UNIVERSIDADE ESTADUAL PAULISTA
Faculdade de Ciências - Bauru
Bacharelado em Ciência da Computação

MARCELO RIBEIRO DOS SANTOS SOARES

AUTOMATIZAÇÃO DE TESTES FUNCIONAIS UTILIZANDO A FERRAMENTA *SIKULI*

UNESP

2012

MARCELO RIBEIRO DOS SANTOS SOARES

AUTOMATIZAÇÃO DE TESTES FUNCIONAIS UTILIZANDO A FERRAMENTA SIKULI

Orientador: Prof. Dr. Wilson Massashiro Yonezawa

Monografia apresentada a Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, campus Bauru, como trabalho de conclusão do Curso de Bacharelado em Ciência da Computação.

BAURU

2012

MARCELO RIBEIRO DOS SANTOS SOARES

AUTOMATIZAÇÃO DE TESTES FUNCIONAIS UTILIZANDO A FERRAMENTA *SIKULI*

Monografia apresentada a Universidade Estadual Paulista “Júlio de Mesquita Filho”, Faculdade de Ciências, campus Bauru, como trabalho de conclusão do Curso de Bacharelado em Ciência da Computação**.**

BANCA EXAMINADORA

Prof. Dr. Wilson Massashiro Yonezawa
Professor Doutor
DCo – FC - UNESP – Bauru
Orientador

Profa. Dra. Simone das Graças Domingues Prado
Professora Doutora
DCo – FC - UNESP – Bauru

Prof. Dr. Renê Pegoraro
Professor Doutor
DCo – FC - UNESP – Bauru

Bauru, 29 de Outubro de 2012.

DEDICATÓRIA

Dedico este trabalho aos meus pais Rubens e Rosa pelos princípios que me ensinaram, pelo apoio, paciência e dedicação com que me conduziram até aqui.

AGRADECIMENTOS

A Universidade Estadual Paulista “Júlio de Mesquita Filho” pela realização deste curso.

Ao Prof. Dr. Wilson Massashiro Yonezawa por toda a ajuda e orientação durante a realização do trabalho.

Aos amigos Affonso, Ivan, Ivo, Paulo, Pedro, Thiago e Wagner pelos anos que passamos juntos, unidos, incentivando e apoiando uns aos outros, na maioria das vezes.

Ao amigo e companheiro de trabalho Carlos Amigo que me auxiliou no projeto e durante todo o aprendizado e trabalho na área de qualidade de *software*.

A minha namorada Karla pelo carinho e paciência nos últimos anos do curso.

RESUMO

Esse trabalho busca demonstrar as vantagens na automação de testes funcionais de *software* utilizando a ferramenta *Sikuli*, que usa reconhecimento de imagem para encontrar os elementos da interface gráfica de um sistema, além de utilizar uma biblioteca personalizada com métodos confeccionados para automatizar a sumarização dos resultados obtidos através dos testes e suas evidências.

Palavras-chave: Teste de *software*, testes automatizados, *Sikuli*

ABSTRACT

This work seeks to demonstrate the advantages in functional *software* test automation using *Sikuli* tool, which uses image recognition to find the graphical elements of a system, in addition to using a custom library with methods made to automate the summarization of obtained results through the tests and their evidence.

Key-words: *Software* testing, automated tests, *Sikuli*.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

PSR – Problem Step Recorder

HTML - HyperText Markup Language

XML - Extensible Markup Language

XSL - Xtensible Stylesheet Language

TC - Test Case

QA – Quality Assurance

IDE – Integrated Development Environment

API – Application Programming Interface

JVM – Java Virtual Machine

GUI – Graphical User Interface

JNI – Java Native Interface

SUMÁRIO

1. INTRODUÇÃO

Sistemas de *software* tornam-se cada vez mais parte do nosso dia-a-dia, desde aplicações comerciais até produtos de consumo. A maioria das pessoas já experimentou um *software* que não funcionou como esperado. *Software*s que não funcionam corretamente podem gerar problemas como: perdas financeiras, perda de tempo e redução da reputação da empresa, etc. Nos piores casos, erros graves em *software* podem ocasionar perdas de vidas humanas. Desta forma, testar corretamente um *software* é uma atividade de crítica no processo de desenvolvimento de *software*.

Entretanto, o processo de teste de *software* também tem seus problemas. Um dos maiores problemas é o retrabalho. O retrabalho ocorre tanto na confecção da documentação dos testes quanto na execução dos testes em si. O retrabalho aumenta os custos e o tempo de desenvolvimento do *software*. Para cada versão do *software* é necessário retestar as funcionalidades que já foram testadas na versão anterior por meio da execução de casos de testes. A reexecução dos testes é necessária visto que novos erros no *software* podem ser introduzidos na nova versão.

Para facilitar o processo de teste e reteste uma estratégia seria a automatização dos testes. A automatização faz com que o processo de realizar alterações durante o desenvolvimento seja facilitado, uma vez que o tempo perdido em reteste da aplicação após correções efetuadas é reduzido e também possibilita prever o impacto de tais alterações que são logo detectadas no momento da sua implementação.

A automatização de teste requer uso de ferramentas. No mercado estão disponíveis diversas delas para este propósito, porém, cada uma possui as suas características específicas e é aplicada em um domínio ou tipo específico de aplicação. Algumas ferramentas de teste dependem da linguagem de programação e/ou da interface utilizada pelos desenvolvedores.

As ferramentas de teste podem ser categorizadas por tipo de teste, como por exemplo, ferramentas para teste de caixa branca e ferramentas para teste de caixa preta. As ferramentas de teste de caixa preta são geralmente utilizadas para teste de funcionalidade. Não é difícil encontrar ferramentas que façam a automatização dos testes funcionais utilizando como base na interface com o usuário da aplicação. Estas

ferramentas simulam as ações do usuário durante o uso da aplicação, registrando os resultados e comparando-as com os resultados esperados. Entretanto, para utilizar tais ferramentas é necessário conhecimento específico a fim de automatizar as tarefas que forem necessárias. Mudar de ferramenta requer que o usuário aprenda novamente a utilizar as funcionalidades da nova ferramenta. Além de aprender a manusear o novo *software*, precisa aprender também uma linguagem de programação, utilizada na confecção dos roteiros (*scripts*) de testes.

Outro problema, e um dos maiores, é o alto custo das ferramentas. Alguns produtos chegam à casa dos R$ 8200,00 por licença de uso.

Mesmo com o alto custo, algumas dessas ferramentas não contemplam alguns tipos de testes, como por exemplo, verificar o resultado de uma operação em outro computador, como mandar uma mensagem e verificar se outro usuário, em outra máquina, a recebeu, o que poderia ser feito utilizando uma ferramenta de acesso remoto por exemplo.

Como dito anteriormente, algumas ferramentas fornecem suporte específico para um ou outro tipo de interface, o que pode impossibilitar o teste de integração entre *software*s que utilizam diferentes linguagens de programação que apenas diferem por suas interfaces, por exemplo, *software*s feitos em C#® com interface em C#®, podem integrar-se com *software*s feitos em C#® com interface em Flash®.

Como resolver esse problema? Dentre as várias possibilidades, este trabalho pretende explorar o uso de uma ferramenta de automatização que utiliza reconhecimento de imagem. Desta forma, seria possível automatizar os testes, independente da interface utilizada pelo *software*.

1.1 OBJETIVOS

Utilizar a ferramenta *Sikuli* para automatizar os testes funcionais através do reconhecimento de elementos gráficos na interface do usuário e desenvolver uma biblioteca para auxiliar na construção da documentação e das evidências dos testes, podendo assim comparar os resultados obtidos através dos testes realizados de forma automática com a execução manual deles, verificando as vantagens e desvantagens de cada um dos métodos.

