AutoTest: Um sistema para testes funcionais e distribuídos de aplicações em ambiente hipermídia

Adriano Camargo Rodrigues da Cunha

Trabalho Final de Mestrado Profissional

Instituto de Computação
Universidade Estadual de Campinas

AutoTest: Um sistema para testes funcionais e distribuídos de aplicações em ambiente hipermídia

Adriano Camargo Rodrigues da Cunha

Dezembro de 2004

Banca Examinadora:

- Profa. Dra. Eliane Martins
  Instituto de Computação – UNICAMP (Orientadora)
- Prof. Dr. Mario Jino
  Faculdade de Engenharia Elétrica e Computação – UNICAMP
- Prof. Dr. Ricardo Anido
  Instituto de Computação – UNICAMP
- Profa. Dra. Claudia Bauzer Medeiros (Suplente)
  Instituto de Computação – UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Cunha, Adriano Camargo Rodrigues da

C914a AutoTest: um sistema para testes funcionais e distribuídos de aplicações em ambiente hipermídia / Adriano Camargo Rodrigues da Cunha -- Campinas, [S.P. :s.n.], 2004.

Orientador: Eliane Martins

Trabalho final (mestrado profissional) - Universidade Estadual de Campinas, Instituto de Computação.

1. Software – Testes. 2. Engenharia de software. 3. World Wide Web (Sistema de recuperação da informação). I. Martins, Eliane. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

AutoTest: Um sistema para testes funcionais e distribuídos de aplicações em ambiente hipermídia

Este exemplar corresponde à redação final do Trabalho Final devidamente corrigido e defendido por Adriano Camargo Rodrigues da Cunha e aprovado pela Banca Examinadora

Campinas, 16 de dezembro de 2004

Eliane Martins

Profa. Dra. Eliane Martins
Instituto de Computação – Universidade Estadual de Campinas (Orientadora)

Trabalho Final apresentado ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Computação na Área de Engenharia de Computação.

Resumo

O teste de software é uma atividade de alto impacto no processo de desenvolvimento de sistemas de grande porte. A automação de parte do teste tem sido vista como a principal medida para melhorar a eficiência desta atividade. Entretanto, o sucesso da aplicação de uma abordagem automatizada depende da utilização de uma estratégia sistemática. Este trabalho final apresenta um sistema para a automação de teste funcional de softwares em ambiente hipermídia, chamado AutoTest, cuja aplicação visa a obtenção de reais ganhos com a automação, baseando-se, para isso, nas melhores técnicas encontradas na literatura. Além disso, são apresentados os resultados da aplicação do sistema criado na automação de teste de três módulos de um sistema de faturamento desenvolvido por uma empresa de telecomunicações.

Abstract

Software testing is an activity with great effect on the development process of large systems. Automation has been seen as the main way to improve testing efficiency. However, the success of an automated approach depends on using a systematic strategy. This dissertation presents a test workbench for the software functional testing automation in hypermedia environments, called AutoTest, based on the best techniques found in the literature, and whose application aims at the achievement of real benefits with the automation. Moreover, the application results of this test workbench on testing automation of three modules of a billing system developed by a telecommunication company are presented.

Em memória do meu saudoso avô Aprígio. Além de uma grande e inigualável pessoa, que esta dissertação inteira não conseguiria descrever, e um exemplo para toda a minha vida, foi o responsável primordial pela minha incursão e perseverança na computação.

Agradecimentos

Agradeço ao meu pai, Hamilton, pelo incentivo constante aos meus estudos e pós-graduação, além de ser sempre um exemplo de seriedade e determinação para mim.

Ao grande profissional, professor, amigo, colega, incentivador, conselheiro e orientador não-oficial, Sindo.

À minha namorada Marilza, pela compreensão, apoio e paciência.

Aos colegas Sueli, André, Marcos, Marcelo, Jayro, Túlio, Ewerton, Vilma, Alexander e Cleida, que direta ou indiretamente me ajudaram na concretização deste trabalho.

À minha orientadora, Eliane, pela credibilidade, ensinamentos, dedicação e profissionalismo, e por ter me oferecido a oportunidade de crescer tanto pessoal quanto profissionalmente.

À Diretoria de Soluções em Billing (DSB), ao Centro de Pesquisa e Desenvolvimento em Telecomunicações (CPqD), que viabilizaram a pesquisa deste trabalho, e ao Fundo para o Desenvolvimento Tecnológico das Telecomunicações (FUNTTEL), que o subsidiou.

Ao Nirvana, Pixies, Capital Inicial e Offspring, pela inspiração.

À ASCII Japan pelo MSX.

Obrigado a todos os que eu não citei, mas que direta ou indiretamente me ajudaram na realização deste trabalho.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução e motivação

Existem várias tentativas no sentido de definir a atividade de teste, desde a visão intuitiva até uma definição formal ([Mye79], [Bei95]). Todas as afirmações, sejam intuitivas ou formais, generalizam uma idéia sobre o que é teste de software e essencialmente conduzem ao mesmo conceito: teste de software é o processo de executar o software de uma maneira controlada com o objetivo de avaliar se o mesmo se comporta conforme o especificado.

O teste de software é uma das principais atividades realizadas para melhorar a qualidade de um produto em desenvolvimento, e seu principal objetivo é revelar a presença de erros no software. O teste de software deve ser efetivo em revelar erros, quaisquer que sejam, mas também deve ser eficiente, sendo executado tão rápido e com tão baixo custo quanto se possa. Por estes motivos, o teste de software tem apresentado progressivamente um maior grau de abrangência e de complexidade dentro do processo de desenvolvimento de software ([Mye79], [Pre92], [Roc01]).

Embora o teste de software seja uma atividade bastante complexa, geralmente ela não é realizada de forma sistemática devido a uma série de fatores como limitações de tempo, recursos e qualificação técnica dos envolvidos. Outros agravantes para a realização desta atividade são a alta complexidade dos sistemas sendo atualmente desenvolvidos e a constante necessidade de sua rápida evolução.

A automação de parte do teste de software tem sido vista como a principal medida para melhorar a eficiência dessa atividade e várias soluções têm sido propostas para esta finalidade. A automação do teste consiste em repassar para o computador tarefas de teste de software que seriam realizadas manualmente.

À primeira vista, automatizar um teste parece fácil: basta comprar uma ferramenta de automação de testes, gravar os testes manuais de alguma forma para, então, executá-los sempre que for necessário. Infelizmente, como várias empresas puderam comprovar, o processo não é tão simples assim. Do mesmo modo que é necessário mais do que simplesmente conhecer uma

linguagem de programação para se criar um software, é necessário mais do que simplesmente conhecer uma ferramenta de automação de testes para se criar testes automatizados.

A automação de testes de software pode reduzir significativamente o esforço exigido para um teste adequado e também aumentar significativamente a quantidade de testes que podem ser executados em um período limitado de tempo. Testes que demoram horas para serem aplicados manualmente podem ser executados em minutos quando automatizados. Economias da ordem de 80% do tempo do teste manual são encontradas na literatura ([Few99]). Em alguns casos o retorno da automação não se revela em economia de custo ou de esforço, mas sim no aumento da qualidade do software testado.

Quando conduzida corretamente, a automação de teste é uma das melhores formas de reduzir o tempo de teste no ciclo de vida do software, diminuindo o custo e aumentando a produtividade do desenvolvimento de software como um todo, além de, conseqüentemente, aumentar a qualidade do produto final ([Hay01]). Estes resultados podem ser obtidos principalmente na execução do teste de regressão, que se caracteriza pelo teste de aplicativos já estáveis que passam por uma correção de defeitos, ou de aplicativos já existentes que são evoluídos pela introdução de novas funcionalidades ([Kan97a]).

Em uma fase já madura, a automação de teste permite que, com um simples clique, se executem testes completos de sistema. Mais ainda, isto pode ser feito em períodos fora do expediente de trabalho, utilizando recursos computacionais que normalmente estão ociosos.

Testes automatizados podem ser repetidos quantas vezes se queira, utilizando-se exatamente as mesmas entradas e executando-se exatamente as mesmas ações, algo que não pode ser garantido no teste manual. A automação de teste permite que mesmo a menor das modificações no software possa ser completamente testada com um mínimo de esforço. Além disso, a automação de teste de software elimina o trabalho repetitivo e entediante dos analistas de teste, permitindo que eles se dediquem a tarefas mais nobres, como criar casos de teste mais abrangentes.

Embora entre os especialistas haja um consenso dos ganhos que podem ser alcançados com o uso de uma boa estratégia de automação de teste, esta é uma área ainda pouco dominada pela indústria de software. Deste modo, as empresas acabam atuando na automação de teste sem a definição de objetivos e expectativas claros e reais e, na maioria das vezes, sem a aplicação de técnicas apropriadas.

1.1 Objetivos

O objetivo principal deste trabalho é propor o AutoTest, um sistema para a automação da execução de teste funcional de software, focando o ambiente hipermídia (a *World-Wide-Web*, ou WWW) – aqui entendido como recurso capaz de unir texto, imagem e áudio em uma mesma estrutura.

O escopo do teste tratado neste trabalho é a execução de teste funcional (também chamado de teste de caixa-preta), no nível de teste de sistema. Assim, a execução de teste estrutural (também chamado de teste de caixa-branca), nos níveis de teste de unidade e de integração, bem como a geração de casos de teste, não serão abordados.

Apesar de focado no ambiente hipermídia, devido à sua arquitetura, o sistema proposto é suficientemente flexível para permitir sua aplicação mesmo em outros ambientes, como o cliente-servidor tradicional.

Uma das vantagens da utilização do sistema AutoTest é que novos projetos de teste de software podem usufruir da infra-estrutura já criada para sua automação. Assim, as atividades específicas ainda necessárias para o novo projeto de teste em questão são apenas a elaboração dos dados e procedimentos de teste e a manutenção do próprio sistema. O objetivo é oferecer um sistema para o engenheiro de automação de teste trabalhar, ao invés de se ter que simplesmente começar o trabalho sempre do ponto inicial, tendo-se que programar tudo o que é necessário e definir novamente os padrões a serem utilizados.

Adicionalmente, o sistema AutoTest não foca apenas o trabalho de criação e manutenção dos testes automatizados, mas também sua execução. Por "execução" não se deve entender simplesmente como o processo mecânico de entrada de dados e verificação de resultados, mas também o trabalho de análise dos erros e falhas detectados e a interação entre a correção dos defeitos da aplicação (ou dos próprios testes automatizados) e a re-execução dos testes.

Atualmente são comercializadas várias ferramentas de automação de teste de software, porém estas apresentam escopo limitado de funcionalidades, as quais são direcionadas a determinadas ações de automação de teste. Diferentemente do sistema AutoTest, a aplicação pura destas ferramentas não oferece garantias de obtenção de melhoria na produtividade e na qualidade de software, aliados a um baixo custo de automação.

1.2 Estrutura da dissertação

No Capítulo 2 é apresentada uma visão geral sobre as atividades de teste e de automação de teste de software, visando oferecer um entendimento da área em que este trabalho está inserido. Para isso são melhor apresentados os objetivos principais do teste de software, conceitos básicos relacionados a essa atividade, as fases de execução do teste de software, as principais técnicas e critérios utilizados no teste de software, as principais diferenças entre as atividades de teste e automação de teste, os problemas comuns e limitações desta última e as principais técnicas de automação de teste funcional de software.

No Capítulo 3 são apresentadas as principais características do ambiente e aplicações hipermídia, além dos métodos de automação de teste apropriados para este contexto, e o sistema AutoTest, com seus requisitos, sua proposta de implementação e alguns componentes candidatos à implementação proposta.

No Capítulo 4 é descrita uma implementação real do sistema proposto, sendo detalhados todos os componentes escolhidos nesta implementação.

No Capítulo 5 são apresentados os resultados da aplicação do sistema proposto em três módulos de um sistema de faturamento. São apresentados os resultados de cada projeto de automação de teste bem como os principais pontos de melhoria identificados no decorrer dos três trabalhos.

No Capítulo 6 é apresentada a conclusão deste trabalho, incluindo sugestões de trabalhos futuros que podem ser conduzidos para melhoria do sistema proposto.

Capítulo 2

Teste e automação de teste de software

O teste de software é uma das atividades essenciais em qualquer processo de engenharia de software, independentemente do paradigma utilizado. De um modo geral, a realização dos testes inicia-se assim que o software é implementado numa forma executável por máquina. Durante essa atividade, o software é testado para que se possam descobrir defeitos de função, de lógica e de implementação ([Pre92]).

Existem várias estratégias que podem ser utilizadas durante a atividade de teste, incluindo a utilização de diferentes técnicas e ferramentas de teste. A escolha de quais delas utilizar depende das características do software sendo desenvolvido. Portanto, apesar da realização propriamente dita do teste ser iniciada somente depois que o código-fonte é gerado, deve existir um planejamento da atividade de teste, a qual deve ser começada no início do processo de desenvolvimento do software.

Embora sejam muito próximas, as atividades de teste e automação de teste são distintas, e exigem habilidades e abordagens diferentes para serem executadas de maneira efetiva. Estas diferenças, quais características de um caso de teste a sua automação afeta, as limitações e problemas mais comuns enfrentados são questões importantes que serão abordadas. Ainda, existem várias técnicas de automação de teste de software que podem ser utilizadas na etapa de execução do teste. Cada uma das técnicas possui características próprias, com vantagens e desvantagens, dependendo do objetivo e do tipo de teste.

Este capítulo está organizado da seguinte forma: a Seção 2.1 apresenta os objetivos do teste e a Seção 2.2 os conceitos básicos; as Seções 2.3, 2.4 e 2.5 apresentam, respectivamente, as fases, técnicas e critérios do teste de software; a Seção 2.6 apresenta as principais diferenças entre teste e automação de teste; a Seção 2.7 comenta os problemas mais comuns na automação de teste; a Seção 2.8 apresenta as limitações da automação de teste e a Seção 2.9, as técnicas de automação de teste funcional mais importantes e utilizadas.

2.1 Objetivos do teste

Como já colocado na introdução, o objetivo principal do teste é revelar a presença de erros no software. Visando atingir este objetivo, o teste deve ser planejado e projetado de modo a definir formas de execução do programa que tenham uma alta probabilidade de detectar erros no software ([Mye79]). Com isso, tem-se que a atividade de teste será realizada com sucesso se ela conseguir descobrir a presença de erros no software ([Pre92]).

Segundo Pressman [Pre92], o objetivo secundário do teste é obter uma boa indicação de confiabilidade e alguma indicação de qualidade do software como um todo. A obtenção desta indicação é possível porque o teste, quando não detecta a presença de erros, demonstra que as funções do software estão aparentemente funcionando de acordo com as especificações.

É importante ressaltar que, por meio da execução do teste, não se pode garantir que um software esteja totalmente correto, pois para a maioria dos softwares o teste não pode mostrar a ausência de erros, mas apenas constatar a sua presença.

2.2 Conceitos básicos

Nesta seção são apresentadas convenções de conceitos básicos da área de teste de software.

- Domínio de entrada: conjunto de valores que podem ser utilizados como entrada para um software;
- Dado de teste: valor, pertencente ao domínio de entrada, que se fornece à execução de um software durante o teste;
- Saída esperada: resultado esperado da execução de um software correto para um dado de teste;
- Caso de teste: par ordenado composto por um dado de entrada e pela respectiva saída esperada;
- Falha: evento notável onde o sistema viola as suas especificações;
- Erro: item de informação ou estado de execução inconsistente;

- Defeito: deficiência algorítmica que pode levar a uma falha se ativada, ou seja, se gera um erro.

2.3 Fases de teste

A atividade de teste pode ser dividida em fases, de modo que, em cada fase, diferentes tipos de defeitos e aspectos do software sejam abordados, culminando no estabelecimento de estratégias adequadas de geração de dados de teste e de medidas de cobertura. De fato, esta divisão da atividade de teste em fases é uma maneira prática de minimizar a complexidade dessa atividade. Essas fases são executadas de forma incremental e complementar, em função das atividades do processo global de engenharia de software ([Roc01]).

De acordo com [Pre92], devem existir quatro fases de teste, que são: teste de unidade – que se concentra na implementação do código-fonte; teste de integração – que se concentra no projeto de software; teste de validação – que se concentra nos requisitos de software; e teste de sistema – que se concentra nos requisitos de sistema. O relacionamento entre as fases de teste e as atividades do processo de engenharia de software é apresentado na Figura 2.1. Cada um desses relacionamentos é explicado a seguir.

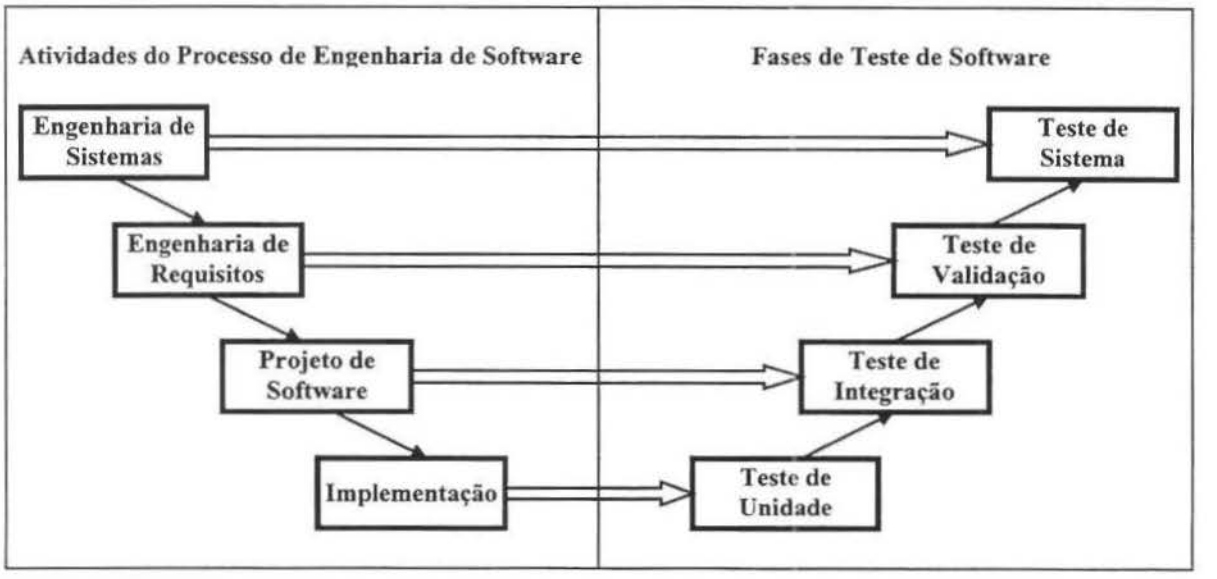

Figura 2.1: Atividades do processo de Engenharia de Software e Fases de Teste

A primeira fase de teste que deve ser executada é o teste de unidade, que tem por objetivo explorar a menor unidade funcional de projeto de software que foi implementada. Segundo o padrão IEEE 610.12-1990 ([Iee90]), uma unidade funcional é um componente de software que não pode ser subdividido. Em programas, uma unidade funcional refere-se a uma sub-rotina ou a um procedimento que é a menor parte funcional de um programa que pode ser executada. Nessa fase o analista de teste deve procurar identificar a presença de erros de lógica e de defeitos de implementação de cada unidade, tentando garantir que cada uma delas funcione adequadamente.

À medida que as unidades vão sendo testadas, pode-se iniciar o teste de integração. Esta fase de teste tem por objetivo explorar as interfaces entre as unidades quando estas são integradas para construir a estrutura do software que foi estabelecida na fase de projeto. Apesar das unidades já terem sido testadas individualmente, o teste de integração é necessário visto que podem surgir problemas na interação entre essas unidades. Nesta fase de teste o analista de teste deve procurar identificar a presença de erros de comunicação entre as unidades, tentando garantir que elas funcionem em conjunto de forma adequada.

Depois que o software estiver totalmente construído, ou seja, todas as unidades tiverem sido integradas, deve-se iniciar o teste de validação. Esta fase de teste tem por objetivo explorar os requisitos estabelecidos como parte da análise de requisitos de software em relação ao software que foi construído. Nesta fase o analista de teste deve procurar identificar falhas, com o objetivo de garantir que o software funcione conforme os requisitos levantados.

Finalmente, depois que o software tiver sido validado, deve-se iniciar o teste de sistema, que tem por objetivo explorar o comportamento do software inserido num contexto mais amplo, contendo, por exemplo, hardware, pessoas, bancos de dados, etc. Nesta fase o analista de teste deve procurar identificar a presença de falhas, tentando garantir que todos os elementos funcionem em conjunto de forma adequada.

Visando a correta execução de cada uma dessas fases, cada uma delas deve envolver quatro etapas básicas – planejamento, projeto de casos de teste, execução e avaliação dos resultados, as quais devem ser conduzidas ao longo de todo o processo de engenharia de software. O planejamento do teste deve fazer parte do planejamento global do sistema, culminando em um plano de teste que constitui um dos documentos cruciais no ciclo de vida de desenvolvimento de software. Nesse documento são estimados recursos e são definidas as estratégias de teste, incluindo as técnicas e ferramentas, a serem utilizadas ([Roc01]).

2.4 Técnicas de teste

A única maneira de se mostrar a corretude de um software seria por meio da execução de um teste exaustivo, ou seja, testar o software com todas as suas combinações de valores de entrada. Entretanto, esta prática é inviável, pois o domínio dos dados de entrada é praticamente infinito ou, pelo menos, muito grande ([Mye79]). Assim, durante o projeto de casos de teste, torna-se necessário selecionar um subconjunto de dados de teste para ser utilizado.

Existem várias técnicas de teste que podem ser utilizadas para a seleção de subconjuntos de dados de teste. Cada uma destas técnicas possui características próprias que tornam sua aplicação mais indicada a diferentes fases de teste. Essas técnicas de teste são agrupadas em classes de técnicas similares. Como exemplos de classes de técnicas de teste pode-se citar o teste funcional e o teste estrutural.

A diferença básica entre as técnicas de teste funcional e as de teste estrutural é a origem das informações utilizadas na seleção dos dados de teste a serem usados, durante a etapa de projeto de casos de teste. O teste funcional leva em consideração os requisitos funcionais do software durante a seleção; por outro lado, o teste estrutural leva em consideração a estrutura interna do software ([Pre92]).

De acordo com [Roc01], um ponto que deve ser ressaltado é que a aplicação dessas técnicas de teste detectam a presença de erros de categorias distintas, uma vez que elas contemplam diferentes perspectivas do software. Deste modo, impõe-se a necessidade de se estabelecer uma estratégia de teste que contemple as vantagens e os aspectos complementares de cada uma das classes, levando a uma atividade de teste de boa qualidade, eficaz e de baixo custo.

Outro exemplo de classe de técnica de teste é o Teste Baseado em Erros, que estabelece os requisitos de teste explorando os erros típicos e comuns cometidos durante o desenvolvimento de software ([Roc01]). Embora as técnicas de Teste Baseado em Erros sejam importantes e bastante pesquisadas, elas não serão tratadas nesta dissertação. A seguir encontra-se uma breve descrição do teste estrutural e do teste funcional.

2.4.1 Teste estrutural

O teste estrutural – também chamado de teste de caixa branca – estabelece os elementos requeridos com base em uma certa implementação, solicitando a execução de partes ou de componentes elementares do programa ([Pre92], [Mye79]).

Usando técnicas de teste estrutural, o analista de teste deriva casos de teste a partir de um exame de elementos da estrutura interna do programa. Embora a funcionalidade do programa seja usada para se determinar qual é a saída esperada para uma dada entrada, a escolha dos dados de teste é realizada olhando-se dentro da "caixa" e escolhendo-se dados de teste para exercitar os elementos requeridos.

Os tipos de defeitos que podem ser revelados por meio do teste estrutural são ([Pre92]): defeitos lógicos, pressuposições incorretas, defeitos de projeto e defeitos tipográficos.

As técnicas estruturais são mais utilizadas nas primeiras fases do processo de teste – teste de unidade e de integração. O teste de unidade faz muito uso das técnicas estruturais para exercitar caminhos específicos da estrutura de controle de um módulo, a fim de garantir uma completa cobertura e máxima detecção da presença de erros. Já o teste de integração faz um menor uso das técnicas estruturais, usadas nessa fase para tentar garantir a cobertura de caminhos importantes de controle entre os módulos integrados ([Pre92]).

2.4.2 Teste funcional

O teste funcional – também chamado de teste de caixa preta – estabelece os elementos requeridos com base na especificação, não utilizando conhecimento algum sobre a implementação. As técnicas desta classe utilizam como base a especificação, ou seja, os requisitos funcionais, para validar o próprio software ([Pre92], [Mye79]).

Por meio da realização do teste funcional, o software ou sistema é tratado como uma "caixa preta": ele é executado com determinadas entradas e suas saídas são verificadas em relação ao comportamento definido por meio da especificação de software. O analista de teste deve estar preocupado somente com a funcionalidade e com características do software, e seus detalhes de implementação não devem ser levados em consideração ([Bei90]).

O teste funcional procura descobrir a presença de defeitos nas seguintes categorias ([Pre92]): funções incorretas ou ausentes, defeitos de interface, defeitos nas estruturas de dados ou no acesso aos bancos de dados externos, defeitos de desempenho e defeitos de inicialização e término.

Ao contrário das técnicas estruturais, as quais são mais utilizadas nas primeiras fases do processo de teste, as técnicas funcionais tendem a ser aplicadas durante as demais fases do processo de teste – teste de integração, teste de validação e teste de sistema. O teste de integração, além de um pouco das técnicas estruturais, faz mais uso das técnicas funcionais. Por outro lado, o teste de validação e o teste de sistema fazem uso exclusivamente das técnicas funcionais ([Pre92]).

2.5 Critérios de teste

De um modo geral, um critério de teste é algum método ou diretriz que serve para direcionar a atividade de teste e/ou tomar decisões relativas ao teste. Um critério de teste define um conjunto de condições que devem ser utilizadas na atividade de teste.

Os critérios de teste podem ser utilizados de duas maneiras:

- Critério de seleção (ou critério de geração): quando o critério é utilizado para selecionar um conjunto de dados de teste;
- Critério de adequação (ou critério de cobertura): quando o critério é utilizado para avaliar a qualidade de um conjunto de dados de teste.

Assim, pode-se dizer que um critério de teste serve para selecionar e/ou avaliar casos de teste de forma a aumentar a probabilidade de se revelar a presença de erros ou, quando isso não ocorre, estabelecer um nível elevado de confiança na corretude do produto. Formalmente, um critério de teste define qual é o conjunto de elementos requeridos do software que deve ser exercitado ([Roc01]).

A partir de um conjunto de casos de teste pode-se realizar uma análise de cobertura, que consiste em determinar o percentual de elementos requeridos que foram exercitados pelo

conjunto de casos de teste. Com essas informações, o conjunto de casos de teste pode ser melhorado para que os elementos ainda não abordados sejam testados com a adição de novos casos de teste ([Roc01]).

Existem vários critérios de teste associados às diferentes técnicas de teste. Cada um desses critérios possui vantagens e desvantagens, as quais têm sido avaliadas por meio de estudos teóricos e empíricos. A seguir, encontra-se um resumo das principais técnicas de teste – do teste estrutural e do teste funcional – juntamente com os principais critérios para cada técnica.

