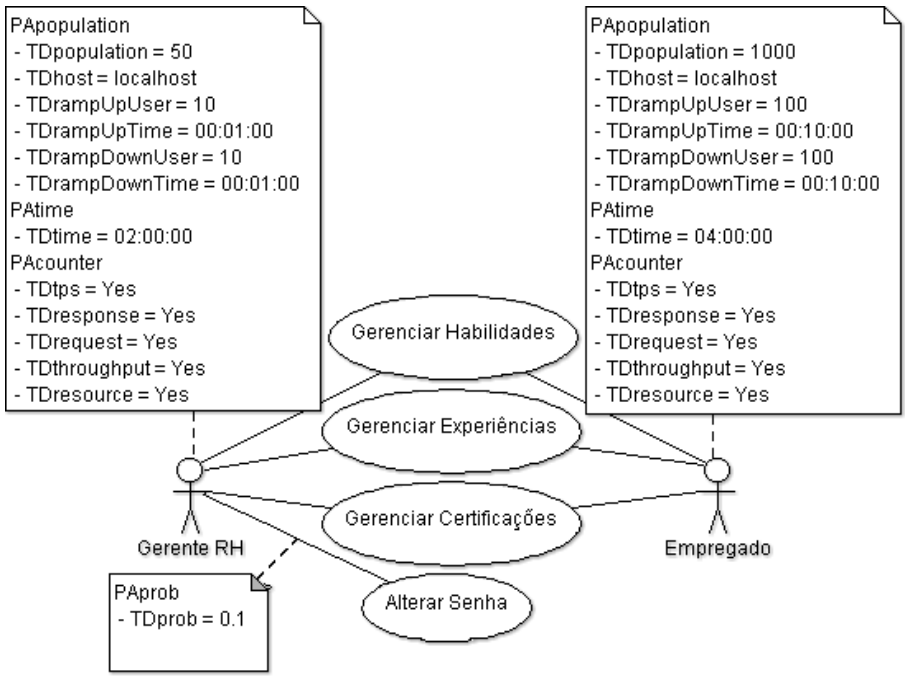

Figura 5.2: Diagrama de caso de uso da aplicação Skills

Além de definirem uma série de informações necessárias para o teste por meio de uma janela de comentários, os atores do diagrama da Figura 5.2 estão vinculados a um conjunto de casos de uso. Neste exemplo, o ator *Gerente RH* está conectado aos quatro casos de uso presentes no diagrama: *Gerenciar Habilidades*, *Gerenciar Experiências*, *Gerenciar Certificações* e *Alterar senha*. Por outro

lado, o ator *Empregado* está conectado somente a três casos de uso. Por questões relativas a permissões de acesso à aplicação, ele não possui acesso à funcionalidade trocar senha e, por este motivo, não está conectado ao caso de uso que descreve tal funcionalidade.

Os casos de uso possuem informações referentes ao perfil de tarefa a ser realizada pelos usuários sob a aplicação a ser testada. Por exemplo, o caso de uso *Gerenciar Habilidades* determina que o conjunto de usuários definidos para ele irá interagir com a aplicação apenas no que se referir ao cadastro ou edição de habilidades. O mesmo é válido para os demais casos de uso. Cada caso de uso também possui uma probabilidade vinculada a um ator, este atributo é utilizado para determinar a quantidade de usuários que irá realizar, sob a aplicação, o perfil de tarefa descrito por um caso de uso específico. Entre o ator *Gerente RH* e o caso de uso *Gerenciar Habilidades*, por exemplo, foi atribuído um valor de probabilidade igual a 40%. Informando, portanto, que 20 dos 50 usuários definidos para o cenário representado pelo ator *Gerente RH* irá interagir com a aplicação e realizar somente tarefas de gerência de habilidades. Também foram atribuídos valores de probabilidade para os demais casos de uso e seus respectivos atores (ver Tabela 5.1).

Tabela 5.1: Probabilidade dos casos de uso

Ator	Probabilidade	Caso de Uso
Gerente RH	40%	Gerenciar Habilidades
	30 %	Gerenciar Certificações
	20 %	Gerenciar Experiências
	10 %	Alterar Senhas
Empregado	40%	Gerenciar Habilidades
	35 %	Gerenciar Certificações
	25 %	Gerenciar Experiências

Cada caso de uso é decomposto em um diagrama de atividades e o conjunto dos quatro diagramas de atividades deste exemplo, os quais podem ser visualizados nas Figuras 5.3, 5.4, 5.5 e 5.6, determina o possível comportamento dos usuários sob a aplicação. Em outras palavras, os diagramas de atividades descrevem os aspectos dinâmicos do sistema sob teste. A seguir, são descritas as etapas que representam o fluxo de atividades da aplicação:

- *Logar*: nesta etapa o usuário insere seus dados de autenticação (usuário e senha) para logar-se na aplicação. Em seguida é redirecionado para a página principal da aplicação, onde são mostradas as habilidades, certificações e experiências cadastradas;

- *Cadastrar habilidade - modo 1*: esta etapa inicia com o usuário clicando no *link Skills*. Com isso, uma lista de tipos de habilidades é apresentada ao usuário. Este navega entre os itens da lista até encontrar a habilidade que deseja cadastrar (*Árvore de Habilidades*). Ao encontrar a habilidade procurada o usuário irá selecioná-la e com isso, um formulário é apresentado. O usuário insere dados referentes ao nível e data de quando adquiriu e última vez que fez uso de tal habilidade, finalizando assim o cadastro;

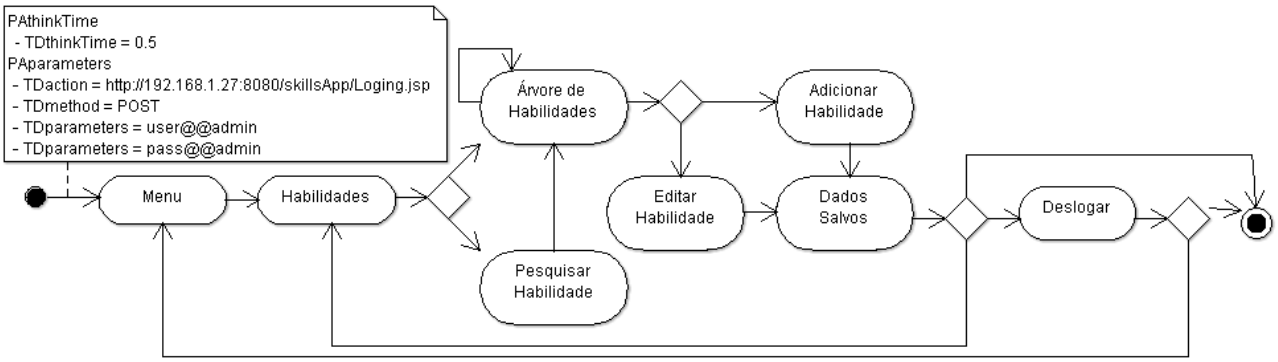

Figura 5.3: Diagrama de atividades para gerência de habilidades

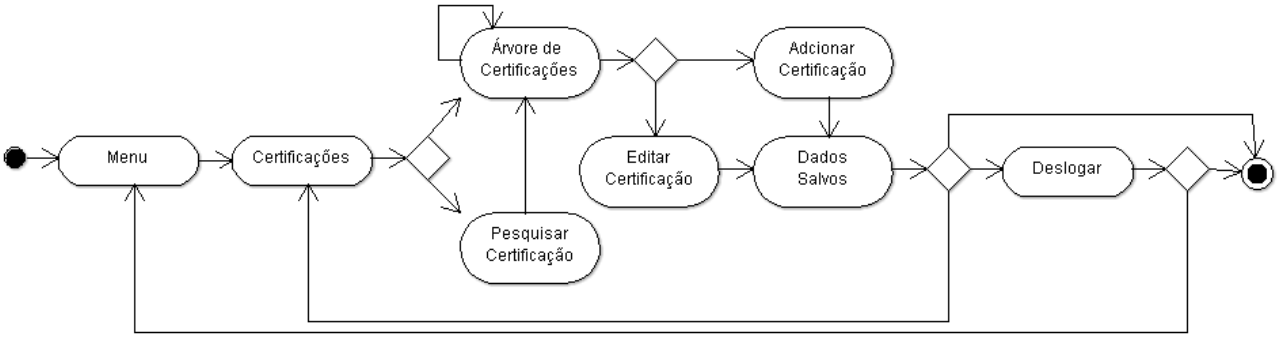

Figura 5.4: Diagrama de atividades para gerência de certificações

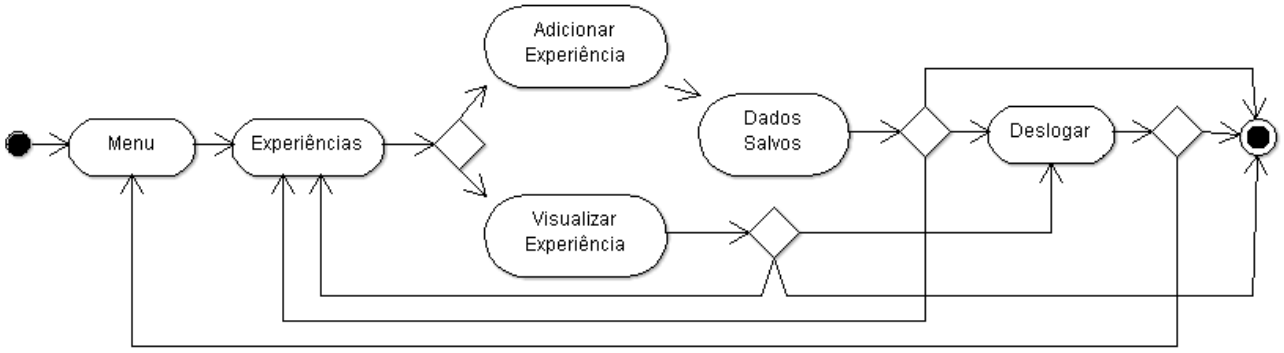

Figura 5.5: Diagrama de atividades para gerência de experiências

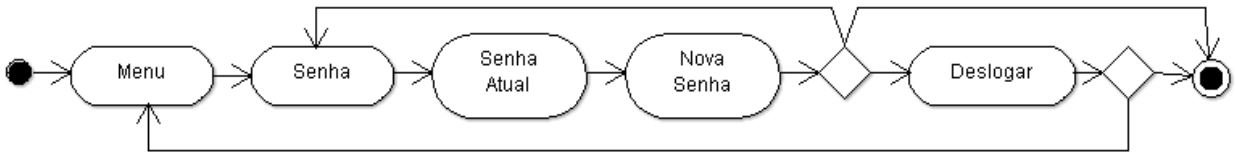

Figura 5.6: Diagrama de atividades para alteração de senha

- *Cadastrar habilidade - modo 2*: esta etapa inicia com o usuário clicando no *link Skills*. Com isso, uma lista de tipos de habilidades é apresentada ao usuário. Este preenche o campo "filter" com o nome da habilidade a ser cadastrada (*Pesquisar Habilidade*). Em seguida, a habilidade pesquisada é mostrada ao usuário que por sua vez a seleciona e com isso, um formulário é apresentado. O usuário insere dados referentes ao nível e data de quando adquiriu e última

vez que fez uso de tal habilidade, finalizando assim o cadastro;

- *Editar habilidade - modo 1*: esta etapa inicia com o usuário clicando no *link Skills*. Com isso, uma lista de tipos de habilidades é apresentada ao usuário. Este navega entre os itens da lista até encontrar a habilidade que deseja editar. Ao encontrar a habilidade procurada o usuário irá selecioná-la e com isso, um formulário é apresentado. O usuário edita os dados referentes ao nível e data de quando adquiriu e última vez que fez uso de tal habilidade, finalizando assim edição das informações;

- *Editar habilidade - modo 2*: esta etapa inicia com o usuário clicando no *link Skills*. Com isso, uma lista de tipos de habilidades é apresentada ao usuário. Este preenche o campo "filter" com o nome da habilidade a ser editada. Em seguida, a habilidade pesquisada é mostrada ao usuário que por sua vez a seleciona e com isso, um formulário é apresentado. O usuário edita os dados referentes ao nível e data de quando adquiriu e última vez que fez uso de tal habilidade, finalizando assim edição das informações;

- *Cadastrar certificação - modo 1*: esta etapa inicia com o usuário clicando no *link Certifications*. Com isso, uma lista com diversas certificações é apresentada ao usuário. Este navega entre os itens da lista até encontrar a certificação que deseja cadastrar (*Árvore de Certificações*). Ao encontrar a certificação procurada o usuário irá selecioná-la e com isso, um formulário é apresentado. O usuário insere dados referentes ao nível e data de quando adquiriu e última vez que fez uso de tal certificação, finalizando assim o cadastro;

- *Cadastrar certificação - modo 2*: esta etapa inicia com o usuário clicando no *link Certifications*. Com isso, uma lista com diversas certificações é apresentada ao usuário. Este preenche o campo "filter" com o nome da certificação a ser cadastrada (*Pesquisar Certificação*). Em seguida, a certificação pesquisada é mostrada ao usuário que por sua vez a seleciona e com isso, um formulário é apresentado. O usuário insere dados referentes ao nível e data de quando adquiriu e última vez que fez uso de tal certificação, finalizando assim o cadastro;

- *Editar certificação - modo 1*: esta etapa inicia com o usuário clicando no *link Certifications*. Com isso, uma lista de diversas certificações é apresentada ao usuário. Este navega entre os itens da lista até encontrar a certificação que deseja editar. Ao encontrar a certificação procurada o usuário irá selecioná-la e com isso, um formulário é apresentado. O usuário edita os dados referentes ao nível e data de quando adquiriu e última vez que fez uso de tal certificação, finalizando assim edição das informações;

- *Editar certificação - modo 2*: esta etapa inicia com o usuário clicando no *link Certifications*. Com isso, uma lista de diversas certificações é apresentada ao usuário. Este preenche o campo "filter" com o nome da certificação a ser editada. Em seguida, a certificação pesquisada é mostrada ao usuário que por sua vez a seleciona e com isso, um formulário é apresentado. O

usuário edita os dados referentes ao nível e data de quando adquiriu e última vez que fez uso de tal certificação, finalizando assim edição das informações;

- *Cadastrar experiência*: esta etapa é muito semelhante ao cadastro de uma habilidade ou certificação. O usuário clica no *link Experience* e uma lista de experiências já cadastradas é apresentada. O usuário tem a opção de editar ou cadastrar uma nova experiência. Neste caso, o usuário opta pelo cadastro de uma nova experiência e insere, no formulário apresentado, dados referentes à companhia, área de atuação, função, nível e período em que adquiriu a experiência;

- *Editar experiência*: esta etapa é muito semelhante edição de uma habilidade ou certificação. O usuário clica no *link Experience* e uma lista de experiências já cadastradas é apresentada. O usuário tem a opção de editar ou cadastrar uma nova experiência. Neste caso, o usuário opta pelo edição de uma experiência já cadastrada e atualiza, no formulário apresentado, dados referentes à companhia, área de atuação, função, nível e período em que adquiriu a experiência;

- *Alterar senha*: o usuário clica no *link Change your password* com isso, um formulário para preenchimento de informações relativas à senha atual e nova é apresentado ao usuário. Ao final a senha é modificada e o usuário é redirecionado ao menu principal da aplicação;

- *Deslogar*: o usuário finaliza seu acesso à aplicação e é redirecionado à página inicial da aplicação.

Assim como os diagramas de casos de uso são fundamentais para a definição dos casos de teste, o mesmo é válido para os diagramas de atividades. Entretanto, diferentemente dos diagramas de caso de uso, os quais definem informações referentes aos cenários de teste, os diagramas de atividades apresentam informações para posterior geração dos *scripts* de teste. Como é possível visualizar na Figura 5.3, a primeira transição do diagrama possui informações para a realização de *login* na aplicação, tais como: tempo de pensamento (*0.5* segundos), página da aplicação (*login.pl*) e parâmetros para inserção de informações relativas a nome de usuário (*admin*) e senha (*admin*). Para as demais transições dos quatro diagramas também foram definidas tais informações, porém com o objetivo de não poluir os diagramas, apenas a primeira transição do diagrama de atividades referente à gerência de habilidades foi ilustrada e utilizada como exemplo.

Com a definição dos casos de teste por meio da modelagem de diagramas de caso de uso e atividades, a geração e execução dos testes são então iniciadas. Em um primeiro momento, optou-se pelo produto que utiliza o LoadRunner para automatizar a geração dos *scripts* e cenários de teste e posteriormente o produto que utiliza o Visual Studio. Durante a execução da ferramenta, as informações referentes aos casos de teste definidos nos diagramas de caso de uso e atividades são exportadas para um arquivo no formato "*XMI*". Em seguida, este arquivo é submetido a um *parser*, o qual armazena as informações deste arquivo na estrutura intermediária. A próxima etapa consiste

na geração dos casos de teste abstratos, onde uma FSM é gerada com as informações referentes às interações do usuário com a aplicação. Posteriormente, o método HSI para geração de sequências de teste é aplicado sob a FSM. A próxima etapa consiste em armazenar as informações das sequências de teste geradas e as informações dos cenários de teste presentes na estrutura intermediária na estrutura de sequência de teste.

Em seguida, uma interface é apresentada ao usuário, o qual tem opção de selecionar o cenário e as sequências de teste que serão utilizadas para gerar os *scripts* de teste. Neste exemplo, foram selecionados o cenário representado pelo ator Gerente RH e quatro *scripts* representados pelo diagrama de atividades para gerência de habilidades. Após a definição do cenário e dos *scripts* cada produto derivado da PLeTs cria uma instância para uma ferramenta específica. Sendo que um produto instancia a interface do LoadRunner e o outro instancia a interface do Visual Studio, onde ambos executam automaticamente os casos de teste definidos anteriormente. Para exemplificar, as Figuras 5.7 e 5.8 apresentam, respectivamente, as interfaces do LoadRunner e Visual Studio instanciadas pelos produtos derivados da PLeTs. Ambas as figuras apresentam um cenário de teste programado para executar por um período de duas horas, onde os *scripts Cadastrar habilidade - modo 1*, *Cadastrar habilidade - modo 2*, *Editar habilidade - modo 1*, *Editar habilidade - modo 2* são utilizados para a interação dos usuários com a aplicação durante o teste.

