2010

LUIDI VILLELA DE ABREU ANDRADE - LUIZ GUSTAVO SCHROEDER VIEIRA

CTC-UFSC

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

# TESTES EM SOFTWARE LIVRE

LUIDI VILLELA DE ABREU ANDRADE
LUIZ GUSTAVO SCHROEDER VIEIRA

**UNIVERSIDADE FEDERAL DE SANTA CATARINA**
**CENTRO TECNOLÓGICO (CTC)**
**DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA (INE)**
**CURSO DE SISTEMAS DE INFORMAÇÃO**
**DISCIPLINA: TRABALHO DE CONCLUSÃO DE CURSO**
**ORIENTADOR: JOSÉ EDUARDO DE LUCCA**

# TESTES EM SOFTWARE LIVRE

***LUIDI VILLELA DE ABREU ANDRADE***
***LUIZ GUSTAVO SCHROEDER VIEIRA***

**FLORIANÓPOLIS, 23 DE MAIO DE 2011**

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

LUIDI VILLELA DE ABREU ANDRADE
LUIZ GUSTAVO SCHROEDER VIEIRA

# TESTES EM SOFTWARE LIVRE

Trabalho apresentado ao Curso de Graduação em Sistemas de Informação da Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Professor José Eduardo De Lucca.

FLORIANÓPOLIS
2011

*DEDICATÓRIAS, dedicamos esse TCC principalmente às nossas família, mas também às namoradas, amigos e sócios, pessoas que estiveram sempre presentes e nos dando apoio nessa caminhada até o final. Pessoas das quais temos orgulho em conviver e que jamais esqueceremos. Pessoas especiais que estão e estarão sempre presentes em nossas vidas e nossos corações.*

# RESUMO

Esse Projeto de Conclusão de Curso aborda temas que envolvem Testes de Software e Software Livre. O tema principal trata de uma abordagem diferenciada para a aplicação de Testes de Software de forma automatizada no contexto de Software Livre, de modo que facilite os testes para o desenvolvedor e que antecipe o processo de correção de falhas.

PALAVRAS-CHAVE: Teste de Software, Software Livre, Automação de Testes, TDD, Test-Driven Development, QUALIPSO, 5CQualiBR, Plano de Testes, Casos de Teste, Ambiente de Testes, Defeitos, Falhas, Comunidades de Software Livre, Open Source, FSF, Free Software Foundation, Richard Stallman, Linus Torvalds, Software Público, Glenford J. Myers, TESTAVO, Test Expert, Selenium, Bugs, Testes de Desempenho, Testador, Analista de Testes

**ABSTRACT**

This Project Completion of course addresses issues involving tests Software and Free Software. The main theme is an approach differentiated for the implementation of Software Testing in an automated way in the context of Free Software in order to favor the testing of and developer to bring forward the process of correcting faults.

KEYWORDS: Software Testing, Free Software, Automation Testing, Test-Driven Development, TDD, QUALIPSO, 5CQualiBR, Test Planning, Test Case, Testing Environment, Defect, Faults, Free Software Communities, FSF, Free Software Foundation, Richard Stallman, Linus Torvalds, Public Software, Glenford J. Myers, TESTAVO, Test Expert, Selenium, Bugs, Perfomance Testin, Tester, Test Analysis.