2. REFERENCIAL TEÓRICO

2.1 TESTE DE *SOFTWARE* E SEUS CONCEITOS

Teste de *Software* é uma das técnicas de Garantia da Qualidade de *Software* e consiste na análise dinâmica do produto, ou seja, na sua execução com o objetivo de provocar a falha nesse produto, contribuindo para uma futura detecção de defeitos através de um processo de depuração e, conseqüentemente, o aumento da confiança de que ele esteja correto (adaptado de ROCHA, 2000).

2.1.1 ATIVIDADE DE TESTE

As atividades de teste possuem um papel fundamental no processo de desenvolvimento de um *software*, pois correspondem a um último momento de corrigir eventuais problemas no produto antes da sua entrega ao usuário final (PRESSMAN, 2005).

A atividade de testes é uma etapa crítica para o desenvolvimento de *software*. Frequentemente, a atividade de testes insere tantos erros em um produto quanto a própria implementação. Por outro lado, o custo para correção de um erro na fase de manutenção é de 60 a 100 vezes maior que o custo para corrigi-lo durante o desenvolvimento (TOMELIN, 2001).

Um objetivo central de toda a metodologia dos testes é maximizar a cobertura destes. Deseja-se conseguir detectar a maior quantidade possível de defeitos que não foram apanhados pelas revisões, dentro de dados limites de custo e prazos. “Os testes não provam que um programa está correto, somente contribuem para aumentar a confiança de que o *software* desempenha as funções especificadas” (SOUZA, 2001)

Existem duas grandes razões para se testar *software*, realizar um julgamento sobre qualidade e descobrir problemas. Na verdade, testa-se porque se sabe que as

atividades humanas são passíveis de falha e, isso é especialmente verdade no domínio de desenvolvimento de *software*.

2.1.2 OBJETIVOS, LIMITES E DESAFIOS DO TESTE DE *SOFTWARE*

O objetivo principal de um teste de *software* é expor um erro ou defeito que não havia sido encontrado. Um erro de *software* também é conhecido como *bug*.

O termo *bug* surgiu em 1878 quando Thomas Edison teve problemas de leitura em seu fonógrafo, mas foi usado em computadores pela primeira vez em 1947 quando engenheiros trabalhavam com a máquina Harvard Mark I e encontraram um inseto em seus circuitos que estava causando um erro nos cálculos da máquina. O inseto foi retirado e colado em um livro, ficando registrado como o primeiro *bug* encontrado.

Para Souza (2000), é impossível testar um programa completamente. Testar um programa completamente significa que não existe absolutamente mais nenhum *bug* a ser encontrado. Para garantir que um programa está completamente testado é necessário testar todas as combinações de entradas válidas (em um programa que soma dois dígitos são aproximadamente 39600 entradas válidas), todas as combinações de entradas inválidas (testar todas as possíveis combinações do teclado que não sejam pares válidos de 1 ou 2 dígitos).

Devem-se testar também todas as possibilidades de edição (se o programa permite editar os valores digitados, é preciso testar todos os recursos de edição e verificar se o resultado de uma edição ainda é uma entrada válida). Todas as possibilidades em relação ao tempo (por exemplo: ao digitar dois números rápido demais, intercalados por <*enter*> e o programa não consegue capturar um dos valores) (OLIVEIRA, 1997).

Finalmente, para este autor, o mais complexo é testar todas as entradas possíveis em qualquer ponto que o programa permita entradas de dados, em todos os estados possíveis que ele pode atingir nesses pontos. Esta tarefa é praticamente

impossível. Como exemplos de situações que normalmente não são testadas, pode-se citar:

a) um certo gerenciador de banco de dados falha quando certas tabelas tem exatamente 512 bytes de tamanho;

b) um certo editor de textos falha se o texto for muito longo (mais de 100.000 caracteres), desde que o texto esteja muito fragmentado no disco (espalhado em setores não contíguos).

O objetivo de um testador de *software*, neste caso um testador profissional e não uma ferramenta de teste, não pode ser ele querer mostrar que o *software* está livre de *bug*s, porque ele jamais conseguirá provar isso. De alguma forma ou maneira ele provavelmente deixará escapar muitos *bug*s, segundo Oliveira (1997). O principal objetivo de um testador de *software* deve ser: encontrar problemas no sistema. E, quando encontrado, é encaminhá-lo para o conserto. Sendo assim: um teste falha quando não acha nenhum *bug*. Os *bug*s estão lá, cabe ao testador encontrá-los.

Pode-se concluir então que alguns dos maiores desafios para os profissionais na área de testes são (VICENZI, 2008):

- Falta de tempo para o teste exaustivo.
- Muitas combinações de entrada podem ser exercitadas.
- Dificuldade em determinar os resultados esperados para cada caso de teste (Falta de documentação).
- Requisitos do *software* inexistentes ou que mudam rapidamente.
- Falta de tempo suficiente para o teste.
- Gerentes que desconhecem teste ou que não se preocupam com qualidade.

2.2 TÉCNICAS DE TESTE

2.2.1 TÉCNICA FUNCIONAL

O teste funcional também é conhecido como teste caixa preta, segundo Myers (1979), pelo fato de tratar o *software* como uma caixa cujo conteúdo é

desconhecido e da qual só é possível visualizar o lado externo, ou seja, os dados de entrada fornecidos e as respostas produzidas como saída.

Na técnica de teste funcional são verificadas as funções do sistema sem se preocupar com os detalhes de implementação.

O teste funcional envolve dois passos principais: identificar as funções que o *software* deve realizar e criar casos de teste capazes de checar se essas funções estão sendo realizadas pelo *software* [7]. As funções que o *software* deve possuir são identificadas a partir de sua especificação. Assim, uma especificação bem elaborada e de acordo com os requisitos do usuário é essencial para esse tipo de teste.

Alguns exemplos de critérios de teste funcional são [7]:

- **Particionamento em Classes de Equivalência:** a partir das condições de entrada de dados identificadas na especificação, divide-se o domínio de entrada de um programa em classes de equivalência válidas e inválidas. Em seguida seleciona-se o menor número possível de casos de teste, baseando-se na hipótese que um elemento de uma dada classe seria representativo da classe toda, sendo que para cada uma das classes inválidas deve ser gerado um caso de teste distinto. O uso de particionamento permite examinar os requisitos mais sistematicamente e restringir o número de casos de teste existentes. Alguns autores também consideram o domínio de saída do programa para estabelecer as classes de equivalência.
- **Análise do Valor Limite:** é um complemento ao critério Particionamento em Classes de Equivalência, sendo que os limites associados às condições de entrada são exercitados de forma mais rigorosa; ao invés de selecionar-se qualquer elemento de uma classe, os casos de teste são escolhidos nas fronteiras das classes, pois nesses pontos se concentra um grande número de erros. O espaço de saída do programa também é particionado e são exigidos casos de teste que produzam resultados nos limites dessas classes de saída.
- **Grafo de Causa-Efeito:** os critérios anteriores não exploram combinações das condições de entrada. Este critério estabelece requisitos de teste baseados nas possíveis. Primeiramente, são levantadas as possíveis condições de entrada (causas) e as possíveis ações (efeitos) do programa.

Um dos problemas relacionado aos critérios funcionais é que muitas vezes a especificação do programa é feita de modo descritivo e não formal. Dessa maneira, os requisitos de teste derivados de tais especificações são também, de certa forma, imprecisos e informais. Como consequência, tem-se dificuldade em automatizar a aplicação de tais critérios, que ficam, em geral, restritos à aplicação manual. Por outro lado, para a aplicação desses critérios é essencial apenas que se identifiquem as entradas, a função a ser computada e a saída do programa, o que os tornam aplicáveis praticamente em todas as fases de teste (unidade, integração e sistema) (DELAMARO, 1997).