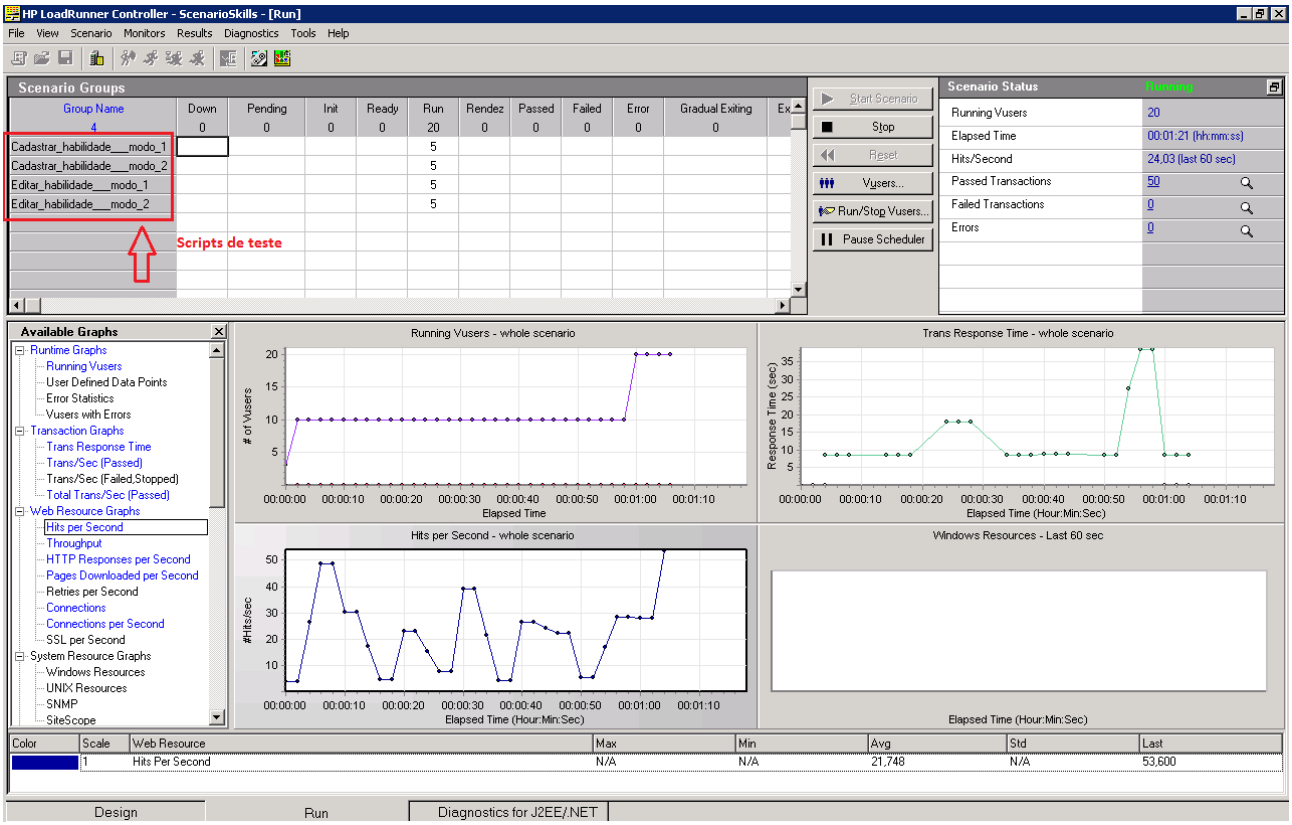

Figura 5.7: Interface do LoadRunner para a execução do teste

Devido à complexidade dos casos de teste criados, observou-se algumas diferenças na configuração dos casos de teste entre os dois produtos gerados pela PLeTs. Algumas informações definidas

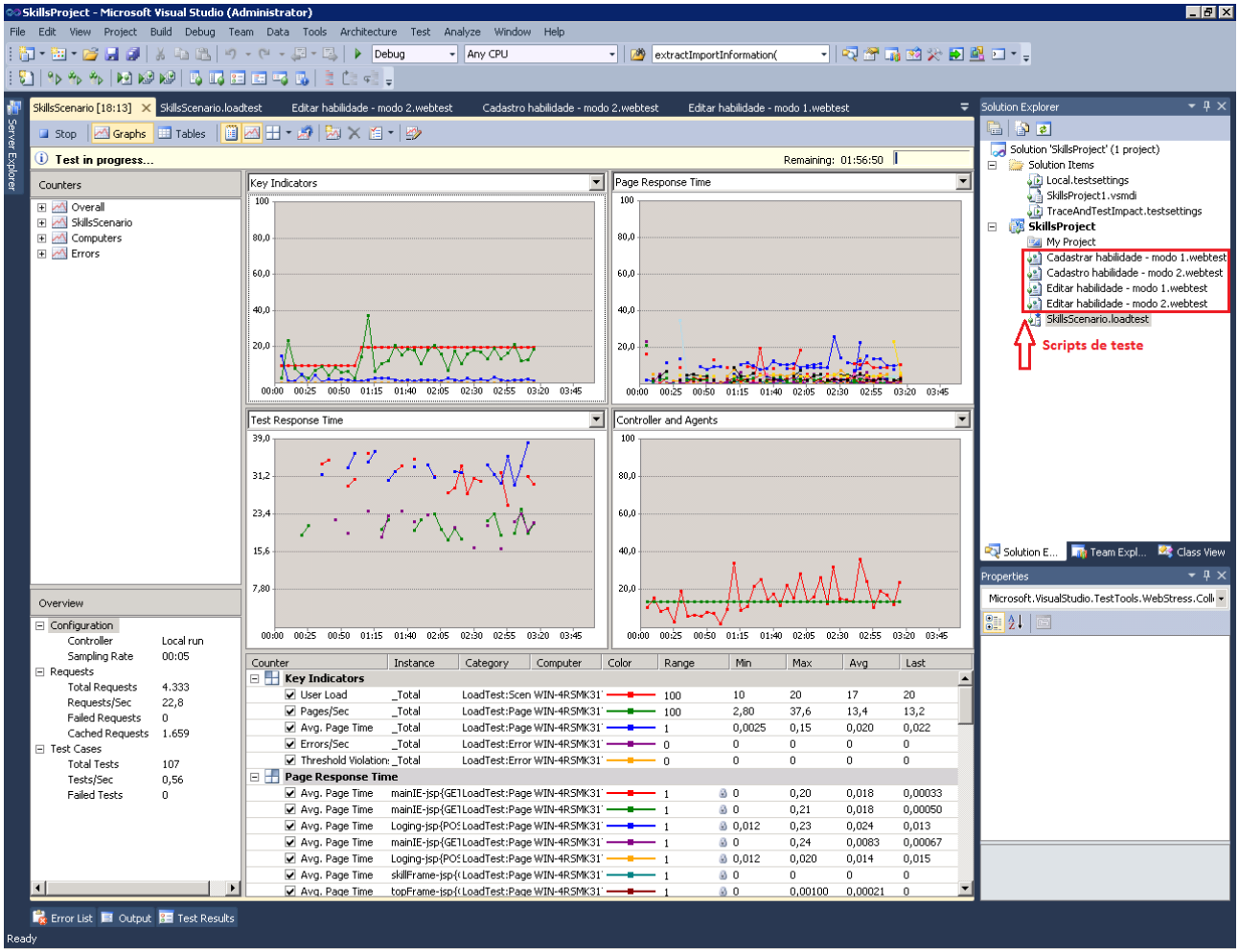

Figura 5.8: Interface do Visual Studio a para execução do teste

no diagrama de caso de uso, tais como, Tempo de rampa de descida e Quantidade de usuários da rampa de descida não foram aproveitadas para a geração do cenário de teste do Visual Studio. Isto se deve ao fato de que a ferramenta de teste da Microsoft não possui tais funcionalidades. Também se observou que o Visual Studio é capaz de configurar o parâmetro chamado *Tempo de Aquecimento* (período de tempo em que o Visual Studio irá coletar informações referentes aos contadores configurados para o teste sem que seja gerada carga na aplicação). Entretanto, esta informação não foi incluída nos diagramas UML. Isto ocorreu devido ao fato desta informação não ter sido identificada no conjunto de características descrito na pesquisa de [29]. Entretanto, o parâmetro referente a esta informação foi identificado no conjunto de características desta dissertação e por este motivo foi incluído na estrutura de sequência de teste. Contudo, a configuração desta informação não é proveniente dos diagramas UML, mas sim realizada por meio da edição de um arquivo de texto, o qual foi criado para armazenar informações específicas de ferramentas e que não puderam ser incluídas nos diagramas UML. Também é importante salientar que os contadores utilizados para verificar e medir o comportamento do ambiente não foram definidos nos diagramas UML. No entanto, foram utilizados para o teste os contadores que as ferramentas LoadRunner e Visual Studio definem por padrão.

Com a definição destes casos de teste para os produtos derivados da PLeTs, foi possível verificar os aspectos funcionais do conjunto de características e os *plugins* desenvolvidos. E, portanto, demonstrar que é possível automatizar casos de teste baseando-se nas informações contidas no conjunto de características implementado.

5.3 Discussão

Tendo apresentado as etapas do processo de criação do conjunto de características para ferramentas de teste de desempenho e a implementação dos *plugins* com base nas informações contidas neste conjunto, o objetivo para esta seção é apresentar uma discussão referente às vantagens e desvantagens em automatizar casos de teste utilizando como referência o conjunto de características implementado.

Com base nas experiências obtidas com a utilização das ferramentas LoadRunner, Visual Studio foi possível estimar aproximadamente o tempo gasto na criação e execução manuais de casos de teste para cada uma dessas ferramentas. Também é correto afirmar que, pelo fato das ferramentas para teste de desempenho da HP e Microsoft possuírem interfaces e características relativamente parecidas, não possuem diferença significativa no tempo gasto para a criação e execução de seus casos de teste.

O tempo gasto na implementação dos *plugins* para as ferramentas LoadRunner e Visual Studio também foi aproximadamente o mesmo. Isso ocorreu, principalmente, devido à semelhança entre elas no que diz respeito à complexidade na forma de estruturar os arquivos e dados necessários para a configuração automática de teste. Apesar do tempo gasto na criação dos *plugins* ser consideravelmente superior quando comparado ao tempo gasto na geração e execução de casos de teste utilizando a abordagem manual, o tempo gasto utilizando a abordagem automática é, por outro lado, muito inferior à abordagem manual.

Com base nestas informações as seguintes considerações podem ser feitas: quando deseja-se automatizar o processo de geração e execução de casos de teste concretizados, inicialmente, o custo de tempo envolvido na implementação de *plugins* para uma determinada ferramenta será consideravelmente superior se comparado ao tempo gasto na execução utilizando somente a abordagem manual. Entretanto, pelo fato da abordagem automática ser consideravelmente mais rápida que a abordagem manual, o tempo gasto na implementação dos *plugins* pode se tornar irrelevante à medida que uma quantidade maior de casos de teste irão sendo definidos para serem executados.

Desta forma, ao longo do tempo a abordagem automática pode se tornar uma alternativa mais atrativa que a geração e execução de casos de teste utilizando a abordagem manual. Outra vantagem na utilização da abordagem automática é a possibilidade de criar e executar casos de teste sem a necessidade de um conhecimento técnico aprofundado da ferramenta de automação de teste. Outra vantagem em utilizar a abordagem automática, diz respeito à documentação das informações dos casos de teste. Visto que as informações referentes ao teste são definidas nos diagramas UML da aplicação, é mais fácil guardar documentos que descrevem o modelo de uma aplicação do que os

arquivos de teste de determinada tecnologia. Por essas razões, a geração e execução automática de casos de teste concretizados torna-se uma alternativa interessante.

5.4 Considerações

Este capítulo apresentou a definição de um conjunto de casos de teste para serem executados pelos produtos derivados da PLeTs. Estes produtos foram gerados com base nos *plugins* implementados, os quais se baseiam nas informações contidas no conjunto de características para ferramentas de teste de desempenho descrito no Capítulo 4.

Ao final foi apresentada uma análise referente às vantagens e desvantagens de gerar e executar casos de teste utilizando a abordagem automática (geração e execução de *scripts* e cenários de teste com os produtos derivados da PLeTs) e a abordagem manual (método convencional de teste por meio da geração e configuração manual de *scripts* e cenários de teste).

6. CONCLUSÃO

6.1 Resumo

Este trabalho apresentou a implementação de um conjunto de características para geração e execução de casos de teste concretizados para ferramentas de teste de desempenho em aplicações *web*. Para isso, destacam-se duas etapas como sendo as principais para efetivação do trabalho. Inicialmente, foi realizado uma análise de diversos trabalhos que descrevem o processo de criação e execução de *scripts* e cenários de teste utilizando ferramentas para automatização de teste de desempenho. Com base nas informações adquiridas dos trabalhos estudados, pôde-se fazer um levantamento das características necessárias para a criação de casos de teste concretizados. Para a efetivação do conjunto, foi adotada uma classificação para essas características, com base no processo de geração de casos de teste de desempenho para aplicações *web* utilizado pela Microsoft. A utilização deste processo serviu para complementar as informações referentes às características e, com isso, definir o conjunto de características apresentado nesta dissertação.

A partir das informações do conjunto apresentado, foi realizada a implementação de *plugins* para as etapas *Script Generation* e *Execution* da PLeTs. Estes *plugins* implementam a geração e execução automática de *scripts* e cenários de teste utilizando duas ferramentas de automatização de teste de desempenho, sendo elas, HP LoadRunner e Microsoft Visual Studio. O objetivo foi utilizar as informações contidas no conjunto para automatizar a geração e execução de casos de teste concretizados. Também foi realizada a implementação de um *plugin* para a geração e execução de casos de teste estrutural utilizando a ferramenta JaBUTi e, foi definido um conjunto com informações específicas para a ferramenta de teste estrutural.

Para complementação do trabalho foram criados casos de teste para as ferramentas de teste de desempenho LoadRunner e Visual Studio, onde a aplicação Skills foi utilizada como exemplo de uso. O objetivo foi verificar as diferenças de configuração existentes durante a criação e execução de casos de teste concretizados para as duas ferramentas. Ao final, foi apresentada uma discussão referente às vantagens e desvantagens em automatizar a geração e execução de casos de teste concretizados. Foram mostrados os custos e os ganhos obtidos com esta automatização quando comparada com a abordagem de geração e execução de casos de teste manuais.

6.2 Contribuição e Trabalhos Futuros

A pesquisa que originou o trabalho apresentado nesta dissertação teve início no estudo dos conceitos relacionados ao teste de *software* e na análise de diversas ferramentas de automatização de teste. Após uma visão geral, a pesquisa focou no teste de desempenho e, neste contexto, fez-se necessária a análise de um conjunto de ferramentas que automatizam este tipo de teste. A etapa seguinte da pesquisa se baseou na implementação de um conjunto de características para ferramentas

de teste de desempenho. O objetivo principal foi implementar um conjunto que auxiliasse a geração e execução automática de *scripts* e cenários de teste de desempenho para diversas ferramentas, pois ainda que as ferramentas de teste existentes automatizem a execução de testes, a geração e criação destes *scripts* e cenários continuam sendo realizadas de forma manual e por este motivo propensa a falhas. Desta forma, sempre que alguém desejar implementar uma ferramenta para automatizar a geração destes *scripts* e cenários, poderá fazê-lo por meio da análise das informações contidas no conjunto de características para ferramentas de teste de desempenho definido na pesquisa desta dissertação.

Como descrito anteriormente, o conjunto de características foi utilizado para a implementação de *plugins* para a PLeTs, onde as informações referentes ao teste são configuradas em modelos UML, tais como, diagramas de casos de uso e atividades. Entretanto, este foi apenas um exemplo onde as informações do conjunto implementado foram aplicadas. Poderia, por exemplo, implementar uma ferramenta onde as informações estivessem, inicialmente, descritas em *logs* de uma aplicação. Desta forma, as informações do teste poderiam ser diretamente extraídas.

Apesar das vantagens adquiridas com a utilização das informações contidas no conjunto de características, existem algumas questões em aberto e pontos que poderiam ser aprimorados. Atualmente, ainda que o conjunto disponibilize um conjunto de informações necessárias para automatizar a geração de *scripts* e cenários, é necessário um conhecimento aprofundado relacionado à ferramenta de automatização de teste que se deseja utilizar. É necessário conhecer os arquivos de configuração de teste da ferramenta e como as informações para o teste estão estruturadas nestes arquivos. Outra questão em aberto diz respeito à falta de informações de análise de resultados do teste no conjunto de características. Atualmente, o conjunto de características apresenta informações necessárias para a geração e configuração do teste, mas não dispõe de informações que podem contribuir na automatização da análise dos resultados. Por exemplo, o conjunto poderia descrever quais as informações do teste são mais relevantes de serem consideradas para análise e qual relação entre esta ou aquela informação pode contribuir em uma análise mais criteriosa do teste.

Os *plugins* desenvolvidos para a PLeTs automatizam a geração e execução de testes para ferramentas, no entanto, estes *plugins* não implementam nenhuma função de análise. Neste contexto, seria interessante a PLeTs disponibilizar *plugins* que implementassem funções que agissem como oráculos, apresentando informações do teste relativas ao cumprimento ou não dos requisitos de desempenho especificados.