# LISTA DE FIGURAS

Figura 1 - Modelo V de Testes .....13
Figura 2 - Ciclo de vida do processo de testes..... 15
Figura 3 - Fluxograma do processo de testes..... 16
Figura 4 – Ciclo do TDD..... 39
Figura 5 – Exemplo aplicação Análise de extremidades.....53
Figura 6 – Exemplo de Partição por Equivalência.....54
Figura 7 – Exemplo de derivação de Test Cases.....57
Figura 8 – Exemplo de Tabela de Decisão..... 58
Figura 9 – Exemplo 2 de Tabela de Decisão.....59
Figura 10 – Exemplo de Grafo de Transição de Estados.....60
Figura 11 – Exemplo da aba Tracker no software Clam Win Free Antivirus.....66
Figura 12 – Fluxograma para automatização de Casos de Teste.....68
Figura 13 – Visualização da Ferramenta da Aspire Systems.....70
Figura 14 – Visualização da Ferramenta osCommerce.....74
Figura 15 – Instalação da Ferramenta osCommerce.....76
Figura 16 – Instalação da base de dados da Ferramenta osCommerce.....77
Figura 17 – Instalação da base de dados da Ferramenta osCommerce.....78
Figura 18 – Configuração da Ferramenta osCommerce.....79
Figura 19 – Término da Configuração da Ferramenta osCommerce.....80
Figura 20 - Instalação do Selenium .....82
Figura 21 – Execução do Selenium IDE.....83
Figura 22 – Execução do Selenium no Firefox.....84
Figura 23 – Login na Ferramenta osCommerce..... 85
Figura 24 – Execução da Ferramenta osCommerce..... 86
Figura 25 – Execução do Selenium na Ferramenta osCommerce.....87
Figura 26 – Execução do Test Case na Ferramenta osCommerce.....96
Figura 27 – Resultado da Execução do Test Case na Ferramenta osCommerce..... 97

# LISTA DE TABELAS

Tabela 1 – Medidas de comparação de utilização do TDD .....46
Tabela 2 – Exemplo de Array Ortogonal .....61
Tabela 3 – Exemplo 2 de Array Ortogonal.....61
Tabela 4 – Exemplo 3 de Array Ortogonal.....61
Tabela 5 – Exemplo 4 de Array Ortogonal.....62

# SUMÁRIO


1. Introdução ..... 10
1.1. Justificativa ..... 13
1.2. Objetivos ..... 14
1.2.1. Objetivos Gerais ..... 14
1.2.2. Objetivos Específicos ..... 14
2. Fundamentação Teórica ..... 14
2.1. TESTES DE SOFTWARE ..... 15
2.1.1. ..... 19
2.2. SOFTWARE LIVRE ..... 7
2.2.1. Desenvolvimento em Software Livre ..... 8
2.2.2. Compartilhando com a comunidade ..... 9
2.2.3. QUALIPSO ..... 10
2.3. SOFTWARE PUBLICO ..... 10
2.3.1. 5CQualiBR ..... 11
2.3.1.1. Testes de Software no 5CQualiBR ..... 12
3. Estado da Arte ..... 13
3.1. Test-Driven Development (TDD) ..... 13
3.2. Descartando os métodos tradicionais ..... 17
4. Proposta ..... 19
4.1. Técnicas de modelagem de casos de teste ..... 19
4.2. Comunidades de Software Livre ..... 20
4.2.1. Imersão na Comunidade de Software Livre ..... 20
4.2.2. Testes em Comunidades de Software Livre ..... 21
4.2.3. Imersão de Testes na Comunidade de Software Livre ..... 22
4.2.4. Criação e Fomento da Comunidade de Testes em Software Livre ..... 22
4.3. Como testar um Software Livre? ..... 23
4.3.1. Automação de Casos de Teste ..... 25
4.3.2. Prós e Contras da Automação de Casos de Teste ..... 28
4.3.3. Ferramentas para Automação de Casos de Teste ..... 29
4.4. Sistema Livre ..... 30
4.4.1. Critérios de Escolha ..... 31
4.4.2. Instalação ..... 31
4.4.3. Selenium ..... 36
4.4.3.1. Selenium IDE ..... 37
4.4.3.2. Como funciona o Selenium: ..... 37
4.4.3.3. Instalações do Selenium IDE ..... 37
4.4.3.4. Exemplo de um Caso de Teste ..... 40
4.4.3.5. Exemplo de um Caso de Teste Automatizado no Selenium ..... 40