2.2.2 TÉCNICA ESTRUTURAL

Na técnica de teste estrutural, também conhecida como teste caixa branca (em oposição ao nome caixa preta), os aspectos de implementação são fundamentais na escolha dos casos de teste. Teste de caixa-branca, algumas vezes chamado de teste da caixa de vidro, é uma filosofia de projeto de casos de teste que usa a estrutura de controle descrita como parte do projeto ao nível de componentes para derivar casos de teste. Usando métodos de teste caixa-branca, o engenheiro de *software* pode derivar casos de teste que (1) garantam que todos os caminhos independentes de um módulo tenham sido executados pelo menos uma vez, (2) exercitem todas as decisões lógicas em seus lados verdadeiro e falso, (3) executem todos os ciclos nos seus limites e dentro de seus intervalos operacionais, e (4) exercitem as estruturas de dados internas para garantir sua validade (PRESSMAN, 2005).

Algumas técnicas de teste de caixa-branca são: teste de caminho básico, teste estrutural de controle, teste de condição, teste de fluxo de dados e teste de ciclo.

2.3 FASES DO TESTE

A seguir estão descritos brevemente os tipos de baterias de testes, ou conjuntos de testes de *software* de determinada categoria, conforme proposto pela IEEE segundo Poston (1996).

2.3.1 TESTE DE UNIDADE

Testes de unidade geralmente são de caixa branca, têm por objetivo verificar um elemento que possa ser localmente tratado como uma unidade de implementação; por exemplo, uma sub-rotina ou um módulo. Em produtos implementados com tecnologia orientada a objetos, uma unidade é tipicamente uma classe. Em alguns casos, pode ser conveniente tratar como unidade um grupo de classes correlatas.

2.3.2 TESTE DE INTEGRAÇÃO

Testes de Integração têm por objetivo verificar as interfaces entre as partes de uma arquitetura de produto. Esses testes têm por objetivo verificar se as unidades que compõem cada liberação funcionam corretamente, em conjunto com as unidades já integradas, implementando corretamente os casos de uso cobertos por essa liberação executável.

2.3.3 TESTE DE ACEITAÇÃO

Testes de Aceitação têm por objetivo validar o produto, ou seja, verificar se este atende aos requisitos especificados. Eles são executados em ambiente o mais semelhante possível ao ambiente real de execução. Os testes de aceitação podem ser divididos em testes funcionais e não funcionais.

2.3.4 TESTE DE REGRESSÃO

Testes de Regressão, que executam novamente um subconjunto de testes previamente executados. Seu objetivo é assegurar que alterações em partes do produto não afetem as partes já testadas. As alterações realizadas, especialmente durante a manutenção, podem introduzir erros nas partes previamente testadas. A maior utilidade

dos testes de regressão aparece durante o processamento de solicitações de manutenção. Entretanto, testes de regressão podem ser executados em qualquer passo do desenvolvimento. Por exemplo, para cada liberação, deve ser feita uma regressão com os testes de integração das liberações anteriores.

2.4 AUTOMATIZAÇÃO DE TESTES

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, sendo feita geralmente por meio do uso de ferramentas de automação de teste. Podem ser consideradas para a automação as atividades de geração e de execução de casos de teste (BINDER, 1999; KANER, 1997).

Quando executada 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. 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 erros, ou de aplicativos já existentes que são evoluídos para uma nova versão e suas funcionalidades são alteradas (KANER, 1997).

2.4.1 TÉCNICAS DE AUTOMAÇÃO DE TESTE

As principais técnicas de automação de teste apresentadas na literatura são: *record & playback*, programação de *scripts*, *data-driven* e *keyword-driven*. Esta seção apresenta uma breve descrição de cada uma destas técnicas.

A técnica *record & playback* consiste em, utilizando uma ferramenta de automação de teste, gravar as ações executadas por um usuário sobre a interface gráfica de uma aplicação e converter estas ações em *scripts* de teste que podem ser

executados quantas vezes for desejado. Cada vez que o *script* é executado, as ações gravadas são repetidas, exatamente como na execução original. Para cada caso de teste é gravado um *script* de teste completo que inclui os dados de teste (dados de entrada e resultados esperados), o procedimento de teste (passo a passo que representa a lógica de execução) e as ações de teste sobre a aplicação. A vantagem da técnica *record & playback* é que ela é bastante simples e prática, sendo uma boa abordagem para testes executados poucas vezes. Entretanto, são várias as desvantagens desta técnica 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. (FEWSTER, 1999, 2001; HENDRICKSON, 1998; MARICK, 1997; PETTICHORD, 2000; FANTINATO et al., 2000)

A técnica de programação de *scripts* é uma extensão da técnica *record & playback*. Através da programação os *scripts* de teste gravados são alterados para que desempenhem um comportamento diferente do *script* 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 testador 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, sua aplicação pura também produz uma grande quantidade de *scripts* de teste, visto que para cada caso de teste deve ser programado um *script* de teste, o qual

também inclui os dados de teste e o procedimento de teste. As técnicas *data-driven* e *keyword-driven*, que são versões mais avançadas da técnica de programação de *scripts*, permitem a diminuição da quantidade de *scripts* de teste, melhorando a definição e a manutenção de casos de teste automatizados (FEWSTER; GRAHAM, 1999; HENDRICKSON, 1998; IBM; TERVO, 2001).

A técnica *data-driven* (técnica orientada a dados) consiste em extrair, dos *scripts* de teste, os dados de teste, que são específicos por caso de teste, e armazená-los em arquivos separados dos *scripts* de teste. Os *scripts* de teste passam a conter apenas os procedimentos de teste (lógica de execução) e as ações de teste sobre a aplicação, que normalmente são genéricos para um conjunto de casos de teste. Assim, os *scripts* de teste não mantêm os dados de teste no próprio código, obtendo-os diretamente de um arquivo separado, somente quando necessário e de acordo com o procedimento de teste 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 manutenção dos *scripts* de teste. Esta técnica de automação permite que o projetista de teste e o implementador de teste trabalhem em diferentes níveis de abstração, dado que o projetista de teste precisa apenas elaborar os arquivos com os dados de teste, sem se preocupar com questões técnicas da automação de teste (FEWSTER, 1999; HENDRICKSON, 1998; IBM; NAGLE, 2000; ZAMBELICH, 1998).

A técnica *keyword-driven* (técnica orientada a palavras-chave) consiste em extrair, dos *scripts* de teste, o procedimento de teste que representa a lógica de execução. Os *scripts* de teste passam a conter apenas as ações específicas de teste sobre a aplicação, as quais são identificadas por palavras-chave. Estas ações de teste são como funções de um programa, podendo inclusive 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 respectivos parâmetros.

Assim, pela técnica *keyword-driven*, os *scripts* de teste não mantêm os procedimentos de teste no próprio código, obtendo-os diretamente dos arquivos de procedimento de teste. A principal vantagem da técnica *keyword-driven* é que se pode

facilmente adicionar, modificar ou remover passos de execução no procedimento de teste com necessidade mínima de manutenção dos *scripts* de teste, permitindo também que o projetista de teste e o implementador de teste trabalhem em diferentes níveis de abstração (FEWSTER; GRAHAM, 1999; IBM Rational; NAGLE, 2000; ZAMBELICH, 1998; KANER, 2000).

3. PROJETO

Inicialmente foi realizada uma pesquisa sobre ferramentas de teste, analisando se alguma delas era adequada para o propósito deste trabalho, de realizar os testes funcionais independente da linguagem em que o programa é escrito. Durante esta fase foram encontradas diversas ferramentas, pagas e gratuitas, porém a grande maioria era utilizada para programas escritos em linguagens específicas, uns para Java, outros para C#, linguagens *web* outras mais. Foi identificada e selecionada a ferramenta *Sikuli* como base para o desenvolvimento do projeto de TCC, visto que atende aos requisitos desse trabalho, pois é gratuita e pode ser usada para testar programas escritos em qualquer linguagem de programação, já que depende apenas de elementos contidos na interface gráfica do sistema, identificando-os por meio de *screenshots*.