REFERÊNCIAS

[1] T. S. Weber, "Tolerância a Falhas: Conceitos e Exemplos," Capturado em: http://www.inf.ufrgs.br/~taisy/disciplinas/textos/ConceitosDependabilidade.pdf, Outubro 2011.

[2] P. van den Broek, "Extended Feature Models," Capturado em: http://www.utwente.nl/ewi/trese/b_referaat/broek1.docx, Dezembro 2011.

[3] E. M. Rodrigues, L. D. Viccari, A. F. Zorzo, and I. M. Gimenes, "PLeTs Tool - Test Automation Using Software Product Lines and Model Based Testing," in *Proceedings of the 22th International Conference on Software Engineering and Knowledge Engineering*, 2010, pp. 483–488.

[4] S. F. A. JMeter, "Apache JMeter User's Manual," Capturado em: http://jmeter.apache.org/usermanual/index.html, Outubro 2011.

[5] I. C. S. Group, "IBM Rational Functional Tester," Capturado em: ftp://aix.boulder.ibm.com/software/emea/de/rational/RAD14072USEN.pdf, Agosto 2010.

[6] JUnit, "Tutorial JUnit," Capturado em: http://www.junit.org, Maio 2010.

[7] I. C. Mercury, "Mercury QuickTest Professional Tutorial," Capturado em: http://thinh1808.files.wordpress.com/2008/03/qtp-tutorial.pdf, Setembro 2010.

[8] ——, "Mercury LoadRunner Tutorial," Capturado em: http://qageek.files.wordpress.com/2007/05/loadrunner_tutorial.pdf, Abril 2011.

[9] D. Hunt, P. Grandjean, S. S. Ordonez, T. Kumar, M. A. May-Pumphrey, and P. Newhook, "Selenium Documentation," Capturado em: http://seleniumhq.org/docs/05_selenium_rc.html, Abril 2010.

[10] A. M. R. Vincenzi, M. E. Delamaro, and J. C. Maldonado, "JaBUTi Java Bytecode Understanding and Testing," Instituto de Ciências Matemáticas e de Computação, USP, Tech. Rep., 2009.

[11] K. Gallagher and B. Shea, *Annual Load Test Market Summary and Analysis*. Newport Group, 2001.

[12] M. Pezzè and M. Young, *Teste e Análise de Software - Processos, Princípios e Técnicas*. John Wiley & Sons, 2008.

[13] A. Romanovsky, P. Periorellis, and A. F. Zorzo, "Structuring Integrated Web Applications for Fault Tolerance," in *Proceedings of the 6th International Symposium on Autonomous Decentralized Systems*, 2003, pp. 99–106.

[14] A. F. Zorzo, P. Periorellis, and A. Romanovsky, "Using Co-ordinated Atomic Actions for Building Complex Web Applications: a Learning Experience," in *Proceedings of the 8th International Workshop on Object-Oriented Real-Time Dependable Systems*, 2003, pp. 288–295.

[15] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transaction on Dependable Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan 2004.

[16] E. Halili, *Apache JMeter*. Packt Publishing, 2008.

[17] C. Davis, D. Chirillo, D. Gouveia, F. Saracevic, J. B. Bocarsley, L. Quesada, L. B. Thomas, and M. v. Lint, *Software Test Engineering with IBM Rational Functional Tester: The Definitive Resource*. IBM Press, 2009.

[18] C. H. Huang and H. Y. Chen, "A Semi-automatic Generator for Unit Testing Code Files Based on JUnit," in *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, 2005, pp. 140–145.

[19] S. R. Mallepally, *QuickTest Professional (QTP) Interview Questions and Guidelines: A Quick Reference Guide to QuickTest Professional*. Parishta, 2009.

[20] R. Zheng, H. Wang, and Y. Pang, "Research on Bio-inspired Multi-net Paralleling Mechanism Based on Web Application," in *Proceedings of the 7th International Conference on Computational Science*, 2007, pp. 117–120.

[21] D. Chadwick, A. Patel, J. Reinstrom, K. Siefkes, P. Silva, S. Ulrich, W. Yeung, C. Davis, M. Dunn, E. Jessee, A. Kofaldt, K. Mooney, and R. Nicolas, *Using Rational Performance Tester Version 7*. IBM Corporation, 2008.

[22] A. Holmes and M. Kellogg, "Automating Functional Tests Using Selenium," in *Proceedings of the 9th International Conference on Agile*, 2006, pp. 270–275.

[23] T. Arnold, D. Hopton, A. Leonard, and M. Frost, *Professional Software Testing with Visual Studio 2005 Team System: Tools for Software Developers and Test Engineers*. Wrox Press, 2007.

[24] M. M. Eler, A. T. Endo, P. C. Masiero, M. E. Delamaro, J. C. Maldonado, A. M. R. Vincenzi, M. L. Chaim, and D. M. Beder, "JaBUTiService: A Web Service for Structural Testing of Java Programs," in *Proceedings of the 33rd IEEE International on Software Engineering Workshop*, 2009, pp. 69–76.

[25] J. Meier, S. Vasireddy, A. Babbar, and A. Mackman, *Improving. NET Application Performance and Scalability*. Microsoft Press, 2004.

[26] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen, "WSDL-based Automatic Test Case Generation for Web Services Testing," in *Proceedings of the IEEE International of Service-Oriented System Engineering Workshop*, 2005, pp. 207–212.

[27] B. M. Subraya and S. V. Subrahmanya, "Object Driven Performance Testing in Web Applications," in *Proceedings of the 1st Asia-Pacific Conference on Quality Software*, 2000, pp. 17–26.

[28] D. Hao, Y. Chen, F. Tang, and F. Qi, "Distributed agent-based performance testing framework on Web Services," in *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences*, 2010, pp. 90–94.

[29] M. B. da Silveira, "Conjunto de Características para Teste de Desempenho: Uma Visão a partir de Modelos," Master's thesis, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2012.

[30] M. R. Blaha and J. R. Rumbaugh, *Object-Oriented Modeling and Design with UML*. Prentice Hall, 2005.

[31] P. Krishnan, "Uniform Descriptions for Model Based Testing," in *Proceedings of the Australian Software Engineering Conference*, 2004, pp. 96–105.

[32] M. B. Silveira, E. M. Rodrigues, A. F. Zorzo, L. T. Costa, H. V. Vieira, and F. M. de Oliveira, "Generation of Scripts for Performance Testing Based on UML Models," in *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*, 2011, pp. 258–563.

[33] C. Meadows and J. McLean, "Security and dependability: then and now," in *Proceedings of the Computer Security, Dependability and Assurance: From Needs to Solutions*, 1998, pp. 166–170.

[34] M. E. Delamaro, J. C. Maldonado, and M. Jino, *Introdução ao Teste de Software*. Elsevier Editora, 2007.

[35] L. D. Viccari, "Automação de Teste de Software Através de Linhas de Produtos e Teste Baseados em Modelos," Master's thesis, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2009.

[36] R. S. Pressman, *Engenharia de Software*. McGraw-Hill, 2006.

[37] L. C. Ascari, "Teste Baseado em Defeitos de Classes Java Utilizando Aspéctos e Mutação de Especificações OCL," Master's thesis, Programa de Pós-Graduação em Informática, UFPR, 2009.

[38] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, Dez 1976.

[39] L. H. Survey, "Applied Performance Management Survey," *Compuware*, vol. 65, no. 6, pp. 644–646, Jun 2006.

[40] M. Woodside, G. Franks, and D. C. Petriu, "The Future of Software Performance Engineering," in *Proceedings of the Future of Software Engineering*, 2007, pp. 171–187.

[41] C. U. Smith and L. G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley Professional, 2002.

[42] I. Molyneaux, *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, 2009.

[43] K. Zhu, J. Fu, and Y. Li, "Research the Performance Testing and Performance Improvement Strategy in Web Application," in *Proceedings of the 2nd International Conference on Education Technology and Computer*, 2010, pp. 328–332.

[44] J. Meier, C. Farre, P. Bansode, S. Barber, and D. Rea, *Performance Testing Guidance for Web Applications: Patterns & Practices*. Microsoft Press, 2007.

[45] B. Software, "Performance Benchmarking Kit Using Incident Management with SilkPerformer," BMC Software, Tech. Rep., 2007.

[46] C. S. Horstmann, *Big Java: Programming and Practice*. John Wiley & Sons, 2001.

[47] T. Nash, *Accelerated C# 2010*. Apress, 2010.

[48] L. Wall, *Programming Perl*. O'Reilly & Associates, 2000.

[49] R. J. Lerdorf, K. Tatroe, B. Kaehms, and R. McGredy, *Programming Php*. O'Reilly & Associates, 2002.

[50] M. Lutz, *Programming Python*. O'Reilly Media, 2006.

[51] D. Thomas and A. Hunt, *Programming Ruby: the Pragmatic Programmer's Guide*. Addison-Wesley Longman Publishing, 2000.

[52] T. J. Bergin and R. G. Gibson, *History of Programming Languages II*. ACM Press, 1996.

[53] N. Brenner, "Visual Basic .NET: one Teacher's Experience," *Journal of Computing Sciences in Colleges*, vol. 21, no. 2, pp. 89–94, Dez 2005.

[54] G. Concas, M. Marchesi, A. Cau, S. Pinna, K. Mannaro, and N. Serra, "XMI for XP project data interchange," in *Proceedings of the Workshop on Quantitative Techniques for Software Agile Process*, 2004, pp. 53–58.

[55] B. Stroustrup, "A History of C++: 1979–1991," in *Proceedings of the 2nd Conference on History of Programming Languages*, 1993, pp. 271–297.

[56] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing, 2001.

[57] Y. Jing, Z. Lan, W. Hongyuan, S. Yuqiang, and C. Guizhen, "JMeter-based Aging Simulation of Computing System," in *Proceedings of the International Conference on Computer, Mechatronics, Control and Electronic Engineering*, 2010, pp. 282–285.

[58] A. Keshk and A. Ibrahim, "Ensuring the Quality Testing of Web Using a New Methodology," in *Proceedings of the 7th IEEE International Symposium on Signal Processing and Information Technology*, 2007, pp. 1071–1076.

[59] Q. Wu and Y. Wang, "Performance Testing and Optimization of J2EE-Based Web Applications," in *Proceedings of the 2nd International Workshop on Education Technology and Computer Science*, 2010, pp. 681–683.

[60] J. Xiao-yun and L. Si-hui, "Research and Realization of Automatic Testing in the Application Software," in *Proceedings of the 29th International Conference of Information Science and Management Engineering*, 2010, pp. 351–353.

[61] D. Miao, W. Chen, D. Tang, Y. Liu, and L. Jia, "Research on HSQLDB Concurrency," in *Proceedings of the 2nd International Conference on Computer Application and System Modeling*, 2010, pp. 477–481.

[62] G. Jiang and S. Jiang, "A Quick Testing Model of Web Performance Based on Testing Flow and its Application," in *Proceedings of the 6th Web Information Systems and Applications Conference*, 2009, pp. 57–61.

[63] Q. Zhou, R. Bian, and Y. Pan, "Design of Electric Power Web System Based on Comet," in *Proceedings of the 2nd International Conference on Intelligent Computation Technology and Automation*, 2009, pp. 42–45.

[64] P. Chen and S. Liu, "Intelligent Vehicle Monitoring System Based on GPS, GSM and GIS," in *Proceedings of the International Conference on Information Engineering*, 2010, pp. 38–40.

[65] S. Gaisbauer, J. Kirschnick, N. Edwards, and J. Rolia, "VATS: Virtualized-Aware Automated Test Service," in *Proceedings of the 5th International Conference on Quantitative Evaluation of Systems*, 2008, pp. 93–102.

[66] Y. Pu and M. Xu, "Load Testing for Web Applications," in *Proceedings of the 1st International Conference on Information Science and Engineering*, 2009, pp. 2954–2957.

[67] D. Schultz, "Hello World: Rational Performance Tester Get to the Bottom of Application Performance Issues," IBM Corporation, Tech. Rep., 2007.

[68] M. Ben-Yehuda, D. Breitgand, M. Factor, H. Kolodner, V. Kravtsov, and D. Pelleg, "NAP: a Building Block for Remediating Performance Bottlenecks Via Black Box Network Analysis," in *Proceedings of the 6th International Conference on Autonomic computing*, 2009, pp. 179–188.

[69] M. Meyers, C. Beyers, and D. Weber, "An Experience Using Rational Performance Tester to Benchmark Oracle EnterpriseOne," IBM Corporation, Tech. Rep., 2008.

[70] M. Ebbers, T. S. Buchanan, A. Greggo, D. Joseph, J. Langer, E. Ong, and M. Wisniewski, "Performance Test of Virtual Linux Desktop Cloud Services on System Z," IBM Corporation, Tech. Rep., 2010.

[71] V. Apte, T. Hansen, and P. Reeser, "Performance Comparison of Dynamic Web Platforms," *Computer Communications*, vol. 26, no. 8, pp. 888–898, Out 2003.

[72] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton, "Middle-tier Database Caching for E-business," in *Proceedings of the International Conference on Management of Data*, 2002, pp. 600–611.

[73] B. C. Ling, E. Kiciman, and A. Fox, "Session State: Beyond Soft State," in *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation*, 2004, pp. 22–26.

[74] G. Kim, H. Moon, G. P. Song, and S. K. Shin, "Software Performance Testing Scheme Using Virtualization Technology," in *Proceedings of the 4th International Conference on Ubiquitous Information Technologies Applications*, 2009, pp. 1–5.

[75] S. C. Borland, "SilkPerformer 2009 Help," Borland Software Corporation, Tech. Rep., 2009.

[76] J. Levinson, *Software Testing With Visual Studio 2010*. Pearson Education, 2011.

[77] S. Subashni and N. S. Kumar, *Software Testing with Visual Studio Team System 2008*. Packt Publishing, 2008.

[78] X. Bai, "Testing the Performance of an SSAS Cube Using VSTS," in *Proceedings of the 7th International Conference on Information Technology: New Generations*, 2010, pp. 986–991.

[79] D. Liu, L. Li, X. Yang, and H. Zhai, "The Applications of Pressure Test in the B/S System," in *Proceedings of the International Conference on Computational Intelligence and Software Engineering*, 2009, pp. 1–4.

[80] D. Thakkar, A. E. Hassan, G. Hamann, and P. Flora, "A Framework for Measurement Based Performance Modeling," in *Proceedings of the 7th International Workshop on Software and Performance*, 2008, pp. 55–66.

[81] J. Krizanic, A. Grguric, M. Mosmondor, and P. Lazarevski, "Load Testing and Performance Monitoring Tools in Use with AJAX Based Web Applications," in *Proceedings of the 33rd International Convention*, 2010, pp. 428–434.

[82] X. Chen and Z. Hu, "Study on Performance Testing of Index Server Developed as ISAPI Extension," in *Proceedings of the IEEE International Conference on Grey Systems and Intelligent Services*, 2009, pp. 1391–1395.

[83] O. Hamed and N. Kafri, "Performance Testing for Web Based Application Architectures (.NET vs. Java EE)," in *Proceedings of the 1st International Conference on Networked Digital Technologies*, 2009, pp. 218–224.

[84] B. L. Romano, G. B. e Silva, H. F. de Campos, R. G. Vieira, A. M. da Cunha, F. F. Silveira, and A. C. B. Ramos, "Software Testing for Web-Applications Non-Functional Requirements," in *Proceedings of the 6th International Conference on Information Technology: New Generations*, 2009, pp. 1674–1675.

[85] K. Bierhoff, M. Grechanik, and E. S. Liongosari, "Architectural Mismatch in Service-Oriented Architectures," in *Proceedings of the International Workshop on Systems Development in SOA Environments*, 2007, pp. 4–9.

[86] A. Koziolek, H. Koziolek, and R. Reussner, "PerOpteryx: Automated Application of Tactics in Multi-Objective Software Architecture Optimization," in *Proceedings of the Quality of Software Architectures*, 2011, pp. 33–42.

[87] S. E. I. (SEI), "Software Product Lines (SPL)," Capturado em: http://www.sei.cmu.edu/productlines, Outubro 2011.

[88] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, CMU, Tech. Rep., 1990.

[89] F. J. v. d. Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag, 2007.

[90] V. Gowadia and C. Farkas, "RDF metadata for XML access control," in *Proceedings of the ACM Workshop on XML Security*, 2003, pp. 39–48.

[91] C. Bertolini, F. Farina, P. Fernandes, and F. M. Oliveira, "Test Case Generation Using Stochastic Automata Networks: Quantitative Analysis," in *Proceedings of the 2nd International Conference on Software Engineering and Formal Methods*, 2004, pp. 251–260.

[92] R. Zurawski and M. Zhou, "Petri Nets and Industrial Applications: A Tutorial," *IEEE Transactions on Industrial Electronics*, vol. 41, no. 6, pp. 567–583, Dez 1994.

[93] S. Kanjilal, S. T. Chakradhar, and V. D. Agrawal, "Test Function Embedding Algorithms with Application to Interconnected Finite State Machines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 9, pp. 1115–1127, Set 1995.

[94] A. Petrenko, N. Yevtushenko, A. Lebedev, and A. Das, "Nondeterministic State Machines in Protocol Conformance Testing," in *Proceedings of the 6th International Workshop on Protocol Test Systems VI*, 1993, pp. 363–378.

[95] O. A. L. Lemos, A. M. R. Vincenzi, J. C. Maldonado, and P. C. Masiero, "Control and Data Flow Structural Testing Criteria for Aspect-oriented Programs," *Journal of Systems and Software*, vol. 80, no. 6, pp. 862–882, Jun 2007.

[96] A. T. Endo, M. Linschulte, A. da Silva Simão, and S. do Rocio Senger de Souza, "Event- and Coverage-Based Testing of Web Services," in *Proceedings of the 4th IEEE International Conference on Secure Software Integration and Reliability Improvement Companion*, 2010, pp. 62–69.