5. Conclusão ..... 55
REFERÊNCIAS ..... 56
APÊNDICE A – Ciclo de Vida do Processo de Testes ..... 59
1. Descrição das atividades ..... 59
1.1. Fase de planejamento ..... 59
1.1.1. Atividade: Elaborar a Estratégia de Testes ..... 60
1.1.2. Atividade: Revisão da documentação ..... 61
1.1.3. Atividade: Elaborar Plano de Testes ..... 63
1.2. Fase de preparação ..... 64
1.2.1. Atividade: Elaborar Casos de Teste ..... 65
1.2.2. Atividade: Preparar o ambiente de testes ..... 67
1.3. Fase de execução ..... 69
1.3.1. Atividade: Executar Casos de Teste ..... 69
1.3.2. Atividade: Reportar Defeitos ..... 70
APÊNDICE B – Criação de um Caso de Teste a partir de um Caso de Uso ..... 72
APÊNDICE C – Técnicas de Modelagem de Casos de Teste ..... 74
5.1.1. Análise de Extremidades ..... 74
5.1.2. Partição de equivalência ..... 75
5.1.3. Tabelas de decisão ..... 77
5.1.4. Grafos de transição de estado ..... 79
5.1.5. Tabelas de pares (ou Array Ortogonal) ..... 80
5.1.6. Checklists de testes ..... 82
APÊNDICE D – Funções e Responsabilidades do Processo de Testes ..... 84
APÊNDICE E – Como funciona a aplicação de um TDD ..... 85
APÊNDICE F – Exemplo Clam Win Free Antivirus ..... 87

# 1. Introdução

Não se pode prever que todo software funcione corretamente, sem a presença de erros, visto que os mesmos muitas vezes possuem um grande número de estados com fórmulas, atividades e algoritmos complexos. O tamanho do projeto a ser desenvolvido e a quantidade de pessoas envolvidas no processo aumentam ainda mais a complexidade. [MYERS].

Em 1979, com o lançamento do livro "*The Art of Software Testing*" por Glenford J. Myers, as práticas técnicas de teste de software começaram a ser difundidas. A crescente complexidade dos sistemas da época e o aumento significativo da busca por qualidade fizeram com que os projetistas começassem a dar mais importância à verificação e validação do software. Enquanto a especificação do software diz respeito ao processo de verificação do software, a expectativa do cliente diz respeito ao processo de validação do mesmo.

O surgimento de aplicações de alto risco (aplicações médicas, controle de tráfego aéreo, etc.), exigiu um maior controle dos defeitos existentes neste tipo de aplicação. No início, as aplicações eram apenas “debugadas”, pois não tinham um processo formal para verificação de defeitos ou não era necessário devido à baixa complexidade dos sistemas em vigor.

Gradualmente o processo de testes foi tomando corpo e passou a demonstrar os defeitos de forma objetiva, mostrando se o software satisfaz ou não os requisitos.. A partir da formalização do processo de testes [MEYERS] foi determinado que, por um dos sete princípios de testes de software (detalhados posteriormente), os testes de software têm o objetivo de encontrar defeitos.

Além de prevenir a presença de defeitos, através de técnicas de testes estáticos e dinâmicos, é necessário também mensurar as atividades de testes. Sempre com o objetivo de levantar informações o mais cedo possível no ciclo de desenvolvimento de um software na busca por defeitos.

Com a globalização e a automatização das atividades corporativas, notou-se uma grande dependência dos softwares das atividades diárias de profissionais de diversas áreas. Na área médica, corporativa, financeira e em todas as outras se viu que um defeito numa aplicação pode gerar prejuízos enormes, podendo até custar vidas. Para as empresas prestadoras de serviço, está implícita a qualidade do produto no contrato, se o cliente contratou uma empresa para desenvolver uma aplicação, é exigido no mínimo que esta funcione sem *bugs*. As fábricas de software também percebem o quão custoso é um defeito quando é descoberto em produção, por isso a necessidade de encontrá-lo precocemente. Esses exemplos fazem crer que, cada vez mais, testar aplicações da maneira correta é importante para a satisfação do usuário.