A ferramenta baseia-se em reconhecimento de imagem para realizar ações na tela do computador do usuário, como clicar, mover o mouse e digitar, entre outras ações, podendo assim ser utilizada para testar *software* independente da interface utilizada, o propósito do *Sikuli* é poder testar qualquer aplicação que apresente uma interface com o usuário.

O *Sikuli* é baseado na linguagem *Jython* (Java + *Python*). A linguagem *Python* será utilizada para a criação da biblioteca, uma vez que apresenta uma curva de aprendizagem curta e pode ser integrada ao ambiente.

Logo, foi iniciado um estudo exploratório sobre a linguagem *Python*, para verificar a possibilidade de desenvolver a biblioteca da maneira desejada e se todos os objetivos podiam ser cumpridos utilizando-a. Foi realizado um estudo completo sobre a linguagem, suas funções e sintaxe.

A partir dai iniciou-se o desenvolvimento das funcionalidades, ou seja, da biblioteca para ser aplicada dentro do ambiente *Sikuli* IDE.

O *Python* foi utilizado para a escrita dos casos de teste que são executados no ambiente *Sikuli* IDE. Os casos de testes exercitam as funcionalidades da aplicação testada, podendo focar os testes nas alterações realizadas no *software*, sem que ocorra

perda significativa de tempo com reteste e documentação durante a realização do processo de testes.

Utilizando as ferramentas descritas, é possível realizar testes utilizando a interface gráfica do *software*, mesmo que aliada a alguma ferramenta de acesso remoto, pode-se verificar, por exemplo, se um teste realizado em um computador teve o resultado esperado, mesmo que o resultado dele seja exibido em outra máquina. Há também a possibilidade de executar testes de integração entre *software*s que utilizam linguagens e interfaces diferentes.

Com a biblioteca que foi desenvolvida, o usuário pode gravar os passos realizados pela ferramenta, para que seja possível, a partir de um resultado, verificar corretamente em que parte do processo surgiu um defeito, assim como evidenciar os testes e gerar um arquivo contendo os resultados de cada teste, sumarizar esses resultados em um documento, onde ficam registrados todos os testes realizados, sua descrição, seus respectivos resultados, a descrição do erro caso haja, e informações que aumentam a rastreabilidade do defeito encontrado.

O usuário, sem ter que ficar observando a realização dos testes, sabe qual foi o resultado do teste e se ele foi diferente do esperado, verificando o relatório gerado pela ferramenta.

Após o desenvolvimento, foi realizado um comparativo, testando três programas com funcionalidades simples, utilizando a ferramenta desenvolvida e manualmente para verificar o quão vantajosa seria a aplicação da ferramenta, conforme mostra a Figura 1.

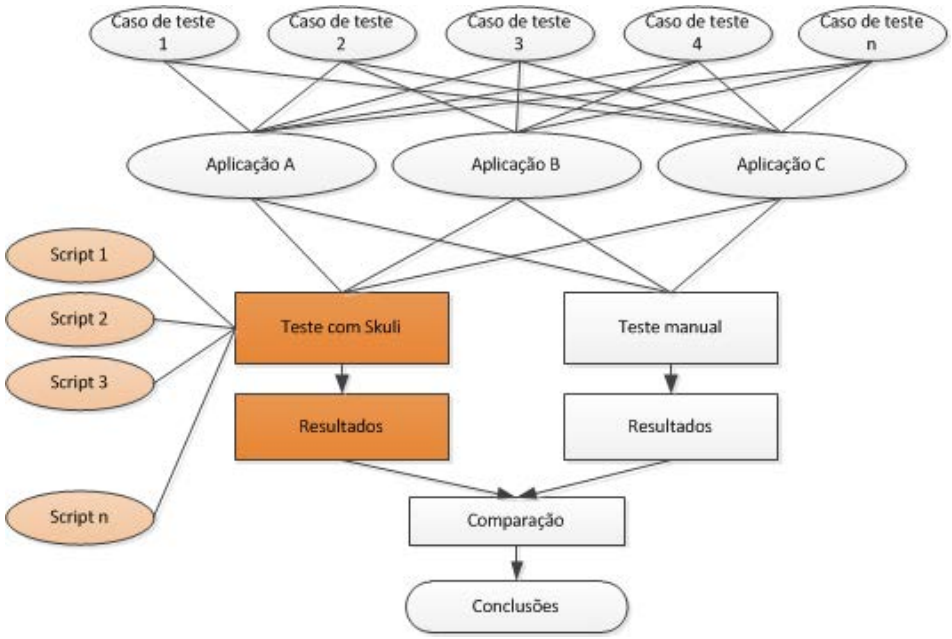

Figura 1 - Processo de avaliação

3.1 TECNOLOGIAS UTILIZADAS

3.1.1 SIKULI

Sikuli (Figura 2) é uma ferramenta para automatizar e testar interfaces gráficas de usuário (*Graphical User Interface*) usando imagens (*screenshot*s). O pacote do *Sikuli* inclui o "*Sikuli Script*", uma API de *script* para *Jython* e o "*Sikuli* IDE", um ambiente de desenvolvimento integrado para escrever *scripts* com imagens facilmente. O *Sikuli Script* automatiza qualquer tela apresentada. Alguns exemplos disso seriam: controlar a navegação em uma página web, controlar um aplicativo desktop para *Windows*, *Linux* ou *Mac OS X*, ou mesmo um aplicativo de *iPhone* ou *Android* rodando em um simulador ou por acesso remoto via VNC (*Virtual Network Computing*).