2.5.1 Critérios para o teste estrutural

As principais técnicas de teste estrutural são: Teste Baseado em Complexidade, Teste Baseado em Fluxo de Controle e Teste Baseado em Fluxo de Dados. Uma visão geral destas técnicas e de alguns dos critérios associados a elas é descrita a seguir.

Teste Baseado em Complexidade

Essa técnica de teste estrutural utiliza informações sobre a complexidade do software para determinar os elementos requeridos. Um dos critérios dessa técnica é o Critério dos Caminhos Básicos, que utiliza uma medida da complexidade do software – chamada Complexidade Ciclomática – para derivar o conjunto de casos de teste. O valor calculado da Complexidade Ciclomática define o número de caminhos independentes existentes no grafo do programa, que são caminhos através do grafo que introduzem pelo menos um novo conjunto de instruções de processamento ou uma nova condição. Este valor oferece um limite máximo para o número de casos de teste que deve ser projetado e executado para garantir que todas as instruções sejam exercitadas pelo menos uma vez ([Pre92]).

Teste Baseado em Fluxo de Controle

Essa técnica de teste estrutural utiliza informações sobre o controle de execução do programa, como comandos ou desvios, para determinar os elementos requeridos. Os critérios mais conhecidos dessa técnica são Todos-os-nós, Todos-os-arcos e Todos-os-caminhos, os quais exigem, respectivamente, que cada nó, cada arco e cada caminho do grafo do programa seja executado pelo menos uma vez durante o teste ([Pre92], [Bri01]).

Teste Baseado em Fluxo de Dados

Essa técnica de teste estrutural utiliza informações sobre as variáveis do programa, como definições e usos das mesmas, para determinar os elementos requeridos ([Bri01]). Os critérios mais conhecidos dessa técnica são os critérios de Rapps & Weyuker, que basicamente exploram associações entre os nós do grafo do programa em que existe uma definição de variável e os nós em que existe um uso da mesma variável. Um exemplo é o critério Todos-os-usos, que exige que pelo menos um caminho entre todas as associações definição-uso de cada variável seja executado pelo menos uma vez. Um conjunto de critérios similares aos de Rapps & Weyuker são os critérios de Potenciais Usos. A diferença é que esses exploram associações entre os nós do grafo do programa em que existe uma definição de variável e os nós em que existe um possível uso dessa variável.

2.5.2 Critérios para o teste funcional

As principais técnicas de teste funcional são: Teste de Particionamento de Equivalência, Teste de Análise de Valores Limites, Teste de Grafo de Causa-Efeito e Teste Baseado em Modelos ([Pre92], [Mye79], [Bri01]). Uma visão geral dessas técnicas e de alguns dos critérios associados a elas é descrita a seguir.

Teste de Particionamento de Equivalência

Essa técnica de teste funcional utiliza informações sobre os dados de entrada do software para determinar os elementos requeridos. Esta técnica baseia-se na hipótese de que o domínio de entrada do programa pode ser dividido em classes de equivalência, cujos elementos possuem a mesma capacidade de detecção da presença de erros. Embora esta hipótese não possa ser garantida, existem diretrizes que podem ser usadas visando a definição de classes de equivalência com esta característica. Uma classe de equivalência representa um conjunto de estados válidos ou inválidos para condições de entrada. Tipicamente, uma condição de entrada é um valor numérico, um intervalo de valores, um conjunto de valores relacionados ou uma condição booleana. Um critério associado a essa técnica consiste em exigir que os casos de teste utilizem pelo menos um dado de entrada de cada classe de equivalência.

Teste de Análise de Valores Limites

Essa técnica de teste funcional complementa a técnica de Teste de Particionamento de Equivalência. A diferença principal do critério dessa técnica é que, em vez de exigir que os casos de teste utilizem qualquer elemento das classes de equivalência, esse critério exige que os casos de teste utilizem dados de entrada pertencentes aos limites de cada classe de equivalência. Como os dados de teste são obtidos a partir dos limites tanto das classes com dados válidos como das classes com dados inválidos para um mesmo parâmetro de entrada de dados, então existirão casos de teste que exercitam a violação dos limites especificados para cada parâmetro. Outra diferença é que, em vez de criar classes de equivalência com base somente no domínio de entrada, também são criadas classes de equivalência de saídas, ou seja, com base no domínio de saída. Esse critério é importante pois, segundo [Pre92], os erros costumam ocorrer com maior freqüência nos limites dos domínios de entrada.

Teste de Grafo de Causa-Efeito

Essa técnica de teste funcional também utiliza informações sobre os dados de entrada do software, verificando o efeito combinado dos dados de entrada sobre sua execução. Primeiramente, são identificadas as causas (condições de entrada) e os efeitos (ações) do software, de acordo com sua especificação. Em seguida, é construído um grafo de causa-efeito, combinando as causas e os efeitos identificados para o software, por meio de operadores lógicos ("e", "ou", "não") e operadores de restrição ("exclusivo", "inclusivo", "somente um", "exige e mascara"). Depois o grafo de causa-efeito é então convertido em uma tabela de decisão. Um critério associado a essa técnica consiste em exigir que os casos de teste coloquem em prova cada uma das regras da tabela de decisão.

Teste Baseado em Modelos

Essa técnica de teste funcional utiliza informações sobre o comportamento funcional do software, descrito por meio de um modelo comportamental, para determinar como será realizado o teste. O modelo comportamental, construído usando os requisitos funcionais do software, determina quais ações são possíveis em sua execução e quais saídas são esperadas. Essa técnica é bastante ampla, visto que existem várias formas de se modelar o comportamento do software, e para cada técnica de modelagem podem existir diferentes critérios associados. Exemplos de

critérios dessa técnica são Todos-os-estados, Todas-as-transições e Todos-os-caminhos, os quais exigem, respectivamente, que cada estado, cada transição e cada caminho do modelo comportamental do programa seja executado pelo menos uma vez durante o teste.

2.6 Teste e automação de teste

Como já visto, um teste exaustivo é impossível de ser feito: o analista de teste deve ser capaz de selecionar um subconjunto de todos os testes possíveis que possa ser não só executado em um tempo razoável, como também seja capaz de encontrar a maior parte dos erros do software.

Selecionar "bons" casos de teste não é uma tarefa fácil. Um bom caso de teste deve possuir quatro atributos de qualidade[1] ([Few99]): eficácia, exemplaridade, economia e manutenabilidade (em inglês, *effective*, *exemplary*, *economic* e *evolvable*).

Um caso de teste é eficaz se ele realmente encontra erros no software ou se, pelo menos, possui alta probabilidade de encontrá-los.

Um caso de teste é exemplar se ele agrega em si verificações que, de outra maneira, necessitariam da existência de outros casos de teste. Em outras palavras: um caso de teste é exemplar se ele reduz o número total de casos de teste necessários.

Finalmente, as últimas duas qualidades de um bom caso de teste são: se ele é econômico em termos de recursos para ser executado, analisado e depurado; e quão fácil é a sua manutenção a cada evolução do software sob teste.

Se um teste é automatizado ou executado manualmente isso não afeta sua eficácia ou exemplaridade, mas sim, sua economia e manutenabilidade. Uma vez implementado, um teste automatizado é, em geral, muito mais econômico, sendo o custo de sua execução uma mera fração do esforço manual correspondente. Entretanto, em geral, testes automatizados custam mais para serem criados e mantidos. Quanto melhor a estratégia de automação de teste, mais barata será a sua implementação a longo prazo. Se a questão da manutenção não for considerada, o trabalho de atualizar um conjunto inteiro de casos de teste automatizados pode custar tanto quanto, ou mais que, executar todos eles manualmente.

---

[1] tradução livre empregada pelo autor

A Figura 2.2 mostra os quatro atributos de qualidade de um caso de teste em um gráfico ([Few01]). A linha cheia representa um caso de teste executado manualmente. Quando este mesmo teste é automatizado pela primeira vez, suas qualidades de econômico e manutenível caem, já que foi despendido esforço para a sua automação. Quando este caso de teste é executado automaticamente várias vezes, ele se torna muito mais econômico que se fosse executado manualmente.

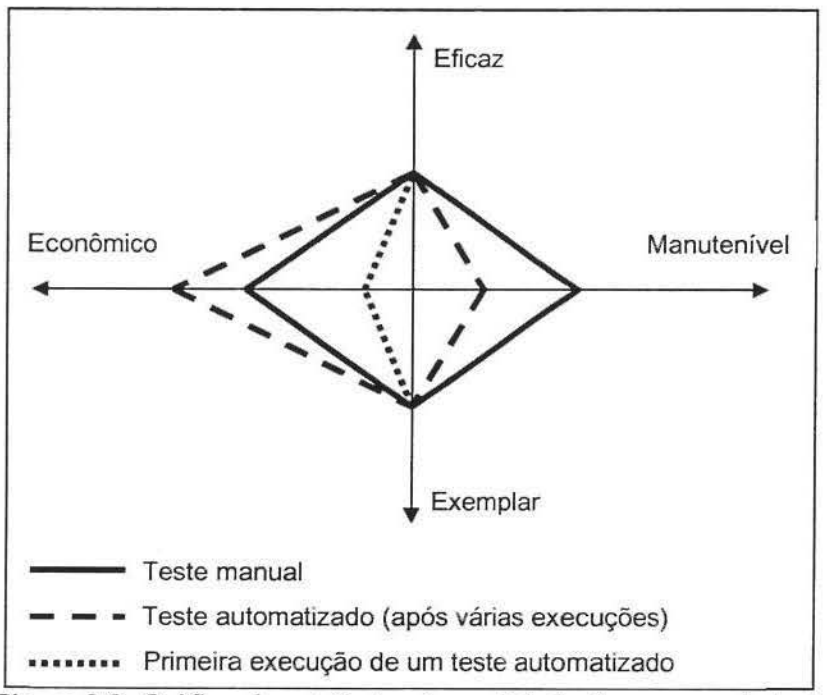

Figura 2.2: Gráfico dos atributos de qualidade de um caso de teste

É possível ter-se um teste de boa ou má qualidade: a habilidade do analista de teste que determina esta qualidade. E também é possível ter-se uma automação de boa ou má qualidade: é a habilidade do engenheiro de automação de teste que determina a facilidade em se adicionar novos testes automatizados, a facilidade de manutenção que o conjunto terá e, no final das contas, quais os benefícios que a automação trará.

Automatizar testes também é uma tarefa que requer habilidade, mas um tanto diferente daquela necessária para testar. Muitas empresas descobrem tardiamente, estarrecidas, que é mais caro automatizar um teste do que executá-lo manualmente ([Kan97a]). Para se obter benefício com a automação de teste é necessário que os testes a serem automatizados sejam escolhidos e implementados cuidadosamente.

2.7 Problemas comuns na automação de teste

Há inúmeros problemas que podem ser encontrados na tentativa de automatizar um teste. [Few99], [Pet01], [Dus99] e [Hen98] citam vários destes. Problemas inesperados são em geral mais difíceis de resolver, por isso ter uma idéia dos tipos de problemas que podem ser enfrentados pode ajudar bastante no trabalho de automação de teste.

Expectativas irreais

É uma atitude um tanto comum nas empresas investir em automação na esperança de que ela resolva os problemas de teste ([Hay01a], [Kan98]). Há uma tendência de superotimismo em relação ao que pode ser feito com a automação de teste. Este otimismo somado às promessas das ferramentas de automação de teste pode gerar expectativas irreais. E, se as expectativas são irreais, então não importa o quão bom o processo de automação de teste seja do ponto de vista técnico, pois ele nunca atenderá as expectativas.

Inexperiência em testes

Se a experiência em testes é pouca, com testes mal organizados, pouca documentação ou documentação inconsistente e testes ineficientes, a automação não é uma boa idéia. Neste caso é muito melhor aumentar a eficácia dos testes que aumentar a economia de testes ruins.

Expectativa de que testes automatizados descobrirão vários erros novos

Um teste provavelmente encontra um erro na primeira vez em que é executado. Se um teste já foi executado com sucesso, executá-lo de novo provavelmente não irá ocasionar a descoberta de um novo erro, a menos que este teste exercite um trecho de código que foi modificado, tenha sido afetado por uma modificação em outra parte do software ou esteja sendo executado em um ambiente diferente. Testes automatizados em geral são testes de regressão. Isto é algo muito útil, mas não é a melhor estratégia para se revelar novos defeitos, particularmente quando a execução se dá no mesmo ambiente de software e hardware.

Falso senso de segurança

Apenas por que um conjunto de testes foi executado sem encontrar nenhum erro não quer dizer que estes inexistem no software. Os testes podem estar incompletos ou podem eles mesmos possuir defeitos. Se os resultados esperados estão incorretos, os testes automatizados simplesmente preservarão estes resultados errados indefinidamente.

Manutenção dos testes automatizados

Quando um software muda, quase sempre se faz necessário atualizar um, ou mesmo todos, os casos de teste de modo que eles possam ser reexecutados com sucesso ([Kan98]). Isto é particularmente verdade para testes automatizados. O esforço de manutenção do teste automatizado é a causa do cancelamento de muitas iniciativas de automação. Quando se gasta mais esforço na atualização do teste automatizado do que na sua reexecução manual, a automação é abandonada.

Problemas técnicos

As ferramentas de automação de teste comerciais são produtos de software. E, como software, elas não estão imunes a erros ou problemas de suporte. Somado a isso existem ainda os problemas técnicos com o software a ser testado. Se o software não for projetado e construído também tendo-se em mente facilitar seu teste pode ser muito difícil fazê-lo, manual ou automaticamente.

Questões organizacionais

A automação de teste não é uma atividade trivial, e por isso precisa de um grande apoio por parte da gerência e ser implantada na cultura da empresa. Deve ser planejada com cuidado, considerando-se tempo para escolha das ferramentas, para treinamento, para experiências e projetos pilotos e para promover o uso da automação de teste na empresa.

2.8 Limitações da automação de teste

Apesar dos benefícios da automação de teste poderem ser grandes, não são infinitos, havendo limitações com as quais deve-se conviver ([Few99]).

Testes automatizados não substituem os testes manuais

Não é possível, e nem desejável, automatizar todos os testes. Sempre haverá testes que são ou muito mais fáceis de se executar manualmente ou tão difíceis de automatizar que seu custo não compensa. Testes que provavelmente não devem ser automatizados incluem: testes que são executados ocasionalmente; testes em softwares que mudam constantemente; testes cujo resultado pode ser facilmente verificado por um ser humano, mas não pela máquina; e testes que envolvem interação física. Nem todos os testes manuais devem ser automatizados – apenas os melhores candidatos ou aqueles que serão reexecutados mais freqüentemente.

Testes manuais encontram mais erros que testes automatizados

Um teste, geralmente, encontra um erro na primeira vez em que é executado. Se um caso de teste vai ser automatizado, primeiro é necessário testá-lo para se ter certeza de que ele está correto, ou seja, que ele é capaz de corretamente revelar os erros a que se propõe caso eles existam. O teste do caso de teste é normalmente feito executando-se o mesmo manualmente. Se o software possui erros que este caso de teste pode revelar, eles o serão neste momento, na execução manual. [Bac97] relata que, em sua extensa experiência, os testes automatizados encontraram apenas 15% dos erros, enquanto que os testes manuais encontraram 85%.

Testes automatizados dependem de uma maior qualidade dos casos de teste

Um teste automatizado pode apenas identificar diferenças entre o resultado atual e o esperado, ou seja, comparar um com outro, por isso há uma necessidade maior de se assegurar a corretude dos resultados esperados. Pode muito bem ser reportado que todos os testes automatizados passaram quando, na verdade, o resultado deles apenas coincide com os resultados esperados que foram definidos, e que podem não espelhar o correto comportamento do software. Portanto, como se torna mais importante a confiança na qualidade e corretude dos resultados dos testes a serem automatizados, eles precisam ser revistos ou inspecionados para se assegurar isto.

A automação de teste não aumenta a eficácia dos casos de teste

Como já foi apresentado, automatizar um teste executado manualmente não torna melhor sua eficácia ou exemplaridade. A automação pode, eventualmente, aumentar a eficiência do teste, ou seja, quanto custa a sua execução e quanto tempo ele demora para ser executado. Mas, provavelmente, a automação terá um efeito adverso na sua manutenção.

A automação de teste pode limitar o desenvolvimento de software

Os testes automatizados são mais "frágeis" que os manuais, ou seja, podem ser invalidados pela mais simples modificação no software testado. As técnicas de automação de teste podem ser utilizadas focando diminuir esta fragilidade, mas os testes automatizados sempre serão mais vulneráveis a mudanças no software que os manuais. Devido ao fato dos testes automatizados exigirem um maior esforço que os manuais para serem criados e atualizados, isto pode acabar por restringir as opções de mudança ou melhoria do software testado. Mudanças no software que causem um grande impacto no conjunto de testes automatizados podem acabar sendo desencorajadas por razões de custo.

Testes automatizados fazem sempre a mesma coisa

Tanto um teste automatizado quanto um analista de teste humano seguem um conjunto de passos, mas um analista de teste humano, na execução do teste, irá segui-los de maneira diferente a cada vez. Diferentemente do teste automatizado, o analista de teste humano pode perceber que os resultados, apesar de coincidirem com os esperados, estão errados, ou pode melhorar os testes à medida em que os executa, seja desviando do procedimento ou, preferivelmente, identificando pontos adicionais de teste. Também, os testadores humanos podem lidar facilmente com eventos inesperados que não fazem parte da seqüência planejada de eventos do teste. Entretanto, um evento inesperado pode interromper um teste automatizado em execução. É claro que os testes automatizados podem ser programados para lidar com alguns tipos de eventos, mas é impossível lidar com todos os casos.

2.9 Técnicas de automação funcional de teste de software

A correta execução de uma fase de teste deve envolver quatro etapas básicas – planejamento, projeto de casos de teste, execução e avaliação dos resultados. O planejamento do teste deve estimar recursos e definir as estratégias de teste a serem utilizadas, incluindo as técnicas e ferramentas de teste ([Roc01]).

Como já apresentado no Capítulo 1, o escopo de teste tratado neste trabalho é a execução de teste funcional, no nível de teste de sistema. Existem várias técnicas de automação de teste funcional de software que podem ser utilizadas na etapa de execução das fases de teste de integração, validação ou sistema, e cada uma das técnicas possui características próprias, com vantagens e desvantagens, dependendo do objetivo e do tipo de teste. Esta seção apresentada uma descrição das duas técnicas de automação de teste mais importantes e utilizadas neste contexto.

2.9.1 A técnica *Capture & Replay* ou *Record & Playback*

Desde o início da utilização da automação de teste funcional em aplicações interativas, a técnica mais óbvia e comum é a que imita o comportamento do analista de teste humano. A idéia mais fácil de se conceber e entender para a automação de teste é a de um robô repetindo as mesmas operações que uma pessoa (mover o mouse, clicar em um botão, apertar uma tecla, etc.).

Várias ferramentas foram desenvolvidas seguindo esta idéia: são as chamadas ferramentas de *Capture & Replay* ou *Record & Playback*. Seu princípio é o mesmo do robô mencionado no parágrafo anterior: a ferramenta grava todos os movimentos e cliques de mouse que o usuário realiza, assim como as teclas que foram apertadas, criando um programa (ou *script*). Uma vez feita esta gravação por um analista de teste humano, tem-se um teste que pode ser repetido quantas vezes se queira, bastando a ferramenta repetir os passos gravados, ou seja, executar o *script* criado.

A vantagem da técnica *Record & Playback* é ser bastante simples e prática, sendo uma boa abordagem para testes executados poucas vezes. Entretanto, são várias as suas desvantagens ao se tratar de um grande conjunto de casos de teste automatizados, tais como: alto custo e dificuldade de manutenção, baixa taxa de reutilização, curto tempo de vida e alta sensibilidade a mudanças no software a ser testado e no ambiente de teste. Como exemplo de um problema desta técnica,

uma alteração na interface gráfica da aplicação poderia exigir a regravação de todos os *scripts* de teste ([Few99], [Few01], [Hen98], [Mar97], [Pet00], [Zam98a]).

2.9.2 A técnica de Programação

A técnica de programação de *scripts* é uma extensão da técnica *Record & Playback*. Como visto na seção anterior, as ferramentas que fazem uso desta técnica geram testes na forma de *scripts*, portanto, por meio de programação estes são alterados para que desempenhem um comportamento diferente do original durante sua execução. Para que esta técnica seja utilizada é necessário que a ferramenta de gravação de *scripts* de teste possibilite a edição dos mesmos. Desta forma, os *scripts* de teste alterados podem contemplar uma maior quantidade de verificações de resultados esperados, as quais não seriam realizadas normalmente pelo analista de teste humano e, por isso, não seriam gravadas. Além disso, a automação de um caso de teste similar a um já gravado anteriormente pode ser feita através da cópia de um *script* de teste e sua alteração em pontos isolados, sem a necessidade de uma nova gravação.

A programação de *scripts* de teste é uma técnica de automação que permite, em comparação com a técnica *Record & Playback*, maior taxa de reutilização, maior tempo de vida, melhor manutenção e maior robustez dos *scripts* de teste. No exemplo de uma alteração na interface gráfica da aplicação, seria necessária somente a alteração de algumas partes pontuais dos *scripts* de teste já criados. Apesar destas vantagens, a aplicação pura desta técnica também produz uma grande quantidade de *scripts* de teste, visto que para cada caso de teste deve ser programado um *script* ([Few99], [Hen98], [Kan97a], [Kan98], [Ter01]).

Várias técnicas e abordagens de desenvolvimento de software podem ser utilizadas para melhorar a eficiência, produtividade e manutenção dos *scripts* de teste, como o uso de *scripts* compartilhados, bibliotecas, código estruturado, orientação a objetos, etc. O uso ou não destes recursos também depende de existir suporte a eles na ferramenta de automação de teste utilizada.

Abordagem orientada a dados

A técnica orientada a dados (ou "*data-driven*") consiste em extrair, dos *scripts* de teste, os dados de entrada e saída, que são específicos por caso de teste, e armazená-los em arquivos separados. Os *scripts* passam a conter apenas os procedimentos do teste (lógica de execução) e as ações sobre a aplicação, que normalmente são genéricos para um conjunto de casos de teste.

Assim, os *scripts* de teste não mantêm os dados no próprio código, obtendo-os diretamente de um arquivo separado, somente quando necessário e de acordo com o procedimento implementado.

A principal vantagem da técnica *data-driven* é que se pode facilmente adicionar, modificar ou remover dados de teste, ou até mesmo casos de teste inteiros, com pequena ou mesmo nenhuma manutenção dos *scripts*. Esta técnica de automação permite que o projetista de teste e o engenheiro de automação trabalhem em diferentes níveis de abstração, dado que o projetista precisa apenas elaborar os arquivos com os dados de teste, sem se preocupar com questões técnicas da automação ([Few99], [Hen98], [Kan97a], [Kan98], [Nag00], [Zam98a]).

Abordagem orientada a palavras-chave

A técnica orientada a palavras-chave (ou "*keyword-driven*") consiste em separar, dos *scripts* de teste, o procedimento que representa a lógica de execução do teste. Os *scripts* passam a conter apenas as ações específicas sobre a aplicação, as quais são identificadas por palavras-chave. Estas ações são como funções de um programa, podendo até mesmo receber parâmetros, que são ativadas pelas palavras-chave a partir da execução de diferentes casos de teste. O procedimento de teste é armazenado em um arquivo separado, na forma de um conjunto ordenado de palavras-chave e seus respectivos parâmetros.

Assim, pela técnica *keyword-driven*, os *scripts* não mantêm os procedimentos de teste no próprio código, obtendo-os diretamente de arquivos em separado. A principal vantagem desta técnica é que se pode facilmente adicionar, modificar ou remover passos de execução no procedimento de teste com necessidade mínima de manutenção dos *scripts*, permitindo também que o projetista de teste e o engenheiro de automação trabalhem em diferentes níveis de abstração ([Few99], [Kan97a], [Kan00], [Kit99], [Nag00], [Zam98a], [Zam98b]).

Exemplos

A fim de ilustrar os diferentes níveis de abstração que podem existir na descrição de um caso de teste, quatro exemplos são apresentados nas Figuras 2.3, 2.4, 2.5 e 2.6.

A Figura 2.3 apresenta a descrição dos passos do caso de teste de forma textual, facilmente compreendido por um analista de testes humano.

A Figura 2.4 apresenta o *script* deste caso de teste, implementado em uma linguagem de programação hipotética. Este *script* pode tanto ter sido criado por um programador como gerado automaticamente por uma ferramenta, pela técnica *Record & Playback*.

A Figura 2.5 apresenta uma descrição textual deste caso de teste e seus dados, separando-os de acordo com o estabelecido pela técnica *data-driven*.

Finalmente, a Figura 2.6 apresenta uma descrição deste caso de teste em função de ações e seus parâmetros, ou seja, na forma utilizada pela técnica *keyword-driven*.

1. Copie todo o conteúdo da base de dados SIS_CAD_TBASE para a base de dados SIS_CAD_T01;

2. Atualize todos os itens da tabela SIS_T_XSW que tenham a coluna ID_GEN contendo o valor 345 para terem a coluna DISP_SN com o valor 1 através do seguinte comando SQL: update SIS_T_XSW DISP_SN=1 where ID_GEN=345.
O usuário e senha para acesso a este banco de dados são, respectivamente, "usuario" e "senha".