Outro ponto a ser explorado neste trabalho é a questão do Software Livre, que teve início com Richard Stallman [STALLMAN] na década de 80 através do projeto GNU e da fundação da FSF (Free Software Foundation).

Software Livre, ou *Free Software*, é o software que pode ser usado, copiado, estudado, modificado e redistribuído sem restrição. A forma usual de um software ser distribuído livremente é sendo acompanhado por uma licença de software livre (como a GPL ou a BSD), e com a disponibilização do seu código-fonte [29].

O Software Livre teve suas origens com a explosão dos softwares proprietários, os usuários tinham a necessidade de uso das aplicações, mas não podiam obtê-la, se não fosse de forma ilegal (pirateando). Então, começou-se a considerar as quatro liberdades para os usuários que

Software Livre. Sendo estas: liberdade de execução do programa, de estudá-lo e adaptá-lo, de redistribuí-lo e vende-lo e de modificá-lo caso achar necessário.

A importância do conceito de Software Livre se acentuou com a internet, pois os entusiastas ao redor do mundo tiveram chance de compartilhar seus códigos e idéias. O Governo Federal do Brasil vem desde 2005 estimulando o uso de Software Livre cuja iniciativa já estava sendo difundida em países como França, México, Índia e Venezuela. O evento "Conferência Latino Americana e do Caribe sobre o desenvolvimento e Uso do Software Livre" em 2005 (que será detalhada nos próximos capítulos) abriu espaço para diversas discussões e abriu a visão para novas possibilidades. Algumas empresas privadas, observando o aumento do movimento em escala internacional e as iniciativas do setor público, começaram a apostar no software livre nos seus negócios. Por um lado, por uma questão de economia como o pagamento de licenças de softwares proprietários e por outro lado como uma forma de incentivar a liberdade de conhecimento e promover a evolução profissional de seus colaboradores e conseqüentemente da empresa.

Pela exigência de qualidade em software, há uma inerente necessidade de implantação de testes no contexto de Software Livre, que hoje é, em sua imensa maioria, composto por usuários puramente desenvolvedores, sem foco explícito nos testes de aplicação.

## 1.1. Justificativa

"*Given enough eyeballs, all bugs are shallow.*" [RAYMOND], Eric S. Raymond costuma dizer que quanto maior o número de colaboradores, menor o número de bugs. De fato essa frase está correta e faz todo o sentido, desde que, esses colaboradores utilizem a aplicação com o objetivo de achar defeitos. Uma vez que um defeito é encontrado no Software Livre, este é reportado para a comunidade, ou então, corrigido pelo próprio usuário que o encontrou, enviando um novo *patch* com a correção da aplicação. Sendo assim, têm-se inúmero olhos voltados para a aplicação e grande parte destes tem interesse na melhoria do software.

Como a busca pela qualidade é inerente em qualquer circunstância, com Software Livre não é diferente. Dentre vários aspectos que definem a qualidade de um produto, teste é uma delas e pode ser aplicada de diferentes formas, em diferentes contextos. Segundo um dos sete princípios de *Software Testing* definidos por Glenford J. Myers, no seu livro *The Art of Software Testing* [MYERS], teste é dependente de contexto. Ou seja, se existe uma forma de testar efetivamente qualquer aplicação, com software livre não é diferente, só é necessário encontrar uma forma ideal.

Apesar da própria política adotada pelo ecossistema do Software Livre já prever qualidade do produto, visto que, uma vez que seu produto tem uma aceitação pela comunidade, ele possuirá um maior auxílio na elaboração deste. Estima-se que seria gasto mais de um bilhão e duzentos milhões de dólares na criação do LINUX se este fosse desenvolvido por uma empresa de padrões tradicionais de desenvolvimento, sendo assim, de forma não colaborativa.

## 1.2. Objetivos

O objetivo deste trabalho é apresentar uma proposta da inserção do mundo de Testes de Software no contexto de Software Livre. Mostrar que é possível utilizar um *approach* preventivo ao reativo através de técnicas das mais variadas (caixa branca, preta e cinza, testes de desempenho, análise de extremidades, dentre outras)..