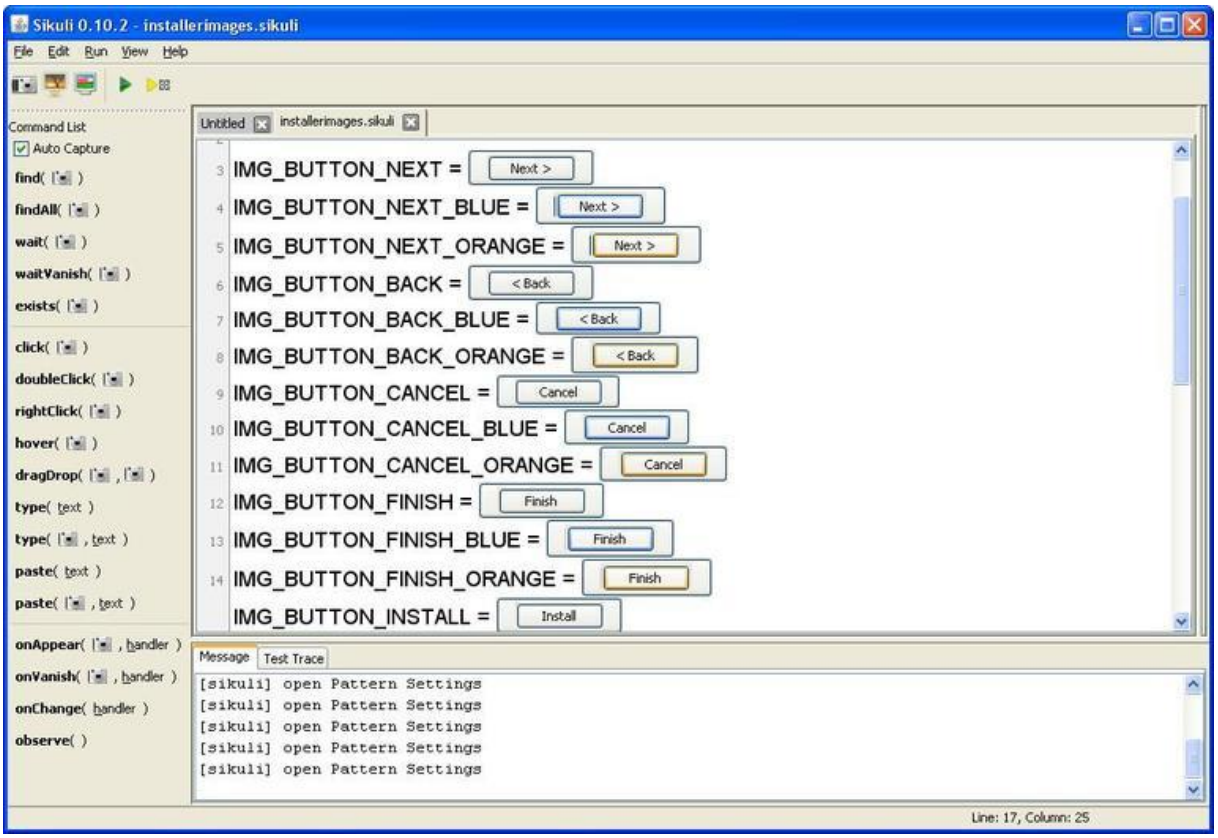

Figura 2 - Sikuli IDE

Sikuli Script é uma biblioteca *Jython* e Java que automatiza a interação com a GUI usando padrões de imagem para direcionar eventos de teclado e mouse. O núcleo do *Sikuli Script* é uma biblioteca Java que consiste de duas partes: *java.awt.Robot*, que dirige eventos de teclado e mouse para locais apropriados, e uma *engine* C++ baseada em *OpenCV*, que procura padrões de imagem exibidas na tela. A *engine* C++ está ligada ao Java via JNI e precisa ser compilada para cada plataforma. No topo da biblioteca Java, uma camada *Jython* é fornecida para o usuário final como um conjunto de comandos simples e claros. Portanto, deve ser fácil adicionar camadas mais para outras linguagens em execução no JVM, por exemplo, *JRuby, Scala, Javascript*, etc.

Um *script Sikuli* (arquivo .*Sikuli*) é um diretório que consiste em um arquivo fonte *Python* (.py), e todos os arquivos de imagem (.png) utilizados por ele. Todas as imagens usadas em um *script Sikuli* são simplesmente um caminho para o arquivo PNG no pacote .*Sikuli*. Portanto, o arquivo fonte *Python* também pode ser editado em qualquer editor de texto.

Ao salvar um *script* usando o *Sikuli* IDE, um arquivo HTML extra também é criado no diretório .*Sikuli*, de modo que os usuários podem compartilhar os *scripts* na web facilmente.

Um *script* executável *Sikuli* (.skl) é simplesmente um arquivo compactado de todos os arquivos no diretório .*Sikuli*. Quando um *script* é passado para o *Sikuli* IDE como um argumento de linha de comando, o *Sikuli* IDE reconhece seu tipo verificando a sua extensão. Se um .skl é visto, o *Sikuli* IDE executa-o sem mostrar a janela do IDE. Se um arquivo .*Sikuli* é visto, o *Sikuli* IDE abre-o em um editor de código fonte.

O *Sikuli* IDE edita e executa *scripts Sikuli*, integra captura de tela e um editor de texto personalizado (*SikuliPane*) para otimizar a escrita dos *scripts*. Para mostrar imagens embutidas no *SikuliPane*, todos os literais de *string* que terminam com ". Png" são substituídos por um objeto *JButton* personalizado (*ImageButton*). Se um usuário ajusta a similaridade padrão de imagem, um "*Pattern*()" é automaticamente criado no topo da imagem.

Para executar um *script Sikuli*, o *Sikuli* IDE cria um interpretador "*org.Python.util.PythonInterpreter*" e automaticamente passa algumas linhas de cabeçalhos (por exemplo, para importar módulos *Sikuli* do *Jython*, e para definir o caminho para diretórios .*Sikuli*) para o interpretador. Uma vez que estes cabeçalhos são definidos, o *script* .py é simplesmente executado pelo "*PythonInterpreter.execfile*()". O funcionamento do *Sikuli* é exibido na Figura 3.

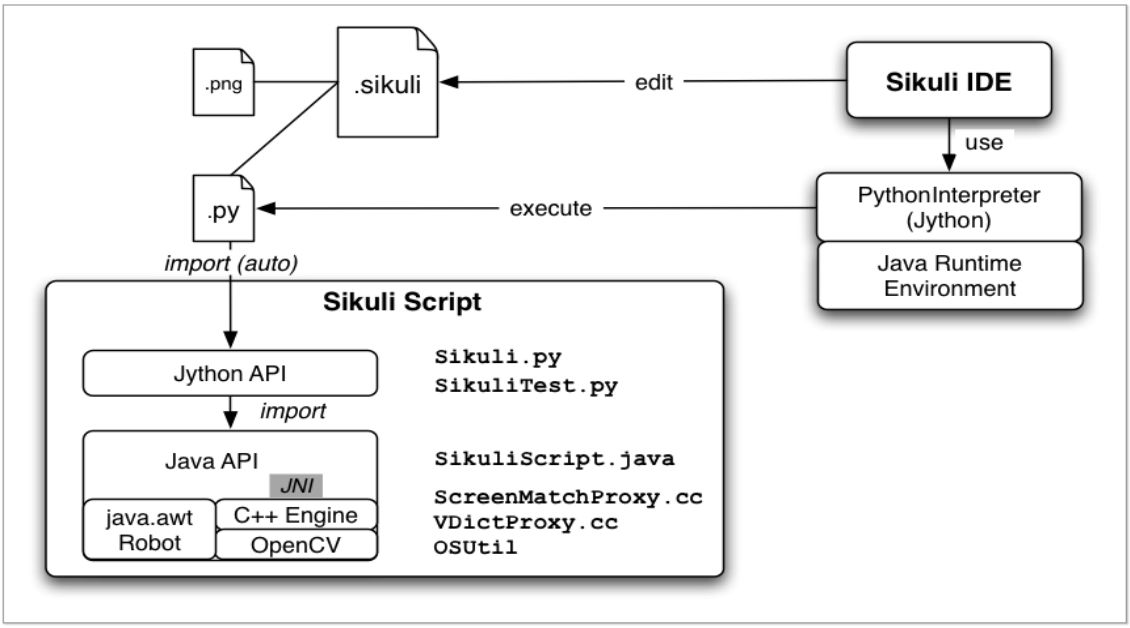

Figura 3 - Como o sikuli funciona

O *Sikuli* funciona através do reconhecimento de imagens da tela do usuário, ele possui funções específicas como click(), find(), exists(), e outras, onde é possível, através de um *screenshot* obtido pelo usuário. O próprio *Sikuli* IDE, oferece esse recurso de capturar de uma imagem que está na tela, utilizando um atalho no teclado [23].

3.1.2 *PYTHON*

A principal tecnologia utilizada na criação do projeto é o *Python* que é uma linguagem de programação de alto nível, interpretada, imperativa, orientada a objetos, de tipagem dinâmica e forte.

O *Python* foi lançado por Guido van Rossum em 1991, e atualmente possui um modelo de desenvolvimento comunitário, aberto e gerenciado pela organização sem fins lucrativos Python Software Foundation. Apesar de várias partes da linguagem possuir padrões e especificações formais, a linguagem como um todo não é formalmente especificada.

A linguagem foi projetada com a filosofia de enfatizar a importância do esforço do programador sobre o esforço computacional. Prioriza a legibilidade do código sobre a velocidade ou expressividade. Combina uma sintaxe concisa e clara com os recursos poderosos de sua biblioteca padrão e por módulos e frameworks desenvolvidos por terceiros.

Python foi escolhido para o desenvolvimento do projeto por ser totalmente compatível com a ferramenta *Sikuli*.

3.2 COMO A BIBLIOTECA FUNCIONA

A biblioteca foi feita para o auxílio na documentação e criação de evidências dos testes realizados.

Os relatórios são criados na linguagem XML, nele primeiramente são descritos os resultados gerais dos testes, dentro da *tag* <*results*>, como o número de casos de teste executados, e quantos obtiveram sucesso, falharam ou ocorreram erros de execução e a *tag* <*tests*>*.* Na *tag* <*tests*>, são exibidos também o número do caso de teste, sua descrição, a data e hora em que ele ocorreu, a duração, e informa se o teste foi executado com sucesso, ocorreu um erro de execução, ou uma falha durante a execução do teste e as *tags* <erro>, onde é descrito o tipo de erro, <*traceback*>, onde descreve em que parte do arquivo de *script* (.py) ocorreu o erro e <caminhoPSR> onde é descrito o caminho do arquivo gerado pelo PSR, onde se encontram os passos do caso de teste.