3. Execute a aplicação "/sistema/bin/cadastro –t –dbms sis_cad_t01" e espere o processo terminar;

4. Verifique se a tabela RES_XSW contém apenas 3 linhas através do comando SQL:
select * from RES_XSW;

5. Verifique se estas 3 linhas possuem os seguintes valores em suas colunas:

RES_ID	RES_VAL	RES_SN	ID_GEN
1	4	S	345
2	5	N	345
3	6	S	345

Figura 2.3: Descrição Textual Simples do Caso de Teste

```
function main

    call database_copy("SIS_CAD_TBASE", "SIS_CAD_T01")
    if error<>0 then fail("Erro ao copiar base de dados")

    call database_login("SIS_CAD_T01", "usuario", "senha")
    if error<>0 then fail("Erro ao acessar base de dados")

    call database_createsql("update SIS_T_XSW DISP_SN=1 where ID_GEN=345")

    call database_execsql()
    if error<>0 then fail("Erro ao inicializar base de dados")

    call database_commit()

    call system("/sistema/bin/cadastro –t –dbms sis_cad_t01")
    if error<>0 then fail("Erro ao executar processo")

    call database_createsql("select * from RES_XSW")

    call database_execsql()
    results = database_fetch()
    if results<>3 then fail("Erro ao obter resultados")

    data = database_getdata()
    if data(0,0)<>1 or data(1,0)<>4 or data(2,0)<>"S" or data(3, 0)<>345 then
        dump_array_line(data, 0)
        fail("Primeiro resultado errado")
    end if

    if data(0,1)<>2 or data(1,1)<>5 or data(2,1)<>"N" or data(3, 1)<>345 then
        dump_array_line(data, 1)
        fail("Segundo resultado errado")
    end if

    if data(0,2)<>3 or data(1,2)<>6 or data(2,2)<>"S" or data(3, 2)<>345 then
        dump_array_line(data, 2)
        fail("Terceiro resultado errado")
    end if

    call database_close()

end function
```

Figura 2.4: Script de Teste (Implementação do Caso de Teste Automatizado)

Procedimento de Teste – utilizar em conjunto com os dados de teste

1. Copie todo o conteúdo da base de dados da linha A para a base de dados da linha B;

2. Execute a SQL da linha C no banco da linha B, com usuário e senha dados pelos conteúdos das linhas D e E;

3. Execute a aplicação dada pela linha F e aguarde o processo terminar;

4. Execute a SQL dada pela linha G e verifique se o número de linhas retornadas coincide com o indicado na linha H;

5. Verifique se as linhas retornadas possuem os valores dados pelas linhas I, J e K;

	Dados de Teste
A	SIS_CAD_TBASE
B	SIS_CAD_T01
C	update SIS_T_XSW DISP_SN=1 where ID_GEN=345
D	usuario
E	senha
F	/sistema/bin/cadastro –t –dbms sis_cad_t01
G	select * from RES_XSW
H	3
I	1, 4, S, 345
J	2, 5, N, 345
K	3, 6, S, 345

Figura 2.5: Descrição do Caso de Teste de acordo com a técnica *data-driven*

Ação	Objetos da Ação
Copie Banco de Dados	origem=SIS_CAD_TBASE, destino=SIS_CAD_T01
Use Banco de Dados	banco=SIS_CAD_T01, usuario=usuario, senha=senha
Execute SQL	update SIS_T_XSW DISP_SN=1 where ID_GEN=345
Execute Aplicação	/sistema/bin/cadastro –t –dbms sis_cad_t01
Execute SQL	select * from RES_XSW
Verifique Resultado	3
Use Tabela	RES_XSW
Procure Linha	1, 4, S, 345
Procure Linha	2, 5, N, 345
Procure Linha	3, 6, S, 345

Figura 2.6: Descrição do Caso de Teste de acordo com a técnica *keyword-driven*

Capítulo 3

O sistema AutoTest

Apesar de muitas práticas tradicionais de automação de teste poderem ser aplicadas ao teste de aplicações de ambiente hipermídia, há várias questões técnicas que são específicas para este ambiente e que devem ser consideradas. As tecnologias do ambiente hipermídia requerem novos métodos de teste e análise dos erros e falhas, e isso tem impacto na automação de teste. Para uma efetiva automação do teste de aplicações em ambiente hipermídia é necessário entender as diferenças tecnológicas entre a automação de teste de aplicações tradicionais e hipermídia.

Tendo este cenário bem definido e claro, este capítulo apresenta a proposta de um sistema (ou *test workbench* [Som00]) de automação de teste funcional de aplicações hipermídia, chamado AutoTest. Esta proposta tem como objetivo definir um sistema com o qual seja possível facilmente se criar, executar, re-executar e atualizar um conjunto de casos de teste automatizados no ambiente em questão.

Este capítulo está organizado da seguinte forma: a Seção 3.1 apresenta o modelo de aplicação do ambiente hipermídia; a Seção 3.2 explora as diferenças tecnológicas entre os sistemas hipermídia/cliente-servidor e os *mainframes* e computadores pessoais; a Seção 3.3 apresenta a arquitetura dos sistemas hipermídia; a Seção 3.4 os testes automatizados mais apropriados para este ambiente; a Seção 3.5 apresenta os requisitos do sistema AutoTest e a Seção 3.6 a proposta de implementação do mesmo; a Seção 3.7 enumera alguns componentes candidatos à implementação proposta e, finalmente, a Seção 3.8 apresenta algumas considerações sobre a escolha dos componentes.

3.1 O modelo da aplicação

Nos sistemas que utilizam *mainframes* todo o processamento, com exceção da interface com o usuário, é feita pelo próprio *mainframe*. A interface com o usuário ocorre por meio de "terminais burros" (*dumb terminals*), que simplesmente ecoam texto vindo do *mainframe* em um terminal não gráfico e enviam para ele as teclas pressionadas, tudo por meio de uma rede.

Nos computadores pessoais convencionais, todo este processo é consolidado em uma única máquina. Não há necessidade de rede e a interface com o usuário pode tanto ser textual como gráfica. Além dos eventos de teclado, aplicações gráficas também recebem eventos de movimento e cliques do mouse.

Por fim, o modelo cliente-servidor, no qual os sistemas hipermídia se baseiam, requerem a existência de uma rede e de, pelo menos, duas máquinas: um computador cliente e um computador servidor, que fornece dados para o primeiro. A maior parte das aplicações hipermídia utiliza um navegador como interface com o usuário no computador cliente. O modelo cliente-servidor, e conseqüentemente o modelo das aplicações hipermídia, não possui uma separação tão rígida como no modelo dos *mainframes* ou dos computadores pessoais. No modelo cliente-servidor tanto o servidor quanto o cliente podem processar informação, assim como o lado servidor pode estar segmentado em várias máquinas diferentes (exemplos: servidor de aplicações, servidor hipermídia, servidor de banco de dados, etc) ([Ngu01]).

3.2 Diferenças entre o modelo cliente-servidor e hipermídia

3.2.1 Aplicações cliente

A maioria dos sistemas cliente-servidor são aplicações de acesso a dados. Um cliente possibilita que os usuários, através de uma interface, enviem e recebam dados e interajam com os processos do servidor. Os clientes dos sistemas cliente-servidor tradicionais são específicos para cada plataforma computacional, ou seja, para cada plataforma cliente suportada (ex.: Windows 9X, Windows NT/2000, Solaris, Linux, Macintosh, etc) uma aplicação cliente deve ser desenvolvida e testada ([Ngu01]).

A maioria dos sistemas hipermídia também são aplicações de acesso a dados. Os clientes baseados em navegadores hipermídia são projetados para lidar com atividades similares àquelas existentes em um cliente tradicional. A grande diferença reside no fato do cliente estar rodando no contexto de um navegador hipermídia.

Os navegadores hipermídia consistem em um software específico para cada plataforma, executados em um computador cliente. Além de serem capazes de exibir HTML estático, também podem processar conteúdo dinâmico através de *applets* Java, controles ActiveX, CSS,

HTML dinâmico, etc. A diversidade de navegadores e de versões dos mesmos contribui para aumentar as questões de incompatibilidade, dificultando o trabalho de codificação da interface do sistema e também aumentando a necessidade dos testes.

3.2.2 Gerenciamento de eventos

Em interfaces gráficas, em geral, as entradas são orientadas a eventos. Como evento podemos citar movimentos do mouse, cliques ou mesmo a entrada de dados pelo teclado. Alguns objetos (como por exemplo, um botão) podem receber eventos especiais, como o tipo *mouse-over* quando se posiciona o cursor do mouse sobre eles.

As aplicações hipermídia introduzem um novo tipo de suporte a eventos. Como os navegadores hipermídia originalmente foram projetados como ferramentas de apresentação de dados, não havia necessidade de maior interação que não o clique simples de mouse para navegação e submissão de dados e o *pop-up* com descrição textual alternativa dos elementos gráficos. Portanto, controles HTML padrão, como *forms* e hipertextos, são limitados a eventos de clique simples. Entretanto, *scripts* do lado cliente podem ser usados para o reconhecimento de outros tipos de evento, como "duplo-clique" ou "arrastar-e-soltar". Além de não ser natural do ambiente hipermídia, esta abordagem ainda traz o problema de incompatibilidade entre os navegadores ([Ngu01]).

Ainda, as aplicações hipermídia possuem suporte muito limitado aos eventos de teclado. A navegação pode ser feita basicamente com as teclas "Tab" e "Shift+Tab", ativando-se os hipertextos e botões pela tecla "Enter". Atalhos de teclado não estão disponíveis, sendo utilizados para o acesso a funções do próprio navegador.

Mas com a popularização do uso de *scripts* do lado cliente, *applets* Java e *plugins* como Shockwave e controles ActiveX, as possibilidades de interação com o usuário (e, portanto, a quantidade de eventos a serem gerenciados) foram expandidas. Por outro lado, todo o gerenciamento de eventos, no caso de *applets*, *plugins* e controles ActiveX, é feito pelos próprios, e não pelo navegador em si. Com isto tem-se um controle não-padrão a mais no sistema, o que pode tornar mais complexa qualquer tarefa de interação automatizada com o mesmo.

Outra implicação do gerenciamento de eventos das aplicações hipermídia está no modelo de "mão-única" de requisição e submissão: o servidor geralmente não recebe comandos ou dados

enquanto o usuário não clicar em um botão ou hipertexto. Este modelo é chamado de "modelo de submissão explícita" ([Ngu01]). Entretanto, o uso de *scripts* do lado cliente permite exatamente quebrar isso, gerando submissões ou requisições ao servidor sem a intervenção (ou mesmo desejo) do usuário.

3.2.3 Múltiplas instâncias e gerenciamento de janelas

Aplicações convencionais podem dar suporte a múltiplas instâncias delas mesmas, ou seja, a mesma aplicação pode ser carregada na memória várias vezes como processos separados. Da mesma maneira, múltiplas instâncias de um navegador podem estar em execução simultaneamente. Assim, é possível acessar mais de uma vez e ao mesmo tempo, a mesma aplicação hipermídia e os mesmos dados como o mesmo usuário ou como usuários diferentes. Do ponto de vista da aplicação isto pode ser problemático, uma vez que ela pode se perder e confundir dados de um usuário com o outro.

Alguns navegadores possuem, ainda, uma opção chamada *tabbed browsing*, que permite a visualização de páginas hipermídia em uma mesma janela, mas como documentos distintos. Esta abordagem é similar ao modo MDI (*multiple document interface*) usado por várias aplicações padrão.

Finalmente, o uso de *scripts* do lado cliente permite também a criação de janelas em separado da janela original da aplicação. Estas janelas, comumente chamadas de *pop-up*, são muito usadas para seleção de dados, apresentação de resultados temporários ou mensagens de erro. Também fazendo uso de *scripts* do lado cliente elas se comunicam e trocam dados com a janela original da aplicação, podendo também ser fechadas por esta última sem intervenção do usuário.

3.2.4 Controles de interface

Basicamente, uma página HTML exibida por um navegador é composta por texto, hipertextos, imagens, tabelas, *frames*, formulários e textos *pop-up*. Os controles de formulário podem ser botões, imagens, campos de texto simples, campos de texto multi-linhas, listas, caixas de seleção ou *combo-boxes*, *check-boxes* e *radio-buttons*. Os controles de formulário, imagens e

texto (*pop-up* ou no corpo da página) são elementos comuns em aplicações tradicionais, residindo a diferença básica, então, nos hipertextos, tabelas e *frames*.

Entretanto, como já mencionado anteriormente, com a popularização do uso de *scripts* do lado cliente, *applets* Java e *plugins* as possibilidades de interação com o usuário e criação de elementos de interface não padrão (como um botão redondo, por exemplo) foram expandidas fortemente.

3.3 Sistemas hipermídia

A complexidade do modelo cliente-servidor é multiplicada exponencialmente nos sistemas hipermídia. Além do problema decorrente de se utilizar múltiplos clientes, com uma razoável gama de opções de navegadores, tem-se ainda que o lado servidor dos sistemas hipermídia apresenta uma grande variedade de hardware e software, incluindo sistemas operacionais, pacotes de serviços e banco de dados.

Tipicamente, um sistema hipermídia é dividido em três camadas: (1) componentes de interface (cliente), (2) componentes de negócio (servidor) e (3) componentes de dados (servidor). Mas assim como no modelo cliente-servidor tradicional, os sistemas hipermídia também se dividem em dois tipos: sistemas com clientes "magros" (*thin-clients*), em que todo o processamento é feito pelo servidor, e sistemas com clientes "gordos" (*thick-clients*), em que alguns componentes do lado cliente realizam uma parte do processamento ([Ngu01]).

A decisão de se utilizar clientes "magros" ou "gordos" depende fortemente do contexto em que a aplicação hipermídia é utilizada. Um assistente pessoal (ou *palmtop*), por exemplo, não tem a capacidade de processamento de um microcomputador sendo, portanto, não recomendável seu uso como cliente "gordo". Por outro lado, o uso de clientes "magros" exige mais do servidor, sendo um teste de desempenho essencial para garantir a qualidade de serviço (QoS) do sistema. Conseqüentemente, o uso de clientes "gordos" requer que os componentes do lado cliente também sejam testados, especialmente quando se faz uso de *applets* Java, uma vez que navegadores clientes diferentes (ou iguais, mas de diferentes versões) podem fazer uso de máquinas virtuais diferentes ([Ngu01]).

3.4 Tipos de testes automatizados para aplicações hipermídia

Nem todos os tipos de testes automatizados são aplicáveis ao ambiente hipermídia. Há várias questões técnicas que são específicas e devem ser consideradas. Nesta seção elencamos os tipos de testes automatizados mais apropriados para as aplicações hipermídia ([Ngu01], [Few99]).

Testes de sanidade

Testes de sanidade são bons candidatos à automação. Seu objetivo é verificar se a aplicação está estável o suficiente para ser testada, ou seja, se ela não apresenta falhas que impedem ou dificultam sobremaneira a execução dos testes. Nos testes de sanidade não são necessários casos complexos ou que exercitem detalhes muito específicos da aplicação.

Testes simples de aceitação funcional

Testes simples de aceitação funcional têm por objetivo verificar se as funcionalidades-chave do sistema estão funcionando corretamente ao menos em uma configuração mínima. Os casos de teste verificam o funcionamento básico de cada comando do sistema, garantindo que testes mais específicos possam ser feitos depois.

Testes de carga, estresse e desempenho

Testes de carga verificam o comportamento da aplicação sob condições extremas, como grande volume de dados ou processamento excessivo, mas não necessariamente no limite da aplicação.

Já o teste de estresse faz a aplicação operar em condições de recursos limitados. Seu objetivo é garantir que o sistema é capaz de operar corretamente em seu limite e também capaz de lidar com situações de erro de forma correta.

Finalmente, os testes de desempenho têm por objetivo coletar métricas da execução do sistema para verificar se ela está dentro das condições aceitáveis (ou exigidas), para melhorá-la ou para prever limites de hardware ou ambiente.

Estes três tipos de teste, em geral, são demasiado caros ou mesmo impossíveis de se realizar manualmente, sendo bons candidatos à automação.

Testes de regressão

Os testes de regressão são usados para se garantir que defeitos foram realmente corrigidos e que isso não causará o aparecimento de novos erros, ou seja, que os defeitos foram corrigidos e que outras funcionalidades não foram afetadas por isso. Dependendo do projeto os testes de regressão podem ser executados a cada marco ou versão interna atingida. Devido ao alto número de vezes que os testes de regressão são executados, eles são bons candidatos à automação.

Testes não recomendáveis para automação

Alguns tipos de teste não são recomendados para serem automatizados por motivos óbvios, como testes de documentação (verificação da corretude das informações, ausência de erros sintáticos e semânticos, estilo lingüístico, aderência aos padrões de documentação, entre outros), instalação (verificação da correta criação de diretórios e arquivos na máquina destino, configuração inicial do sistema, gerenciamento de erros, indicação de progresso da instalação para o usuário, entre outros), desinstalação (verificação da correta remoção de diretórios e arquivos na máquina destino, gerenciamento de erros, indicação de progresso da desinstalação para o usuário, entre outros), compatibilidade e configuração (se a aplicação opera como esperado em diferentes ambientes de software e hardware), usabilidade de interface (facilidade de uso da interface com o usuário, facilidade de navegação pelo sistema, adequação dos elementos da interface ao padrão visual, entre outros) ou funcionamento correto em outros navegadores (se as funcionalidades do sistema de comportam da maneira esperada em outros navegadores hipermídia). Todos estes devem ser executados manualmente.

Outros tipos não listados nesta seção, como testes funcionais via interface ou testes de adequação de interface, podem ser automatizados, mas é necessário avaliar-se com cuidado o esforço exigido e o ganho esperado.

3.5 Requisitos do sistema AutoTest

Enumerar os requisitos do sistema de automação de teste perfeito pode até ser feito, mas, cumprir estes requisitos não o é. Assim, até pela real necessidade de certos requisitos em diferentes trabalhos de automação de teste, esta lista pode ser bem reduzida. Por exemplo, para

testar uma aplicação que opera em uma máquina isolada, o requisito "verificação de QoS da rede" para o sistema de automação é dispensável.

Coletando e analisando as proposições e idéias apresentadas por vários autores, como [Ngu01], [Few99], [Kan97a], [Kan00], [Kit99], [Nag00] e [Zam98b], chegou-se ao conjunto de requisitos para um sistema para a automação de teste funcional de aplicações hipermídia, chamado AutoTest, detalhados a seguir.

Requisitos para a criação de testes automatizados

- Uso da técnica *Record & Playback*: Como apresentado no Capítulo 2, a técnica *Record & Playback* tem a vantagem de ser bastante simples e prática, sendo uma boa abordagem para testes executados poucas vezes, testes exploratórios em aplicações e mesmo para o treinamento de engenheiros de automação de teste iniciantes;
- Uso de linguagem de programação para a criação de testes automatizados: Como também apresentado no Capítulo 2, scripts de teste criados por programação podem contemplar uma maior quantidade de verificações de resultados esperados, as quais não seriam realizadas normalmente pelo analista de teste humano. Além disso, o uso de programação permite maior taxa de reutilização, maior tempo de vida, melhor manutenção e maior robustez dos scripts de teste;
- Uso das técnicas *keyword-driven* e *data-driven*: Ainda como apresentado no Capítulo 2, o uso das técnicas *data-driven* e *keyword-driven* permite a diminuição da quantidade de *scripts* de teste, melhorando a definição e a manutenção de casos de teste automatizados, além de permitir que o projetista de teste e o engenheiro de automação trabalhem em diferentes níveis de abstração, melhorando e facilitando o trabalho de ambos.

Requisitos de interação com aplicações

- Interação com a aplicação sob teste via interface gráfica: Pela sua própria natureza, testes que interagem diretamente com elementos da interface gráfica da aplicação

são essenciais em um ambiente hipermídia, fazendo deste um requisito chave para testes funcionais;

- Interação com aplicações sem interface gráfica: Como já foi visto, os sistemas hipermídia apresentam duas camadas de responsabilidade do servidor, cujas aplicações são batch, ou seja, não há interação com o usuário, porém uma grande quantidade de processamento é realizada sobre os dados existentes em uma base de dados. Para um completo teste funcional de uma aplicação hipermídia, faz-se necessário também o teste das aplicações batch do servidor. Este contexto é diferente do contexto do cliente, onde há uma interface gráfica, sendo necessário, para a execução dos testes, comunicar-se com o processo batch de outras maneiras, como linha de comando ou via rede.

Requisitos de interação com ambientes

- Interação com outros ambientes: Pesquisas como a da Unisys ([Uni03]) e da Netcraft ([Net04]) confirmam uma conclusão bastante lógica: poucas empresas utilizam um único ambiente (sistema operacional e/ou máquina) para seus servidores, em especial no caso de servidores hipermídia. Portanto, é importante que os testes automatizados possam interagir com outros ambientes que façam parte do sistema hipermídia sob teste;
- Acesso ao banco de dados: O acesso ao banco de dados é essencial para se verificar a consistência entre as informações apresentadas na interface da aplicação sob teste e as persistidas por ela. Ainda, o acesso ao banco de dados permite ao próprio teste automatizado criar situações de teste e satisfazer pré e pós-condições de estado das informações persistidas.

Requisitos de manutenção dos testes automatizados

- Fácil expansão de pontos de verificação: Dada a diversidade de verificações que o teste funcional de uma aplicação pode requerer, e aliado ao fato da contínua evolução e criação das tecnologias voltadas ao ambiente hipermídia, a possibilidade

de se estender ou expandir facilmente os pontos de verificação (seja devido a componentes novos ou proprietários) é outro fator importante. Sem isso corre-se o risco de se limitar em demasia as possíveis verificações de resultados apresentados pela aplicação sob teste;

- Fácil manutenção e criação de dados para uso das técnicas *keyword-driven* e *data-driven*: Para uma efetiva utilização das técnicas *keyword-driven* e *data-driven* é fundamental que os arquivos de dados a serem utilizados possam ser facilmente criados e manipulados pelo projetista de teste. Deve existir também, para não dificultar o trabalho do engenheiro de automação, uma maneira fácil de acessar estes dados nos scripts de teste;
- Abstração dos elementos de interface gráfica: Pelo requisito de interação com aplicações com interface gráfica, os scripts de teste devem ser capazes de reconhecer e manipular componentes da interface gráfica da aplicação sob teste. De alguma maneira, cada componente deve possuir um identificador único, para que seja possível este reconhecimento e manipulação. Não é conveniente que os scripts de teste apresentem estes identificadores espalhados no meio do seu corpo, pois, se a aplicação ou alguma propriedade de um componente da interface é alterada (a cor, por exemplo), o script também deve ser alterado, pois o identificador também se altera. Por isso é necessário haver uma abstração para que todo componente da interface gráfica da aplicação seja referenciado univocamente, e de maneira independente, pelos scripts de teste. Esta abstração entre os identificadores dos componentes da interface gráfica da aplicação e os scripts de teste é chamado genericamente de "Mapa de Interface" e consiste basicamente em uma relação "de-para" entre um nome simbólico do componente e seu identificador. Fazendo uso deste mapeamento e referenciando os elementos da interface gráfica apenas por seus nomes simbólicos, alterações na aplicação sob teste (atualização, inclusão de novas funcionalidades, correção de defeitos, etc.) refletem-se apenas no "Mapa de Interface". Assim, o trabalho passa a ser de manutenção deste mapa, o que pode ser feito facilmente por meio de uma ferramenta, e não de código ou planilhas;
- Interação com sistemas de controle de versão: Uma vez que, assim como a própria aplicação sob teste, os testes automatizados da mesma também evoluem, a

argumentação para o uso de sistemas de controle de versão também é parecida. Outro ponto forte desta interação é permitir que se possa executar testes em versões anteriores do aplicativo ou módulo sem interferir nos trabalhos atuais, e ainda utilizando-se exatamente os testes automatizados criados para a versão específica;

- Uso de base de dados versionadas de referência: Em aplicações que trabalham com banco de dados de grande volume pode ser inviável, por questões de tempo e/ou esforço, povoar o banco com os dados necessários para a realização do teste. Assim, é prática comum manter-se bases de dados "congeladas" e versionadas que contêm todos os dados necessários para a realização dos testes ([Few99]). Esta realidade também se aplica aos testes automatizados: por isso faz-se necessária uma maneira não só prática, rápida e confiável de se usar, manipular, manter e interagir com tais bases "congeladas", mas também automática, de modo que possa ser usada pelos testes automatizados.

Requisitos de execução dos testes automatizados

- Execução remota de testes automatizados: A possibilidade de iniciar testes automatizados remotamente, ou seja, em uma máquina que não a do interessado no teste, não só otimiza o uso dos recursos computacionais, não atrapalhando o trabalho do usuário interessado no teste, mas também permite que qualquer pessoa dispare a execução, dando liberdade aos analistas de teste para o fazerem quando lhes for conveniente;
- Execução em paralelo ou distribuída dos testes automatizados: Sendo possível disparar testes automatizados remotamente, uma conseqüência natural disto é poder disparar testes automatizados em paralelo (seja o mesmo teste, sejam testes diferentes), diminuindo o tempo total necessário para a completa execução dos testes, ou de maneira distribuída, para otimização do uso dos recursos computacionais ou em caso de aplicações específicas.

Requisitos de obtenção de resultados

- Coleta de métricas de cobertura de código durante a execução dos testes: A fim de se ter uma medida quantitativa da qualidade dos casos de teste automatizados, é interessante poder-se coletar métricas de cobertura do código exercitado durante a execução dos testes automatizados.
- Visualização dos erros e falhas detectados após a execução do teste: Apesar de se poder interromper um teste automatizado quando um erro ou falha é detectado, esta não é uma abordagem prática, pois requer a intervenção do analista de teste tantas vezes quantos forem os erros e falhas detectados. Ainda, esta abordagem impede que os testes automatizados sejam executados fora do horário de expediente ou em máquinas que não tenham fácil acesso pelos analistas. Assim, é preciso que todos os erros e falhas detectados durante a execução do teste possam ser visualizados após o término do mesmo, para que então possam ser analisados, otimizando o tempo do analista de teste;
- Relatórios de execução detalhados/resumidos de fácil acesso: Para a análise dos erros e falhas detectados durante a execução do teste automatizado, o relatório de erros deve conter o maior número de informações possíveis, a fim de facilitar o trabalho do analista de teste e não requerer, se possível, a re-execução manual dos casos de teste que falharam. Por outro lado, um relatório resumido dos erros detectados também é útil por permitir uma consulta mais rápida e fácil pelo analista de teste, além de fornecer uma visão geral e instantânea dos resultados do teste executado.

Requisitos de integração

- Integração com o processo de desenvolvimento: Na busca pela qualidade do software desenvolvido é de fundamental importância que os resultados dos testes automatizados sirvam de insumo tanto para fases posteriores (por exemplo, métricas de número de defeitos do software) quanto anteriores à de teste (por exemplo, atividade de *bug tracking* e métricas de reincidência de defeitos) no ciclo de

desenvolvimento. Integrando os testes automatizados com outras ferramentas utilizadas no processo de desenvolvimento, não só este objetivo é atingido, mas também os próprios testes automatizados podem se beneficiar das informações recebidas ao longo da cadeia (por exemplo, impacto nos testes automatizados por alterações em requisitos ou em código da aplicação sob teste). Mais do que simplesmente obter as métricas, esta integração permite obtê-las automaticamente.

3.6 Arquitetura e implementação

Muitos sistemas e ferramentas existentes no mercado atendem vários dos requisitos apresentados, mas não todos. Para isso é necessário criar-se, de alguma forma, um novo sistema. A codificação, a partir do zero, deste novo sistema é uma solução, mas outra solução, mais barata e mais simples de se implementar, é utilizar componentes prontos.

A arquitetura do sistema proposto, o AutoTest, está apresentada na Figura 3.1. Para a implementação de fato do sistema deve-se escolher os componentes necessários para cumprir cada um dos requisitos apresentados. Esta escolha deve feita cuidadosamente, uma vez que quanto mais requisitos forem atendidos por um componente, menos componentes serão necessários para a instanciação completa do sistema.

3.7 Componentes

Nesta seção são analisados alguns componentes candidatos à instanciação do sistema AutoTest.

3.7.1 Ferramentas de automação de teste

Nesta seção são analisadas algumas ferramentas candidatas a instanciar o componente de criação e execução de *scripts* de teste do sistema AutoTest. Quanto mais requisitos forem atendidos pela ferramenta, menos componentes extras serão necessários para a instanciação completa do sistema.

As ferramentas analisadas foram escolhidas dentre as mais populares atualmente, principalmente baseando-se no relatório do IDC de 2003 ([Hei03]) sobre o mercado de ferramentas de automação de teste.