[97] A. M. R. Vincenzi, J. C. Maldonado, W. E. Wong, and M. E. Delamaro, "Coverage Testing of Java Programs and Components," *Science of Computer Programming*, vol. 56, no. 1–2, pp. 211–230, Abr 2005.

[98] M. Ahmed, M. M. Uddin, M. S. Azad, and S. Haseeb, "MySQL Performance Analysis on a Limited Resource Server: Fedora vs. Ubuntu Linux," in *Proceedings of the International Conference Spring Simulation Multiconference*, 2010, pp. 99–106.

[99] D. Chetty, *Tomcat 6 Developer's Guide*. Packt Publishing, 2009.

[100] S. F. Apache, "Apache Software Fundation," Capturado em: http://www.apache.org, Março 2011.

[101] R. C. Gronback, *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009.

[102] H. Böck, *The Definitive Guide to the NetBeans Platform 7*. Apress, 2011.

[103] G. B. Shelly and S. M. Freund, *Windows Internet Explorer 9: Introductory*. Course Technology, 2011.

[104] G. B. Shelly, T. J. Cashman, S. G. Forsythe, and S. M. Freund, *Mozilla Firefox: Introductory Concepts and Techniques*. Course Technology, 2005.

[105] R. Barbuti and S. Cataudella, "Java Bytecode Verification on Java Cards," in *Proceedings of the ACM Symposium on Applied Computing*, 2004, pp. 431–438.

[106] J. Sharp and A. Longshaw, *Microsoft Visual J# .Net (Core Reference)*. Microsoft Press, 2002.

A. FERRAMENTAS PARA AUTOMATIZAÇÃO DE TESTE DE *SOFTWARE*

Com a necessidade de aprimorar a qualidade dos sistemas computacionais, diversas ferramentas para automatização e verificação foram criadas com o intuito de aumentar a eficiência na execução de casos de teste das aplicações. Atualmente, existem diversas ferramentas para a geração e execução automática de casos de teste. A seguir, são descritas as características e funcionalidades de algumas das mais conhecidas ferramentas para automatização de testes existentes. Em virtude do foco desta dissertação estar direcionado ao teste de desempenho, foi dada uma importância maior para as ferramentas referentes a este tipo de teste.

A.1 Apache JMeter

O JMeter [16] é uma ferramenta *open source* integrada ao projeto *Jakarta* da *Apache Software Fundation* [100] capaz de executar testes de desempenho em aplicações *web*. Tem a capacidade de gerar requisições *web* automaticamente, simulando o comportamento de um número de usuários que pode ser determinado. Assim, ele cria um conjunto de *threads*, cada uma simulando um usuário que faz requisições aos endereços de páginas configuradas previamente.

O JMeter possui uma interface simples de se utilizar e com isso, a criação e configuração de casos de teste tornam-se tarefas intuitivas. Esta ferramenta possui um controle de *threads*, chamado *Thread Group*, com o qual é possível determinar o número de usuários fictícios (*threads*) que fazem requisições à aplicação; o tempo de inicialização do conjunto de *threads*; tempo de execução de cada grupo de *threads*; bem como, o número de iterações que cada conjunto irá executar;

Além disso, o JMeter disponibiliza diversos recursos para análise e coleta de resultados, uma das funcionalidades mais utilizadas para esse fim é o *Agregate Report*. Com ele é possível analisar graficamente e em tempo de execução diversos contadores, tais como os descritos a seguir (ver Figura A.1):

- Número de amostras;
- Média do tempo de resposta em milissegundos;
- Percentual de erros ocorridos durante o teste;
- Percentual de utilização de CPU;
- Vazão de dados na rede em KB/s;
- Dados estatísticos como, por exemplo, mediana e intervalo de confiança.

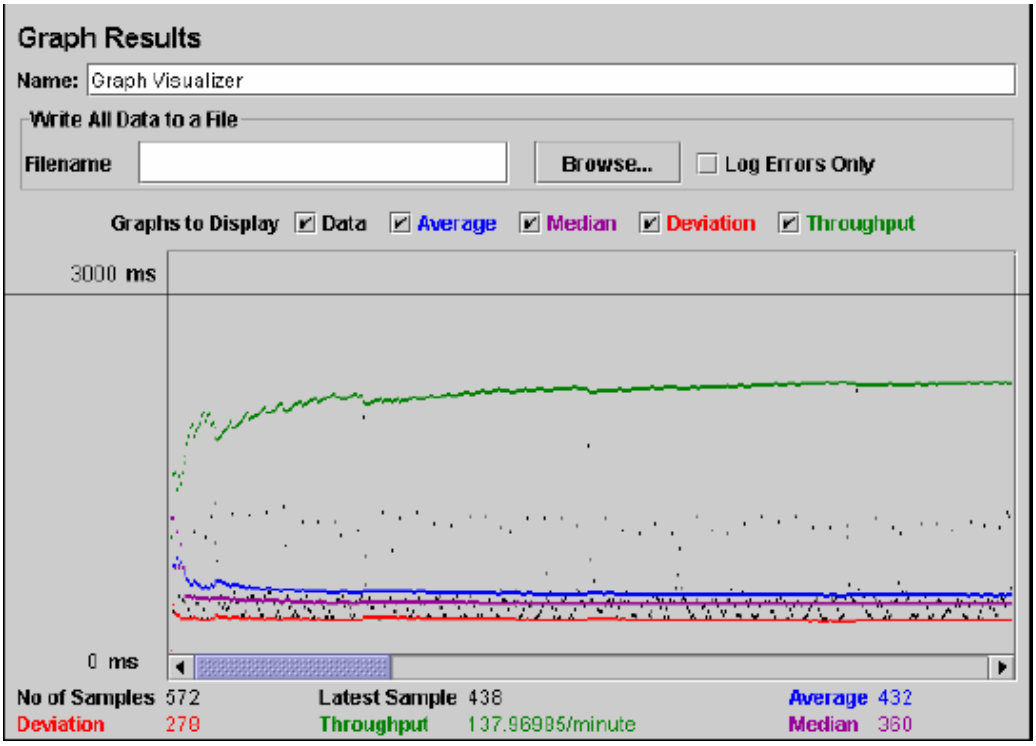

Figura A.1: Gráfico de análise do JMeter [4]

É importante mencionar que para a geração e execução de casos de teste o JMeter necessita que uma série de informações sejam configuradas. Entretanto, os principais parâmetros utilizados para a criação de casos de testes para a ferramenta são:

- *HTTP Request*: define o caminho (endereço destino) para onde será gerada a carga pelo JMeter;
- *Number of Threads*: define o número de usuários que farão requisições ao(s) endereço(s) definido(s) em *HTTP Request*;
- *Ramp-Up period*: define o tempo que levará para que todas as *threads* (usuários) estejam ativas, ou seja, realizando requisições ao endereço previamente configurado.
- *Cycle Number (Loop Count)*: define o número de interações de cada usuário com a aplicação;
- *Duration (seconds)*: define o tempo total de execução do teste;
- *Listeners*: define o conjunto de contadores que serão utilizados para medir o desempenho como, por exemplo, número de amostras, tempo de resposta da aplicação, *throughput*, etc.

Utilizando, basicamente, os parâmetros descritos anteriormente é possível criar cenários de teste para o JMeter, os quais iniciam-se a partir da configuração de um plano de teste. Nesse plano de teste é adicionado o controle de *threads*, sendo este chamado *Thread Group*. Nele (*Tread Group*) são configurados os parâmetros *Number of Threads*, *Ramp-Up period*, *Loop Count* e *Duration*,

descritos anteriormente. Os demais parâmetros são configurados adicionando-se as informações de *HTTP Request* ao controle *Thread Group*. Como descrito anteriormente, com *HTTP Request* é possível realizar a configuração do(s) endereço(s) destino para onde será gerada a carga, neste caso alguns endereços (*links*) referentes à aplicação a ser testada são adicionados. Para melhor compreensão, é apresentado na Figura A.2 os parâmetros a pouco descritos.

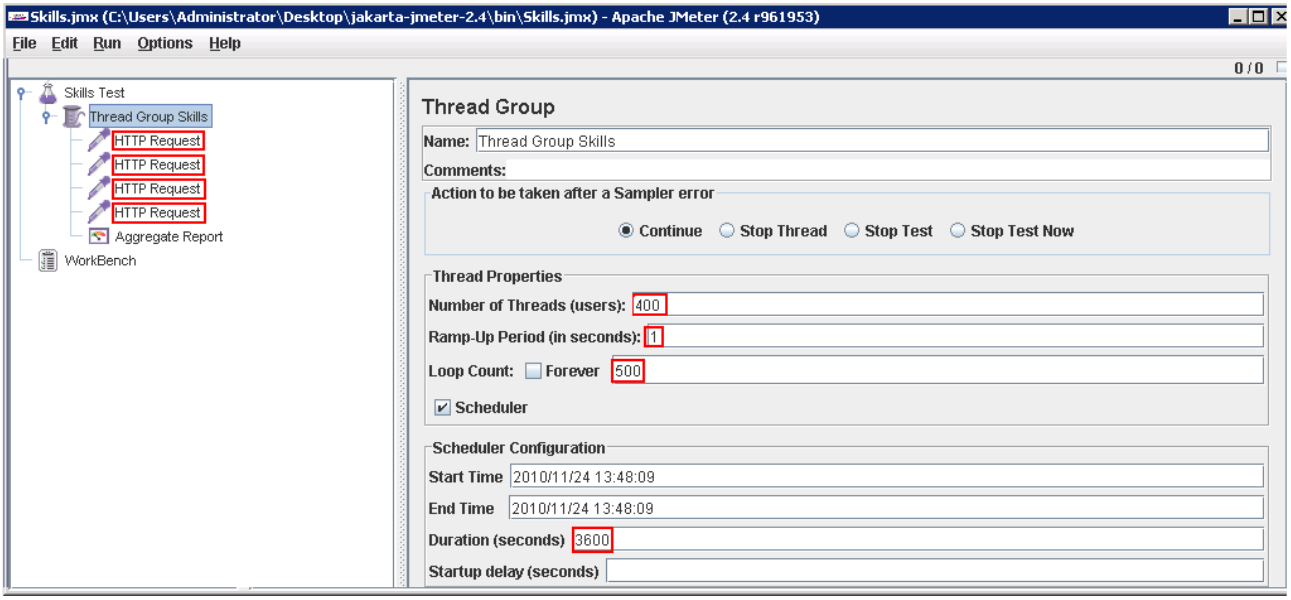

Figura A.2: *Layout* do cenário de teste do JMeter

Para finalizar a configuração do cenário de teste, é necessária a adição de alguns contadores. Ao adicionar o *Listener Aggregate Report*, uma série de contadores são utilizados para monitoração do ambiente, tais como, *Average* (tempo de resposta), *Samples* (número de amostras), *Throughput* (vazão). Contudo, uma série de *Listeners*, diferentes do *Aggregate Report*, podem ser configurados para o teste.

A.2 IBM Rational Functional Tester

O IBM Rational Functional Tester (RFT) [17] é uma ferramenta para a automação de testes funcionais que utiliza a técnica *Record&Playback* para a geração de *scripts* de teste. Esta técnica consiste na gravação de todas as interações do usuário com a aplicação a ser testada. O RFT possui um eficiente gravador capaz de registrar as atividades do usuário e gerar um *script* que pode ser modificado e personalizado. Esta ferramenta é bastante utilizada para teste em aplicações com tecnologia Java, Microsoft Visual Studio e *web*.

A geração de um caso de teste para a ferramenta baseia-se, em um primeiro momento, na criação de *scripts* através da gravação das interações de um usuário com a aplicação que se deseja testar. Toda gravação inicia quando o botão "*record*" da interface da ferramenta é ativado pelo profissional de teste que, em seguida, faz acesso à aplicação e cada uma de suas interações e atividades é gravada.

Quando este processo termina, através da intervenção do próprio profissional de teste, por exemplo, as informações gravadas durante esta etapa são armazenadas em um *script* o qual, pode ser opcionalmente modificado e parametrizado. Na Figura A.3 a seguir é apresentado um exemplo de *script* gerado após o processo de gravação. Neste exemplo, todo o código do *script* gerado é gravado no método "testMain" da classe "PurchaseMauiAdventure".

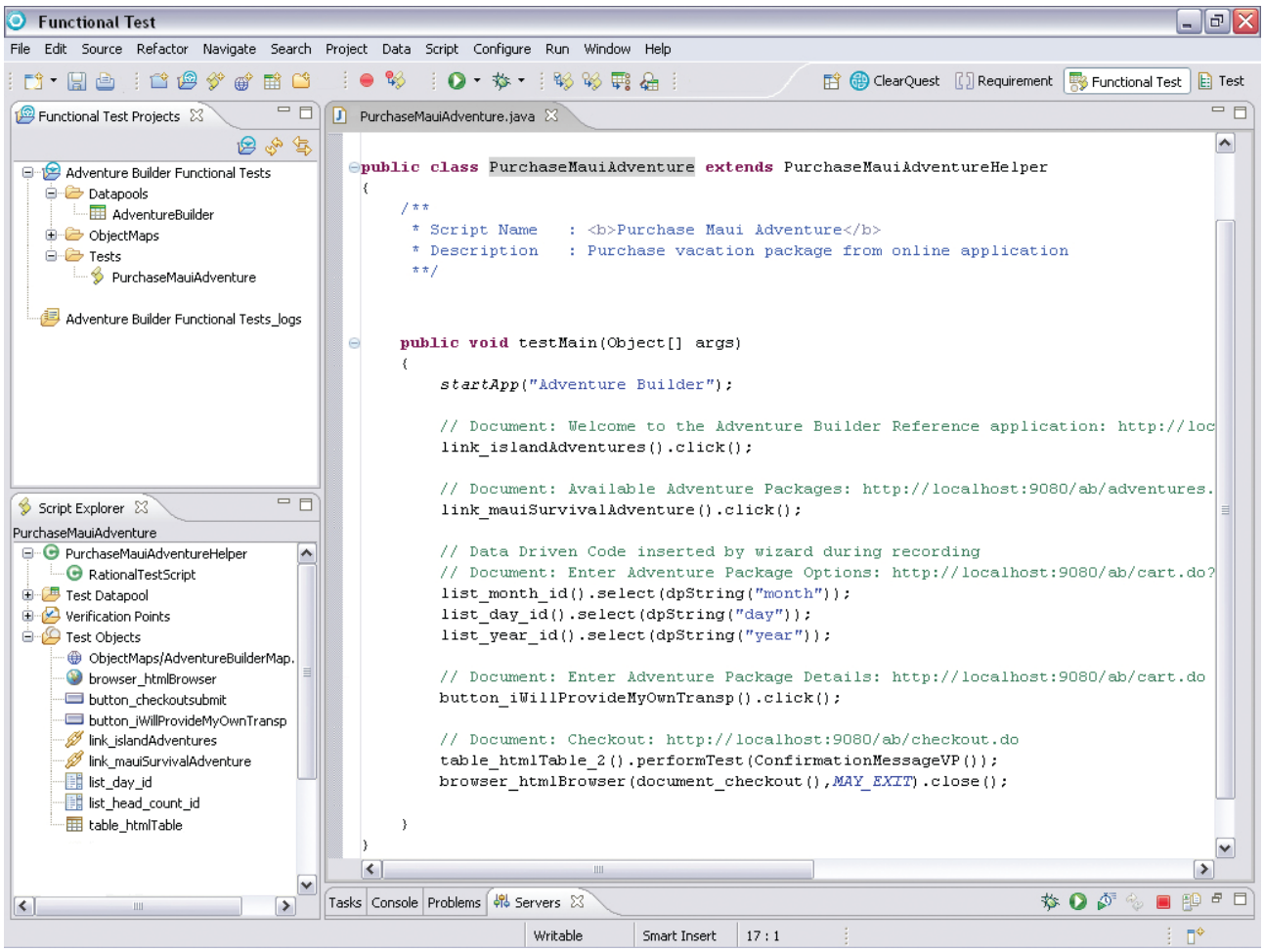

Figura A.3: Exemplo de *Script* gerado para o RFT [5]

O teste pode ser executado após o processo de gravação e geração do *script*. Nesta etapa pode ser definido o número de usuários (*threads*), tempo de duração do teste entre outros atributos para o teste. Após a execução, a ferramenta RFT gera um relatório com uma série de informações de interesse.

A.3 JUnit

O JUnit [18] é um *framework open source* capaz de automatizar testes em código Java. Criado por Eric Gamma e Kent Beck, o foco deste *framework* está na criação e execução de testes unitários. Com o JUnit é possível testar cada método de uma classe e verificar para um determinado conjunto de entradas se o resultado está ou não de acordo com o esperado, caso não esteja o JUnit exibe um relatório dos defeitos ocorridos referentes aos métodos testados.

Para um melhor entendimento relativo ao funcionamento e funcionalidades do JUnit é necessário antes conhecer sua arquitetura, a qual é composta por três classes: "Test", "TestCase" e "TestSuit", onde as duas últimas herdam métodos da primeira (ver Figura A.4). A classe "Test" possui um método chamado *runTest* o qual, é responsável por executar um conjunto de testes particulares. A classe "TestCase" possui dois métodos, "setUp" e "tearDown". O primeiro método desta classe é responsável por informar o início de um processo de teste, já o segundo é responsável por assinalar o final de um processo de teste. A classe "TestCase" é capaz de testar os resultados de um método apenas, enquanto a classe *TestSuit* é responsável por testar os resultados de um conjunto de métodos.

Além destes métodos, a arquitetura do JUnit possui um método de comparação chamado *assertEquals* que recebe dois parâmetros, o resultado do método testado e o resultado esperado para esse método.