### 1.2.1. Objetivos Gerais

- Apresentar técnicas de Testes de Software que sejam aplicáveis ao contexto de desenvolvimento de Software Livre;

### 1.2.2. Objetivos Específicos

- Fomentar uma comunidade de Testadores de Sofware Livre
    - Indicar pontos críticos do software
    - Desenvolver casos de testes, mesmo que superficial
    - Fomentar a utilização de ferramentas de Testes Automatizados
- Encontrar bugs precocemente
- Automatizar os casos de teste para que sejam rodados após a finalização do pacote
    - Uso de técnicas apropriadas para obter maior cobertura do código
    - Testes de regressão baseados em casos de uso de maior risco
    - Utilização de relatórios de execução dos testes automatizados (maior confiabilidade)
- Eliminar a necessidade de interação com analista ou desenvolvedor

# 2. Fundamentação Teórica

## 2.1. TESTES DE SOFTWARE

Para Myers [MYERS], teste é a atividade que consiste em executar um programa com a intenção de encontrar erros. Teste de software é o processo, ou a série de processos, descritos para mostrar que um código de computador está fazendo o que foi projetado para fazer e que não está fazendo nada que não tenha sido definido.

Os motivos para os quais um software deve ser testado são vários, um deles é o problema que um defeito pode causar numa aplicação, levando a grandes prejuízos ao usuário e custo de correção excessivamente alto se encontrado tardiamente.

De acordo com um estudo comissionado pelo *National Institute of Standards and Technology* em 2002, os custos de um defeito num software doméstico chegou a atingir os 59.6 bilhões de dólares nos EUA. Esse estudo também sugere que um terço desse gasto poderia ser evitado se essas aplicações fossem testadas particularmente mais cedo no ciclo de vida de desenvolvimento do software e que boa parte dos gastos com falhas no desenvolvimento de sistemas poderiam ter sido evitadas se aumentassem o tamanho da equipe de testes em pelo menos um terço

O destaque crescente do software como elemento de sistema e os "custos" envolvidos associados às falhas de software são forças propulsoras para uma atividade de teste cuidadosa e bem planejada [Presmann, 1995]. Não é incomum que uma organização de software gaste 40% do esforço de projeto total em teste.

Segundo Pressman [PRESSMAN, 1995], a atividade de teste de software é um elemento crítico da garantia de qualidade de software e representa a última revisão de especificação, projeto e codificação. Ou seja, visando uma melhor qualidade no produto final, as atividades de testes devem, idealmente, andar em paralelo com as demais atividades do projeto, e sincronizadas com as atividades de desenvolvimento para que não tenha perda de produtividade.

O desenvolvimento de sistemas de software envolve uma série de atividades de produção em que as oportunidades de injeção de falhas humanas são enormes (Pressman, 2007). Erros podem começar a acontecer logo no começo do processo, onde os objetivos podem estar

errônea ou imperfeitamente especificados, além de erros que venham a ocorrer em fases de projeto e desenvolvimento posteriores.

Os conceitos de testes de software começam a ser entendidos a partir do momento que se sabe a diferença entre erro, defeito e falha. O erro (ou engano) é causado por ser humano, isso ocasiona um defeito na aplicação, e a partir do momento que essa aplicação é executada com esse defeito, o usuário terá acesso a uma falha.

Sendo assim, o impacto de um defeito numa aplicação crítica (que envolva risco de vida) pode causar danos irreversíveis para a sociedade. Qualquer *software* que trabalhe com um *hardware* que tenha como princípio se mover, é considerado uma aplicação crítica. Pode-se usar o exemplo de um elevador, caso a aplicação não esteja corretamente configurada, pode acarretar conseqüências sérias à saúde dos indivíduos que estão sendo transportados por ele.Beizer [BEIZER, 1990] descreve a situação efetivamente quando afirma: Há um mito segundo o qual, se fôssemos realmente bons para programar, não haveria *bugs* a ser procurados. Se pudéssemos realmente nos concentrar, se todos usassem programação estruturada, projeto *top-down*, tabelas de decisão, se os programas fossem escritos em SQUISH, se tivéssemos as balas de prata certas, então não haveria *bugs*. Assim segue o mito.