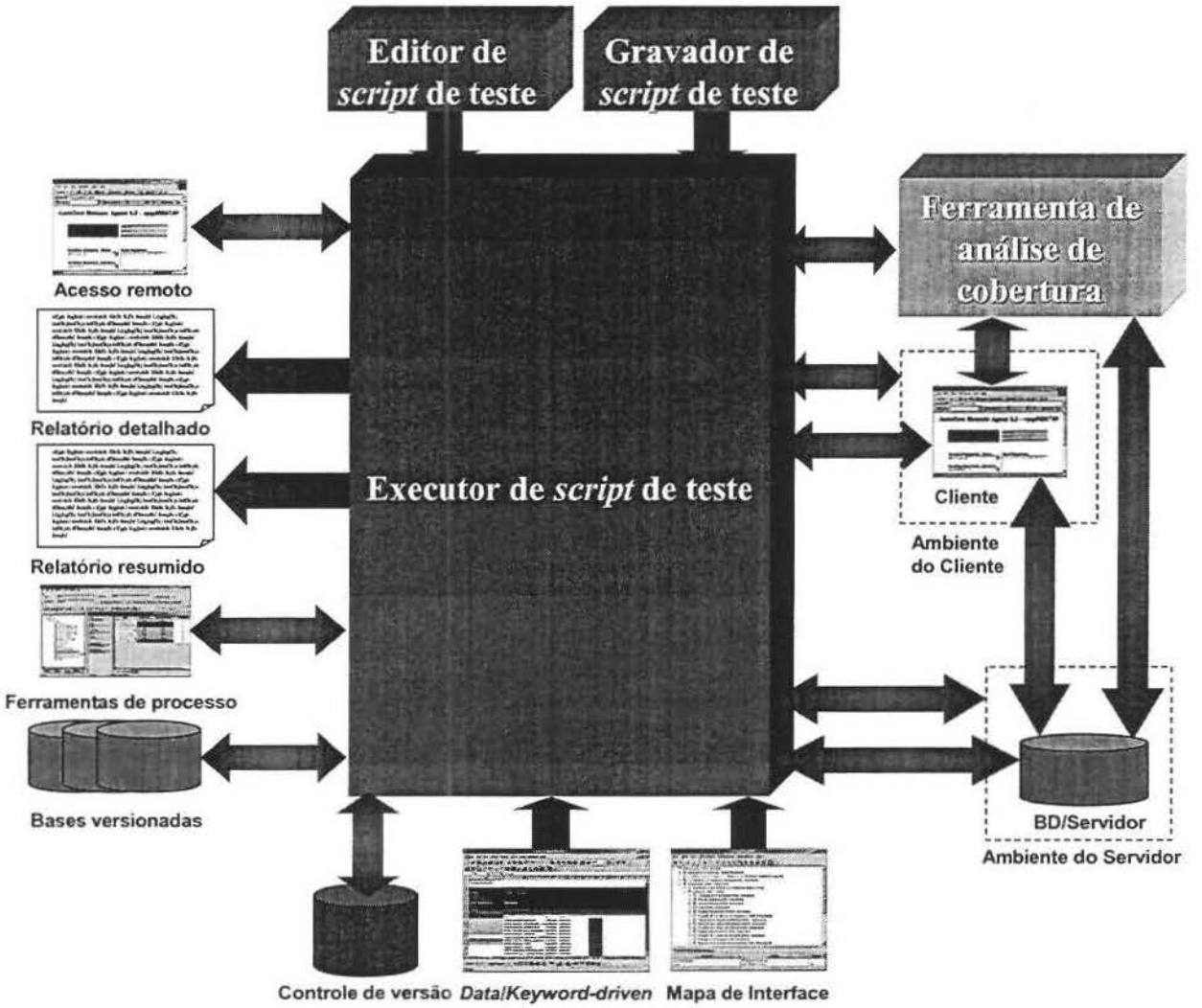

Figura 3.1: Componentes do sistema proposto

É interessante notar que, nativamente, nenhuma das ferramentas analisadas apresenta suporte à técnica *keyword-driven*, mas todas dão suporte à técnica *Record & Playback*. Uma possível explicação para este fato é que os fabricantes focam muito a baixa necessidade de treinamento e a facilidade de criação de testes automatizados com suas ferramentas, o que de certa forma é verdade para a técnica *Record & Playback* mas não para a técnica *keyword-driven*.

e-Tester ([Emp04])

O e-Tester é uma ferramenta com foco na automação de teste funcional de aplicações hipermídia, Java e .NET. Possui suporte nativo à técnica *Record & Playback* de automação de testes e usa uma linguagem proprietária, baseada em ações e assistente, como linguagem dos *scripts* de teste. O e-Tester possui uma IDE ("Integrated Development Environment") também proprietária, usa o conceito de "Mapa de Interface" e permite o acesso a banco de dados. Não apresenta suporte nativo à técnica *keyword-driven*, mas apresenta suporte à técnica *data-driven* por meio de assistente e arquivos CSV ("Comma Separated Values"). A extensão dos pontos de verificação pode ser feita por meio de programação em linguagem VBScript, C++ ou J++. O e-Tester se integra de maneira transparente ao e-Manager Enterprise, um produto da própria Empirix para gerenciamento de teste. O relatório de erros pode ser consultado no próprio e-Tester.

Functional Tester for Java and Web ([Ibm04a])

O Functional Tester for Java and Web é uma ferramenta com foco na automação de teste funcional de interface, exclusivamente para aplicações escritas em Java ou para o ambiente hipermídia. Apesar de dar suporte nativo à técnica *Record & Playback* de automação de testes, o Functional Tester usa Java como linguagem dos *scripts* de teste. Ainda, o Functional Tester utiliza o Eclipse ([Ecl04]) como IDE e o conceito de "Mapa de Interface". Não existe suporte nativo à técnica *keyword-driven* e nem ao acesso a banco de dados, mas a adoção da linguagem Java permite tanto a sua implementação como a extensão dos pontos de verificação. O suporte à técnica *data-driven* existe, mas requer o uso do TestManager, o gerenciador de teste da IBM Rational. O Functional Tester se integra de maneira transparente à suíte de desenvolvimento da própria IBM Rational. O relatório de erros pode ser gerado em formato HTML ou consultado no TestManager.

QA Wizard ([Sea04])

O QA Wizard é uma ferramenta com foco na automação de teste funcional de interface de aplicações Java ou escritas para o sistema operacional Windows, incluindo aplicações hipermídia. Possui suporte nativo à técnica *Record & Playback* de automação de testes, e usa uma linguagem proprietária, baseada em palavras-chaves e assistente, como linguagem dos

scripts de teste. O QA Wizard possui uma IDE também proprietária, utiliza o conceito de "Mapa de Interface" e permite o acesso a banco de dados. Não apresenta suporte nativo à técnica *keyword-driven*, mas apresenta suporte à técnica *data-driven* por meio de assistente. A extensão dos pontos de verificação não é possível. O QA Wizard se integra de maneira transparente ao TestTrack Pro, um produto da própria Seapine para gerenciamento de *bug tracking*. O relatório de erros pode ser consultado no próprio QA Wizard.

QARun ([Com04a])

O QARun é uma ferramenta com foco na automação de teste funcional de interface de aplicações Java, CRM e hipermídia. Possui suporte nativo à técnica *Record & Playback* de automação de testes e usa uma linguagem proprietária, similar a VisualBasic, como linguagem dos *scripts* de teste. O QARun possui uma IDE também proprietária, utiliza o conceito de "Mapa de Interface" e permite o acesso a banco de dados. Não apresenta suporte nativo à técnica *keyword-driven*, mas apresenta suporte à técnica *data-driven* por meio de arquivos CSV. A extensão dos pontos de verificação pode ser feita por meio da criação de bibliotecas dinâmicas do Windows. O QARun se integra de maneira transparente ao QADirector, um produto da própria Compuware para gerenciamento de teste. O relatório detalhado de erros pode ser consultado no próprio QARun.

QuickTest Professional ([Mer04a])

O QuickTest é uma ferramenta com foco na automação de teste funcional de interface de aplicações Java ou escritas para o sistema operacional Windows, incluindo aplicações hipermídia. Possui suporte nativo à técnica *Record & Playback* de automação de testes e usa VBA ("VisualBasic for Applications") como linguagem dos *scripts* de teste. O QuickTest possui uma IDE proprietária, utiliza o conceito de "Mapa de Interface" e permite o acesso a banco de dados. Não apresenta suporte nativo à técnica *keyword-driven*, mas apresenta suporte à técnica *data-driven* pelo uso de planilhas do Microsoft Excel. A extensão dos pontos de verificação pode ser feita por meio da criação de bibliotecas dinâmicas do Windows. O QuickTest se integra de maneira transparente ao TestDirector, um produto da própria Mercury Interactive para gerenciamento de teste. O relatório de erros detalhado pode ser consultado no próprio QuickTest e exportado em formato texto.

Robot ([Ibm04b])

O Robot é uma ferramenta com foco na automação de teste funcional de interface de aplicações Java ou escritas para o sistema operacional Windows, incluindo aplicações hipermídia. Possui suporte nativo à técnica *Record & Playback* de automação de testes e usa uma linguagem proprietária, similar a Visual Basic, como linguagem dos *scripts* de teste. O Rational Robot possui uma IDE também proprietária, não utiliza o conceito de "Mapa de Interface" e permite o acesso a banco de dados. Também não apresenta suporte nativo à técnica *keyword-driven*, mas para a técnica *data-driven*, sim, por um assistente de criação. A extensão dos pontos de verificação pode ser feita por meio da criação de bibliotecas dinâmicas do Windows. O Robot se integra de maneira transparente à suíte de desenvolvimento da própria IBM Rational. O relatório de erros detalhado pode ser gerado em formato texto ou consultado no TestManager, o gerenciador de teste da IBM Rational.

SilkTest ([Seg04a])

O SilkTest é uma ferramenta com foco na automação de teste funcional de interface de aplicações escritas para o sistema operacional Windows, incluindo aplicações hipermídia, e Java. Possui suporte nativo à técnica *Record & Playback* de automação de testes, e usa uma linguagem própria para os *scripts* de teste. O SilkTest possui uma IDE proprietária, utiliza o conceito de "Mapa de Interface" e permite o acesso a banco de dados. Não apresenta suporte nativo à técnica *keyword-driven*, mas apresenta suporte à técnica *data-driven* através de arquivos CSV. A extensão dos pontos de verificação pode ser feita por meio da criação de bibliotecas dinâmicas do Windows. O SilkTest se integra de maneira transparente ao SilkPlan Pro e ao SilkRadar, dois produtos da própria Segue para, respectivamente, gerenciamento de teste e de *bug tracking*. O relatório detalhado de erros pode ser consultado no próprio SilkTest e exportado em diferentes formatos.

SilkTest International ([Seg04b])

O SilkTest International possui as mesmas características do SilkTest da Segue, com a adição do forte suporte à internacionalização, incluindo código Unicode.

TestPartner ([Com04b])

O TestPartner é uma ferramenta com foco na automação de teste funcional de interface de aplicações Java, SAP, .NET e hipermídia. Possui suporte nativo à técnica *Record & Playback* de automação de testes e usa VBA como linguagem dos *scripts* de teste. O TestPartner possui uma IDE proprietária, utiliza o conceito de "Mapa de Interface" e permite o acesso a banco de dados. Não apresenta suporte nativo à técnica *keyword-driven*, mas apresenta suporte à técnica *data-driven* através de arquivos CSV. A extensão dos pontos de verificação pode ser feita por meio da criação de bibliotecas dinâmicas do Windows. O TestPartner se integra de maneira transparente ao QADirector, um produto da própria Compuware para gerenciamento de teste. O relatório detalhado de erros pode ser consultado no próprio TestPartner.

TestSmith ([Qua04])

O TestSmith é uma ferramenta com foco na automação de teste funcional de interface de aplicações escritas para o sistema operacional Windows, incluindo aplicações hipermídia. Possui suporte nativo à técnica *Record & Playback* de automação de testes, e pode tanto usar uma linguagem proprietária para os *scripts* de teste quanto Java ou C++. O TestSmith possui uma IDE também proprietária, mas apenas utilizada para codificação se a linguagem dos *scripts* de teste não for Java ou C++. Não apresenta suporte nativo à técnica *keyword-driven*, mas para a técnica *data-driven*, sim, com o uso arquivos CSV, além do conceito de "Mapa de Interface" e acesso a bancos de dados. A extensão dos pontos de verificação pode ser feita com *scripts* de teste em linguagem Java ou C++. Um ponto negativo forte é que não existe integração nativa entre o TestSmith e qualquer outro sistema de gerenciamento de teste. O relatório de erros detalhado pode ser gerado em formato texto ou HTML, com diferentes níveis de detalhe.

WinRunner ([Mer04b])

O WinRunner é uma ferramenta com foco na automação de teste funcional de interface de aplicações Java ou escritas para o sistema operacional Windows, incluindo aplicações hipermídia. Possui suporte nativo à técnica *Record & Playback* de automação de testes e usa uma linguagem proprietária, similar a C, para os *scripts* de teste. O WinRunner possui uma IDE também proprietária, utiliza o conceito de "Mapa de Interface" e permite o acesso a banco de dados. Não apresenta suporte nativo à técnica *keyword-driven*, mas para a técnica *data-driven*,

sim, com o uso de planilhas do Microsoft Excel. A extensão dos pontos de verificação pode ser feita por meio da criação de bibliotecas dinâmicas do Windows. O WinRunner se integra de maneira transparente ao TestDirector, um produto da própria Mercury Interactive para gerenciamento de teste. O relatório de erros detalhado pode ser consultado no próprio WinRunner e exportado em formato texto.

3.7.2 Ferramentas e tecnologias complementares

Nesta seção são analisadas algumas ferramentas e tecnologias complementares para a instanciação do sistema AutoTest. Dependendo da escolha da ferramenta de automação de teste como componente de criação e execução de *scripts*, uma ou mais destas ferramentas ou tecnologias pode ser escolhida para a completa instanciação do sistema.

Não serão abordadas ferramentas de gerenciamento de teste ou *bug tracking*, pois estas são dependentes da escolha da ferramenta de automação de teste, como visto na seção anterior.

As ferramentas e tecnologias analisadas foram escolhidas dentre as mais populares atualmente, com base em pesquisas na Internet.

Ferramentas de controle de versão

- CVS ([Ber89]): é a mais popular ferramenta de controle de versão existente e, além de gratuita, pode ser integrada de maneira transparente (ou semi) a praticamente qualquer aplicação, principalmente por contar com clientes para vários ambientes e plataformas;
- ClearCase ([Ibm04c]): outra ferramenta de controle de versão bastante utilizada pelas empresas desenvolvedoras de software. Apesar de não possuir clientes para tantas plataformas quanto o CVS, pode ser utilizada tanto em ambiente UNIX quanto Windows, além de se integrar ao sistema operacional, ao Robot e ao Functional Tester for Java and Web de maneira quase transparente.

Ferramentas para manipulação de dados das técnicas *data-driven* e *keyword-driven*

- Editor de textos simples: qualquer editor de textos, por mais simples que seja, pode ser utilizado para criar e manter os arquivos de dados utilizados pelas técnicas *data-driven* e *keyword-driven*. Apesar desta abordagem não ter custo, a manutenção de grandes arquivos de dados pode se tornar tão mais trabalhosa quanto mais simples for o editor. Ainda, pode ser necessário incluir códigos de controle entre os dados do arquivo, tornando sua manutenção mais delicada e onerosa;
- Microsoft Word ([Mic03a]): o poderoso editor de textos da Microsoft permite a gravação e leitura de arquivos em vários formatos, além de contar com *macros* que podem ser usadas para automatizar e facilitar a criação e manutenção dos arquivos de dados utilizados pelas técnicas *data-driven* e *keyword-driven*. Uma das vantagens do uso do Microsoft Word é seu uso já difundido na maioria das empresas desenvolvedoras de software, não requerendo, na maioria dos casos, treinamento ou mesmo nova compra do produto;
- OpenOffice Writer ([Ope03]): apresenta as mesmas vantagens do Microsoft Word. Apesar de pouco difundido atualmente, é gratuito e, por ter uma interface com o usuário muito próxima à do Microsoft Word, requer um esforço muito pequeno de treinamento;
- Microsoft Excel ([Mic03b]): assim como o editor de textos da mesma família de produtos da Microsoft, o Excel permite a gravação e leitura de arquivos em vários formatos, além de contar com *macros* que podem ser usadas para automatizar e facilitar a criação e manutenção dos arquivos de dados utilizados pelas técnicas *data-driven* e *keyword-driven*. Além da mesma vantagem, compartilhada com o Word, de uso já difundido na maioria das empresas desenvolvedoras de software, a utilização do formato planilha, e não texto, garante um formato mais rígido e menos sujeito a erros na criação e manutenção dos arquivos de dados;
- OpenOffice Calc ([Ope03]): apresenta as mesmas vantagens do Microsoft Excel. Assim como o Writer, apesar de pouco difundido atualmente, é gratuito e, por ter uma interface com o usuário muito próxima à do Microsoft Excel, requer um esforço muito pequeno de treinamento.

Tecnologias para acesso aos arquivos de dados das técnicas *data-driven* e *keyword-driven*

- Manipulação de arquivos: desde que a ferramenta de automação dê suporte à manipulação de arquivos, os arquivos de dados utilizados pelas técnicas *data-driven* e *keyword-driven* podem ser acessados por meio de funções nativas da linguagem de programação utilizada pela ferramenta. Apesar desta abordagem não ter custo, dependendo da complexidade dos arquivos de dados pode ser necessário um esforço considerável para a criação das rotinas necessárias para este fim;
- OLE/DDE ([Boy98]): estes são dois protocolos do sistema operacional Windows que permitem que um programa inicie outro, receba e envie dados e execute comandos. Desde que a ferramenta de automação dê suporte ao uso destes protocolos (por meio de código no próprio *script* de teste ou de uma biblioteca dinâmica) é possível que se utilize o próprio Microsoft Word, Microsoft Excel ou outra ferramenta para acessar os arquivos de dados. As desvantagens desta abordagem são: só pode ser usada em sistemas operacionais Windows, requer um conhecimento avançado para a criação das rotinas de manipulação e a aplicação utilizada para criar os arquivos de dados deve necessariamente estar instalada na máquina em que o teste automatizado é executado;
- Jakarta POI ([Apa02]): esta biblioteca Java permite o acesso direto a arquivos do Microsoft Word e Microsoft Excel por programas Java. Apesar de poderosa e gratuita, esta biblioteca possui uma certa complexidade de uso e está em desenvolvimento;
- Java Excel API ([Kha03]): esta outra biblioteca Java permite o acesso direto a arquivos do Microsoft Excel por programas Java. Além de gratuita e tão poderosa quanto o Jakarta POI, tem a vantagem de ser bem mais simples de usar, mas, assim como o POI, está em desenvolvimento.

Ferramentas para execução remota de testes

- SilkTest Runtime ([Seg04a], [Seg04b]): também produto da Segue, pode ser utilizado com o SilkTest ou SilkTest International para a execução remota de testes automatizados criados nestas ferramentas;
- Rational Test Manager ([Ibm04d]): também produto da IBM Rational, pode ser utilizado com o Robot ou Functional Tester for Java and Web para a execução remota de testes automatizados criados nestas ferramentas;
- DameWare Mini Remote Control ([Dam04]): esta ferramenta permite o acesso remoto a máquinas com sistema operacional Windows, podendo ser utilizada para se iniciar, manualmente, a execução remota de testes automatizados;
- VNC ([VNC02]): possui o mesmo objetivo e as mesmas vantagens do Mini Remote Control, com a diferença de ser gratuita, possuir clientes para praticamente todos os sistemas operacionais existentes, ser acessível via navegadores hipermídia e ter um desempenho melhor que o Mini Remote Control;
- Microsoft Terminal Services ([Mic00]): esta ferramenta que permite o acesso remoto a máquinas com sistema operacional Windows 2000, inclusive através do navegador hipermídia Internet Explorer. Em comparação com o Mini Remote Control e o VNC, é a ferramenta com melhor desempenho.

Ferramentas para interação com outros ambientes

- Servidor telnet ([Bra89]): o telnet é uma maneira fácil e simples de se acessar remotamente outras máquinas e ambientes. Servidores telnet são encontrados não só nativamente em vários sistemas operacionais, mas inclusive, gratuitamente na Internet;
- Servidor FTP ([Pos85]): já o FTP é uma maneira fácil e simples de se trocar arquivos com máquinas e ambientes remotos. Servidores FTP são encontrados não só nativamente em vários sistemas operacionais, mas inclusive, gratuitamente na Internet;

- Samba ([Ts03]): através deste software é possível que máquinas Windows enxerguem diretórios de máquinas UNIX como se Windows também fossem. Em empresas de ambiente misto (Windows e UNIX) é comum o seu uso para facilitar a transferência de arquivos entre os dois sistemas operacionais.

Tecnologias para interação com outros ambientes

- Telnet ([Bra89]): o protocolo simples, mas eficiente, não apresenta grande complexidade caso se opte por implementá-lo, mas soluções baratas, e mesmo gratuitas, são facilmente encontradas na Internet;
- FTP ([Pos85]): assim como o telnet, o FTP é um protocolo simples e eficiente, podendo ser facilmente implementado ou utilizadas soluções baratas, e mesmo gratuitas, encontradas na Internet;
- jCIFS ([All00]): esta biblioteca Java permite o acesso direto a arquivos compartilhados via Samba por programas Java. Apesar de simples, gratuita e poderosa, tem a desvantagem de estar em desenvolvimento.

Tecnologias para acesso aos bancos de dados

- ODBC ([Mic97]): esta é uma especificação aberta para acesso a bancos de dados muito utilizada em ambientes Windows (possuindo, inclusive, suporte nativo neste sistema operacional), apesar de existirem iniciativas para sua adoção em ambientes UNIX;
- JDBC ([Sun03]): esta biblioteca Java permite a conexão a uma variedade de bancos de dados, por programas Java, sem necessidade de componentes externos, sendo uma unanimidade nessa linguagem.

Ferramentas para gravação visual de execução de teste

- Camtasia Studio ([Tec04]): esta ferramenta, para Windows, permite a geração de vídeos da tela de uma máquina, sendo capaz de gravar os vídeos em uma variedade de formatos e ser tanto iniciada quanto terminada através de outros programas ou combinações de teclas. Duas outras vantagens desta ferramenta são seu baixo consumo de processamento e flexibilidade de definição da gravação (tela inteira, janela, adição de notas, entre outros);
- My Screen Recorder ([Des04]): também para Windows, esta ferramenta é mais simples, e também mais barata, que o Camtasia Studio, mas apresenta as mesmas funcionalidades básicas necessárias.

Ferramentas para manipulação de bases de dados versionadas

Ferramentas para manipulação de bases de dados completas são altamente dependentes do sistema gerenciador de banco de dados (SGBD) utilizado, e também específicas para o SGBD que acompanham. Apesar de existirem alternativas genéricas, elas são menos eficientes e mais limitadas que as providas pelos próprios fabricantes dos sistemas gerenciadores de banco de dados. E uma vez que estas ferramentas integram o próprio pacote do SGBD, utilizar componentes genéricos em seu lugar é um custo desnecessário.

3.8 Seleção dos componentes

Conforme descrito na Seção 3.6, muitas ferramentas existentes no mercado atendem a um subconjunto dos requisitos apresentados para o sistema AutoTest. A escolha de quais serão utilizadas na implementação do sistema deve feita cuidadosamente, uma vez que quanto mais requisitos forem atendidos por um componente, menos componentes serão necessários para a instanciação completa do sistema. Não é possível, e nem prudente, definir um conjunto rígido e imutável de componentes para toda e qualquer implementação do sistema AutoTest, pois a escolha dos mais adequados depende também das necessidades de teste e características de desenvolvimento das aplicações a serem testadas.

Por exemplo, em uma empresa cujo produto é uma aplicação nativa para ambiente Windows, a escolha da ferramenta de automação de teste poderia recair entre o Robot e o WinRunner, e os fatores de decisão entre uma ou outra poderiam ser: a proximidade da linguagem utilizada para a criação dos *scripts* de teste com a utilizada no desenvolvimento do produto da empresa; a já existência, na empresa, de outros produtos do mesmo fabricante. Estes dois fatores têm grande importância tanto nas questões da curva de aprendizado da ferramenta quanto no custo de aquisição da mesma. Assim, se esta empresa cria aplicações utilizando a linguagem Visual Basic, e já possui outras ferramentas da IBM Rational, a escolha do Robot pode ser a melhor opção. Já se ela utiliza a linguagem C++, e não possui ferramentas de nenhuma das duas empresas, a escolha do WinRunner pode ser a melhor opção.

O Capítulo 4 apresenta as ferramentas e as justificativas de sua seleção numa implementação do sistema AutoTest. Novamente, estas seleções podem variar de implementação para implementação, sempre focando no conjunto que traga mais vantagens na implementação do sistema AutoTest.

Capítulo 4

Implementação do sistema AutoTest

Este capítulo apresenta uma implementação do sistema AutoTest, proposto no Capítulo 3, desenvolvida internamente na Gerência de Testes de Sistemas da Diretoria de Soluções em Billing do CPqD pela equipe de automação de teste.

Este capítulo está organizado da seguinte forma: a Seção 4.1 apresenta a ferramenta de automação de teste escolhida e as adaptações necessárias; as Seções 4.2, 4.3 e 4.4 apresentam, respectivamente, os componentes de registro visual de execução, controle de versão e criação e manutenção de dados para as técnicas *data-driven* e *keyword-driven* escolhidos; a Seção 4.5 apresenta a ferramenta de execução remota/distribuída escolhida para esta implementação do sistema; as Seções 4.6 e 4.7 apresentam os componentes de gerenciamento de bases de dados versionadas e cobertura de código escolhidos; finalmente, a Seção 4.8 apresenta algumas considerações sobre esta implementação e os requisitos do sistema AutoTest atendidos.

4.1 Ferramenta de automação de teste

A ferramenta de automação de teste escolhida para esta implementação do sistema AutoTest foi o IBM Rational Functional Tester for Java and Web. Os motivos desta escolha foram: ser uma ferramenta focada na automação de teste funcional de interface para aplicações hipermídia, ter suporte nativo à técnica *Record & Playback* de automação de testes, utilizar o Eclipse como IDE, implementar o conceito de "Mapa de Interface", se integrar transparentemente à suíte de desenvolvimento da IBM Rational e, principalmente, utilizar Java como linguagem dos *scripts* de teste.

Como a suíte de desenvolvimento da IBM Rational é adotada corporativamente pelo CPqD, a integração com o Functional Tester é interessante em vários pontos, entre eles: geração de relatórios, importação e exportação de dados e resultados e expansibilidade das funcionalidades (por exemplo, controle de versão e execução remota).

O fato de utilizar Java e Eclipse dá ao engenheiro de automação de teste não só um forte e poderoso ambiente de desenvolvimento, mas também flexível e facilmente configurável e expansível. Não se optou por uma ferramenta com linguagem de programação proprietária uma vez que, em geral, a criação de ações ou verificações mais complexas ou específicas exige soluções também mais complexas de se criar, como bibliotecas dinâmicas ou mesmo processos externos.