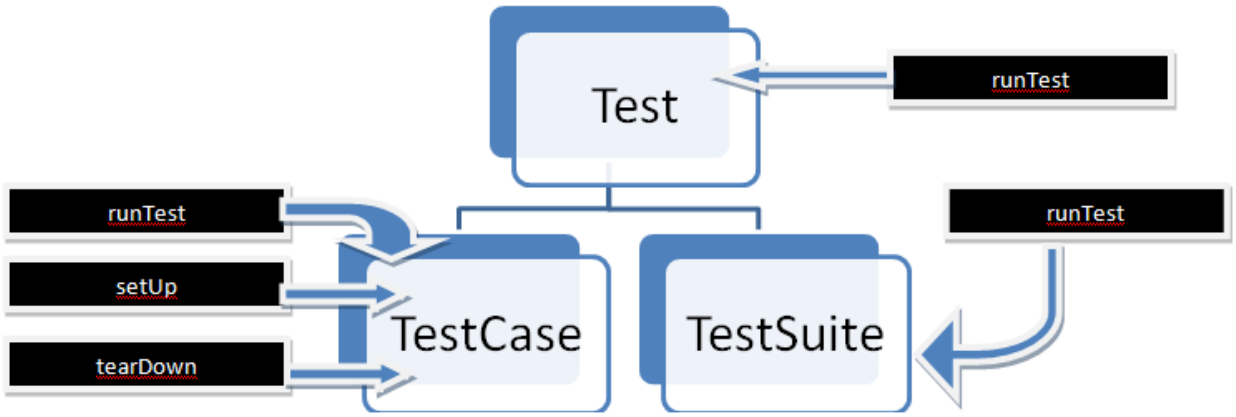

Figura A.4: Arquitetura da API do JUnit [6]

Além de ser muito intuitivo e fácil de utilizar, o JUnit possui suporte a diversas *IDE's* Java, entre eles citam-se o Eclipse [101] e NetBeans [102], utilizados por grande parte dos desenvolvedores de aplicações Java. Além disso, o JUnit possui uma série de outras vantagens, como:

- Permite escrever código de teste mais rapidamente e com isso aumentar a produtividade e a qualidade do *software*;
- É simples e ao mesmo tempo elegante, um incentivo à criação de testes;
- Verifica os resultados dos testes e retorna uma resposta de forma imediata. Realizar testes de forma manual pode atrasar o andamento de um projeto, por outro lado o JUnit executa automaticamente cada teste retornando imediatamente seus resultados, facilitando e agilizando todo o processo de desenvolvimento;
- Contribui de forma eficiente à etapa de desenvolvimento, uma vez que o programador não necessita perder tempo depurando o código;

- É um *framework* livre e todos os casos de teste são escritos em Java.

A.4 HP Quick Test Professional

Popularmente conhecido como QTP [19], o Quick Test Professional é uma das ferramentas mais utilizadas para automatizar testes funcionais. Foi originalmente desenvolvida pela empresa Mercury Interactive e posteriormente adquirida pela empresa Hewlett Packard (HP) em 2006. Atualmente, o QTP é mais uma entre outras ferramentas de teste que fazem parte da HP Quality Center. Esta ferramenta possui uma interface nativa (similar a uma interface *web*), a qual é utilizada pelo profissional de teste para interagir com a aplicação e criar *scitpts* de teste. Sendo estes gerados na linguagem Visual Basic Scripting Edition (VBScript). A ferramenta também possui outras funcionalidades, como por exemplo, a capacidade de gerar relatórios com as informações dos resultados após a execução de cada caso de teste.

Assim como outras ferramentas de teste, o QTP possui um processo para a criação e execução de casos de teste. Para a ferramenta, este processo envolve três etapas: criação dos testes (*scripts de teste*), execução dos *scripts* de teste e análise dos resultados. A seguir, é apresentada a descrição referente a cada uma das etapas deste processo (ver Figura A.5).

- *Criação dos testes (scripts de teste):* Inicialmente, é necessário definir o que deve ser testado no sistema, por exemplo, supondo que a aplicação a ser testada se trata de uma aplicação *web* para reserva de passagens aéreas e a página inicial dessa aplicação se refere à interface para definir informações de *login* de usuário. Pode-se definir como sendo um caso de teste, que tratamento será dado quando o campo *login* receber um valor inválido, ou seja, receber, por exemplo, um valor inteiro assumindo que o respectivo campo só permite valores do tipo String. Com o caso de teste definido, pode-se utilizar a técnica de gravação da ferramenta e executar a aplicação inserindo valores de entrada nos campos referentes à operação de *login*. Realizada esta operação a aplicação pode ser fechada e cada interação do usuário é gravada em um *script*.

- *Execução dos scripts:* Nesta etapa é realizada a verificação da funcionalidade da ferramenta para o caso de teste definido na etapa anterior. Executa-se o *script* gerado e a partir dessa execução são informados os erros ou se estava correto o tratamento para caso de teste determinado.

- *Análise dos resultados dos testes:* Por último, é gerado um relatório com diversas informações da execução do teste, como por exemplo, dados de entrada referentes ao campo utilizado para o caso de teste definido, os parâmetros e tipos de variáveis utilizadas para o teste, onde ocorreu o defeito (no caso de uma funcionalidade não implementada), etc.

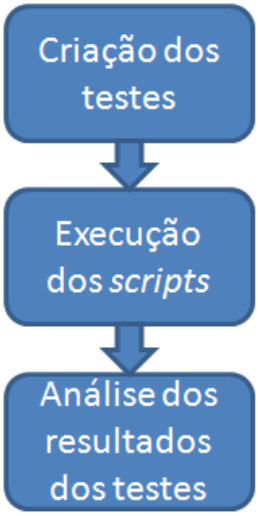

Figura A.5: Etapas para a execução de um caso de teste completo para o QTP [7]

A.5 HP LoadRunner

O LoadRunner [20], a exemplo do Quick Test Professional, é mais uma das ferramentas criadas pela Mercury Interactive e que hoje integra o conjunto de ferramentas da HP Quality Center. A diferença está no fato de que para o Loadrunner o foco está na geração e execução de casos de teste de desempenho. Entretanto, assim como o QTP, o LoadRunner também utiliza a técnica *Record&Playback* para geração de seus *scripts* de teste, os quais podem ser gerados nas linguagens Java [46] e C [52].

O processo de geração e execução de testes para o LoadRunner consiste de cinco etapas: Planejamento, criação do *script* de teste, definição do cenário, execução do cenário e análise de resultados (ver Figura A.6):

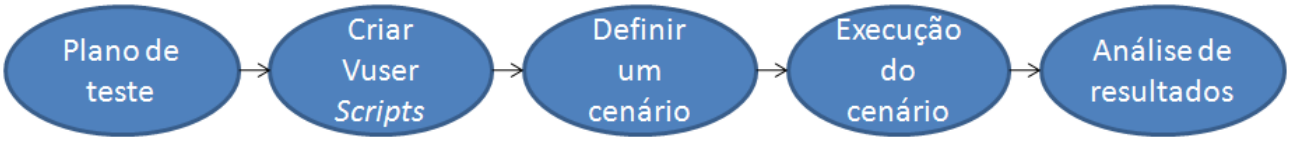

Figura A.6: Etapas para a execução de casos de teste para o LoadRunner [8]

- *Plano de teste:* definição dos requisitos para o teste de desempenho, por exemplo, definição de determinado tempo de resposta para certo número de usuários acessando aplicação simultaneamente;
- *Criar VUser Scripts:* Gravar as atividades do usuário sobre a aplicação a ser testada e gerar *scripts* automatizados;
- *Definir um cenário:* Configurar o ambiente de teste utilizando o Controller do LoadRunner;
- *Execução do cenário:* Gerenciar e monitorar o teste de carga utilizando o Controller do LoadRunner;

- *Análise de resultados:* Utilizar o Analyser do LoadRunner para a manipulação dos gráficos de resultados, gerar relatórios e avaliar o desempenho.

Para auxiliar no processo descrito, existem três ferramentas que compõem a arquitetura do LoadRunner: *Virtual User Generator* (VUGen), *Controller* e *Analyser*. Elas são utilizadas, respectivamente, para a geração, execução e análise dos resultados dos casos de teste. A seguir, são descritas as funcionalidades de cada uma destas ferramentas.

- *Virtual User Generator:* esta ferramenta é responsável por capturar as informações referentes às interações do usuário com a aplicação a ser testada. Posteriormente, estas informações são gravadas em um *script* de teste, também conhecido como *script* de usuário virtual;
- *Controller:* é responsável pela configuração das informações de carga de trabalho e definição/monitoramento de contadores para avaliar o comportamento do sistema a ser testado;
- *Analyser:* gera relatórios com as informações do teste executado, apresentando gráficos e os valores correspondentes às métricas e contadores definidos e monitorados no Controller.

Para melhor compreensão do funcionamento da ferramenta é apresentado, a seguir, na Figura A.7 uma imagem do *layout* inicial do LoadRunner, onde cada uma das três ferramentas que compõem sua arquitetura pode ser selecionada.

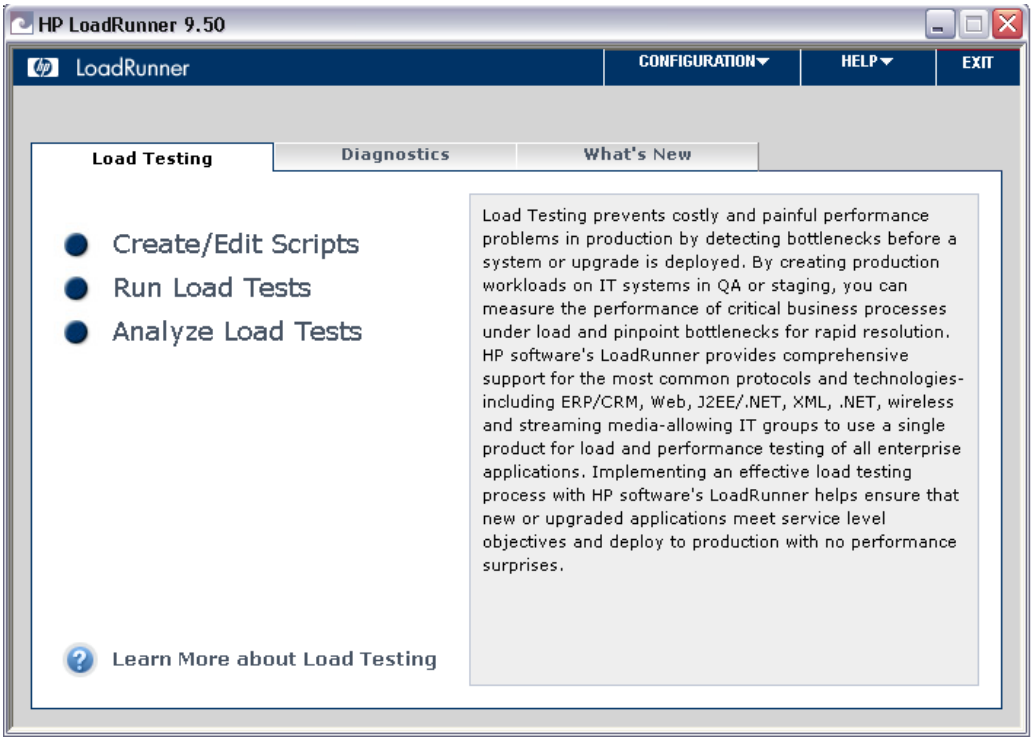

Figura A.7: *Layout* inicial do LoadRunner

Em posse do conhecimento de alguns termos e das ferramentas supracitadas, o processo de execução de um caso de teste para o LoadRunner pode ser apresentado. A criação de um plano de

teste para o LoadRunner consiste na criação de *scripts* e cenários de teste. Entretanto, inicialmente, é necessária a escolha do protocolo específico para teste de desempenho *web*, o protocolo *Web (HTTP/HTML)* (ver Figura A.8).

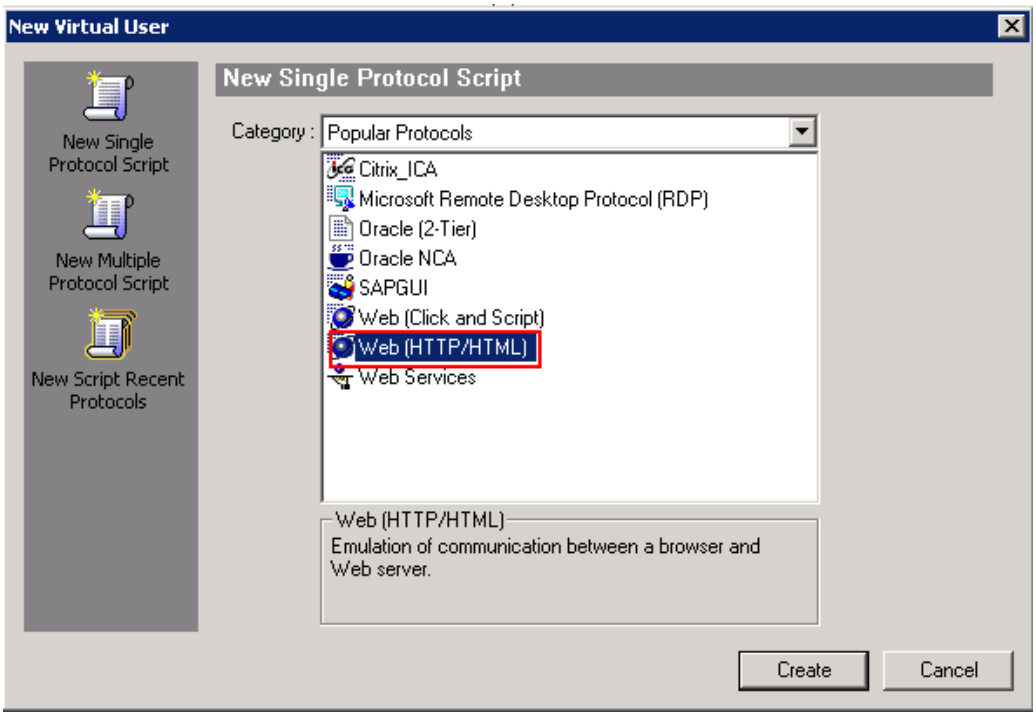

Figura A.8: *Layout* referente aos protocolos disponíveis para o LoadRunner

Após a seleção do protocolo de teste, realiza-se o processo de gravação, onde o botão *"Record"* do VUGen é ativado pelo testador e em seguida, automaticamente, a aplicação a ser testada é executada, nesta etapa todas as operações efetuadas pelo usuário (testador) sobre a aplicação são armazenadas no banco de informações do VUGen para posteriormente gerar o *script*. Esta etapa termina quando o testador ativa o botão *"stop"* do VUGen, com isso, a gravação é interrompida e o *script* de teste é gerado (ver Figura A.9).

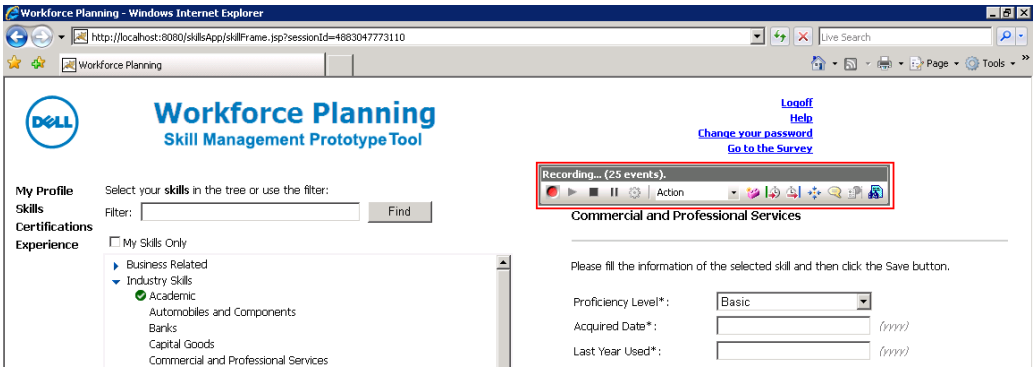

Figura A.9: *Layout* referente ao processo de gravação do LoadRunner

A etapa seguinte é onde, de fato, o teste de carga é executado, portanto, é necessário neste momento a execução do Controller, que dever ser executado de modo a "carregar" o *script* gerado na etapa anterior. Antes da execução do caso de teste, é necessário configurar o cenário de

teste, através da definição de alguns parâmetros, tais como: quantidade de *threads* que simulam o usuário acessando a aplicação, tempo de inicialização para um conjunto de *threads* e tempo total de execução de teste. Para finalizar a configuração do cenário de teste, é necessária a adição de alguns contadores como, por exemplo, *Trans Response Time* (tempo de resposta), *Hits per second* (número de requisições por segundo), *Throughput*. Além desses, uma série de outros contadores podem ser adicionados ao teste. Uma característica do Controller é que os dados relativos a cada tipo de contador configurado para o teste podem ser visualizados em tempo de execução por gráficos e tabelas. Na Figura A.10, é possível observar um teste em execução, onde estão destacados alguns contadores ativos.