Há também um método para transformar o arquivo XML em um relatório HTML, facilitando assim sua visualização pelo usuário final. Essa transformação é feita pela ferramenta *xsltproc*, que processa um arquivo XSL, responsável por transformar o arquivo XML em HTML. No método é passado o comando *xsltproc*.*exe* por linha de comando, passando o caminho do arquivo XSL que fará a tranformação, o caminho do arquivo XML que será transformado e o caminho e nome do arquivo HTML que será gerado.

Para criar as evidências foi utilizada uma ferramenta do Windows chamada PSR que captura as ações do usuário como um pacote .zip com *screenshot*s e a sua

descrição, ela é nativa do sistema operacional e é chamada por linha de comando dentro dos *scripts* de teste, passando como parâmetros o caminho do arquivo e informando que se deseja tirar *screenshot*s dos passos do TC (*Test Case*).

3.3 ARQUITETURA DOS SCRIPTS

Serão escritos dois *scripts* para automatização dos testes, um para cada caso de teste e um onde eles serão executados.

No *script* dos casos de teste são definidos o nome e a descrição do TC e uma variável que define o sucesso da execução. A seguir, os passos do TC são executados dentro de um "*try/except*" para que, caso ocorra um erro durante a execução do teste, a situação possa ser identificada e escrita corretamente no relatório, o mesmo ocorre com o resultado esperado, que também é escrito dentro de um "*try/except*". Sendo assim, dependendo do resultado, será escrito no relatório se ocorreu sucesso, falha, ou erro de execução no teste, ao final do *script* é fechado o PSR (Figura 4)

```
from sikuli import *
import sys,re,traceback

def TC_001_00(diretorioPSR,testes):
    import AutoTests,datetime
    Tini = datetime.datetime.now().replace(microsecond=0) #hora em que se inicia a execucao do teste
    nomeTC = "TC_001_00" #nome do caso de testes
    descTC = "Teste soma 1 + 1" #descricao do caso de teste
    success = 1 #variavel que define o sucesso do teste
    testes.abrePSR(diretorioPSR,nomeTC) #abre o PSR
    try:
        #passo 1
        #passo 2
        #passo 3
        #...
        #passo n
        try:
            #resutado esperado do teste
        except:
            descErro = "O resultado foi diferente de 2" #descricao do erro
            testes.Falha(nomeTC,Tini,descTC,descErro) #escreve no XML que o teste falhou
            success = 0 #define o sucesso como 0
    except:
        descErro = "Erro ao executar os passos do TC"
        #diz que houve uma falha na execucao do teste
        testes.Erro(nomeTC,Tini,descTC,descErro) #escreve no XML que ocorreu um erro no teste
        success = 0 #define o sucesso como 0
    if success == 1: #se o teste passou
        descErro = "" #nao ocorreu erro
        testes.Passou(nomeTC,Tini,descTC,descErro)#escreve no XML que o teste passou
    testes.fechaPSR() #fecha o PSR
```

Figura 4 - Script do caso de teste

No início do *script* de execução do teste (Figura 5) serão configurados os nomes dos arquivos que serão usados e criados e os caminhos onde eles serão gerados, em seguida, no método principal do *script* são criadas as pastas para os arquivos de relatório e evidências, caso não existam, serão importados e executados os casos de teste e ao final do *script* o arquivo XML gerado é transformado em um arquivo HTML para que seja exibido amigavelmente ao usuário.

```
import sys,re,traceback,datetime
caminhoLibsTCs = "c:\\Users\\Marcelo\\Desktop\\Testes Sikuli Calc\\TCs"
if not caminhoLibsTCs in sys.path: sys.path.append(caminhoLibsTCs)  #faz com que os TCs possam ser importados
                                                                    # dentro do script
global diretorioPSR,caminhoXML,nomearqXML,XSLT,relatorio,arqXSL,XMLtransf
#variaveis setadas antes do inicio dos testes para configuracoes
version = '1.0.0.9' #versao do produto sob teste
nomearqXML = "Testes Funcionais_" + version + "_Calc" #nome do arquivo XML
dirTestes = "C:\\Users\\Marcelo\\Desktop" #Diretorio onde fica o arquivo .sikuli
diretorioPSR = dirTestes + "\\Testes Sikuli Calc\\PSR\\" + version + "\\" #diretorio onde grava os logs do PSR
caminhoXML = dirTestes + "\\Testes Sikuli Calc\\Relatorios\\" + version + "\\" #diretorio onde grava os arquivos XML
XSLT = dirTestes + '\\XSLTransform\\xsltproc.exe' #caminho do xsltproc.exe
arqXSL = dirTestes + '"\\RelatorioTestes.xsl"' #arquivo xsl
relatorio = caminhoXML + nomearqXML #local onde sera gerado o relatorio HTML
XMLtransf = relatorio + ".xml" #caminho do arquivo XML que sera transformado

def testes_Calc(self):
    import os
    if not os.path.exists(diretorioPSR):#cria o diretorio do PSR caso nao exista
        os.makedirs(diretorioPSR)
    if not os.path.exists(caminhoXML):#cria o diretorio do XML caso nao exista
        os.makedirs(caminhoXML)
    import AutoTests #importa a biblioteca
    testes = AutoTests.LibTestes(nomearqXML,caminhoXML)
    import TC_001_00,TC_002_00,...,TC_N #importa os arquivos de script dos casos de teste
    TC_001_00.TC_001_00(diretorioPSR,testes) #executa o caso de teste
    TC_002_00.TC_002_00(diretorioPSR,testes) #executa o caso de teste
    testes.transformaXML(XSLT,relatorio,arqXSL,XMLtransf) #transforma o XML em um HTML
```

Figura 5 - Script de execução

4. APLICAÇÃO DA FERRAMENTA

Foram selecionadas três aplicações onde foi feito o uso da ferramenta e da biblioteca desenvolvida, duas delas são aplicações "*desktop*" e foram construídas em C#, enquanto a outra é uma aplicação "*Web*", todas com *bugs* conhecidos para que fosse aplicada a automatização dos testes funcionais de acordo com o trabalho proposto.

Para cada uma dessas aplicações, foram criados dez casos de teste (Figura 6), escritos tanto em português como em *Python*, para que possam ser executados através do *Sikuli* IDE.

3.1.1. TC_001_00: Criar um arquivo selecionando a opção Bytes

Atores	Qualquer usuário
Precondições	O FileCreator deve estar aberto
Dados de entrada	Tamanho do arquivo Local e nome do arquivo
Passos	Selecionar o botão de rádio "Bytes" Preencher o tamanho do arquivo Preencher o local e o nome do arquivo Clicar em "Criar Arquivo"
Resultado esperado	Deve ser criado um arquivo com o tamanho dado, no local e com o nome que foram dados
Notas	

Figura 6 - Exemplo de caso de teste

A primeira aplicação é o programa "*FileCreator*" (Figura 6), no qual o usuário seleciona o tamanho de um arquivo e o local onde ele será criado no computador, a segunda se chama "VerdeMilitar" (Figura 7), um programa onde o usuário escolhe uma cor pré-selecionada para ser exibida na tela do próprio software ou então coloca o valor em hexadecimal no modelo de cores RGB, há também o sistema "*TriangleWeb*" (Figura 8), onde têm-se como entrada os lados de um triângulo e como saída o tipo de triângulo que foi dado pelo usuário de acordo com as medidas fornecidas.