Ainda, a adoção de Java traz vantagens significativas, uma vez que é uma linguagem difundida e com profissionais competentes no mercado, orientada a objetos e com um amplo parque de soluções já prontas, muitas gratuitas, e que podem ser integradas transparentemente ou de maneira fácil aos *scripts* de teste (e ao próprio sistema).

Uma vantagem específica para a DSB é que a própria diretoria utiliza Java (e a IDE Eclipse) no desenvolvimento de seus produtos, o que permite a troca de experiências e soluções entre a equipe de automação de teste e a de desenvolvimento, tornando o trabalho mais fácil e produtivo.

Em contrapartida, devido à ferramenta não dar suporte nativo à técnica *keyword-driven*, e também ao acesso a bancos de dados, e devido ao suporte simplista e proprietário à técnica *data-driven*, faz-se necessário a criação de rotinas e soluções para sanar estas deficiências e cumprir os requisitos do sistema.

4.1.1 Suporte às técnicas *data-driven* e *keyword-driven*

O suporte à técnica *keyword-driven* foi implementado como uma classe Java capaz de interpretar uma seqüência de comandos e tomar a ação correspondente em tempo de execução do teste. Planilhas de teste *keyword-driven* especificam os comandos a serem executados, de forma que elas controlam todo o fluxo e seqüência de execução dos comandos de testes a serem realizados na aplicação. Cada uma das palavras-chave das planilhas *keyword-driven* dispara uma ação, implementada pelo *script* correspondente (chamados "*scripts* orientados a palavras-chave" ou *scripts keyword-driven*), o qual por sua vez interage com a aplicação sob teste.

Foi criado um repositório de comandos de propósito geral, comum a todas as aplicações hipermídia, os quais podem ser invocados independentemente por meio das palavras-chave correspondentes em uma planilha *keyword-driven* ou via código em um *script*. Estes são

denominados comandos internos. Por outro lado, existem também comandos criados unicamente para um propósito específico: o teste de uma funcionalidade de uma aplicação em particular. Estes comandos, implementados pelos "*scripts* orientados a dados" ou *scripts data-driven*, são denominados comandos externos e podem estar associados a planilhas *data-driven*. O conjunto destes dois tipos de *scripts* forma uma espécie de linguagem de programação bastante simples, com associação direta à língua portuguesa, como já exemplificado no Capítulo 2.

Objetivando a facilidade de criação de novos *scripts keyword-driven*, os comandos internos são construídos de forma mais genérica e flexível possível. Esta promoção do reuso de comandos é extremamente vantajoso do ponto de vista de economia no tempo no desenvolvimento dos *scripts* de teste, bem como na redução de sua complexidade, promovendo um aumento de sua confiabilidade. Além da reusabilidade tem-se expansibilidade, uma vez que o sistema pode ser facilmente expandido pela criação de novos comandos, que passarão a ser integrados a ele: uma vez criado um novo comando, este poderá ser utilizado transparentemente em uma planilha ou *script keyword-driven*.

O suporte à técnica *data-driven* é simplesmente uma classe capaz de acessar dados em uma planilha de teste *data-driven*. A organização desta planilha será comentada adiante.

4.1.2 Arquitetura

Uma vez que o *framework* original do Functional Tester é um tanto limitado, e a ferramenta não apresenta algumas funcionalidades exigidas pelo sistema AutoTest (ou as apresenta de forma muito limitada), optou-se por criar um *framework* em cima do Functional Tester. Este *framework*, chamado AutoTestScript, é apresentado na Figura 4.1.

Foi estendida a classe RationalTestScript para permitir a manipulação de objetos da interface gráfica da aplicação sob teste. Os comandos (internos ou externos) são especializações da classe AutoTestScript. Os comandos externos, por provavelmente manipularem planilhas de dados, descendem da classe ScriptDataDriven, que fornece métodos e facilidades para isso. Ainda, comportamentos ou ações específicas a um grupo de aplicações podem ser reunidos no que foi chamado nesta implementação de *skin*, ou seja, um conjunto de padrões visuais e de operações sob a interface que caracterizam um certo padrão de comportamento da aplicação. Isto

permite a reutilização de código e a centralização dos pontos de mudança em testes automatizados de diferentes aplicações que façam parte da mesma família de aplicativos.

A Figura 4.2 apresenta a arquitetura das classes criadas para permitir o uso das técnicas *data-driven* e *keyword-driven*: as classes DataDrivenSheet e KeywordDrivenSheet, respectivamente, representam e manipulam estes dois tipos de planilhas de teste utilizadas pelo sistema AutoTest.

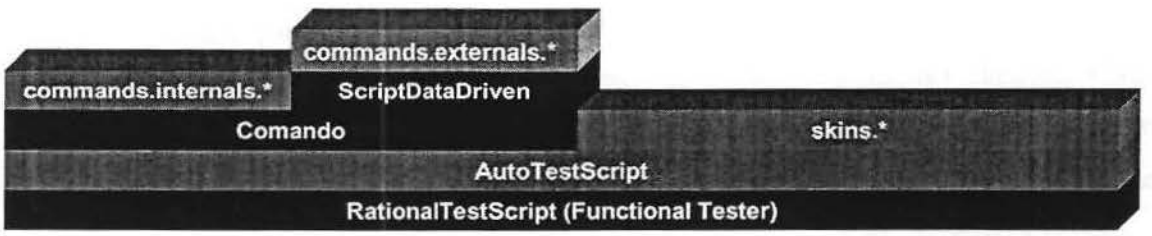

Figura 4.1: Arquitetura do *framework* AutoTestScript, implementado no Functional Tester

Figura 4.2: Classes para uso das técnicas *data-driven* e *keyword-driven*

4.1.3 Ambiente

Por utilizar o Functional Tester como base, esta implementação do sistema AutoTest está restrita ao ambiente desta ferramenta. Apesar desta ser escrita em Java, seu uso em sistemas operacionais que não Windows é restrito apenas à execução de testes automatizados em aplicações também em Java, uma vez que a ferramenta oferece suporte à gravação de *scripts* via *Record & Playback* apenas nesta família de sistemas operacionais. Entretanto, esta limitação não causa impacto nos objetivos desta implementação do sistema, uma vez que os ambientes utilizados, tanto de desenvolvimento quanto de teste, têm Windows como sistema operacional padrão.

4.1.4 Mapa de Interface

O Functional Tester oferece nativamente suporte à abstração de "Mapa de Interface" da aplicação sob teste. Este mapa pode ser editado e modificado facilmente usando um utilitário que acompanha a ferramenta. Entretanto, o "Mapa de Interface" da aplicação sob teste deve ser ligado a um *script* criado na ferramenta para que possa ser utilizado. Ainda, os nomes fictícios dos objetos da interface, se não modificados pelo usuário, são crípticos e não apresentam informação nenhuma de sua hierarquia na interface gráfica da aplicação.

Para sanar estes dois problemas, esta implementação do sistema AutoTest, além de realizar um pré-processamento do arquivo de "Mapa de Interface" para criar nomes de objetos que contenham informação de hierarquia, realiza a ligação do mapa com ele próprio para que seja possível fazer uso do mapeamento criado. Tudo isto é feito sem alterar o arquivo original, permitindo, portanto, que o utilitário do Functional Tester seja plenamente utilizado na manutenção do mapa da interface gráfica da aplicação.

O uso de nomes que contêm informação de hierarquia apresenta duas vantagens: o mapa pode conter objetos distintos com mesmos nomes (por exemplo, "Confirmar" para botões em páginas diferentes) e a referência a um objeto pode ser feita simplesmente pelo seu nome final. Por exemplo, o elemento "Página de Cadastro/Quadro Principal/Confirmar" pode ser referenciado na planilha *keyword-driven* ou no *script data-driven*, estando a janela "Página de Cadastro" ativa, como simplesmente "Confirmar"; se houverem dois *frames* nesta janela, cada um contendo um botão "Confirmar", eles podem ser referenciados simplesmente por "Frame Principal/Confirmar" e "Frame Secundário/Confirmar".

4.1.5 Relatório de execução

Apesar do Functional Tester já possuir a funcionalidade de geração de relatório de execução, tanto em formato HTML quanto integrado ao TestManager, esta não foi utilizada por ser muito simplista e limitada.

Assim, optou-se por criar um formato próprio: no decorrer da execução do teste todas as ações executadas são registradas em um relatório em formato texto, criado no diretório da planilha *keyword-driven*, com o mesmo nome desta. Este relatório também apresenta a data e

hora em que cada uma das ações foi executada, bem como seu resultado (sucesso ou falha). Além disso, o relatório apresenta, nas duas primeiras colunas, códigos alfanuméricos que identificam a natureza das mensagens (informação, erro, início de caso de teste, etc) para tornar fácil a filtragem do seu conteúdo por meio de editores de texto ou sua conversão para um formato específico. A Figura 4.3 apresenta um exemplo de relatório gerado.

Ao término da execução do teste automatizado são gerados mais dois relatórios, resumidos, baseados no relatório detalhado. Um dos relatórios, em formato texto simples, apresenta apenas as mensagens de erro, separadas por caso de teste. Este pode ser encaminhado para o analista de teste responsável para a abertura de requisições de correção ou alteração da aplicação.

O outro relatório resumido, em formato HTML, apresenta uma estatística geral da execução de teste (número total de casos de teste, número de casos de teste que passaram e falharam e o tempo decorrido) e os identificadores dos casos de teste que passaram e falharam em forma de hipertextos. Para os casos de teste que falharam, além do seu nome são apresentadas as mensagens de erro e hipertextos para imagens da tela, registradas no momento da falha do teste. Este relatório permite uma consulta mais rápida e fácil pelo analista de teste, sendo que detalhes maiores podem ser obtidos no relatório detalhado ou no vídeo de execução (Seção 4.2).

```
I 13.02.04 14:12:59          AutoTest - Início da execução: Fri Feb 13 14:12:59 GMT-02:00 2004
S 13.02.04 14:13:03          Mapeadas 33 instâncias Oracle para acesso.
X 13.02.04 14:13:06          Analisando conteúdo da planilha 'CBILL_~1.XLS'.
S 13.02.04 14:15:41          Conteúdo da planilha 'CBILL_~1.XLS' sem erros.
I 13.02.04 14:15:41          +--------------------------------------------------------------------------
I 13.02.04 14:15:41          |Projeto: Sistema de Faturamento
I 13.02.04 14:15:41          |Módulo: Entrada
I 13.02.04 14:15:41          |Planilha de Teste: Planilha.xls
I 13.02.04 14:15:41          |Data da Criação:
I 13.02.04 14:15:41          |Última Atualização: 02/13/2004
I 13.02.04 14:15:41          |Versão: 1.0
I 13.02.04 14:15:41          +--------------------------------------------------------------------------
C 13.02.04 14:15:41          --- 1 - Apaga todo o conteúdo do banco de dados de ---
X 13.02.04 14:15:53          Executar SQL 'delete from jcl_constante' no banco 'pr08auto'.
X 13.02.04 14:15:56          SQL 'delete from jcl_constante' executada no banco 'pr08auto'.
X 13.02.04 14:15:56          Executar SQL 'delete from jcl_identificador_sumarizado' no banco 'pr08auto'.
X 13.02.04 14:15:56          SQL 'delete from jcl_identificador_sumarizado' executada no banco 'pr08auto'.
I 13.02.04 14:16:15          >>> Tempo decorrido (seção): 34s
C 13.02.04 14:16:15          --- 2 - Testa funcionalidade "Incluir " ---
X 13.02.04 14:16:33          Abrir a página 'http://localhost/app' ('Entrada').
```

Figura 4.3: Exemplo de relatório detalhado de execução

4.1.6 Acesso aos bancos de dados

Nesta implementação do sistema AutoTest todos os acessos aos bancos de dados utilizam a tecnologia JDBC, não necessitando de nenhum componente externo ou de suporte do próprio sistema operacional.

No caso específico da DSB, foi possível facilitar a interação com os bancos de dados pela abstração de particularidades de conexão como protocolo, instância e servidor.

Um sistema hipermídia chamado QControleBD, mantido na *intranet* da DSB, permite a consulta das características de todos os bancos de dados da diretoria. Pela simulação de um cliente hipermídia fictício, esta implementação do sistema AutoTest obtém as informações de conexão com o banco desejado em tempo real. Assim, em caso de mudança de conexão com um banco de dados (por exemplo, troca de instância, servidor ou porta), o teste automatizado não falha em decorrência da impossibilidade de contatar o banco em questão. Para evitar um comprometimento de desempenho em operações em banco de dados, faz-se uso de *cache* local temporária das informações de conexão dos bancos de dados, após serem obtidos diretamente do QControleBD.

4.1.7 Acesso ao sistema de arquivos

A implementação em questão do sistema AutoTest faz acesso ao sistema de arquivos utilizando-se de caminhos relativos. Localizações absolutas de arquivos (como as planilhas de teste, por exemplo) são passadas pelo usuário, e qualquer acesso aos diretórios em que eles se localizam também é feito através de caminhos relativos.

Isto permite que a implementação do sistema possa existir fisicamente em qualquer máquina ou diretório, não sendo necessária instalação, cópia de arquivos ou restrição de drive. É delegada ao sistema operacional a tarefa de converter os caminhos relativos em absolutos, dependendo da referência. Com isso pode-se manter toda a implementação do sistema em uma única máquina, com o diretório compartilhado, tendo-se várias outras acessando o mesmo código, mas cada uma executando um teste diferente. Qualquer alteração ou melhoria na implementação do sistema é automaticamente refletida nestas máquinas "clientes". Por conseqüência, o próprio repositório único da implementação do sistema (e inclusive as planilhas de teste e mapas de interface das aplicações) pode ser colocado no sistema de controle de versão.

A biblioteca jCIFS foi utilizada para permitir o acesso ao sistema de arquivos UNIX das máquinas servidoras. O motivo de se usar esta abordagem, e não simplesmente delegar o acesso ao sistema operacional, deve-se ao fato que existe a questão de segurança no acesso a diretórios compartilhados no Windows: caso um usuário não autorizado tente acessar um diretório compartilhado via Samba em uma máquina UNIX, o Windows exige que seja digitado um usuário e senha autorizados para a realizar o acesso. Por limitação do próprio Functional Tester, não é possível simular a entrada do nome/senha válidos na caixa de diálogo de autenticação. Esta limitação, somada ao fato de que é comum o uso de usuários e senhas especiais para o acesso aos diretórios compartilhados em máquinas UNIX, e que freqüentemente o usuário que executa o teste na máquina Windows não tem acesso ao diretório desejado, foi decisiva para a escolha do cliente Samba. Além do diretório que se quer acessar, a planilha ou *script* de teste deve também especificar um usuário e senha válidos para o acesso.

4.1.8 Uso de processos externos

Alguns processos são suficientemente complexos a ponto de inviabilizar o seu processamento pela própria implementação do sistema, outros são impossíveis de se realizar por motivos diversos. Como exemplo, citemos a transferência de 150 arquivos de 100MB da máquina local para um servidor UNIX em um diretório não compartilhado por Samba. Neste caso a melhor solução é, sem dúvida, a transferência dos arquivos por FTP.

Para viabilizar este tipo de operação e também permitir a interação com outros processos (como processos *batch* a serem testados), esta implementação do sistema AutoTest também dá suporte à execução de processos externos (no exemplo acima, o utilitário "ftp" do Windows).

4.1.9 Comunicação com outros ambientes

Esta implementação do sistema AutoTest também apresenta uma versão simples do protocolo telnet, com métodos para autenticação e registro completo das entradas e saídas. Com isso é possível que as planilhas ou *scripts* de teste executem comandos e aplicações em ambiente UNIX, para teste de aplicações *batch*, inclusive registrando a saída dos mesmos para verificação posterior de resultados.

4.2 Ferramenta de registro visual de execução

No decorrer da execução do teste também é gravado um vídeo da área de trabalho da máquina pela ferramenta Camtasia Publisher. Aliado ao relatório detalhado de execução, o vídeo é uma poderosa informação para a descoberta da causa de falha dos testes. Seu uso praticamente elimina a necessidade de execução manual do caso de teste falho para identificação e análise do erro. Pontos específicos de execução são facilmente encontrados através da data e hora de ocorrência do erro ou falha, uma vez que o vídeo também apresenta esta informação em seu canto inferior direito.

O Camtasia Publisher também é utilizado para registrar a imagem da área de trabalho para o relatório resumido de erro. Apesar do vídeo de execução também apresentar esta informação visual, a consulta às imagens é bem mais rápida que ao vídeo, por não exigir busca, principalmente quando consultada pelo relatório resumido.

4.3 Ferramenta de controle de versão

A ferramenta de controle de versão escolhida é o IBM Rational ClearCase, por ser utilizada corporativamente pelo CPqD. Apesar de esta ferramenta também ser utilizada em ambiente UNIX, ela é mais comumente utilizada no CPqD em ambiente Windows, além de se integrar ao sistema operacional e ao próprio Functional Tester de maneira quase transparente.

4.4 Ferramenta para criação e manutenção de dados

Os arquivos de dados das técnicas *data-driven* e *keyword-driven* – chamados aqui genericamente de "planilhas de teste" – são mantidos como documentos da ferramenta Microsoft Excel, e para a sua elaboração foram adotadas regras de sintaxe e de semântica claras, que permitem seu fácil entendimento, servindo também como guia para a execução manual dos casos de teste.

A escolha da ferramenta Microsoft Excel deveu-se a ela ser amplamente utilizada e disponibilizada no CPqD, além das outras vantagens já citadas.

A Figura 4.4 apresenta um exemplo de planilha de teste *keyword-driven*. A primeira coluna contém as palavras-chave, que executam ações específicas através dos comandos internos e externos. As outras colunas contêm parâmetros para as palavras-chave. A ordem e significado destes parâmetros variam para cada palavra-chave. Alguns destes, na verdade, têm caráter apenas informativo, não desencadeando nenhuma ação, e seu objetivo é dar maior legibilidade à planilha: por exemplo, o comando "Apertar Botão" instrui o AutoTest a disparar o *script* que realiza a operação de apertar um botão em uma janela determinada pelos parâmetros, mas o comando "Projeto" não desencadeia nenhuma ação, servindo apenas como informação na planilha para o usuário que acessá-la.

1	A	B	C	D	E	F	G
2	Projeto	Sistema de Faturamento					
3	Módulo	Desconto					
4	Data da Criação	01/04/2003					
5	Status	Válido					
6	Versão	1.0					
7	Seção	1 - Inicia o Módulo de Descontos					
8		Executável	Parâmetros	Janela	Estado		
9	Iniciar Aplicação	C:\Tst\Dct\run.bat	–gui	Aplicação de Descontos	Maximizado		
10	Seção	2 - Testa os casos válidos de cadastro de descontos					
11		Planilha					
12	Cadastrar Descontos	CsTeste1PR FAT inclusão de descontos (dados válidos).xls					
13	Seção	3 - Tenta cadastrar um desconto com data inválida (detecção automática de erro)					
14		Janela	Menu	SubMenu 1			
15	Selecionar Menu	Aplicação de Descontos	Desconto	Criar desconto			
16		Janela	Botão				
17	Apertar Botão	Desconto	Maximizar Janela				
18		Janela	Campo	Texto			
19	Editar Campo	Desconto	Validade Final	01012002			
20		Janela	Teclas				
21	Digitar	Desconto	^{F4}				
22		Abortar planilha?	Operação	Janela	Campo	Texto	
23	Em Caso de Erro	Não	Editar Campo	Desconto	Validade Inicial	{HOME}+{END}{DEL}	

Figura 4.4: Exemplo de planilha de teste *keyword-driven*

A Figura 4.5 apresenta um exemplo de planilha de teste *data-driven*. Estas planilhas contêm dados puros que são utilizados pelos comandos externos do sistema. Cada linha da planilha, ou conjunto de linhas, descreve detalhadamente cada caso de teste por meio do uso de três grupos de colunas: o primeiro grupo descreve um caso de teste principal, o qual pode ser dividido hierarquicamente em vários casos de teste derivados; o segundo grupo descreve os dados de entrada do teste; e o terceiro grupo de colunas descreve os resultados esperados. A quantidade de colunas no segundo e no terceiro grupo depende da aplicação para a qual é realizado o projeto de teste. O processo de teste da DSB impõe que a identificação dos casos de teste seja feita de maneira hierárquica: por isso a planilha *keyword-driven* reflete esta organização e as classes DataDrivenSheet e ScriptDataDriven do *framework* AutoTestScript são responsáveis

por sua representação e manipulação. Para acessar diretamente planilhas Excel no formato nativo desta ferramenta, esta implementação do sistema AutoTest utiliza a biblioteca Java Excel API.

Planilhas *keyword-driven* e *data-driven* podem estar no mesmo arquivo do Microsoft Excel, uma vez que a ferramenta permite (uma "pasta de trabalho" ou *workbook* pode conter várias planilhas). O sistema também dá suporte a esta característica, uma vez que a centralização das planilhas em um único arquivo aumenta a organização e facilita o acesso às mesmas.

	A	B	C	D	E	F	G
2		Dados de Teste					
3		Funcionalidade: A – Incluir					
4							
5	CT Pai (ID)	CT Pai (Nome)	CT Filho (ID)	CT Filho (Nome)	Descrição	Tipo	Prioridade
6	1) Incluir Entrada Tipo 1						
7	1	Entrada válida 1			EV1_1	1	1
8	2	Entrada válida 2	a	Prioridade 6 com código 1	EV2a_1	1	6
9			b	Prioridade 6 com código 2	EV2b_1	1	8
10	3	Entrada válida 3			EV3_1	1	3
11	4	Entrada válida 4			EV4_1	1	2
12	2) Incluir Entrada Tipo 2						
13	1	Entrada válida 1			EV1_2	2	1
14	2	Entrada válida 2	a	Prioridade 6 com código 1	EV2a_2	2	6
15			b	Prioridade 6 com código 2	EV2b_2	2	8
16	3	Entrada válida 3			EV3_2	2	3
17	4	Entrada válida 4			EV4_2	2	2
18	3) Incluir Entrada Tipo 3						
19	1	Entrada válida 1			EV1_3	3	1
20	2	Entrada válida 2	a	Prioridade 6 com código 1	EV2a_3	3	6
21			b	Prioridade 6 com código 2	EV2b_3	3	8
22	3	Entrada válida 3			EV3_3	3	3
23	4	Entrada válida 4			EV4_3	3	2

Figura 4.5: Exemplo de planilha de teste *data-driven*

4.4.1 Consultas SQL em testes automatizados

Outra atividade comumente realizada nos testes de aplicações que trabalham com banco de dados é a verificação de resultados e dados no próprio banco, por meio de consultas SQL. Portanto, esta implementação do sistema AutoTest e as planilhas *data-driven* devem dar suporte adequado para facilitar e tornar prático o uso de consultas SQL na verificação de resultados dos testes automatizados.

Uma vez que o resultado de uma consulta SQL pode ser grande, mas os pontos a serem verificados necessariamente não o são, e que esta linguagem de consulta permite uma ampla gama de respostas, a abordagem utilizada foi o a criação de uma linguagem simples, baseada em apenas três palavras-chave e atribuições de valores, para definir as entradas e saídas de uma SQL. A Tabela 4.1 apresenta estas palavras-chave. Outra vantagem desta abordagem é a de se poder utilizar as consultas SQL escritas na mesma forma em que são usadas nas ferramentas de desenvolvimento de PL/SQL.

Por exemplo, suponha que uma série de casos de teste utiliza a seguinte consulta SQL para verificação de resultado:

```
select distinct fk_documento_denum doc from conta_objeto c where
c.fk0objeto_de_fanum || c.fk0objeto_de_faano in (&ag_ano) order by
fk_documento_denum
```

A definição das entradas e saídas desta SQL para um determinado caso de teste pode ser simplesmente escrita como:

```
Entrada=ag_ano:8062002;Saída=5;Resultado=<linha 5>;DOC:237393
```

Palavra-chave	**Ação**	**Exemplo**
campo:valor	Atribui um valor fixo a um campo da consulta SQL ou compara o conteúdo de uma coluna com o valor dado	ag_ano:3752002
<variável>:coluna	Atribui o conteúdo da coluna da tabela-resposta da consulta SQL a um nome	<codigo_item>:cd_item
campo:<variável>	Atribui o valor de uma variável a um campo da consulta SQL ou compara o conteúdo de uma coluna com o conteúdo da variável dada	cd_item:<codigo_item>
<linha x>	Identifica a linha da tabela-resposta da consulta SQL a interagir	<linha 1>
Entrada=	Identifica os valores de entrada da SQL	Entrada=ag_ano:3752002
Saída=	Identifica quantas linhas são esperadas na resposta da consulta SQL	Saída=7
Resultado=	Identifica os resultados da consulta SQL a serem verificados	Saída=<linha 1>; num_item_fat:187

Tabela 4.1: Palavras-chave para definição de entradas e saídas de uma consulta SQL

Apesar de simples e prática, as consultas escritas desta forma começam a se tornar demasiadamente trabalhosas de se manter quando a quantidade de verificações nos resultados de uma SQL é muito grande, principalmente devido a erros de digitação.

Por este motivo foi criada a ferramenta SQLgen, integrada às planilhas *data-driven* utilizadas por esta implementação do sistema AutoTest. Por meio de uma interface gráfica, como mostra a Figura 4.6, ela permite que o analista de teste crie e execute consultas SQL às bases de dados, identifique os resultados a serem verificados e crie dependências de valores entre SQLs do mesmo caso de teste. A ferramenta gera, automaticamente, a definição de entradas e saídas da SQL no formato utilizado por esta implementação do sistema AutoTest na própria planilha, poupando o analista de teste de qualquer trabalho de digitação.