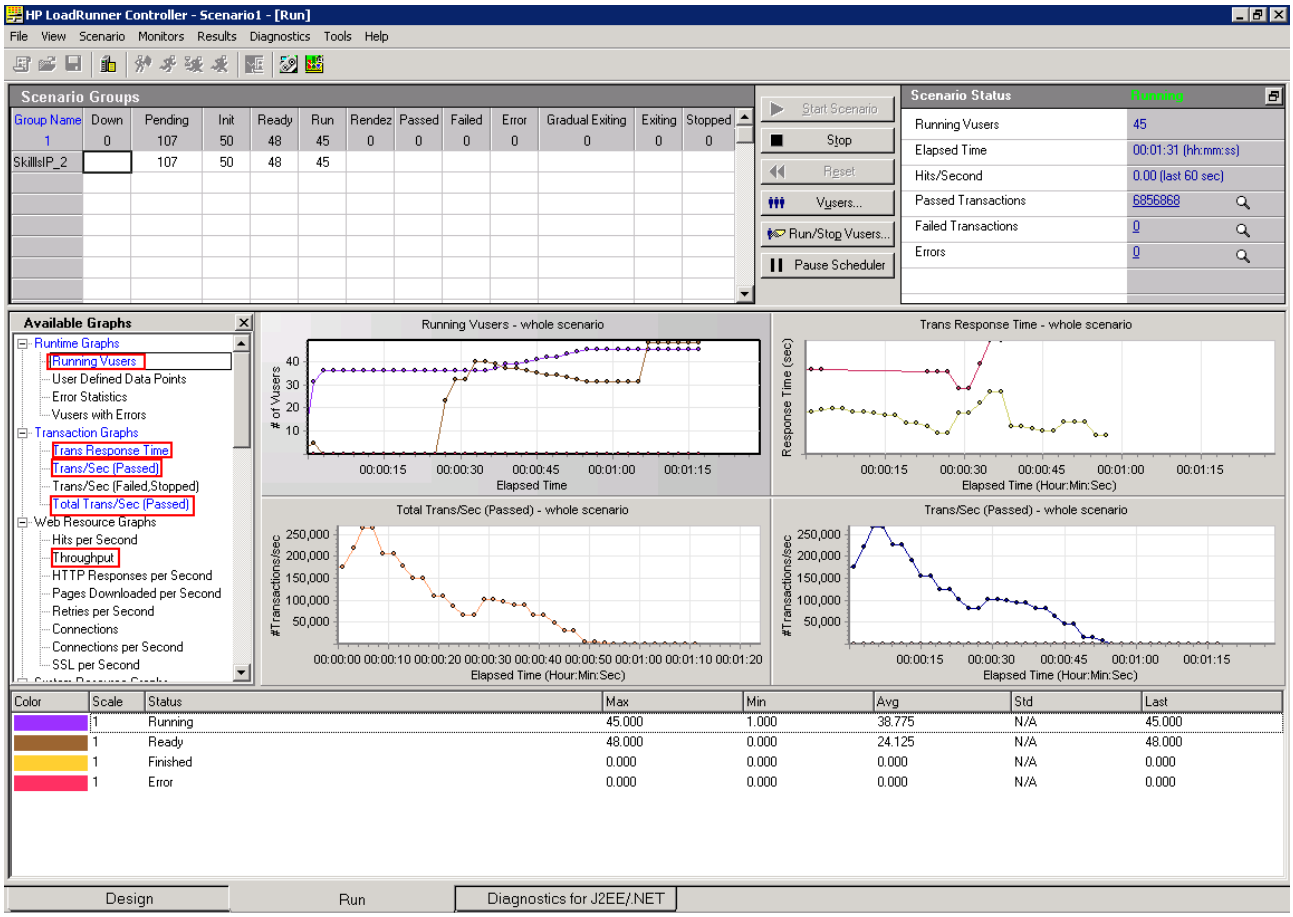

Figura A.10: *Layout* do cenário de teste do LoadRunner com a adição de contadores de desempenho

Após a execução do teste os resultados do teste podem ser observados por meio da utilização da ferramenta Analyser. Esta ferramenta é capaz de agrupar os vários gráficos reproduzidos durante o teste, a fim de determinar algum tipo de relação e comportamento da aplicação, com isso, determinar a causa de algum defeito ocorrido e responsável, por exemplo, por uma eventual queda de desempenho.

A.6 IBM Rational Performance Tester

O IBM Rational Performance Tester [21], comumente conhecido como RPT, é uma ferramenta para a criação e execução de testes de desempenho. Provê análise dos testes realizados e com isso, auxilia equipes de teste na validação da escalabilidade e confiabilidade de aplicações. Assim como outras ferramentas de teste, o RPT permite a utilização de vários navegadores *web* para a geração de *scripts* de teste da aplicação, como Internet Explorer [103], Mozilla [104], entre outros.

Com o intuito de facilitar e melhor organizar a geração de casos de teste para o RPT, é apresentado em [21] a definição de um conjunto de cinco etapas contendo documentos e especificações para auxiliar na criação de planos de teste para a ferramenta:

- *Especificação do teste:* A especificação do teste é um documento que contêm toda a modelagem para o teste de desempenho. Contêm os objetivos e as metas de desempenho, a descrição formal dos cenários de teste, valores máximo e mínimo aceitáveis para o tempo de resposta da aplicação para determinado número de usuários, entre outros fatores determinantes para o teste;

- *Plano de teste:* Nesta etapa é descrito como será organizado o conjunto de fatores descritos na etapa anterior e a lista de testes planejados. Também contém um conjunto de testes para cada tipo de teste e a ordem que devem ser executados;

- *Teste ativo:* Nesta etapa são gerados os *scripts* de teste, através das ações do usuário sobre a aplicação que se deseja testar. Para o RPT o teste ativo consiste em um projeto de teste contendo as gravações dos cenários de teste já configurados. Estes cenários são executados e ao final dessa execução as informações do teste são gravadas em um arquivo com um formato específico do RPT;

- *Resultados dos testes:* Ao término de cada execução, o RPT gera um conjunto de resultados e diversas análises podem ser feitas a fim de verificar o desempenho da aplicação e desta forma, fazer algum tipo de inferência com relação ao comportamento da aplicação. Por fim, as informações de análise geram relatórios padrão e personalizáveis para o RPT. Esses relatórios devem ser salvos para posteriormente serem utilizados como referência para o documento de análise do teste;

- *Análise do teste:* O documento de análise do teste descreve e define um conteúdo importante a partir dos resultados da execução de cada teste. O ponto mais importante deste documento são as conclusões sobre a análise do desempenho da aplicação pois, servirá como base para a geração de futuros casos de teste.

As informações descritas nestas etapas também promovem um maior entendimento relativo ao processo de criação de cenários de teste para o Rational Performance Tester (RPT), o qual é muito semelhante ao processo das ferramentas de desempenho apresentadas anteriormente. Inicialmente,

define-se o protocolo específico para teste de desempenho *web*, o protocolo *HTTP Test* (ver Figura A.11).

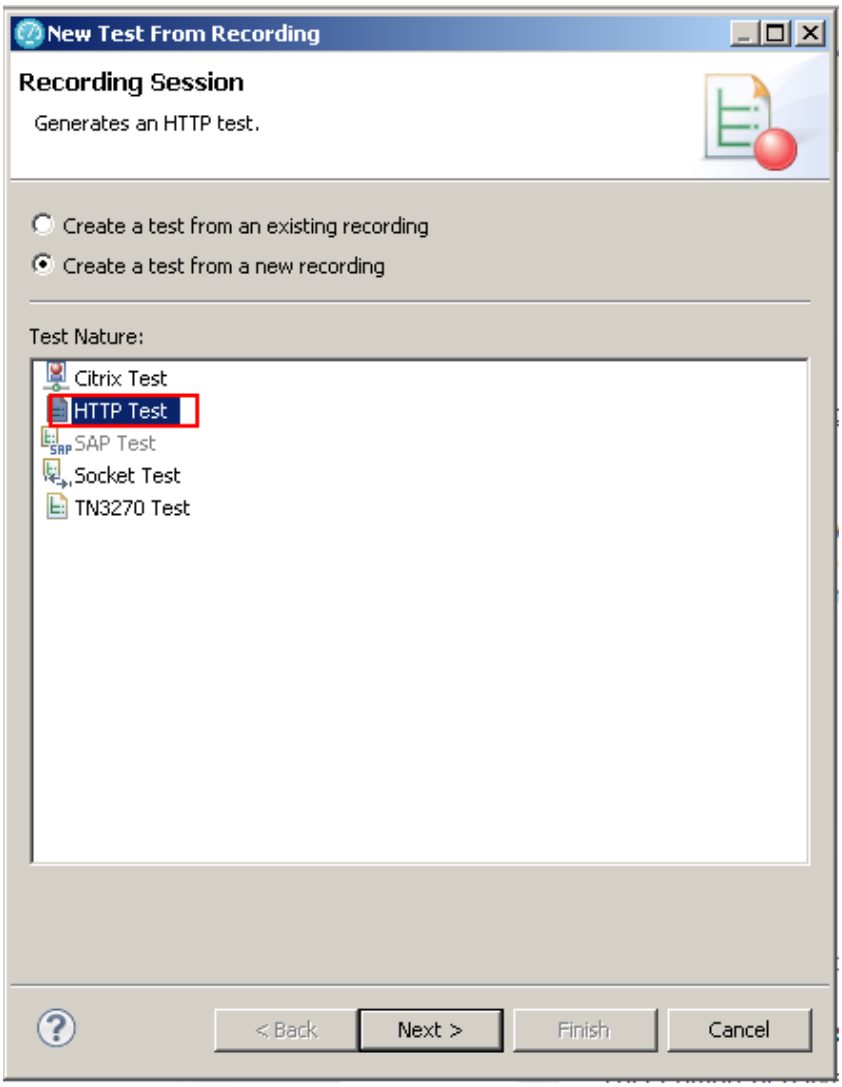

Figura A.11: *Layout* referente aos protocolos disponíveis para o RPT

Após a definição do protocolo, a gravação do *script* é iniciada automaticamente e, da mesma forma como é realizada para o LoadRunner, a geração de *scripts* para o RPT é feita por meio da utilização da técnica *Record&Playback*. Portanto, assim como é realizado para a ferramenta de desempenho da HP, toda a criação de um caso de teste para o RPT inicia a partir da gravação das interações do usuário com a aplicação que se deseja testar. Ao final da gravação, as informações referentes às interações são registradas no formato de um *script*.

Uma vez gravado o *script*, o mesmo é utilizado para a execução do cenário. Entretanto, anterior à execução do teste, todo o ambiente necessita ser configurado. Nesta etapa, diversas informações referentes ao cenário de teste são ajustadas como, por exemplo, quantidade total de usuários (*Number of users*), que define o número de usuários que farão requisições à aplicação; e tempo de duração do teste (*Run for specified period of time*), que define o tempo de duração do teste para um determinado cenário (ver Figura A.12).

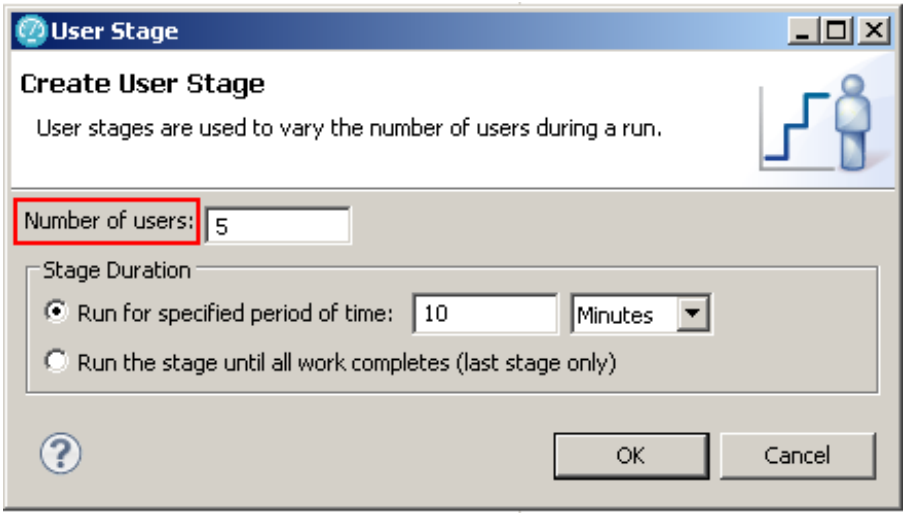

Figura A.12: *Layout* referente à configuração dos parâmetros do cenário de teste do RPT

Para completar a configuração de um cenário de teste, é necessária a adição de contadores de desempenho. Como pode-se observar na Figura A.13 podem ser definidos diversos tipos de contadores para avaliar o ambiente como, por exemplo, memória utilizada (*Pages/Sec*), percentual de utilização de CPU (*% Processor Time*), acesso ao disco (*%Read/Write Time*), entre outros.

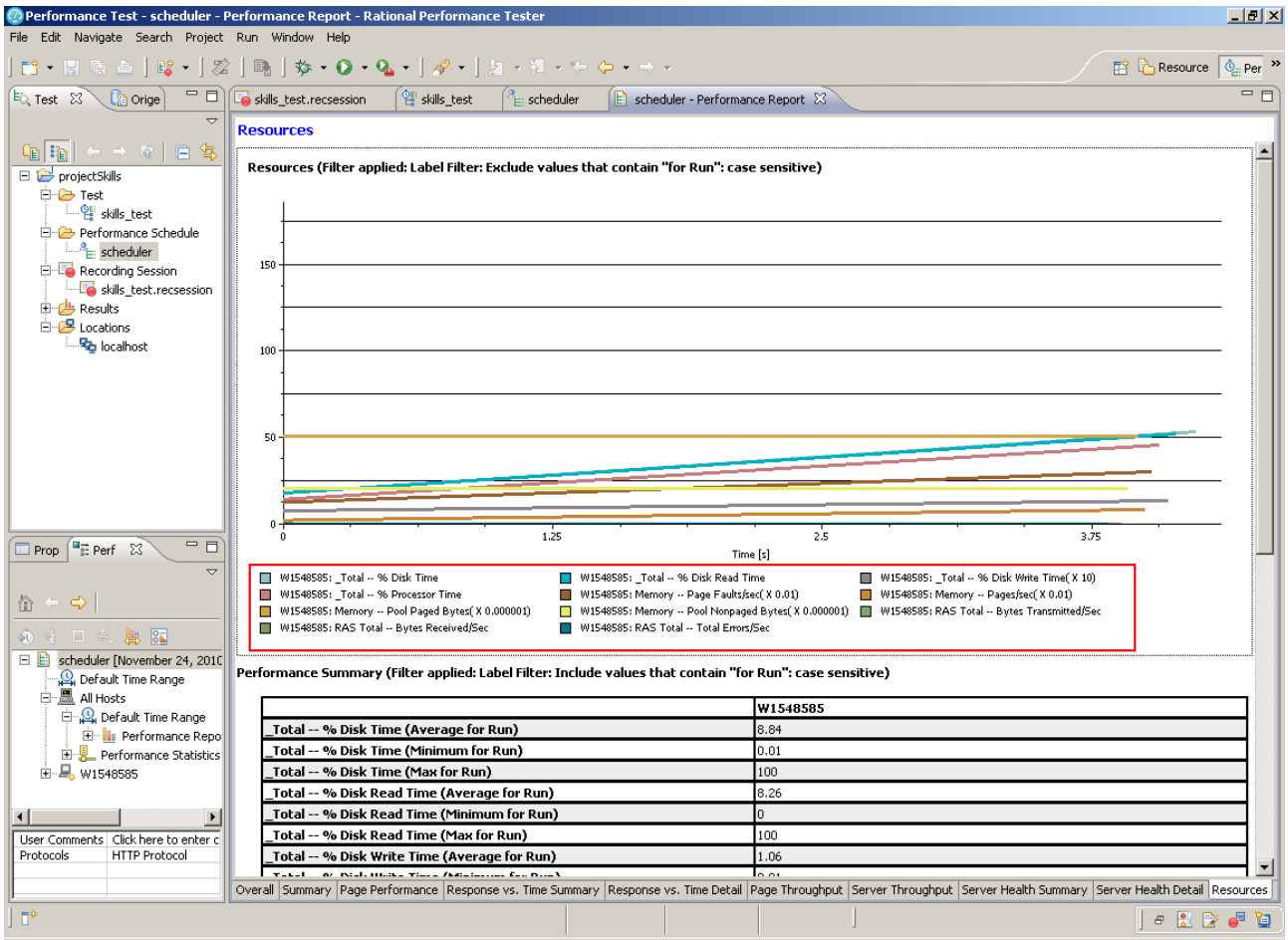

Figura A.13: *Layout* referente à execução do cenário de teste do RPT

A.7 Selenium

O Selenium [22] é uma ferramenta *open source* que provê a automação de testes funcionais para aplicações que executam sob protocolos *web* como, por exemplo, HTTP ou HTTPS. Esta ferramenta é capaz de criar *scripts* de teste, os quais são gerados por meio da utilização da técnica *Record&Playback*. Além disso, o Selenium tem suporte a diversos navegadores, e.g. Mozilla Firefox e Internet Explorer. Outra importante característica desta ferramenta, está relacionada à sua compatibilidade com diversas plataformas. Pelo fato de ser desenvolvido na linguagem Java, pode ser executado em sistemas operacionais Windows, Linux, e MacOS.

Sua arquitetura é composta por três ferramentas, com as quais é possível criar e executar casos de teste: Selenium-IDE, Selenium-RC (Remote Control) e Selenium-Grid. O Selenium-IDE é um ambiente de desenvolvimento do Selenium e a partir dele é realizada a criação de casos de teste e suítes de teste. Ele funciona como uma extensão do *browser* e possui um recurso de gravação que registra cada interação do usuário com a aplicação. Posteriormente, cada interação registrada é utilizada para a geração dos *scripts* de teste.

Para um público iniciante, é recomendado o uso do Selenium-IDE, pois ele é fácil de trabalhar e a criação de casos de teste pode ser realizada em poucos instantes, uma vez que não é necessário um conhecimento prévio em programação para utilizá-lo. Entretanto, a criação de alguns casos de teste para o Selenium-IDE pode ser relativamente complexa, principalmente quando se faz necessária a utilização de lógica de programação. Neste sentido, a melhor alternativa é o Selenium-RC, pois ele tem suporte para criação de *scripts* em diversas linguagens como Java, C#, Perl, PHP, Python, e Ruby. Ainda assim, o Selenium-IDE é uma boa opção para o treinamento de funcionários com pouca experiência.