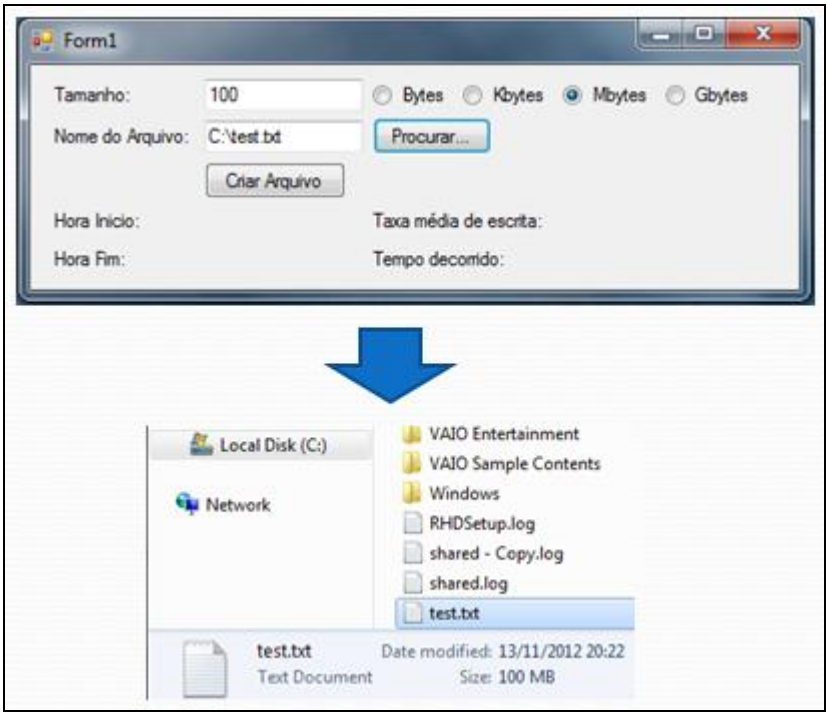

Figura 7 – FileCreator

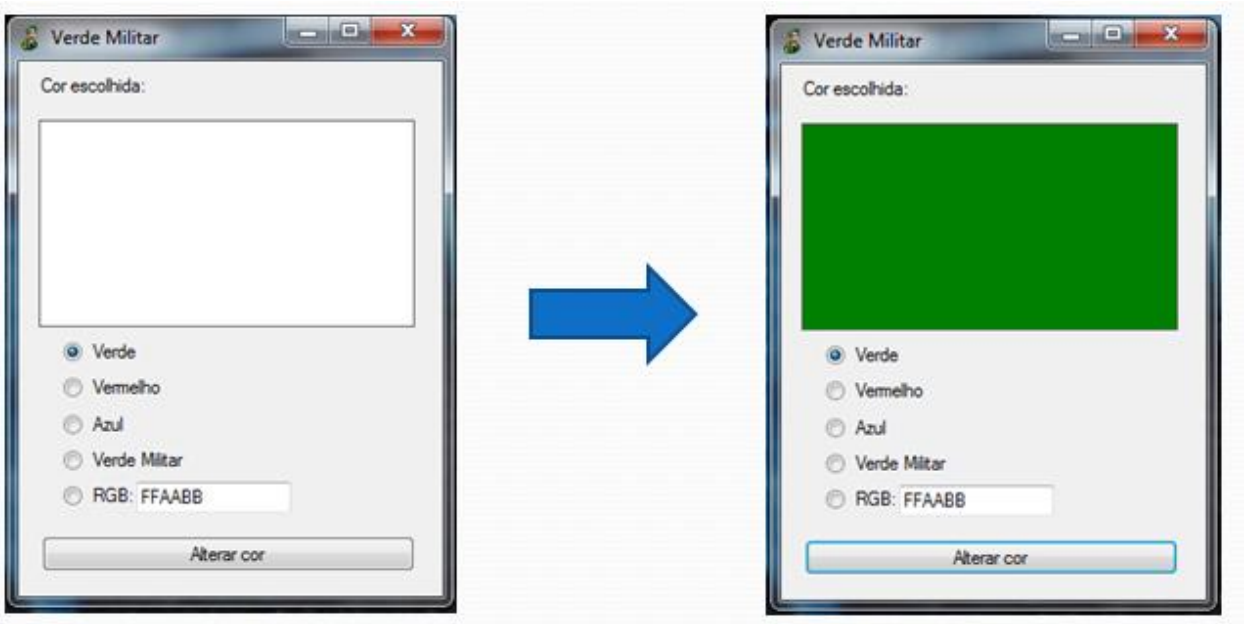

Figura 8 - VerdeMilitar

Figura 9 - TriangleWeb

Foi realizada a comparação do teste manual documentado com o teste automatizado, calculando apenas o tempo de execução dos testes, mas, para que se possa obter um resultado mais consistente, foram estipulado tempos de planejamento, elaboração e manutenção dos testes, que são importantes para demonstrar se existe vantagem na aplicação desse trabalho.

5. RESULTADOS

O tempo de planejamento independe do tipo de teste que foi executado, portanto foi tratado como constante e igual em ambos os tipos de teste, sobre o tempo de elaboração, foi constatado durante o desenvolvimento do projeto que para escrever os *scripts* dos testes automatizados leva-se praticamente o dobro do tempo em relação a escrever os casos de teste no contexto do teste manual.

Para o cálculo do tempo de execução foram levadas em consideração todas as etapas que são feitas tanto manualmente quanto pela ferramenta junto á biblioteca, ou seja, a execução do teste em si, a captação das evidências dos testes e o registro da execução.

Nos testes manuais uma pessoa seguiu um roteiro contendo os dez casos de testes de cada uma das aplicações, e documentando também manualmente os resultados da execução de cada um deles, esse tempo está exibido na coluna “Execução” das tabelas.

O tempo de manutenção foi considerado apenas para os testes automatizados, visto que, caso o *software* sofra alguma alteração o caso de teste escrito não precisaria ser alterado, enquanto o *script* de automatização sim. No entanto, o tempo gasto com manutenção pode variar de acordo com a arquitetura utilizada para automatizar os testes, caso os testes estejam mal escritos, pode-se considerar um tempo de manutenção de até quarenta minutos, ao passo que aliada a uma boa arquitetura a manutenção pode levar apenas cinco. Neste trabalho foi considerado que foram utilizadas boas práticas, porém, como o aprendizado sobre como escrever os *scripts* da melhor maneira foi adquirido durante o desenvolvimento, o tempo de manutenção considerado foi intermediário e de acordo com a complexidade da ferramenta.

Para coletar os resultados foram medidos os tempos (em minutos) de execução dos dez casos de teste de cada uma das três aplicações, como mostram as tabelas 1, 2 e 3:

Tipo de	Planejamento	Elaboração	Execução	Manutenção	Total

teste/Tempo(min)					
Teste manual	5	15	30	0	50
Teste automatizado	5	30	5,5	15	50,5

Tabela 1 – Resultados FileCreator

Tipos de teste/Tempo(min)	Planejamento	Elaboração	Execução	Manutenção	Total
Teste manual	5	9	20	0	34
Teste automatizado	5	18	2,9	10	35,9

Tabela 2 - Resultados VerdeMilitar

Tipo de teste/Tempo(min)	Planejamento	Elaboração	Execução	Manutenção	Total
Teste manual	5	12	25	0	42
Teste automatizado	5	24	3,3	11	43,3