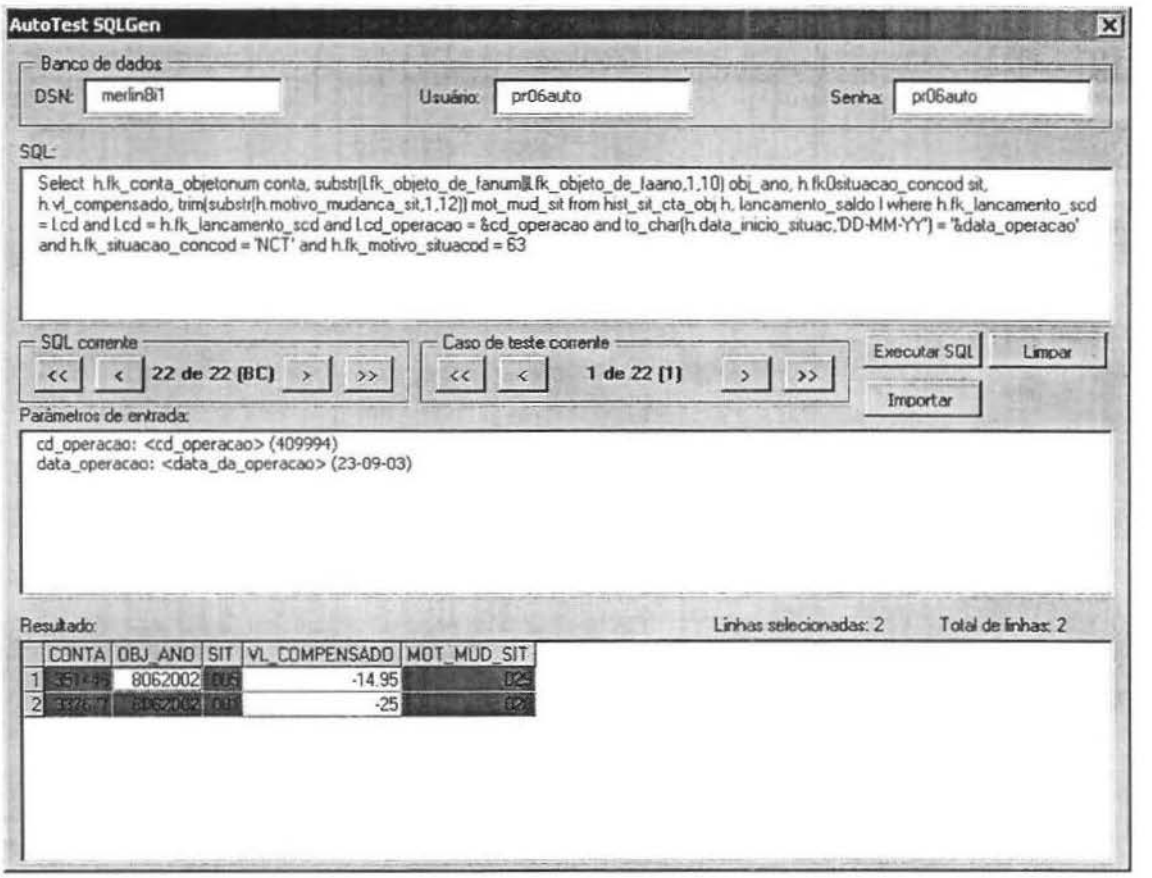

Figura 4.6: Interface com o usuário do AutoTest SQLgen

4.5 Ferramenta para execução remota/distribuída de testes

Por ser criado em cima do Functional Tester, esta implementação do sistema AutoTest requer a existência da ferramenta na máquina onde serão executados os testes, e esta, por sua vez, exige que o usuário inicie a execução do teste automatizado manualmente, por meio de sua interface gráfica.

A execução remota ou distribuída dos testes automatizados pode ser feita por meio da ferramenta TestManager da própria IBM Rational. Entretanto, esta se revelou muito complexa e limitada a ponto de não ser utilizada nesta implementação.

A solução adotada foi a criação de uma ferramenta, chamada AutoTest Remote Agent. Com uma interface hipermídia, mostrada na Figura 4.7, acessada por meio de um navegador, ele permite a qualquer pessoa iniciar e acompanhar a execução de testes automatizados, sem a necessidade de alocação de outros recursos, e com a facilidade de se fazer isso com apenas alguns cliques de mouse.

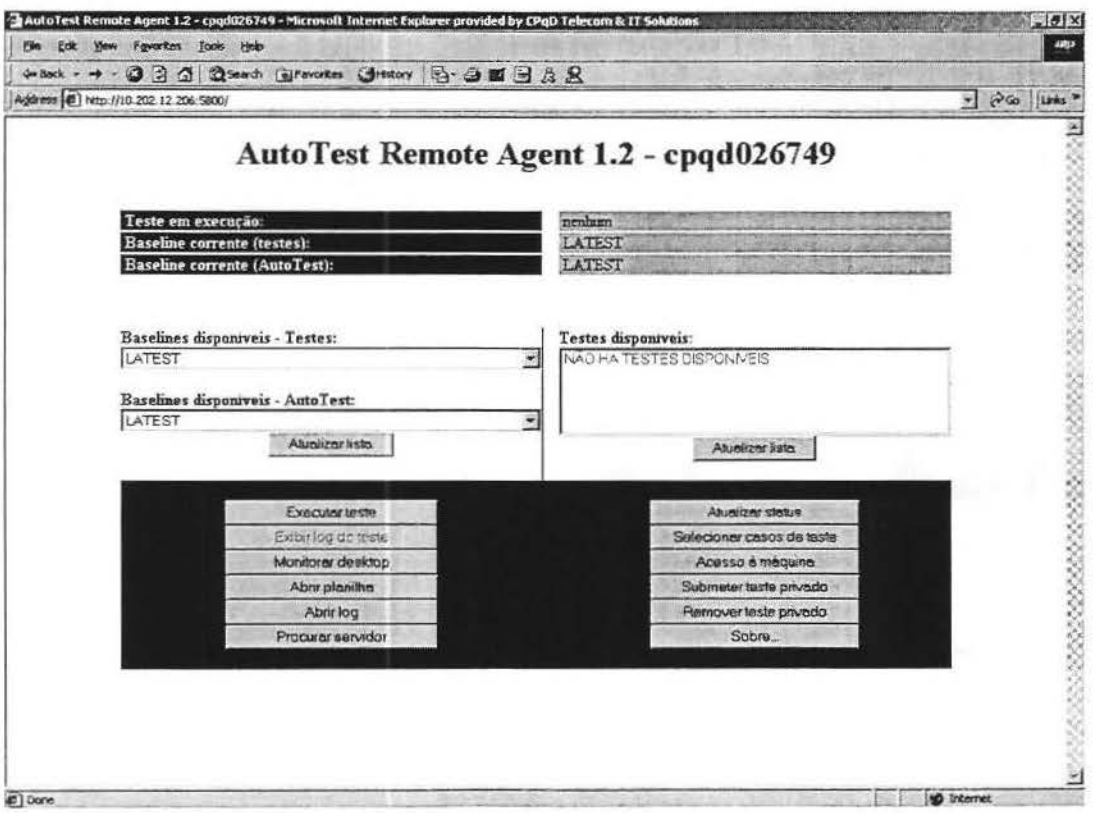

Figura 4.7: Interface hipermídia do AutoTest Remote Agent

4.5.1 Arquitetura do AutoTest Remote Agent

O AutoTest Remote Agent é composto por dois elementos: servidor e cliente. A Figura 4.8 ilustra esta divisão. Os clientes, que realmente executam o teste automatizado, têm instalado o Remote Agent e o Functional Tester, sendo máquinas dedicadas exclusivamente à execução de testes automatizados. Já o servidor, que é apenas uma máquina, não é capaz de executar nenhum teste automatizado. Entretanto, ele exerce três funções de grande importância:

- Mantém registro de clientes ativos;
- Atende requisições relacionadas às bases versionadas de testes;
- Arbitra semáforos;

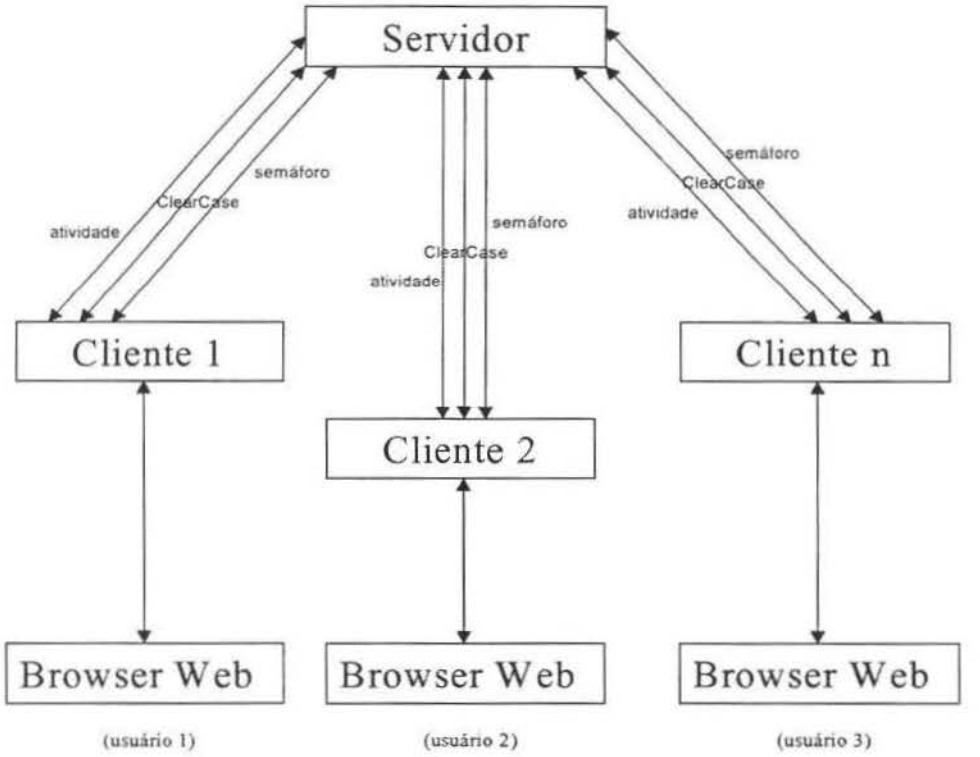

Figura 4.8: Interação cliente/servidor do AutoTest Remote Agent

A Figura 4.9 apresenta a arquitetura do AutoTest Remote Agent. O núcleo do sistema é um micro-servidor hipermídia *multithread* que roda na porta 5800. Ele pode receber três tipos de requisições:

- Requisições de páginas (tanto para servidor quanto para cliente);
- Requisições de interação com o ClearCase (apenas o servidor);
- Requisições de *download* ou *upload* (apenas o cliente);

Paralelamente ao micro-servidor hipermídia existe um outro servidor *multithread* específico, na porta 5700, responsável por também atender três tipos de requisições:

- Requisições de semáforos (apenas o servidor);
- Requisições de verificação de atividade (apenas o cliente);
- Anúncio de registro de cliente (apenas o servidor);

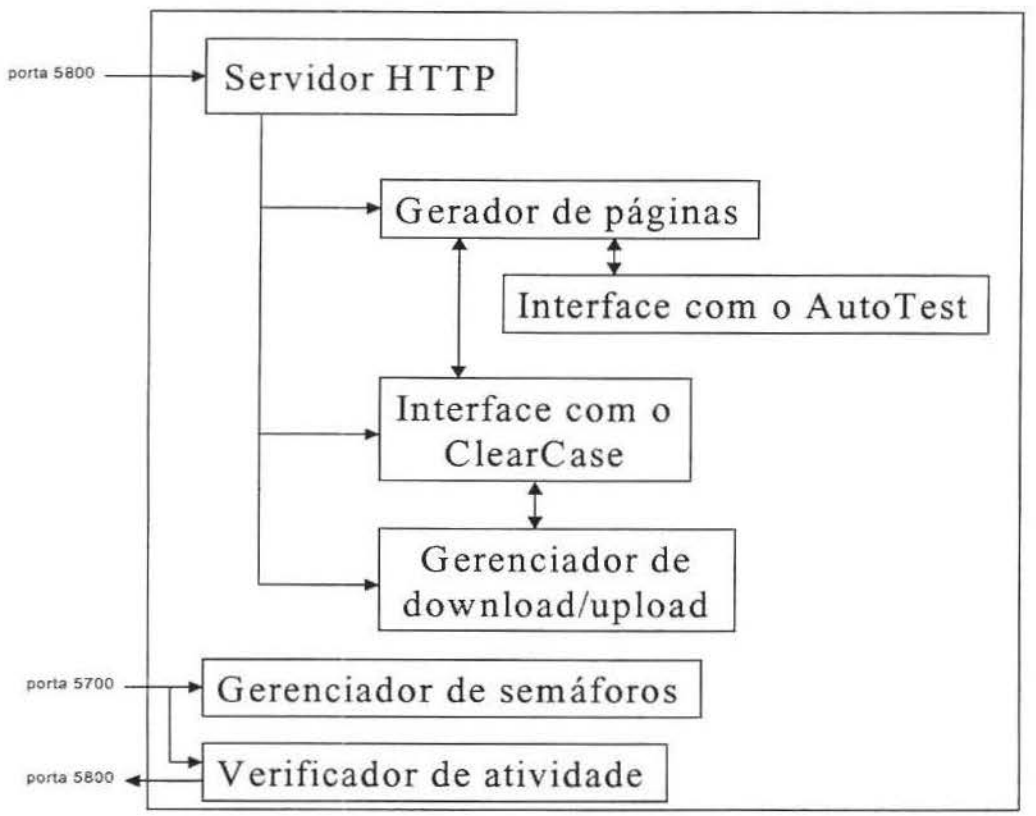

Figura 4.9: Arquitetura do AutoTest Remote Agent

O atendimento das requisições e as funções desempenhadas tanto pelo servidor quanto pelos clientes é detalhado nas próximas seções.

4.5.2 Funções do servidor

Como já apresentado, o servidor exerce três funções importantes no sistema AutoTest Remote Agent.

Registro de clientes ativos

Quando uma máquina cliente do AutoTest Remote Agent é ligada, ela envia, por *broadcast* UDP pela rede, uma mensagem de procura pelo servidor. Além de responder a este cliente qual o seu IP, o servidor também registra este cliente em uma tabela, passando a monitorar, em intervalos de 30s, se ele está ativo e atualizando esta tabela, caso necessário. A verificação se faz por uma requisição HTTP especial com a porta 5800 do cliente.

A resposta do cliente também inclui informação se ele está executando ou não um teste. A tabela de clientes disponíveis pode ser visualizada por meio de um navegador hipermídia simplesmente conectando-se ao servidor na porta 5800.

Esta abordagem, além de permitir a inclusão e exclusão de máquinas clientes no sistema sem necessidade de se alterar a configuração do servidor ou de reiniciá-lo, também unifica a lista de clientes ativos e disponíveis, não obrigando os usuários do sistema a decorarem ou manterem esta lista em separado.

Requisições de interação com o ClearCase

O servidor também possui instalados tanto a ferramenta ClearCase quanto repositórios versionados para cada uma das máquinas clientes. Por isso, todas as requisições dos clientes relacionadas ao controle de versão (como mudança de *baseline*, identificação de *baselines* disponíveis ou atualização do repositório) são atendidas por ele.

Além de facilitar a manutenção dos repositórios, concentrando-os fisicamente em uma só máquina, esta abordagem não exige instalação extra de software e configuração nas máquinas clientes, diminui o número de licenças necessárias do ClearCase e permite o uso de máquinas com menos recursos (como espaço de armazenamento) como clientes.

Toda a interação do servidor com o ClearCase é feita pela API deste, dando mais confiabilidade e desempenho às operações executadas.

Arbitragem de semáforos

Pela porta 5700 o servidor também atende pedidos de criação e verificação de semáforos feitas por métodos providos por esta implementação do sistema AutoTest. Os semáforos podem ser usados pelos testes automatizados para controle de concorrência em execuções paralelas e operações atômicas.

4.5.3 Funções dos clientes

Como já apresentado, os clientes podem receber requisições de páginas, *download*, *upload* ou verificação de atividade. Usando estes quatro tipos de requisição são montadas todas as funcionalidades disponibilizadas pelos clientes do AutoTest Remote Agent para o usuário.

Seleção de *baseline* de teste

Por duas caixas de seleção é possível selecionar um *baseline* específico, tanto para os testes automatizados quanto para a implementação do sistema AutoTest. Isto permite que se possa executar testes de qualquer versão de um aplicativo ou módulo, como também, de maneira independente, utilizar qualquer versão da implementação do sistema para isto.

Um botão "Atualizar lista" permite que, a qualquer momento, se obtenha uma lista atualizada das *baselines* dos testes automatizados e das implementações do sistema AutoTest constantes nas bases versionadas do ClearCase.

Seleção de teste

Uma lista das planilhas disponíveis para o *baseline* de testes selecionado é mostrado na página e qualquer uma delas, mas apenas uma, pode ser selecionada, sendo então alvo de outras operações disponibilizadas para o usuário e que a envolvam. Um botão "Atualizar lista" permite que, a qualquer momento, se obtenha uma lista atualizada das planilhas existentes sob a *baseline* de testes selecionada.

Execução de teste

O botão "Executar teste" dispara a execução, no cliente acessado, da planilha de teste selecionada. A partir deste momento o status de execução é mostrado no topo da página, com o

nome da planilha de teste e o tempo decorrido na execução. Também o botão "Exibir log do teste", explicado abaixo, é habilitado, e o botão "Executar teste" é substituído por "Parar teste", permitindo abortar a execução a qualquer momento. Ainda, a página passa a ser atualizada automaticamente a cada 30 segundos para refletir o status atual de execução da planilha de teste escolhida.

O disparo do teste automatizado é feito na máquina cliente pela execução de um processo externo, que inicia a máquina virtual Java e todo o ambiente do Functional Tester necessário para esta implementação do sistema AutoTest. A IDE do Functional Tester não é carregada para não consumir recursos desnecessariamente.

Uma vez que o usuário tem liberdade para modificar as *baselines* de testes e da própria implementação do sistema a qualquer tempo, e como as classes Java que compõem a implementação do sistema não são versionadas, faz-se necessária a compilação do *framework* AutoTestScript imediatamente antes da sua execução. Entretanto, essa operação não é necessária se não houve modificação no *baseline* da implementação do *framework*. Portanto, torná-la obrigatória antes de toda execução de teste constitui-se em um dispêndio desnecessário de tempo e esforço computacional.

Para se resolver isso foi utilizado o ant [Apa03], uma poderosa ferramenta de compilação de sistemas que, entre outras funcionalidades, elimina passos desnecessários (ou já executados anteriormente) no processo. Ainda, foi utilizado o jdt [Ecl03] como compilador devido ao seu alto desempenho e ao fato de ele já estar disponível através do Functional Tester, eliminando a necessidade de se também instalar o kit de desenvolvimento Java da Sun na máquina cliente. Como resultado são gastos 30 segundos para a compilação completa do *framework*, e apenas 3 segundos caso não seja necessária a compilação (verificação feita automaticamente pela ferramenta ant).

Acompanhamento da execução por relatório em tempo real

O AutoTest Remote Agent permite o acompanhamento do teste em tempo real, apresentando as mensagens de execução, um resumo com o tempo decorrido, total de casos de teste existentes, executados e falhos, assim como o número de erros e falhas detectados. A Figura 4.10 ilustra esta funcionalidade. A comunicação com a implementação do sistema AutoTest é realizada por um *applet* Java, e a formatação do relatório por meio de Javascript e DHTML.

Para viabilizar esta funcionalidade, a implementação do *framework* AutoTestScript foi estendida de modo que, no decorrer da execução do teste, as mensagens adicionadas ao relatório detalhado também são enviadas a qualquer cliente TCP/IP conectado a uma porta específica da máquina.

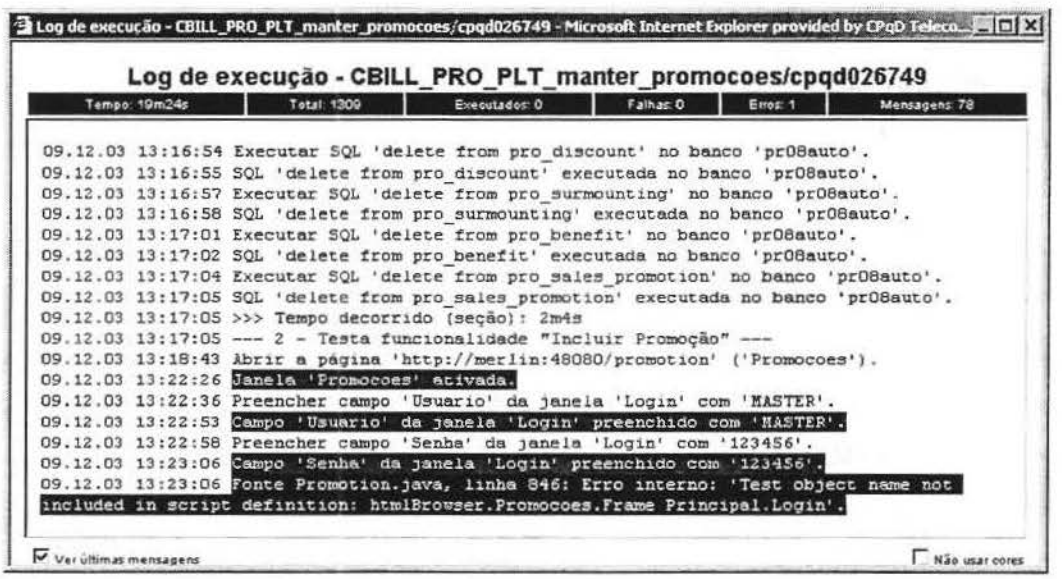

Figura 4.10: Relatório de execução em tempo real do AutoTest Remote Agent

Seleção individual de casos de teste

Para flexibilizar a execução de testes automatizados, o AutoTest Remote Agent também apresenta a funcionalidade de permitir a seleção individual dos casos de teste. Para implementar esta funcionalidade foram adicionadas duas opções à implementação do *framework* AutoTestScript: a listagem hierarquizada de casos de teste existentes em uma planilha e a passagem opcional de um arquivo-texto contendo os casos de teste que não devem ser executados. Através do uso destas duas opções, e utilizando-se JavaScript e DHTML para a interface com o usuário, foi implementada esta funcionalidade, como exemplificado na Figura 4.11.

***Download* e *upload* de arquivos**

O micro-servidor hipermídia do AutoTest Remote Agent também é capaz de gerenciar o *download* e o *upload* de arquivos binários. Isto permitiu a implementação de mais três funcionalidades ao sistema:

- *download* da planilha de testes selecionada, permitindo que o usuário possa verificá-la, obter dados ou mesmo copiá-la para a máquina local para trabalhos futuros, sem a necessidade de configurar o ClearCase para acessá-la;
- *download* do relatório detalhado de execução de teste, permitindo o acesso de maneira fácil e não obrigando o usuário a conhecer a localização do repositório do mesmo na rede;
- criação de planilhas de teste privadas pelo *upload* das mesmas para o diretório correto (tarefa gerenciada pelo próprio AutoTest Remote Agent);

As planilhas privadas não estão sob controle do ClearCase, e são removidas ou pelo próprio usuário ou na atualização da *baseline* de testes. Entretanto, uma vez feito o *upload* o sistema AutoTest as trata da mesma maneira que as outras. Isto permite ao analista de teste criar, modificar ou excluir planilhas de teste, em caráter temporário, de maneira fácil e rápida, sem a necessidade de interação com o ClearCase; portanto, sem alterar *baselines* ou versões de elementos.

Figura 4.11: Seleção de casos de teste pelo AutoTest Remote Agent

Acompanhamento visual remoto da execução de testes

Para permitir o acompanhamento visual da execução de testes foi integrado ao AutoTest Remote Agent o VNC, e também seu cliente *applet* Java.

Apesar do acompanhamento visual não ser tão efetivo quando o relatório de execução, ele é interessante para se verificar visualmente qualquer comportamento anormal da aplicação sob teste, como mostra a Figura 4.12. Outra possibilidade é utilizar máquinas remotas sem monitor para a execução dos testes.

Para evitar interações indevidas com a interface gráfica da aplicação sob teste o *applet* foi modificado para não transmitir movimentos e cliques do mouse ou o pressionamento de teclas para a máquina cliente monitorada. E por questões de espaço ocupado na tela, o *applet* também foi modificado para fazer o redimensionamento em tempo real da área de trabalho remota para 25% do tamanho original.

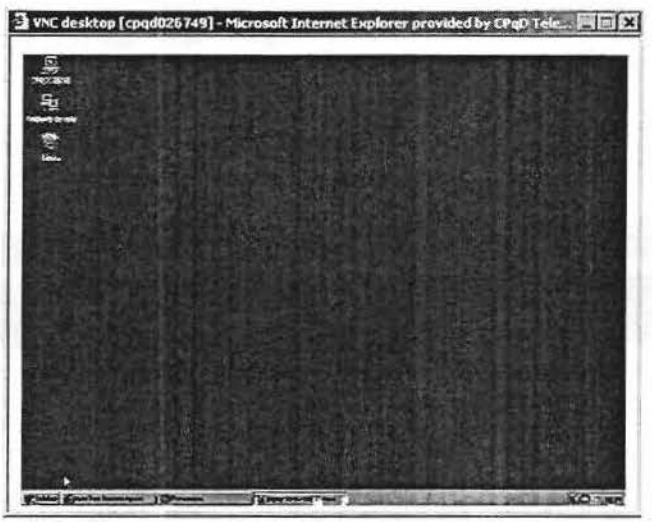

Figura 4.12: Monitoramento remoto do *desktop* de uma máquina cliente

Acesso via navegador WAP

O micro-servidor hipermídia do AutoTest Remote Agent também foi expandido para gerenciar conteúdo WML, e não apenas HTML. Assim, é possível acessar o sistema, executar testes e acompanhá-los a partir de um telefone celular com navegador WAP, como mostra a Figura 4.13.

Obviamente que as funções avançadas (*upload* e *download* de planilhas, seleção de casos de teste e acompanhamento visual e via relatório em tempo real de execução) não estão disponíveis devido aos recursos limitados dos navegadores WAP.

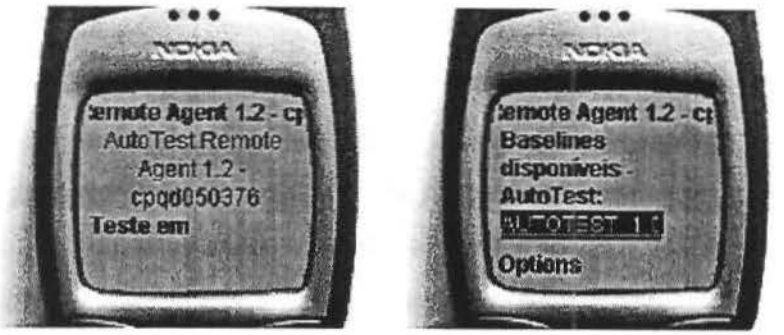

Figura 4.13: Acesso ao AutoTest Remote Agent via navegador WAP

4.6 Ferramenta para bases de dados de referência

Devido às necessidades e características específicas da DSB, optou-se por, nesta implementação do sistema AutoTest, criar um software específico, chamado "amenu", para a manutenção das bases versionadas utilizadas nos testes automatizados. As ferramentas de gerenciamento de banco de dados utilizadas pela DSB não possuem a flexibilidade necessária para permitir uma forte interação com o sistema AutoTest, e a troca do SGBD é descartada devido ao profundo impacto que isso causaria em toda a diretoria.

O processo de cópia do conteúdo de uma base de dados para um arquivo, denominado "arquivo de *dump*", é chamado de "exportação". O processo inverso, de cópia deste arquivo de *dump* para a base de dados, é chamado de "importação".

Exportando-se uma base de dados corretamente populada da aplicação sob teste, e mantendo-se o arquivo de *dump* versionado pela ferramenta ClearCase, é possível não só voltar-se o estado da base de dados ao ponto inicial do teste após ele ter sido executado, mas também voltá-la ao estado inicial em outras versões da aplicação.

O amenu facilita o trabalho de importação e exportação das bases de dados por já gerenciar todo o processo de *dump* e versionamento. Executado em modo não interativo, pela linha de comando pelo *framework* AutoTestScript, o amenu permite que os testes automatizados já

realizem a importação da base de dados antes da execução do teste, não exigindo intervenção manual para isso.

O uso de arquivos de *dump*, e não de bancos de dados completos, não só torna o versionamento mais simples pelo ClearCase como também não exige o uso de recursos computacionais preciosos para sua manutenção.

A atualização do conteúdo das bases de dados congeladas também se torna simples, bastando importar os dados do banco, realizar as alterações necessárias e exportá-los novamente.