A segunda ferramenta que compõe a arquitetura do Selenium é o Selenium-RC. É uma ferramenta que permite a execução de casos de teste e, conforme descrito anteriormente, é recomendada para a geração de casos de teste mais complexos. Esta ferramenta é utilizada quando se faz necessária uma análise mais profunda e criteriosa dos casos de teste ou até mesmo a realização de consultas em banco de dados ou outros sistemas integrados.

Para melhor entendimento da ferramenta, um diagrama simplificado de sua arquitetura é apresentado na Figura A.14. O diagrama mostra cada comando do *script* sendo passado ao servidor que em seguida repassa ao Selenium Core, o qual é responsável pela execução dos casos de teste. O Selenium Core é uma ferramenta comum tanto ao Selenium-IDE como ao Selenium-RC, assim como o Selenium-RC permite a execução de *scripts* de teste, porém sua execução pode ser realizada em ambientes diferentes do ambiente de desenvolvimento.

A terceira ferramenta da arquitetura do Selenium é o Selenium-Grid. Ele permite maior escalabilidade dos testes, uma vez que possibilita a execução simultânea de diversos Selenium-RC. Com o Selenium-Grid, várias instâncias do Selenium-RC são executadas em paralelo de forma a simular um ambiente de teste mais próximo do real.

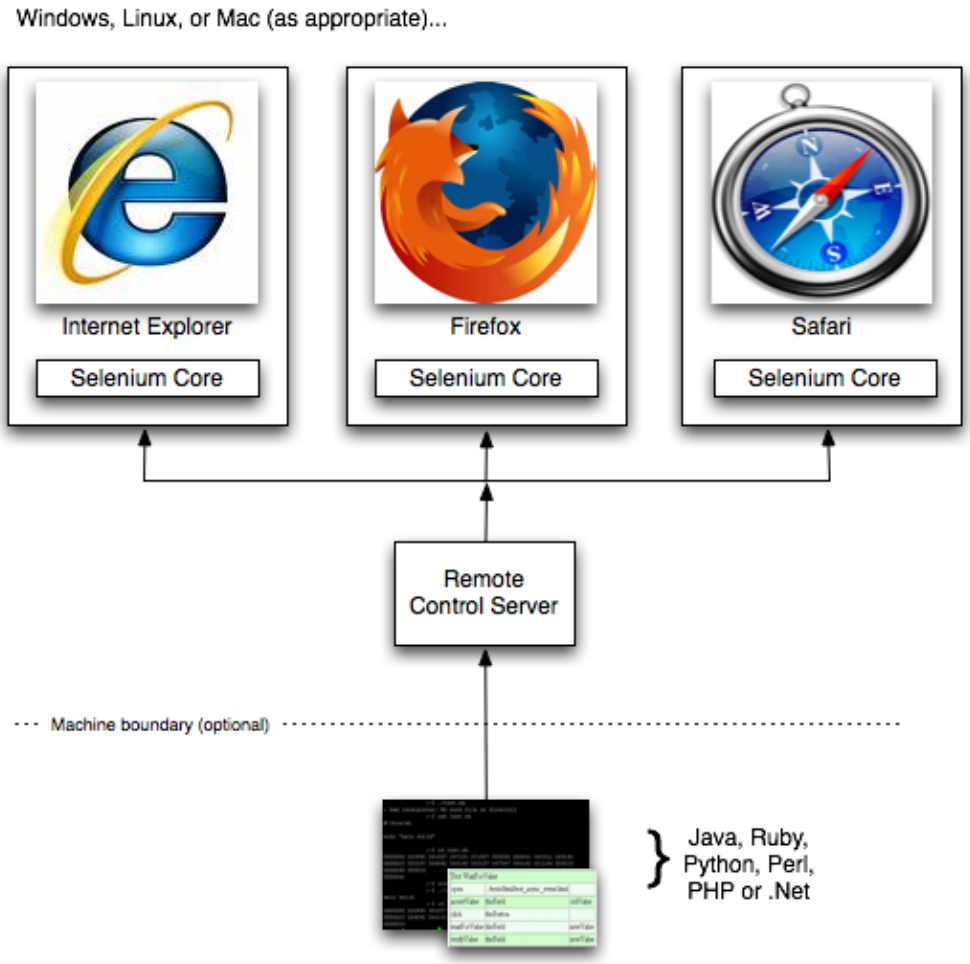

Figura A.14: Diagrama simplificado da arquitetura do Selenium-RC [9]

A.8 JaBUTi

Desenvolvida pelo grupo de Engenharia de *Software* da USP-São Carlos, a JaBUTi [24] [95] (*Java Bytecode Understanding and Testing*) é uma ferramenta utilizada para execução de teste estrutural em aplicações Java. Ao contrário de outras ferramentas de teste estrutural, a JaBUTi é capaz de executar testes sem que seja necessária uma análise do código fonte, pois a execução de testes pela ferramenta é realizada a partir da análise do Java *Bytecode*, um conjunto de instruções expressas em um código binário gerado pelo compilador. O Java Bytecode é uma forma intermediária de código, o qual é interpretado pela JVM (*Java Virtual Machine*), esta que permite que todo e qualquer programa escrito em Java possa ser executado independentemente de plataforma [105].

Para a análise de cobertura das classes e componentes dos programas a serem testados, a Jabuti se baseia em oito critérios de teste estrutural, sendo quatro critérios de análise de fluxo de dados e outros quatro para análise de fluxo de controle [97]. A seguir, são descritos cada um destes oito critérios, a começar pelos critérios referentes à análise de fluxo de controle:

- *Todos os nodos primários (all-primary-nodes):* Todo nó primário tem que ser visitado pelo

menos uma vez. Este critério exige a cobertura de todas as declarações não relacionadas a exceções.

- *Todos os nodos secundários (all-secondary-nodes):* Todo nó secundário tem que ser visitado pelo menos uma vez. Este critério exige a cobertura de todas as declarações relacionadas a exceções.

- *Todas as arestas primárias (all-primary-edges):* Toda aresta secundária tem que ser percorrida pelo menos uma vez. Este critério exige a cobertura de todas as expressões condicionais avaliadas como verdadeiras e falsas não relacionadas ao lançamento de uma exceção.

- *Todas as arestas secundárias (all-secondary-edges):* Toda aresta secundária tem que ser percorrida pelo menos uma vez. Este critério exige a cobertura de todas as expressões condicionais avaliadas como verdadeiras e falsas relacionadas ao lançamento de uma exceção. A seguir, são descritos os critérios para análise de fluxo de dados.

- *Todos os usos primários (all-primary-uses):* Este critério exige a cobertura de todas as definições-uso não relacionadas a exceções.

- *Todos os usos secundários (all-secondary-uses):* Este critério exige a cobertura de todas as definições-uso relacionadas a exceções.

- *Todos os potenciais usos primários (all-primary-pot-uses):* Este critério exige a cobertura de todas as associações potenciais de definições-uso não relacionadas a exceções.

- *Todos os potenciais usos secundários (all-secondary-pot-uses):* Este critério exige a cobertura de todas as associações potenciais de definições-uso relacionadas a exceções.

A definição de um caso de teste para a JaBUTi se inicia por meio da criação de um arquivo de projeto, onde o *bytecode* das classes do programa a ser testado deve ser "carregado". Para exemplificar este processo, são apresentadas nas Figuras A.15a e A.15b as janelas referentes à criação de um projeto de teste, o qual utiliza as classes (*Dispenser* e VendingMachine) de uma típica aplicação de máquina de venda.

Após a criação do projeto, é possível realizar o processo de teste para a JaBUTi. Uma das características desta ferramenta é que ela utiliza diferentes cores para a representação dos requisitos de teste referente a cada um dos critérios de cobertura definidos para ela. Este conjunto distinto de cores representa diferentes pesos, os quais indicam o grau de cobertura atingido. Na Figura A.16 é apresentado o gráfico do fluxo de controle referente ao grau de cobertura de um dos métodos da classe *Dispenser*.

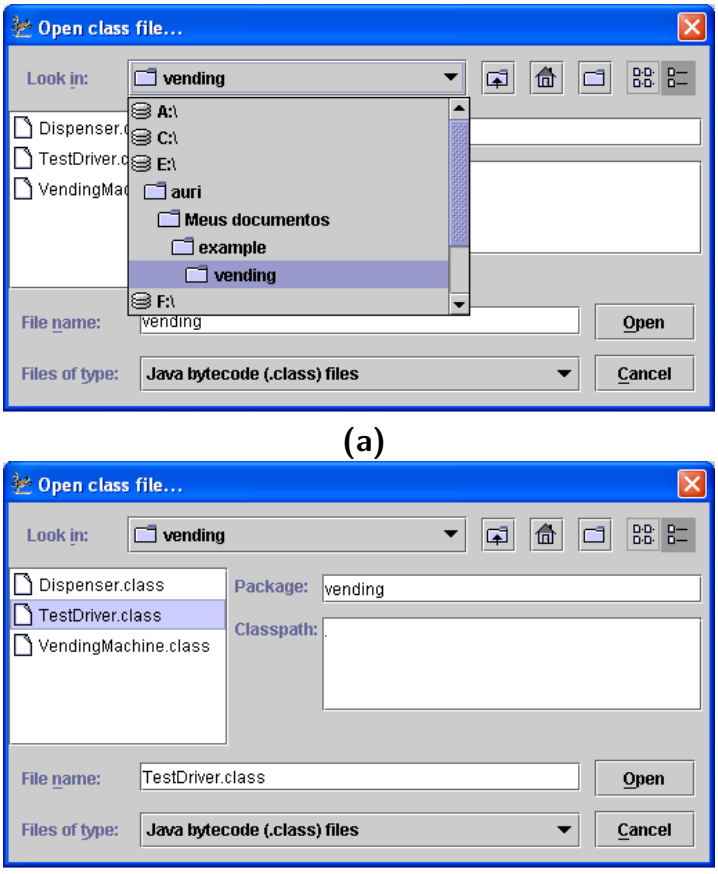

Figura A.15: Selecionando os arquivos *.classs* do projeto *Vending* para criação dos casos de teste [10]

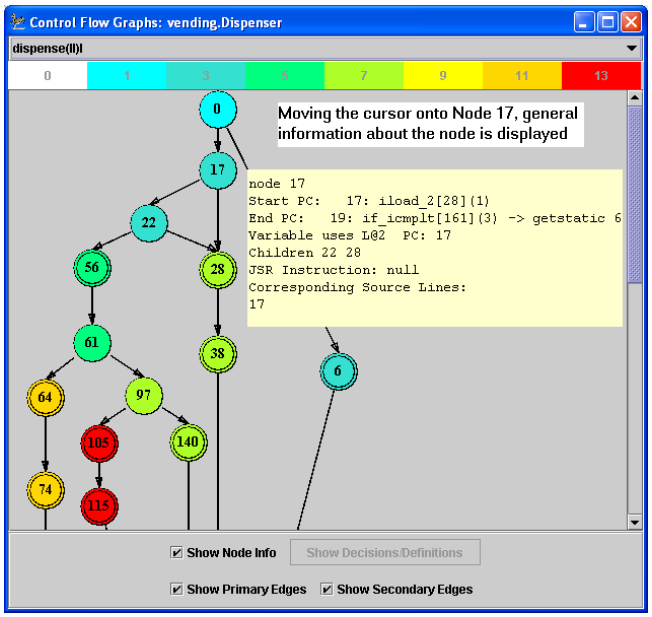

Figura A.16: Gráfico de fluxo de controle [10]

A.9 Borland SilkPerformer

O SilkPerformer [45] é uma ferramenta criada pela Borland e tem como finalidade a criação e execução de testes de desempenho. Assim como o LoadRunner e o RPT, essa ferramenta utiliza a

técnica *Record&Playback* para a criação de seus *scripts* de teste, sendo esses gerados na linguagem *Benchmark Description Language* (BDL), linguagem própria da ferramenta. Além disso, o SilkPerformer tem suporte a diversos protocolos e interfaces de monitoramento das aplicações, tais como: HTTP(S)/HTML, SOAP/XML, streaming media (MS, Real), MAPI, IMAP, SMTP/POP, J2EE, .NET, CORBA, DCOM, Citrix MetaFrame, Oracle Forms, Macromedia Flex/AMF, WAP2, ODBC, OCI, DB2-CLI, ADO, TCP/IP, TN3270E, TN5250, VT100/200+.

O SilkPerformer ainda provê monitoramento dos ambientes de teste por meio do *SilkCentral Performance Manager*, um produto que quando integrado ao SilkPerformer é capaz de gerar relatórios de tendências e características de desempenho [45]. Toda a criação e execução de casos de teste para o SilkPerformer requer que três passos sejam executados: (a) Gravação das ações de usuário sob uma determinada aplicação *web*, através da iteração entre esse usuário e um navegador qualquer; (b) Configuração do cenário de teste, onde características do teste como número de usuários e tempo de teste são parametrizáveis; e (c) Análise dos relatórios gerados a partir da execução do teste.

Diferentemente do LoadRunner, onde toda criação de cenários de teste consiste na execução de duas ferramentas (*Virtual User Generator* para a criação do *script* e *Controller* para a criação do cenário), o SilkPerformer realiza todo esse processo através de um *framework*. Desta forma, a criação de cenários de teste, desde a criação do *script* até a parte de análise de resultados, é realizada em uma única interface e em várias etapas.

Assim como ocorre com a ferramenta de desempenho da HP, para o SilkPerformer é necessária a escolha do parâmetro referente ao protocolo para teste de desempenho *web*, chamado *Web business transaction (HTML/HTTP)* (ver Figura A.17). Após a seleção do protocolo, a gravação do *script* pode ser iniciada, como pode-se observar na Figura A.18.

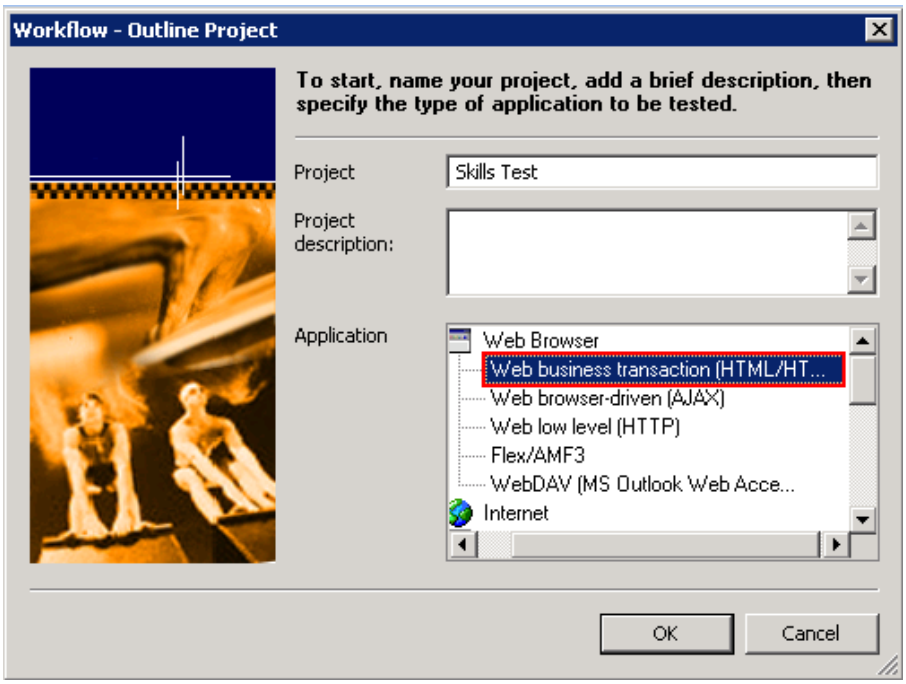

Figura A.17: *Layout* referente aos protocolos disponíveis para o SilkPerformer

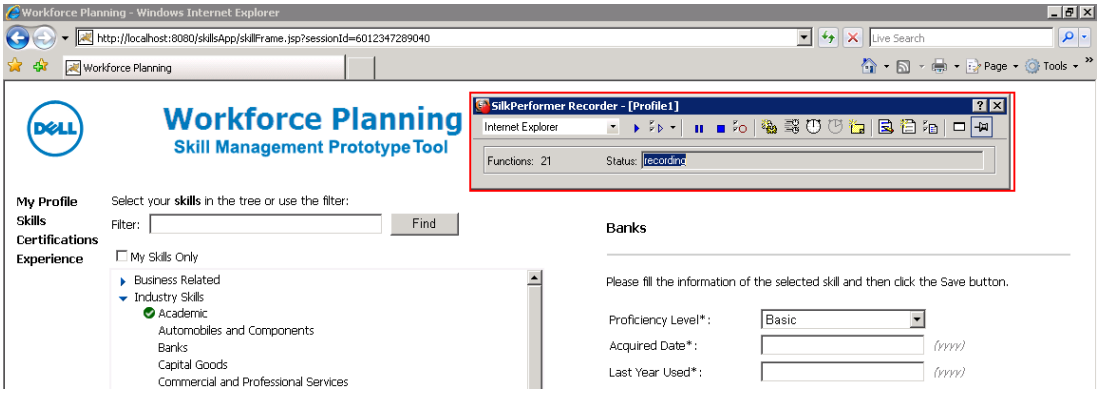

Figura A.18: *Layout* referente à etapa de gravação do *script* de teste para o SilkPerformer

Após o processo de gravação, o *script* é gerado e a etapa de configuração do cenário é iniciada. Com isso, os demais parâmetros necessários para o teste podem ser adicionados. Conforme se observa na Figura A.19, diversos os parâmetros são configurados para o cenário de teste, tais como: *Vusers*, que define o número de usuários que farão requisições ao(s) endereço(s) HTTP configurado(s) para o *script* da ferramenta; *Simulation time*, que corresponde ao tempo de simulação do teste, ou seja, o tempo total do teste; *Ramp-up time*, que define o tempo que levará para que todas os usuários estejam ativos, ou seja, requisitando o endereço previamente configurado; *Warmup time*, que corresponde ao período de tempo em que a ferramenta de teste irá coletar informações referentes aos contadores configurados para o teste, sem que ainda seja realizada carga no sistema; e *Type of workload to run* (perfil *Increasing*), que define o perfil do teste que será executado, por exemplo, usuários iniciam suas iterações gradativamente (perfil *Increasing*), todos os usuários iniciam suas iterações simultaneamente (perfil *Steady State*).