Tabela 3 - Resultados TriangleWeb

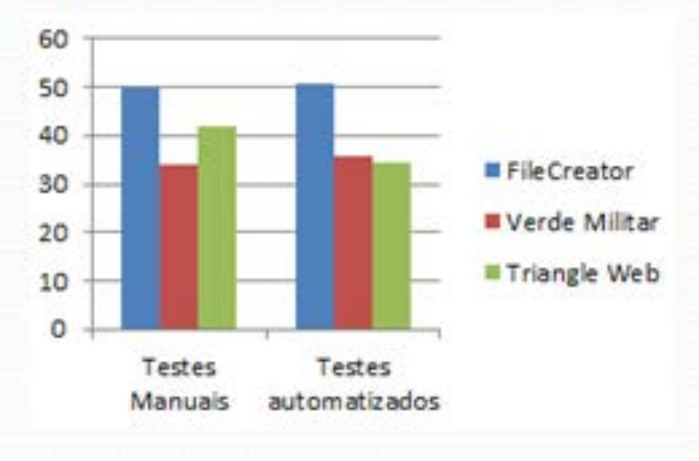

Figura 10 - Gráfico de resultados gerais com uma execução

Pode-se verificar que os tempos obtidos com a aplicação da ferramenta foram muito próximos, porém, como foi levado em consideração o tempo de manutenção, que só seria gasto quando houvesse alguma mudança no *software*, deve-se considerar que os testes devem ser executados a cada versão do *software* que é lançada e então temos:

Tipo de teste/Tempo(min)	Plan.	Elab.	Exec.1	Exec.2	Exec.3	Manut.	Total
Teste manual	5	15	30	30	30	0	110
Teste automatizado	5	30	5,5	5,5	5,5	15	66,5

Tabela 4 - Resultados FileCreator após três execuções

Tipo de teste/Tempo(min)	Plan.	Elab.	Exec.1	Exec.2	Exec.3	Manut.	Total
Teste manual	5	9	20	20	20	0	74
Teste automatizado	5	18	2,9	2,9	2,9	10	41,7

Tabela 5 - Resultados VerdeMilitar após três execuções

Tipo de teste/Tempo(min)	Plan.	Elab.	Exec.1	Exec.2	Exec.3	Manut.	Total
Teste manual	5	12	25	25	25	0	92
Teste automatizado	5	24	3,3	3,3	3,3	11	49,9

Tabela 6 - Resultados TriangleWeb após três execuções

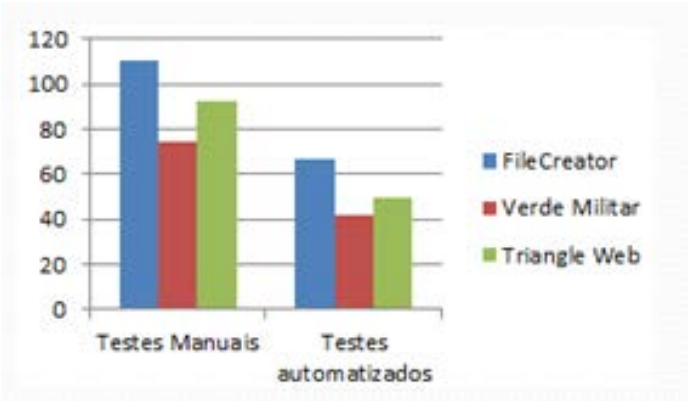

Figura 11 - Gráfico de resultados gerais com três execuções

6. CONCLUSÃO

A automatização de testes funcionais, em curto prazo pode não trazer muito ganho no tempo total da execução dos testes, porém, é possível notar com os resultados desse trabalho que ao utilizar ferramentas de automatização pode-se reduzir esse tempo ao longo do desenvolvimento de um projeto.

Além da vantagem da diminuição do tempo gasto na execução e documentação dos testes, podem ser citadas outras vantagens, como a possibilidade de executar as técnicas de integração contínua e entrega contínua, onde a cada *build* do projeto são executados automaticamente os testes e o desenvolvedor tem um *feedback* quase instantâneo sobre o funcionamento do *software*.

É importante ressaltar que a automatização dos testes não pode ser considerada a solução dos problemas para a garantia da qualidade de *software*, visto que é inviável automatizar todas as situações que um programa pode enfrentar. É recomendado então automatizar as partes repetitivas e que não exijam criatividade de uma pessoa, deixando o restante por conta de testes exploratórios feitos por um profissional da área de QA (*Quality Assurance*).

Analisando os resultados, podemos observar que houve uma redução média de cinquenta e sete por cento no tempo gasto com os testes, e conforme vão sendo produzidas novas versões essa redução pode ser maior, levando a conclusão de que a automatização é vantajosa em longo prazo, lembrando que ainda pode-se diminuir mais ainda o tempo gasto, utilizando melhores técnicas de programação e padrões de projeto como *Data-Driven Testing* e *Page-Object Model*, entre outros.

7. REFERÊNCIAS

ROCHA, A. R., MALDONADO, J. C., WEBER, K. C., et al., "Qualidade de *software* – Teoria e prática", Prentice Há ll, São Paulo, 2001.

PRESSMAN, R. S., "*Software* Engineering: A Practitioner's Approach", McGraw-Hill, 6th ed, Nova York, NY, 2005.

TOMELIN, MARCIO. "TESTES DE *SOFTWARE* A PARTIR DA FERRAMENTA ." Trabalho de conclusão de curso . Bumenau, Santa Catarina, Junho de 2001.

SOUZA, Simoni do Rocio Senger. Introdução ao teste de *software*, Paraná, out. 2000. Disponível em: <www.pbnet.com.br/openline/cefet/horario_sbes_englihs.htm>. Acesso em: 18 abr. 2001.

OLIVEIRA, Flávio Moreira de. Teste de *software*, Porto Alegre, dez. [1997]. Disponível em: <http://www.inf.pucrs.br/~flavio/teste/>. Acesso em: 14 mar. 2012.

MYERS, G. J. The Art of *Software* Testing. Wiley, New York, 1979.

PRESSMAN, R. S. *Software* Engineering – A Practitioner's Approach. McGraw-Hill, 4 edition, 1997.

DELAMARO, M. E. Mutação de Interface: Um Critério de Adequação Interprocedimental para o Teste de Integração. PhD thesis, Instituto de Física de São Carlos – Universidade de São Paulo, São Carlos, SP, Junho 1997.

POSTON, R.M. Automating specification-based *software* testing. IEEE computer society. Chicago: Books, 1996.

BINDER, R. V., *"Testing Object-Oriented Systems – Models, Patterns, and Tools"*, Addison-Wesley,1999.

KANER, C., *"Improving the Maintainability of Automated Test Suites"*, Proceedings of the Thenth International Quality Week, 1997.

FEWSTER, M. & GRAHAM, D., *"Software Test Automation"*, Addison-Wesley, 1999.

FEWSTER, M., *"Common Mistakes in Test Automation"*, Proceedings of Fall Test Automation Conference, 2001.

HENDRICKSON, E., *"The Differences Between Test Automation Success And Failure"*, Proceedings of STAR West, 1998.

MARICK, B., *"Classic Testing Mistakes",* Proc. of STAR Conference, 1997.

PETTICHORD, B., *"Capture Replay - A Foolish Test Strategy"*, Proc. of STAR West, 2000.

FANTINATO, M. et al, *"AutoTest – Um Framework Reutilizável para a Automação de Teste Funcional de Software"*, Artigo, 2000.

IBM Rational. Acesso em 07 de Abril de 2012. Disponível em: http://www-306.ibm.com/*software*/rational/.

TERVO, B., *"Standards For Test Automation"*, Proc. of STAR East, 2001.

NAGLE, C., *"Test Automation Frameworks"*, disponível em http://members.aol.com /sascanagl/Data DrivenTestAutomationFrameworks.htm, 2000.

ZAMBELICH, K., *"Totally Data-driven Automated Testing"*, disponível em http://www.sqatest.com/w_ paper1.html, 1998.

KANER, C., *"Architectures of Test Automation"*, Proceedings of Los Altos Workshops on *Software* Testing, 2000.

SIKULI (2009). Project *Sikuli*. Acesso em 07 de Abril de 2012, disponível em Site do projeto *Sikuli*: http://*Sikuli*.org/index.shtml

VICENZI, A. M. R. & DE DEUS G. D Processo de teste de *software* (2008*)*. Acesso em 05 de Abril de 2012, disponível em: http://www.inf.ufg.br/~auri/curso/arquivos/introducao.pdf