Para maior flexibilização do trabalho, o amenu também apresenta funcionalidades, quando executado interativamente, de agendamento dos processos de importação e exportação, permitindo um melhor aproveitamento dos recursos computacionais dos servidores.

4.7 Ferramenta de análise de cobertura de código

O processo de teste definido pela Gerência de Testes de Sistemas da DSB não utiliza a medida de cobertura de código como uma métrica de qualidade para testes de sistema. Esta métrica, pelo processo definido, é utilizada apenas nos testes de unidade. Portanto, esta implementação do sistema AutoTest não contempla este componente.

4.8 Considerações sobre a implementação

No decorrer desta implementação do sistema AutoTest não foram colhidas métricas de codificação, por isso não foi possível medir o reuso e a necessidade de criação de código novo. Entretanto, uma medida qualitativa, obtida a partir do depoimento dos envolvidos nesta implementação, é que o reuso é alto, chegando a casos em que ele é praticamente 100%.

Para o teste do sistema implementado, por questão de tempo e esforço, optou-se por não se criar projeto ou casos de testes. O teste foi feito de maneira simples e direta pelos próprios implementadores à medida em que o sistema foi sendo construído.

A Tabela 4.2 apresenta os requisitos do sistema AutoTest atendidos por esta implementação. Todos eles, com exceção da coleta de métricas de cobertura de código, como apresentado na Seção 4.7, foram atendidos.

Requisito	Atendido por
Uso da técnica *Record & Playback*	IBM Rational Functional Tester
Uso de linguagem de programação para a criação de testes automatizados	IBM Rational Functional Tester
Uso das técnicas *keyword-driven* e *data-driven*	*framework* AutoTestScript
Interação com a aplicação sob teste via interface gráfica	IBM Rational Functional Tester
Interação com aplicações sem interface gráfica	execução de processos externos, protocolo telnet
Interação com outros ambientes	biblioteca jCIFS, protocolo telnet
Acesso ao banco de dados	tecnologia JDBC
Fácil expansão de pontos de verificação	IBM Rational Functional Tester
Fácil manutenção e criação de dados para uso das técnicas *keyword-driven* e *data-driven*	Microsoft Excel, biblioteca Java Excel API
Abstração dos elementos de interface gráfica	IBM Rational Functional Tester
Interação com sistemas de controle de versão	IBM Rational Functional Tester, IBM Rational ClearCase
Uso de base de dados versionadas de referência	software amenu
Execução remota de testes automatizados	software AutoTest Remote Agent
Execução em paralelo ou distribuída dos testes automatizados	software AutoTest Remote Agent
Coleta de métricas de cobertura de código durante a execução dos testes	N/A
Visualização dos erros e falhas detectados após a execução do teste	ferramenta Camtasia Publisher
Relatórios de execução detalhados/resumidos de fácil acesso	*framework* AutoTestScript
Integração com o processo de desenvolvimento	IBM Rational Test Manager

Tabela 4.2: Requisitos atendidos por esta implementação do sistema AutoTest

Capítulo 5

Resultados obtidos

Este capítulo apresenta os resultados da automação dos testes funcionais, utilizando a implementação do sistema AutoTest descrita no Capítulo 4, de três módulos do sistema de faturamento desenvolvido pela Diretoria de Soluções em Billing (DSB) do CPqD. A realização de todo o trabalho para a obtenção dos resultados apresentados só foi possível graças à colaboração da Gerência de Testes de Sistemas da DSB, em particular da equipe de automação de teste.

Este capítulo está organizado da seguinte forma: a Seção 5.1 apresenta algumas considerações e explicações necessárias para o correto entendimento das métricas apresentadas; as Seções 5.2, 5.3 e 5.4 apresentam os resultados propriamente ditos e a Seção 5.5 os principais pontos de melhoria identificados no decorrer dos três trabalhos.

5.1 Considerações sobre as métricas apresentadas

Algumas explicações se fazem necessárias para o correto entendimento dos números e resultados que serão apresentados nas próximas seções deste capítulo.

Projeto de teste

O projeto de teste é comum para o teste manual ou automatizado, ou seja, é utilizado o mesmo projeto de teste para as duas abordagens. Entretanto, o projeto de teste completo também envolve um teste manual, justamente para validar a corretude dos casos de teste ([Few99]).

Cobertura dos testes

A cobertura dos testes é baseada nas funcionalidades a serem testadas, e não no código exercitado. Assim, por exemplo, uma cobertura dos testes de 50% significa que metade das funcionalidades do módulo são testadas, e não que os testes exercitam 50% do seu código. É importante ressaltar que uma funcionalidade pode requerer mais de um caso de teste para ser

testada, portanto esta medida não fornece informação de quanto esforço é necessário para se executar manualmente os casos de teste não automatizados. Por este dado não ter sido coletado durante a realização dos trabalhos, a medida de cobertura deve ser interpretada como uma grandeza relativa da complexidade do teste.

Tempos

Os tempos apresentados são para apenas uma pessoa realizar o trabalho. Se o projeto envolve mais pessoas, os tempos individuais são somados.

Tempos de execução automática e manual

Os tempos apresentados incluem não só a execução do teste, mas também a importação de dados, se necessária, e a análise e registro dos erros e falhas.

Retestes

É considerado necessário retestar a aplicação quando:

- existe uma evolução da mesma;
- um defeito é corrigido e a aplicação modificada.

5.2 Automação de testes do módulo "CPqD Revenue Match"

O módulo "CPqD Revenue Match" realiza basicamente duas operações fundamentais: a comparação e a identificação de serviços. Trata-se de um processo entre duas empresas de telefonia que visa verificar a origem e propriedade de cada uma das ligações envolvidas. Para tanto são usadas as funcionalidades de comparação e de identificação de serviços.

Além disso, o módulo também conta com várias outras funcionalidades de cadastros, que têm por objetivo configurar a maneira como as funcionalidades de comparação e identificação de serviços operam. Alguns destes cadastros são pré-requisitos para que estas funcionalidades possam ser usadas. Todas as funcionalidades deste módulo são interativas, por meio de uma interface hipermídia.

5.2.1 Projeto de automação de teste

O objetivo da automação de teste do "CPqD Revenue Match" foi, a princípio, realizar os testes de comparação de serviços e, posteriormente, de identificação. Entretanto, algumas funcionalidades da aplicação são pré-requisitos para a comparação de serviços: o cadastro de regras de comparação e o cadastro de agendamento. Do mesmo modo, outras funcionalidades são pré-requisitos para a identificação de serviços: cadastro de horários, cadastro de intervalos, cadastro de serviços, cadastro de regras de identificação e cadastro de agendamento.

Apesar do projeto não contemplar o teste das funcionalidades de cadastro, o teste automatizado precisa ser capaz de realizá-lo a fim de satisfazer os pré-requisitos para realizar uma comparação ou identificação de serviços.

Utilizando o sistema AutoTest foram criados *scripts* para os cadastros e também para a comparação e identificação de serviços.

A execução completa do teste automático para a comparação de serviços consiste em, inicialmente, cadastrar as regras de comparação e, posteriormente, cadastrar um agendamento e dispará-lo. Note-se que estes cadastros são específicos para o teste que se deseja realizar, por isso a necessidade de se desenvolver os *scripts* que os executem. Após a comparação temos o processo de verificação, que confere os resultados da comparação com os esperados.

Para a identificação de serviços o processo é um pouco mais complicado: deve-se cadastrar os horários que serão utilizados na identificação de serviços, os serviços relativos a esses horários, os intervalos, as regras de identificação e o agendamento, que será disparado para a efetiva realização do processo de identificação. Assim como na comparação, estes cadastros são específicos para o teste que se deseja realizar. Após o término do processo de identificação temos a conferência e verificação dos resultados.

5.2.2 Resultados

A Tabela 5.1 apresenta as métricas mais importantes da automação dos testes funcionais do "CPqD Revenue Match". A Tabela 5.2 sumariza os resultados, em termos de tempo, da automação dos testes funcionais do módulo.

Todo o esforço para criação dos *scripts* de testes automatizados, incluindo estudo de viabilidade, desenvolvimento e testes dos *scripts* e da aplicação, consumiu 88 horas, o que equivale a aproximadamente meio mês de trabalho. Esse tempo é justificado pela complexidade dos testes de comparação e identificação e também pela grande quantidade de *scripts* auxiliares de cadastro.

Comparando a criação dos testes automatizados e manuais temos uma diferença de 88 horas a mais para o primeiro. Entretanto, pode-se criar o projeto de teste em paralelo com os *scripts*, caindo então esse tempo para 88 horas e praticamente se igualando ao tempo da criação dos testes manuais. Entretanto, esta otimização só foi percebida, e concluída como viável neste caso, ao final dos trabalhos de automação.

Métrica	**Medida**
Número de casos de teste	131
Cobertura dos testes automatizados[1]	65%
Tempo de projeto de teste	88 h
Tempo de execução manual	8 h
Tempo de execução automática	13 m 6 s
Esforço de automação	88 h
Número de retestes	1
Erros/falhas detectados	1
Severidade dos erros/falhas detectados	alta
Tempo de análise de relatório/ajuste/registro	10 m

Tabela 5.1: Métricas do trabalho de automação de testes do "CPqD Revenue Match"

Métrica	**Medida**
Tempo do teste manual	96 h
Tempo do teste automatizado	176 h
Economia	-80 h / -84 %

Tabela 5.2: Ganho de tempo com a automação de testes do "CPqD Revenue Match"

[1] cobertura das funcionalidades testadas, conforme definido na Seção 5.1

O tempo necessário para se realizar o teste manualmente é de, aproximadamente, 8 horas. Já automaticamente se gasta, em média, apenas 13m12s. Evidentemente que o ganho de tempo é muito grande, já que o tempo de execução do teste automático não representa nem 3% do tempo do teste manual. Mas, para compensar o esforço gasto na criação dos testes automatizados, seria necessário executar pelo menos 12 retestes, um número elevado. A Figura 5.1 apresenta as estimativas de tempo gasto no teste manual e automático e o conseqüente ganho em relação ao número de retestes feitos.

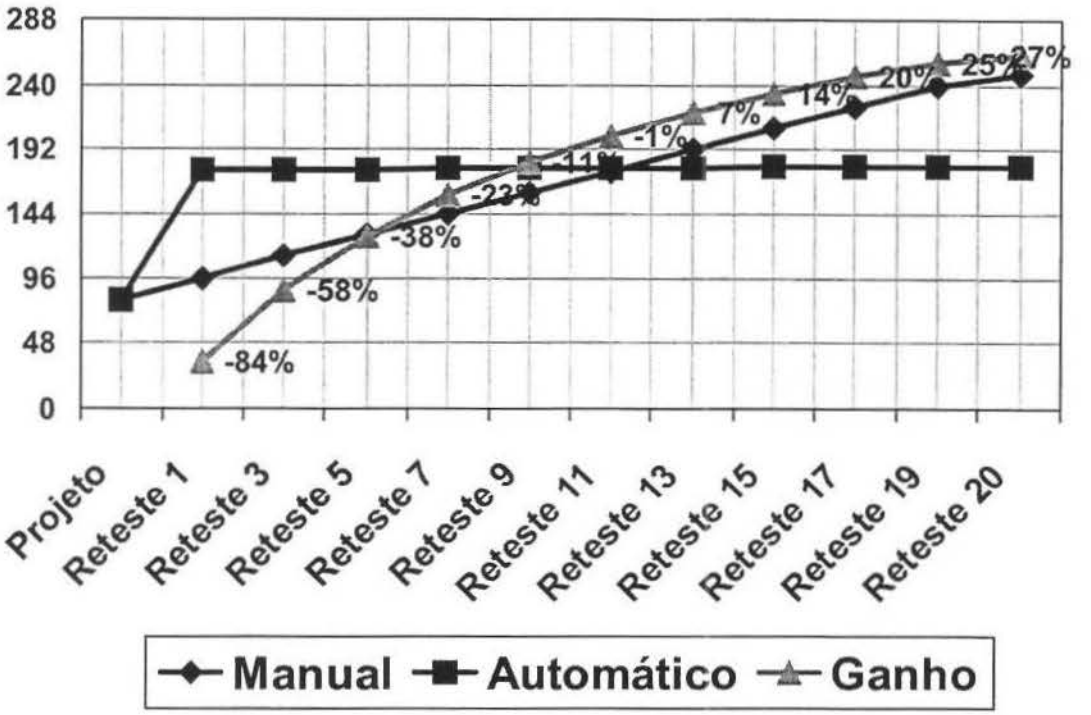

Figura 5.1: Estimativas de tempo e ganho nos retestes do "CPqD Revenue Match"

Embora tenha-se conseguido um ganho grande em termos de tempo de execução do teste, o tempo de projeto foi muito mais significativo, principalmente por que foi realizado um único reteste da aplicação. Um fator que contribuiu fortemente para o tempo de projeto foi a necessidade de se criar *scripts* para realizar os cadastros necessários para a execução das funcionalidades de comparação e identificação de serviços. O projeto, desenvolvimento e testes desses *scripts* consumiram mais de 60% do tempo de projeto do teste automatizado.

Um grande problema enfrentado no trabalho de automação dos testes foi a natureza da própria aplicação: por ser um sistema de comparação cujo algoritmo ainda não estava totalmente estável e sofrendo constantes alterações, os testes automatizados necessitariam de muitos ajustes durante o decorrer do projeto. Estes ajustes acabariam por aumentar o tempo de manutenção por demasiado. Somado a um problema interno da DSB de indisponibilidade de analistas de teste para se dedicarem a esta tarefa, os trabalhos foram suspensos.

Apesar de não haver métricas quantitativas de indicação do aumento da qualidade do software, este é inexpressivo devido ao baixo número de erros e falhas detectados.

5.3 Automação de testes do módulo "CPqD Promotion"

O módulo "CPqD Promotion" é responsável pela implementação das promoções lançadas pelas empresas de telecomunicações com o objetivo de fidelizar seus clientes e cativar novos. Cada promoção é formada por um conjunto de benefícios, que podem ser descontos ou franquias sobre os serviços usados pelo cliente e cobrados em conta telefônica.

O módulo conta com as funcionalidades de criação, exclusão, edição e aplicação de promoções. As três primeiras são realizadas de forma interativa, por meio de uma interface hipermídia. A última é realizada automaticamente durante a execução do ciclo de faturamento da companhia telefônica, e não tem interação com o usuário, sendo um processo *batch*. Como resultado, itens de faturamento resultantes da aplicação das promoções são criados no banco de dados do sistema de faturamento.

5.3.1 Projeto de automação de teste

Dado que o módulo "CPqD Promotion" pode ser dividido em duas funcionalidades básicas, "manter promoções" (ou seja, a criação, edição e exclusão das mesmas) e "aplicar promoções" (ou seja, a criação de itens de desconto com base nas promoções que se aplicam ao cliente em questão), com objetivos diferentes, o projeto e a automação dos testes também foi dividido.

O projeto de teste abrange as funcionalidades de criação válida, criação inválida, alteração válida, alteração inválida, exclusão válida, exclusão inválida e aplicação válida de promoções.

O teste das funcionalidades de criação, alteração e exclusão de promoções é feito de maneira linear: os casos de teste são executados em seqüência e os resultados verificados imediatamente após cada ação, de acordo com o procedimento de teste.

Para o teste da funcionalidade de aplicação válida de promoções é necessário, primeiramente, realizar o cadastro das promoções a serem aplicadas para, então, realizar a aplicação propriamente dita das mesmas. Para isolar cada caso de teste, antes da aplicação da promoção são invalidadas todas as outras.

Ainda, os casos de teste da interface do sistema também incluem verificações no banco de dados para verificar a sincronia entre o estado da base de dados e os resultados exibidos na interface.

A funcionalidade "aplicar promoções", por também utilizar dados de entrada já existentes no banco de dados, exigiu a criação de uma base de dados exclusiva, versionada e congelada, para os testes automatizados.

5.3.2 Resultados

A Tabela 5.3 apresenta as métricas mais importantes da automação dos testes funcionais do "CPqD Promotion". A Tabela 5.4 sumariza os resultados, em termos de tempo, da automação dos testes funcionais do módulo.

A criação dos *scripts* de teste em si requereu 257 horas para as funcionalidades de "manter promoções" e 94 horas para a funcionalidade de "aplicar promoções". Todo o esforço para criação dos *scripts* requereu 351 horas de trabalho. Isto equivale a aproximadamente dois meses de trabalho. Esse tempo é justificado pela complexidade dos testes, além do fato da aplicação, no momento da criação e teste dos *scripts*, ter apresentado problemas de instabilidade.

Comparando a criação dos testes automatizados com os manuais temos uma diferença de 351 horas a mais para o primeiro. Entretanto, neste projeto não pôde ser aproveitado o paralelismo entre as atividades de criação de *scripts* e projeto de teste, pois este último foi feito antes de iniciados os trabalhos de automação por motivos de cronograma do produto.

O tempo gasto para realizar todo o teste manualmente é de 123 horas. Já automaticamente necessita-se, em média, de 14 horas. Evidentemente que o ganho de tempo é muito grande, uma vez que o tempo de execução do teste automático é cerca de 10% do tempo do teste manual. Mas

para compensar o esforço gasto da criação dos testes automatizados seria necessário executar mais quatro retestes, o que é perfeitamente razoável. A Figura 5.2 apresenta as estimativas de tempo gasto no teste manual e automático e o conseqüente ganho em relação ao número de retestes feitos.

Métrica	Medida
Número de casos de teste	1644
Cobertura dos testes automatizados[1]	88%
Tempo de projeto de teste	224 h
Tempo de execução manual	123 h
Tempo de execução automática	14 h
Esforço de automação	351 h
Número de retestes	1
Erros/falhas detectados no teste manual	174
Erros/falhas detectados no teste automatizado	39
Severidade dos erros/falhas detectados	alta: 13 média: 14 baixa: 12
Tempo de ajuste/registro de erro/falha (manual)	34 h
Tempo de análise de relatório/ajuste/registro	28 h

Tabela 5.3: Métricas do trabalho de automação de testes do "CPqD Promotion"

Métrica	Medida
Tempo do teste manual	347 h
Tempo do teste automatizado	617 h
Economia	-270 h / -77 %

Tabela 5.4: Ganho de tempo com a automação de testes do "CPqD Promotion"

[1] cobertura das funcionalidades testadas, conforme definido na Seção 5.1

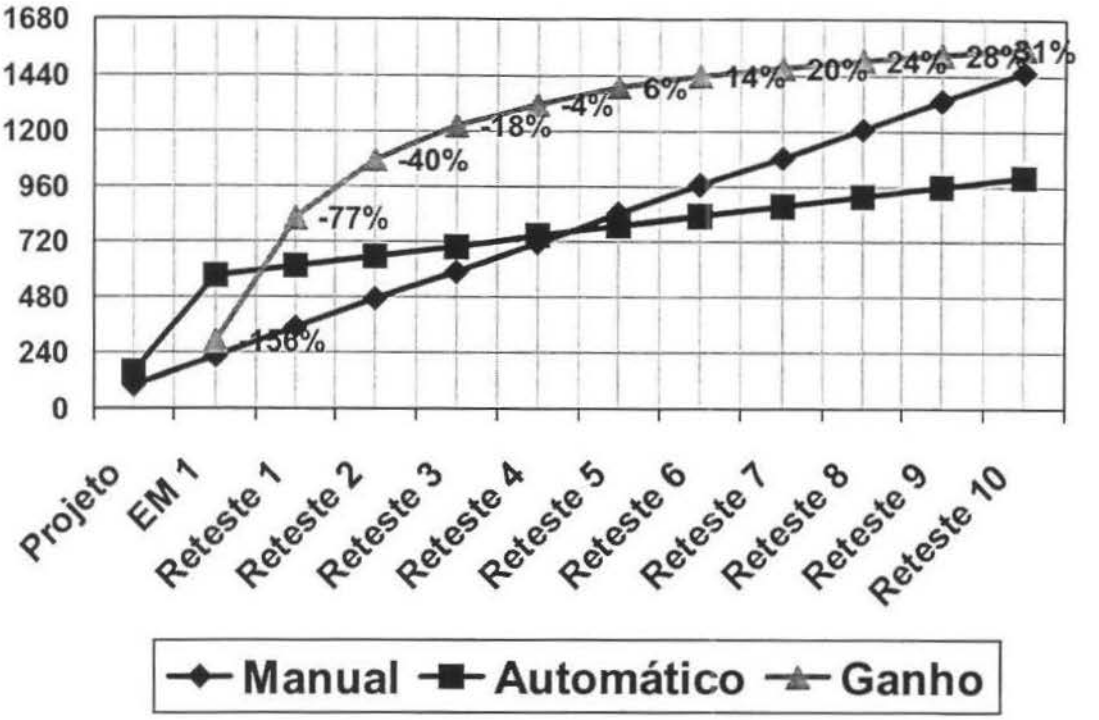

Figura 5.2: Estimativas de tempo e ganho nos retestes do "CPqD Promotion"

Justamente pelo fato do projeto de teste ter sido criado antes de iniciados os trabalhos de automação de teste, ele sofreu uma evolução para melhorar sua cobertura quando automatizado. A Tabela 5.5 apresenta as métricas desta evolução. Dois fatores foram decisivos para a evolução do projeto de testes: a necessidade de reduzir o tempo gasto nos testes integrados de componentes por motivos de cronograma, substituindo-os por testes sistêmicos; a facilidade com que novos casos de teste puderam ser inseridos nas planilhas de teste, graças ao modelo utilizado pelo sistema AutoTest.

Apesar de não ter havido economia no tempo de teste, dado o baixo número de retestes da aplicação até o momento, o resultado positivo da automação de teste se refletiu em outro ponto muito importante: a qualidade do produto de software. O teste automatizado do "CPqD Promotion" detectou 39 erros, distribuídos igualmente em três categorias de severidade (alta, média e baixa). Destes 39 erros, 13 haviam sido detectados no teste manual e, supostamente, corrigidos, o que não se revelou verdade pelo teste automatizado. A análise destes 13 erros revelou que mais da metade deles possuía severidade média ou alta.

Apesar de não haver métricas quantitativas de indicação do aumento da qualidade do software, foi obtida uma medida qualitativa deste fato: não houve, por parte dos clientes, nenhuma reclamação de problemas no módulo que, inclusive, foi elogiado por sua alta estabilidade.

Métrica	Medida
Número de casos de teste adicionados	714
Tempo de evolução do projeto de teste	24 h
Cobertura inicial dos casos de teste automatizados[1]	65%

Tabela 5.5: Evolução do projeto de teste do "CPqD Promotion"

5.4 Automação de testes do módulo "Atendimento ao Cliente"

O módulo "Atendimento ao Cliente" é um dos mais importantes módulos que compõem o sistema de faturamento desenvolvido pela DSB. Ele é utilizado pela equipe da empresa operadora de telecomunicações no atendimento a clientes que possuem dúvidas ou reclamações sobre serviços ou valores cobrados em sua conta telefônica. As reclamações podem ocasionar a geração de uma nova conta ou alterações a serem refletidas em uma conta futura.

O módulo é uma aplicação com interface hipermídia e requer interação com o servidor de banco de dados para consulta ou atualização de informações referente às contas dos clientes.

Basicamente o módulo "Atendimento ao Cliente" é composto de três funcionalidades principais: "reclamação de conta", "retificação de conta" e "cancelamento de reclamação/retificação de conta". Entretanto, pelo fato da DSB possuir clientes internacionais, as funcionalidades de reclamação e retificação apresentam duas variantes no algoritmo de cálculo, que se adequam a dois tipos de modelos contábeis utilizados pelos clientes da diretoria: o modelo "*on balance*" e o modelo "por documento". Para os objetivos desta seção não se faz necessário explicar as diferenças e detalhes dos modelos, mas tão somente salientar que, devido a elas, o módulo "Atendimento ao Cliente" acaba se dividindo em duas variantes independentes.

[1] cobertura das funcionalidades testadas, conforme definido na Seção 5.1

5.4.1 Projeto de automação de teste

Dado que o módulo "Atendimento ao Cliente" possui três funcionalidades básicas e duas variantes de cálculo, ele pode ser dividido em oito funcionalidades: "reclamação *on balance*", "cancelamento de reclamação *on balance*", "reclamação por documento", "cancelamento de reclamação por documento", "retificação *on balance*", "cancelamento de retificação *on balance*", "retificação por documento", "cancelamento de retificação por documento".

O teste das funcionalidades é feito de maneira linear: os casos de teste são executados em seqüência e os resultados verificados imediatamente após cada ação, de acordo com o procedimento de teste.

Ainda, os casos de teste da interface do sistema também contemplam verificações no banco de dados para testar a sincronia entre o estado da base de dados e os resultados apresentados na interface.

O "Atendimento ao Cliente" está atualmente em sua terceira evolução dos testes automatizados. A Tabela 5.6 sumariza as principais mudanças para cada versão do sistema.

Versão do sistema	**4.2**	**4.3**	**4.4**
Número de casos de teste	19	244	246
Esforço para evolução dos testes automatizados	205 h	1431 h	1037 h
Cobertura dos testes automatizados[1]	9%	86%	71%
Esforço por caso de teste	32 h	8 h	4 h
Número de funcionalidades testadas	1	8	8

Tabela 5.6: Evolução do projeto de teste do "Atendimento ao Cliente"

Da versão 4.2 para a versão 4.3 destaca-se o expressivo aumento do número de casos de teste automatizados, e conseqüentemente do esforço para a sua automação, mas também a acentuada queda deste esforço por caso de teste. Esta queda é explicada pela institucionalização e melhoria do processo de automação de teste (incluindo-se a criação de ferramentas para isto), o aumento de experiência da equipe envolvida e o caráter de evolução, e não criação, do projeto de teste. Da versão 4.3 para a versão 4.4 não houve evolução significativa no número de casos de

[1] cobertura das funcionalidades testadas, conforme definido na Seção 5.1

teste, apesar de outras funcionalidades terem sido incluídas no sistema, por motivos de estratégia da gerência de testes, o que se reflete também na cobertura do teste automatizado. Entretanto, o número de horas necessárias para a evolução dos testes automatizados, que teoricamente deveria ser baixo, não o foi devido ao grande número de problemas de ambiente computacional enfrentados durante este trabalho.

5.4.2 Resultados

As Tabelas 5.7, 5.8, 5.9, 5.10, 5.11 e 5.12 apresentam as métricas mais importantes, e também sua sumarização em termos de tempo, da automação dos testes funcionais das três versões do módulo "Atendimento ao Cliente".