Existem *bugs*, diz o mito, porque somos ruins naquilo que fazemos; e, se somos ruins nisso, devemos sentir-nos culpados por isso. Por conseguinte, a atividade de teste e o projeto de casos de teste são uma admissão de falha, o que promove uma boa dose de culpa. E o tédio de testar é apenas uma punição por nossos erros. Punição por quê? Por sermos humanos? Culpa por quê? Por deixarmos de conseguir uma perfeição sobre-humana? Por não distinguirmos entre o que outro programador pensa e o que ele diz? Por deixarmos de ser telepáticos? Por não resolvermos os problemas de comunicação humana que têm existido por aí, por 40 séculos?

Durante a modelagem dos casos de teste, deve-se levar em consideração a aplicabilidade das técnicas de caixa-preta, fazendo com que cada caso de teste, tenha uma maior cobertura do código que está sendo testado. Para Myers [MYERS], para objetivos de teste, um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um erro ainda não descoberto.

Mesmo sabendo que desenvolvimento em software livre (que será abordado posteriormente) é diferente do ciclo de desenvolvimento de software tradicional (modelo cascata,

por exemplo), é importante entender como que o processo de desenvolvimento adotado nas corporações privadas pode ser abstraído para a forma de produzir um software livre.

Partindo desse princípio, é importante entender o Modelo V de Testes de Software (imagem a seguir):

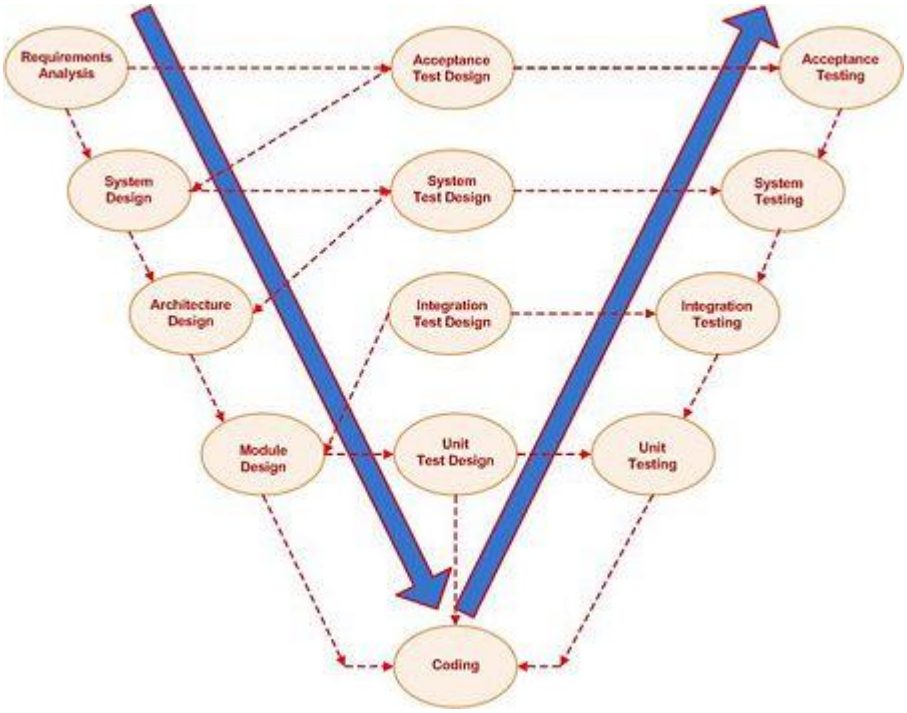



**Figura 1 - Modelo V de Testes**

Basicamente esse Modelo apresenta cronologicamente quais são as etapas do ciclo de desenvolvimento de software que envolve as atividades de teste. O que o modelo prevê é que as atividades descritas à direita do Modelo devem ser definidas nas etapas descritas à esquerda. As próprias fases de definição de projetos devem ter atividades de inspeção e revisão dentro do modelo tradicional.