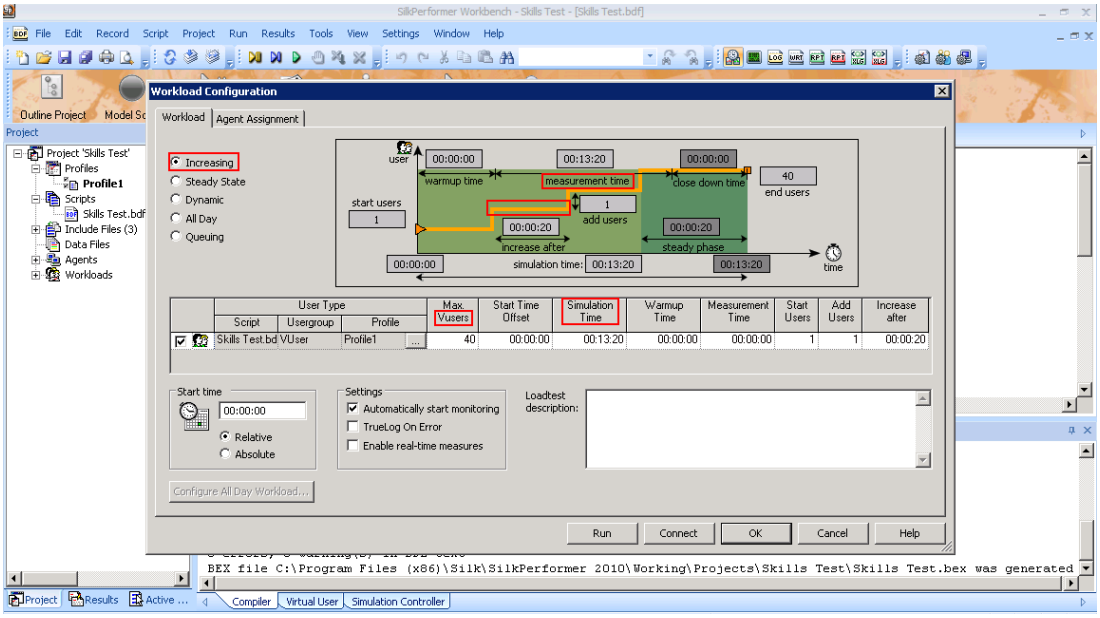

Figura A.19: *Layout* referente à configuração dos parâmetros do cenário de teste do SilkPerformer

Para finalizar a configuração do cenário de teste, ainda se faz necessária a configuração de

alguns contadores necessários para avaliar o desempenho do sistema. Diversos contadores podem ser adicionados para este teste como, por exemplo, utilização de CPU/Memória, número de transações efetuadas com sucesso, tempo de resposta de uma transação ou requisição, entre outras. Estes contadores, assim como os demais, podem ser vistos na Figura A.20.

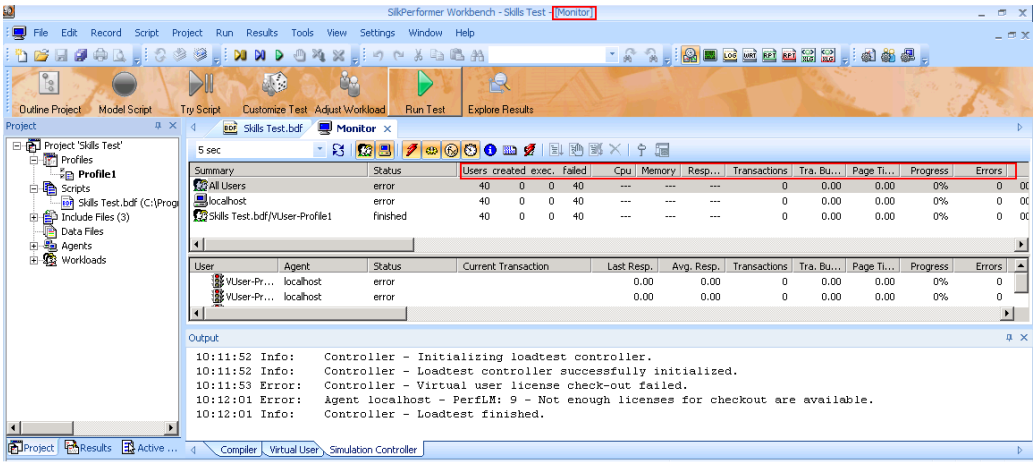

Figura A.20: *Layout* do cenário de teste do SilkPerformer com a adição de contadores de desempenho

A.10 Microsoft Visual Studio

O Visual Studio [23] é um ambiente integrado de desenvolvimento criado pela Microsoft e dedicado ao .NET Framework. Além de possuir um ambiente completo para suporte ao desenvolvimento de aplicações nas plataformas Windows, o Visual Studio possui ferramentas que permitem a geração e execução de diversos tipos de testes como, por exemplo, teste unitário, teste de cobertura de código, entre outros. As versões mais recentes da ferramenta (Visual Studio 2010/Ultimate 2010) possuem dois módulos para a geração e execução de testes de desempenho, são os módulos *Web Test* e *Load Test*.

O *Web Test* é o modulo responsável pela criação dos *scripts* de teste, os quais são gravados utilizando a técnica *Record&Playback*. Os *scripts* de teste do Visual Studio podem ser gerados nas linguagens: Visual Basic [53], C# [47] e C++ [55]. Outras linguagens também são suportadas, no entanto, não há suporte da IDE para executar os testes nessas linguagens, um exemplo é o Visual J# [106].

O módulo responsável pela configuração do cenário de teste é o *Load Test*. Este módulo utiliza os *scripts* gerados pelo *Web Test* para a execução dos cenários. Nele (*Load Test*) são configuradas informações como: número de usuários, tempo de duração do teste, entre outras. A vantagem de se utilizar o módulo *Load Test* do Visual Studio, se comparado com as outras ferramentas, é a possibilidade de executar diversos cenários de teste em paralelo, ou seja, ele permite a execução de várias instâncias do *Load test* ao mesmo tempo.

Para o Visual Studio, a criação de casos de teste consiste na criação de *scripts* e cenários, tal qual é realizado para as ferramentas descritas nas seções anteriores. Entretanto, diferentemente

das ferramentas apresentadas anteriormente, o Visual Studio utiliza dois módulos para a criação e execução de teste de desempenho *web*, são os módulos *Web Test* e *Load Test*, mencionados há pouco. Para facilitar o entendimento, um exemplo referente à seleção do módulo *Web Test* é apresentado (ver Figura A.21).

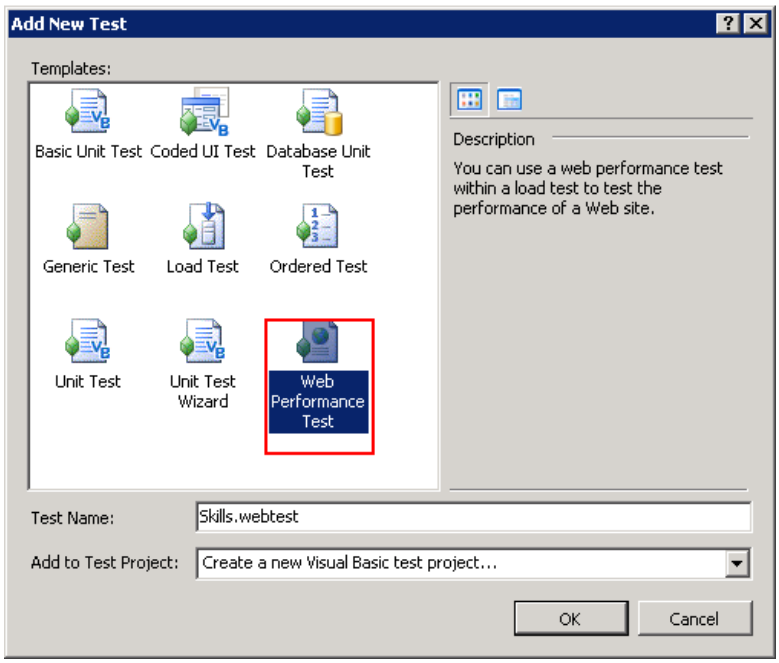

Figura A.21: *Layout* referente aos módulos de teste do Visual Studio

Tão logo o módulo *Web Test* é selecionado, o processo de gravação do *script* é iniciado e, todas as informações referentes aos *links* das páginas navegadas são mostradas durante o processo de gravação (ver Figura A.22). Após o término da gravação do *script*, o cenário de teste pode ser configurado, para isso o módulo Load Test da ferramenta deve ser selecionado (ver Figura A.23).

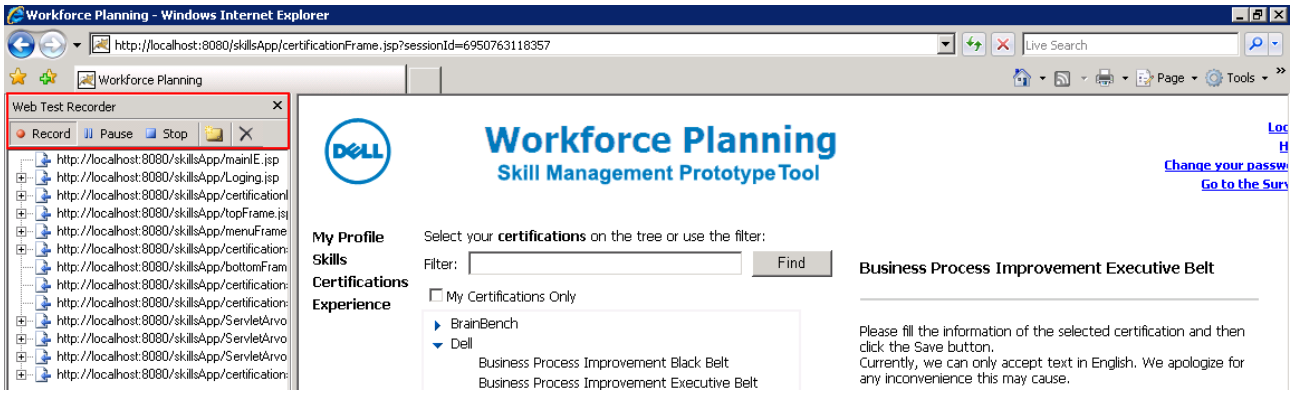

Figura A.22: *Layout* referente ao processo de gravação do Visual Studio

O processo de configuração do cenário de teste para o Visual Studio consiste, inicialmente, na configuração dos parâmetros *Load Pattern*, referente à escolha do perfil de teste e *User Count*, que define o número de usuários que farão requisições à aplicação (ver Figura A.24). Ao final, os

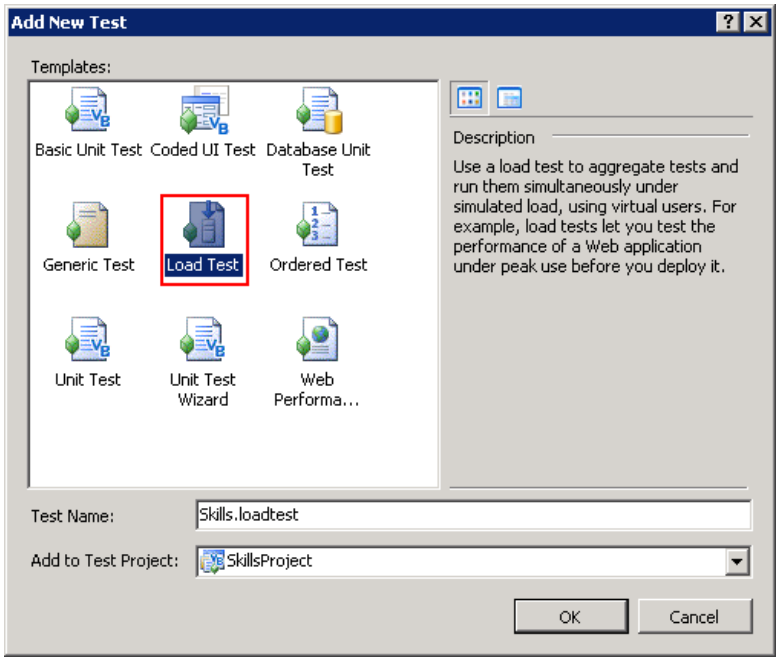

Figura A.23: *Layout* referente à seleção do módulo Load Test do Visual Studio

parâmetros *Warm-up Duration*, que define o tempo necessário para ativar todos os usuários e *Run Duration*, que define o tempo de duração do teste, são também configurados (ver Figura A.25).

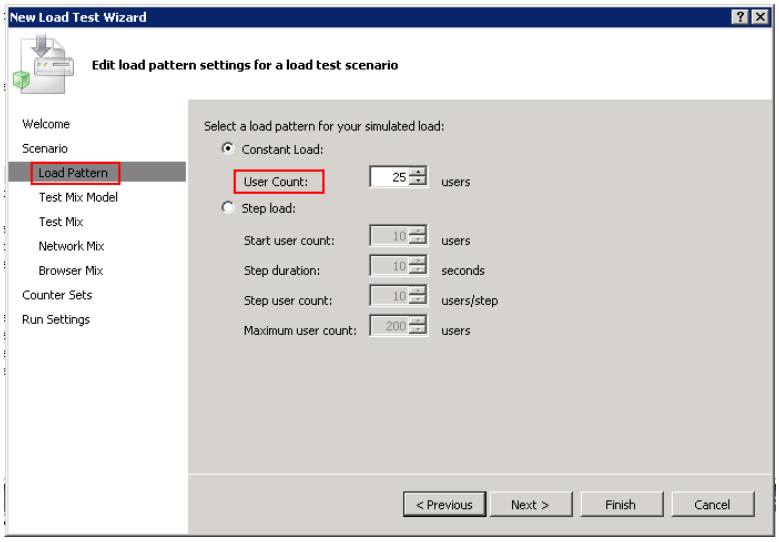

Figura A.24: *Layout* referente à configuração dos parâmetros do cenário de teste do Visual Studio

Para finalizar a configuração do cenário, é necessária a adição de alguns contadores de desempenho. Assim como as demais ferramentas, uma série de contadores podem ser adicionados, como se observa na Figura A.26, onde é apresentado um teste sendo executado.

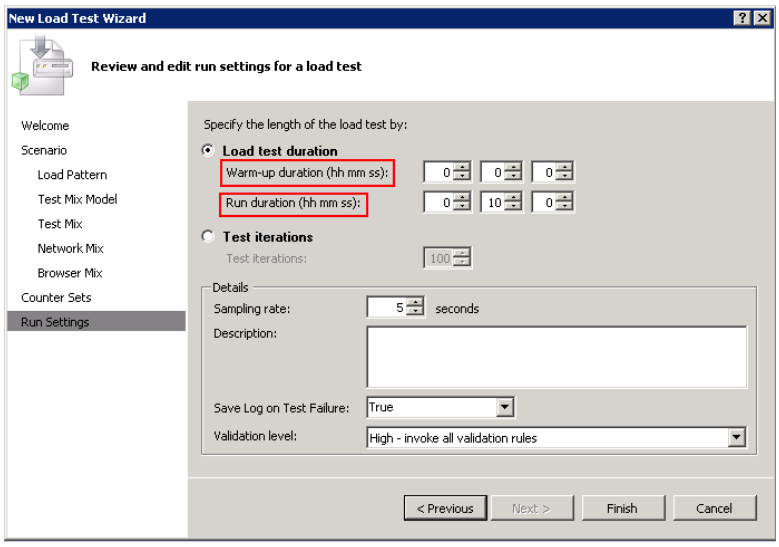

Figura A.25: *Layout* referente à configuração dos parâmetros do cenário de teste do Visual Studio

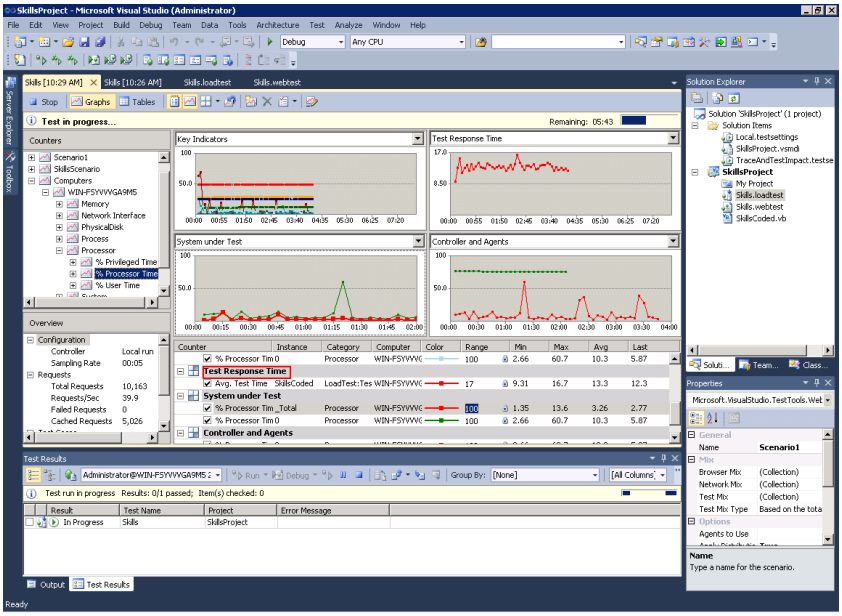

Figura A.26: *Layout* referente à execução do cenário de teste do Visual Studio