Métrica	Medida
Número de casos de teste	19
Cobertura dos testes automatizados[1]	9%
Tempo de projeto de teste	205 h
Tempo de execução manual	45 h
Tempo de execução automática	1h 30 m
Esforço de automação	398 h
Número de retestes	3
Erros/falhas detectados no teste automatizado	0
Severidade dos erros/falhas detectados	N/A
Tempo de análise de relatório/ajuste/registro	2 h

Tabela 5.7: Métricas do trabalho de automação do "Atendimento ao Cliente", versão 4.2

[1] cobertura das funcionalidades testadas, conforme definido na Seção 5.1

Métrica	Medida
Tempo do teste manual	296 h
Tempo do teste automatizado	605 h
Economia	-309 h / -104 %

Tabela 5.8: Ganho de tempo com a automação do "Atendimento ao Cliente", versão 4.2

Métrica	Medida
Número de casos de teste	244
Cobertura dos testes automatizados[1]	86%
Tempo de evolução	1431 h
Tempo de execução manual	562 h
Tempo de execução automática	9 h
Esforço de automação	520 h
Número de retestes	4
Erros/falhas detectados no teste automatizado	21
Severidade dos erros/falhas detectados	alta 10 média 4 baixa 7
Tempo de análise de relatório/ajuste/registro	110 h

Tabela 5.9: Métricas do trabalho de automação do "Atendimento ao Cliente", versão 4.3

Métrica	Medida
Tempo do teste manual	3117 h
Tempo do teste automatizado	2153 h
Economia	965 h / 31 %

Tabela 5.10: Ganho de tempo com a automação do "Atendimento ao Cliente", versão 4.3

[1] cobertura das funcionalidades testadas, conforme definido na Seção 5.1

Métrica	Medida
Número de casos de teste	246
Cobertura dos testes automatizados[1]	71%
Tempo de evolução	1037 h
Tempo de execução manual	562 h
Tempo de execução automática	9 h
Esforço de automação	28 h
Número de retestes	7
Erros/falhas detectados no teste automatizado	18
Severidade dos erros/falhas detectados	alta 6 média 8 baixa 7
Tempo de análise de relatório/ajuste/registro	62 h

Tabela 5.11: Métricas do trabalho de automação do "Atendimento ao Cliente", versão 4.4

Métrica	Medida
Tempo do teste manual	4409 h
Tempo do teste automatizado	1266 h
Economia	3142 h / 71 %

Tabela 5.12: Ganho de tempo com a automação do "Atendimento ao Cliente", versão 4.4

A Tabela 5.13 apresenta os tempos gastos exclusivamente com o projeto de teste do módulo "Atendimento ao Cliente" (ou seja, não considerando o teste manual do projeto). Para a versão 4.2 este tempo é alto, em relação ao reduzido número de casos de teste, por se tratar da criação, e não evolução, do projeto. Para a versão 4.3, em que foram adicionados 225 novos casos de teste, o tempo sobe significativamente. Finalmente, para a versão 4.4, em que houve apenas a equalização dos resultados dos casos de teste com as novas funcionalidades do sistema, este valor cai bastante. A queda não é mais acentuada devido ao grande número de casos de teste

[1] cobertura das funcionalidades testadas, conforme definido na Seção 5.1

que tiveram que ser equalizados e também devido a problemas de infra-estrutura enfrentados durante a realização do trabalho.

A Tabela 5.13 também apresenta os tempos gastos exclusivamente com a criação ou manutenção dos *scripts* de teste do módulo "Atendimento ao Cliente". Comprova-se que, para a criação dos *scripts*, o esforço é grande, ao contrário do necessário para sua manutenção apenas.

Por fim, a Tabela 5.13 apresenta ainda o tempo gasto na criação ou manutenção dos ambientes exclusivos para os testes automatizados. Em face do que já foi apresentado sobre o comportamento normal dos números, o único ponto discrepante é o alto tempo gasto para a manutenção do ambiente da versão 4.4 do módulo "Atendimento ao Cliente". Ele é explicado pelos inúmeros problemas enfrentados durante a migração do ambiente de automação para esta versão do módulo.

Versão	Projeto de teste	*Scripts* de teste	Ambiente
4.2	96 h	398 h	64 h
4.3	371 h	520 h	498 h
4.4	68 h	28 h	407 h

Tabela 5.13: Tempos de projeto de testes e ambiente do módulo "Atendimento ao Cliente"

Na questão do esforço para a criação ou manutenção do projeto de teste automatizado comparado com o manual, os números apresentados nas tabelas anteriores reforçam a discrepância entre eles, não sendo novidade. Para o "Atendimento ao Cliente", nas versões 4.3 e 4.4, foi aproveitado o paralelismo entre as atividades de criação de *scripts* e projeto de teste.

O resultado quantitativo mais significativo é justamente a economia de tempo na execução dos testes. Consideradas as três versões do módulo, temos um tempo acumulado de 8991 horas para o teste manual e 3723 horas para o teste automatizado, resultando em uma economia de 5268 horas, ou 59% do tempo do teste manual. A Figura 5.3 apresenta as evoluções de tempo gasto no teste manual e automático acumulado nas três versões do módulo, e o conseqüente ganho também acumulado.

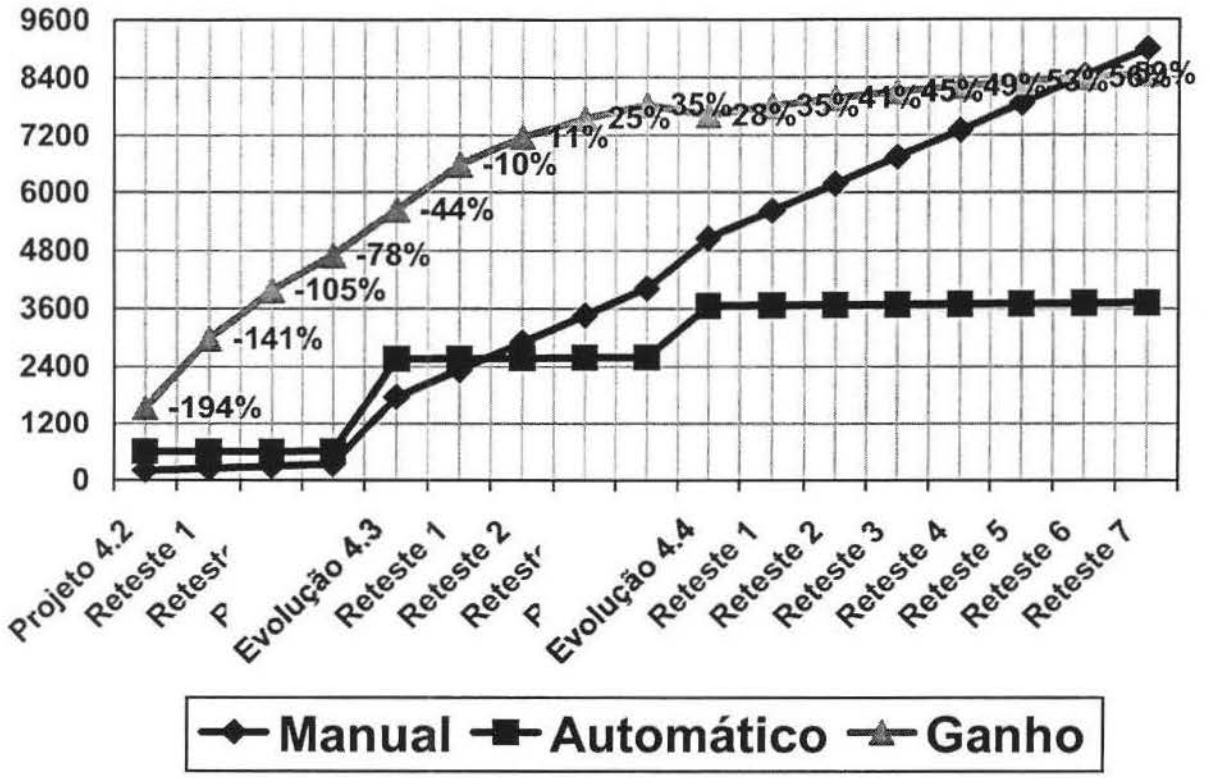

Figura 5.3: Tempos e ganho em retestes do módulo "Atendimento ao Cliente"

Apesar de a economia em tempo de execução de teste ter sido o maior ganho neste caso, existem ainda outros dois resultados positivos: o aumento da qualidade do produto de software e o aumento do moral dos analistas de teste. O primeiro resultado pôde ser comprovado tanto pela queda de 40% no número de chamados ao *helpdesk* relativos ao módulo "Atendimento ao Cliente" quanto pelas características de alguns erros que, segundo os analistas de teste, nunca seriam detectados pelo teste manual. O segundo resultado é comprovado pelo testemunho dos próprios analistas de teste envolvidos com o módulo: a não necessidade de se executar o teste manual, que é bastante trabalhoso, entediante e cansativo, e a conseqüente mudança de foco dos analistas para um trabalho mais criativo e investigativo, contribuíram bastante para a elevação do moral entre os membros da equipe.

5.5 Pontos de melhoria

No decorrer dos trabalhos de automação de teste dos três módulos apresentados anteriormente, vários pontos de melhoria no processo e no próprio sistema AutoTest foram identificados, inclusive alguns já trabalhados, e são apresentados a seguir.

Tempo de preenchimento da planilha

O preenchimento dos dados de entrada e saída das SQLs de consulta a banco de dados torna-se demasiadamente trabalhoso e altamente sujeito a erros de digitação quando a quantidade de verificações nos resultados é muito grande. Este ponto foi melhorado com a criação da ferramenta SQLgen, tornando o esforço nesta tarefa praticamente nulo.

Preparação da massa de dados

A preparação de massa de dados, apesar de ser um trabalho cansativo e demorado, uma vez realizada não requer manutenção, a menos de uma evolução do sistema. Para melhorar o trabalho, é interessante o estudo e a adoção de ferramentas que facilitem as atividades dos analistas. Como exemplo podemos citar ferramentas de desenvolvimento PL/SQL ou mesmo interfaces especiais para a entrada de dados no sistema.

Qualidade dos requisitos

É importante que os requisitos da aplicação a ser testada estejam claros e corretos, de modo que o projeto de teste não necessite de alterações futuras que poderiam ser evitadas. Isto pode aumentar de sobremaneira o custo da criação dos testes automatizados.

Modificações de interface

No ambiente hipermídia existem várias maneiras de se obter um mesmo resultado visual. Por exemplo, a centralização de um gráfico em uma página pode ser feita tanto usando uma tabela quanto DHTML. Entretanto, para a ferramenta de automação de teste, cada uma destas abordagens gera um mapa de interface diferente, todos incompatíveis entre si.

Alterações cosméticas na interface gráfica com o usuário devem ser estudadas de modo a não tornar muito grande o esforço de manutenção do mapa de interface (ou mesmo dos *scripts* de teste) dos sistemas que já possuem testes automatizados.

Também, a codificação das interfaces gráficas pode ser feita seguindo-se um conjunto de regras que facilitem, o máximo possível, a utilização da ferramenta de geração e manutenção de mapa de interface, minimizando o retrabalho neste tipo de atividade. Obviamente, este conjunto de regras varia de acordo com a ferramenta adotada, exigindo um estudo prévio da mesma para a sua definição.

Tempo de vida das funcionalidades

Os sistemas a terem seus testes automatizados devem ser estudados para se verificar a existência das funcionalidades alvo de teste em versões futuras. Caso esse estudo não seja feito, o trabalho acaba sendo sobre exceções e não regras, e conseqüentemente o custo torna-se maior que o benefício obtido.

Intercalação de trabalhos

O trabalho do analista de teste não termina com a conclusão do projeto de teste automatizado. É importante considerar o tempo de análise dos relatórios de execução. Se o analista é alocado para outras atividades, pode não ter tempo hábil para esta análise, impactando no cronograma dos trabalhos de automação de teste.

Alocação da automação de testes em cronograma

Os trabalhos de automação de teste não podem ser incluídos no cronograma de um novo produto. Pelos motivos já citados de estabilidade da aplicação, eficiência do teste automatizado, ganho com o número de retestes e necessidade do teste manual na criação do projeto de teste, fica claro que os trabalhos devem começar após o término da primeira versão do sistema, focando o teste de regressão.

Uso de SSH

Para a execução de testes entre *sites* diferentes e mesmo por questões de segurança, é interessante dotar o sistema AutoTest da opção de uso do protocolo SSH [Bar01], e não apenas telnet.

Coleta de métricas para medida de qualidade

Um ponto de melhoria importante é a coleta de métricas, para que seja possível quantificar o aumento de qualidade do produto de software, e não só com o uso de testes automatizados. Algumas opções são a integração com o sistema de *bug tracking* ou a partir das chamadas no *helpdesk*.

Interdependência dos casos de teste

Outro ponto de melhoria é a criação de casos de teste independentes entre si. Esta abordagem, apesar de mais trabalhosa, permite a execução de casos de teste isolados, sem pré-requisitos atendidos apenas por outros casos de teste. Isto também permite a execução de conjuntos de casos de teste que não os definidos originalmente no projeto, de modo a atender necessidades específicas dos analistas de teste.

Capítulo 6

Conclusão

Embora a automação de teste de software seja vista como uma forma de melhorar a produtividade e a qualidade de software, ela não é indicada para o teste de qualquer tipo de funcionalidade. As funcionalidades mais propícias para o uso de tal abordagem são aquelas que envolvem a execução de tarefas repetitivas e cansativas, facilmente suscetíveis a erros, ou impossíveis de serem realizadas manualmente. Além disso, existem várias técnicas que podem ser utilizadas na automação do teste de uma funcionalidade, as quais possuem vantagens e desvantagens dependendo da natureza da funcionalidade em questão.

Portanto, o ganho com a automação depende fortemente de sua implantação sistemática. Conseqüentemente, esta atividade apresenta as mesmas características comuns a outros projetos de software, sendo necessários planejamento, análise, projeto, implementação e até mesmo teste. Segundo Kaner [Kan97a], um trabalho mínimo de criação, manutenção e documentação de testes automatizados é, em média, de 3 a 10 vezes mais longo que o mesmo trabalho manual.

Este trabalho e sua contribuição consistiram na definição (Capítulo 3) e implementação (Capítulo 4) do AutoTest, um sistema para automação de teste funcional que visa facilitar a aplicação do teste de software automatizado de modo a se obter maiores ganhos em relação a uma abordagem de teste puramente manual.

Os ganhos da abordagem de teste automatizado em relação à abordagem manual foram mostrados pela automação de testes de três módulos de um sistema de grande porte bastante complexo (Capítulo 5). De uma forma geral, os resultados obtidos com este estudo de caso se apresentaram muito satisfatórios, superando em alguns pontos as expectativas. O sistema se mostrou robusto e expansível o suficiente para permitir sua reutilização na automação de teste de várias funcionalidades do sistema em questão. Além disso, foi possível obter uma grande economia no esforço e tempo necessários para a realização do teste, ganhando-se eficiência e confiança nos resultados – principalmente na aplicação de teste de regressão.

Mas a comparação das abordagens de teste manual e automatizada se mostrou um tanto limitada. Como, depois que o sistema para automação de teste é estabelecido para um

determinado projeto, a execução manual normalmente não é mais realizada, não há a coleta de métricas para a execução manual. Assim, a comparação das demais re-execuções para as duas abordagens de teste pode ser apenas estimada. Não há muitas garantias, apesar das fortes indicações, de que esta estimativa leve ao real ganho na qualidade do produto, por ser uma medida mais subjetiva que a produtividade. Uma forma de melhor avaliá-la seria a comparação da satisfação do cliente antes e depois da implantação da automação de teste, realizada apenas em um caso devido à necessidade de existência de histórico formal de reclamações do cliente.

Com base neste trabalho foram publicados dois artigos em simpósios nacionais ([Cun02a], [Fan04]), uma comunicação breve em congresso internacional ([Cun02b]) e uma submissão de projeto para o Programa Brasileiro da Qualidade e Produtividade em Software – PBQP Software ([Dia04]).

Como trabalho futuro pode-se estender o sistema AutoTest para contemplar a automação de teste funcional dos sistemas legados, uma vez que estes ainda têm uma representatividade considerável nas empresas de desenvolvimento de software hoje em dia. Faz-se necessária a realização de estudos para se identificar quais as características que os sistemas legados possuem e que causam impacto na estratégia de automação de teste suportada pelo sistema AutoTest. Além disso, pode-se também avaliar a viabilidade de extensão do sistema para suportar os testes estruturais, ou seja, testes de caixa-branca, mas voltados para a tecnologia Java.

Referências

[All00] Allen, Michael, *"jCIFS – Implementing CIFS in Java"*.
http://jcifs.samba.org (acessado em 20 de dezembro de 2004)

[Apa02] Apache Software Foundation, *"Jakarta POI – Java API To Access Microsoft Format Files"*.
http://jakarta.apache.org/poi (acessado em 20 de dezembro de 2004)

[Apa03] Apache Software Foundation, *"The Apache Ant Project"*.
http://ant.apache.org (acessado em 20 de dezembro de 2004)

[Bac97] Bach, James, *"Test Automation Snake Oil"*. Windows Technical Journal, págs. 40 a 44, outubro de 1997.

[Bar01] Barret, Daniel e Silverman, Richard, *"SSH, The Secure Shell: The Definitive Guide"*. O'Reilly & Associates, 2001.

[Bei90] Beizer, Boris, *"Software testing techniques"*, segunda edição. New York, 1990.

[Bei95] Beizer, Boris, "*Black-Box Testing: techniques for funcional testing of software and systems*", John Wiley & Sons, New York, 1995.

[Ber89] Berliner, Brian, Grune, Dick e Polk, Jeff, *"CVS – Concurrent Versions System"*.
http://www.cvshome.org (acessado em 20 de dezembro de 2004)

[Boy98] Boyce, Jim, *"Inside Windows 98"*. New Riders Publishing, 1998.

[Bra89] Braden, Robert, *"RFC 1123: Requirements for Internet Hosts – Application and Support"*, Internet Engineering Task Force, 1989.

[Bri01] British Computer Society Specialist Interest Group in Software Testing, *"Standard for software component testing"*. 2001.

[Com04a] Compuware Corporation, *"QARun"*. http://www.compuware.com/products/qacenter/qarun.htm (acessado em 20 de dezembro de 2004)

[Com04b] Compuware Corporation, *"TestPartner"*. http://www.compuware.com/products/qacenter/375_ENG_HTML.htm (acessado em 20 de dezembro de 2004)

[Cun02a] Cunha, Adriano et al. *"Um Framework para a Automação de Teste Funcional de Aplicações Batch"*. VII Simpósio de Informática, Uruguaiana, Brasil, novembro de 2002.

[Cun02b] Cunha, Adriano et al. *"Um Framework para Automação de Testes Funcionais de Interface Gráfica em Plataforma Windows"*. VIII Congreso Argentino de Ciencias de la Computación, Buenos Aires, Argentina, outubro de 2002.

[Dam04] DameWare Development, *"Mini Remote Control"*. http://www.dameware.com (acessado em 20 de dezembro de 2004)

[Des04] DeskShare, *"My Screen Recorder v.2.21"*. http://www.deskshare.com/msr.aspx (acessado em 20 de dezembro de 2004)

[Dia04] Dias, Sindo et al. *"Automação de Teste de Software como Suporte para o Incremento de Qualidade e Produtividade em Sistemas de Faturamento para Telecomunicações"*. Programa Brasileiro da Qualidade e Produtividade em Software – PBQP Software 2003, projeto nº 4.05, categoria "Serviços Tecnológicos".

[Dus99] Dustin, Elfriede, *"Lessons in Test Automation"*. Better Software Magazine, págs. 16 a 22, setembro/outubro de 1999.

[Ecl03] Eclipse Foundation, *"Java Development Tools (JDT) Subproject"*. http://www.eclipse.org/jdt/index.html (acessado em 20 de dezembro de 2004)

[Ecl04] Eclipse Foundation, *"Eclipse IDE"*. http://www.eclipse.org/ (acessado em 20 de dezembro de 2004)

[Emp04] Empirix, *"e-Tester"*. http://www.empirix.com/default.asp?action=article&ID=419 (acessado em 20 de dezembro de 2004)

[Fan04] Fantinato, Marcelo et al. *"AutoTest - Um Framework Reutilizável para Automação de Teste Funcional de Software"*. III Simpósio Brasileiro de Qualidade de Software, Brasília, Brasil, junho de 2004.

[Few99] Fewster, Mark e Graham, Dorothy, *"Software Test Automation"*. Addison-Wesley, 1999.

[Few01] Fewster, Mark, *"Common Mistakes in Test Automation"*. Software Test Automation Conference & EXPO, San Jose, Califórnia, 2001.

[Hay01] Hayes, Linda, *"Does Test Automation Saves Time and Money?"*. StickyMinds.com, agosto de 2001. http://www.stickyminds.com/sitewide.asp?ObjectId=2735&Function=DETAILBROWSE&ObjectType=COL (acessado em 20 de dezembro de 2004)

[Hay01a] Haynes, Dawn, *"Automated Testing: A Silver Bullet?"*. The Rational Edge e-zine, setembro de 2001.

[Hei03] Heiman, Richard, *"Worldwide Distributed Automated Software Quality Tools Forecast and Analysis, 2003-2007"*. IDC #29739, volume 1, 2003.

[Hen98] Hendrickson, Elisabeth, *"The Differences Between Test Automation Success And Failure"*. STAR West, San Jose, Califórnia, 1998.

[Ibm04a] IBM Rational, *"Rational Functional Tester for Java and Web"*. http://www.ibm.com/software/awdtools/tester/functional/ (acessado em 20 de dezembro de 2004)

[Ibm04b] IBM Rational, *"Rational Robot"*. http://www-306.ibm.com/software/awdtools/tester/robot/ (acessado em 20 de dezembro de 2004)

[Ibm04c] IBM Rational, *"IBM Rational ClearCase"*. http://www.ibm.com/software/awdtools/clearcase/ (acessado em 20 de dezembro de 2004)

[Ibm04d] IBM Rational, *"IBM Rational Test Manager"*. http://www-306.ibm.com/software/awdtools/test/manager/ (acessado em 20 de dezembro de 2004)

[Iee90] IEEE-AS Standards Board, *"IEEE Std 610.12-1990 – IEE standard glossary of software engineering terminology"*. Software Engineering Technical Committee of the IEEE Computer Society, 1990.

[Kan97a] Kaner, Cem, *"Improving the Maintainability of Automated Test Suites"*. Software QA, volume 4, nº 4, 1997.

[Kan98] Kaner, Cem, *"Avoiding Shelfware: A Manager's View of Automated GUI Testing"*. STAR East, Orlando, Flórida, 1998.

[Kan00] Kaner, Cem, *"Architectures of Test Automation"*. Los Altos Workshops on Software Testing, San Jose, Califórnia, 2000.

[Kha03] Khan, Andy, *"Java Excel API"*. http://www.jexcelapi.org (acessado em 20 de dezembro de 2004)

[Kit99] Kit, Edward, *"Integrated, Effective Test Design and Automation"*. Software Development Magazine, pág. 27, fevereiro de 1999.

[Mar97] Marick, Brian, *"Classic Testing Mistakes"*. STAR East, Orlando, Flórida, 1997.

[Mer04a] Mercury Interactive, *"QuickTest Professional"*. http://www.mercury.com/us/products/quality-center/functional-testing/quicktest-professional/ (acessado em 20 de dezembro de 2004)

[Mer04b] Mercury Interactive, *"WinRunner"*. http://www.mercury.com/us/products/quality-center/functional-testing/winrunner/ (acessado em 20 de dezembro de 2004)

[Mic97] Microsoft Corporation, *"Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference: Software Development Kit and Programmer's Reference"*, Microsoft Press, 1997.

[Mic00] Microsoft Corporation, *"Windows 2000 Terminal Services"*. http://www.microsoft.com/windows2000/technologies/terminal/default.asp (acessado em 20 de dezembro de 2004)

[Mic03a] Microsoft Corporation, *"Microsoft Office Word 2003"*. http://office.microsoft.com/word (acessado em 20 de dezembro de 2004)

[Mic03b] Microsoft Corporation, *"Microsoft Office Excel 2003"*. http://office.microsoft.com/excel (acessado em 20 de dezembro de 2004)

[Mye79] Myers, Glenford, *"The Art of Software Testing"*. John Wiley & Sons, 1979.

[Nag00] Nagle, Carl, *"Test Automation Frameworks"*. http://safsdev.sourceforge.net/FRAMESDataDrivenTestAutomationFrameworks.htm (acessado em 20 de dezembro de 2004)

[Net04] Netcraft, *"Internet Data Mining Services"*. http://www.netcraft.com (acessado em 20 de dezembro de 2004)

[Ngu01] Nguyen, Hung, *"Testing Applications on the Web"*, John Wiley & Sons, 2001.

[Ope03] OpenOffice.org, *"OpenOffice"*. http://www.openoffice.org (acessado em 20 de dezembro de 2004)

[Pet00] Pettichord, Bret, *"Capture Replay - A Foolish Test Strategy"*. STAR West, San Jose, Califórnia, 2000.

[Pet01] Pettichord, Bret, *"Are you ready to automate?"*. STAR West, San Jose, Califórnia, 2001.

[Pos85] Postel, Jon e Reynolds, Joyce, *"RFC 959: File Transfer Protocol (FTP)"*, Network Working Group, 1985.

[Pre92] Pressman, Roger, *"Engenharia de Software"*. Makron Books do Brasil Editora Ltda, 1992.

[Qua04] Quality Forge, *"TestSmith"*. http://qualityforge.com/testsmith/index.html (acessado em 20 de dezembro de 2004)

[Roc01] Rocha, Ana, Maldonado, José e Weber, Kival, *"Qualidade de software – Teoria e prática"*. Prentice Hall, 2001.

[Sea04] Seapine Software, *"QA Wizard"*. http://www.seapine.com/qawizard.html (acessado em 20 de dezembro de 2004)

[Seg04a] Segue Software, *"SilkTest"*. http://www.segue.com/products/functional-regressional-testing/silktest.asp (acessado em 20 de dezembro de 2004)

[Seg04b] Segue Software, *"SilkTest International"*. http://www.segue.com/products/functional-regressional-testing/silktest-international.asp (acessado em 20 de dezembro de 2004)

[Som00] Sommerville, Ian, *"Software Engineering"*, sexta edição. Addison-Wesley, 2000.

[Sun03] Sun Microsystems, *"JDBC Technology"*. http://java.sun.com/products/jdbc/ (acessado em 20 de dezembro de 2004)

[Ts03] Ts, Jay, Eckstein, Robert e Collier-Brown, David, *"Using Samba"*, O'Reilly & Associates, 2003.

[Tec04] TechSmith, *"Camtasia Studio"*. http://www.techsmith.com/products/studio/default.asp (acessado em 20 de dezembro de 2004)

[Ter01] Tervo, Brian, *"Standards For Test Automation"*. STAR East, Orlando, Flórida, 2001.

[Uni03] Unisys, *"Trends in Data Center Operating Systems Standardization"*. http://www.unisys.com/eprise/main/admin/corporate/doc/Trends_in_Data_Center_OS_Standardization_White_Paper.pdf (acessado em 20 de dezembro de 2004)

[VNC02] Real VNC, *"Virtual Network Computing"*.
http://www.realvnc.com/ (acessado em 20 de dezembro de 2004)

[Zam98a] Zambelich, Keith, *"Totally Data-driven Automated Testing"*. Automated Testing Specialists Whitepaper, 1998.
http://www.sqa-test.com/w_paper1.html (acessado em 20 de dezembro de 2004)

[Zam98b] Zambelich, Keith, *"Methodologies for Automated Testing"*.
http://www.sqa-test.com/method.html (acessado em 20 de dezembro de 2004)