Dentre as fases definidas por inúmeros autores, principalmente referenciada nas principais certificações do mercado, é importante destacar as fases de testes mais conhecidas:

Testes unitários (ou de unidade) são aqueles no qual são testadas basicamente as entradas e saídas (*inputs* e *outputs*) do sistema, sendo idealmente aplicado por desenvolvedores. São chamados testes de unidade,  essa é a denominação feita para a menor parte testável de um sistema, sendo ela uma classe, um módulo ou um componente. Os testes unitários hoje são popularmente aplicáveis através de técnicas (como o TDD, que será descrito posteriormente) ou programas como JUnit e afins.

Testes de integração são aqueles onde os módulos, já testados, são integrados e devem se comunicar entre si, avaliando critérios de desempenho, confiabilidade e a verificação segundos os requisitos funcionais.

Testes de sistema são aqueles feitos onde os módulos integrados devem estar hospedados num ambiente de produção (ou uma simulação deste, idealmente chamado de ambiente de testes). Entende-se por ambiente de testes, aquele que tem a configuração idêntica ou proporcional ao do ambiente oficial (ou de produção). Para fins de desempenho, entende-se que essa configuração pode ser proporcional à oficial, por exemplo, metade da memória, metade da banda larga, metade do processamento, metade da capacidade, etc.

Testes de aceitação são aqueles idealmente feitos pelo cliente, com fins de validar o sistema de acordo com os requisitos pré-estabelecidos na concepção do projeto. São bastante conhecidos os testes alfa e beta, onde o sistema é disponibilizado ao usuário dentro (alfa) e fora (beta) da organização do sistema em testes. Testes beta são famosos nas empresas de jogos, estas disponibilizam uma versão beta dos jogos de alguns usuários pré-estabelecidos para testes.

Visto os níveis básicos de teste, é importante entender que existem mais padrões que envolvem o teste no ciclo de desenvolvimento de uma aplicação. Como por exemplo, os testes de caso de uso, segurança e desempenho (são essencialmente feitos durante a fase de testes de sistema) bem como a aplicação de outras técnicas de caixa-preta e caixa-branca.

Mas para este estudo em específico, é importante destacar os Testes de Regressão, são aqueles aplicados quando algum novo módulo é adicionado ao sistema ou algum em vigor sofre alguma alteração. Tendo por objetivo garantir que o programa ainda satisfaz seus requisitos [Rothermel, 94].

Segundo Binder [BINDER, 1999], existem cinco abordagens selecionáveis para aplicação de testes de regressão, são estas:

- Reteste total - Re-executar o conjunto de teste base inteiro no sistema;
- Reteste baseado em casos de uso de maior risco - Usa algoritmos baseados no risco para selecionar o conjunto de teste de regressão parcial;
- Reteste por perfil - Algoritmos baseados no perfil operacional dos requisitos dos casos de uso;
- Reteste de segmentos modificados - Análise do código modificado para selecionar o conjunto de teste;

- Reteste com firewall - Analisa dependência de código para selecionar o conjunto de firewall (limite imaginário que cerca software modificado que contém falha de regressão).

  Não utilizar termos ofensivos ao reportar um defeito

  Ver criação de um caso de teste a partir de um caso de uso no Apêndice A.

  Ver matriz de papéis e responsabilidades no Apêndice B.

### 2.1.1.

| | |
|---|---|
| | |
| | |
| | |
| | |
| | |

## 2.2. SOFTWARE LIVRE

Para ser um considerado um Software Livre, necessariamente este deve ter seu código aberto, sendo assim, a Fundação de Software Livre (FSF - *Free Software Foundation*) descreve que Software Livre é qualquer programa de computador que pode ser usado, copiado, estudado, modificado e redistribuído com suas devidas limitações.

A utilização desse termo foi concebida na época em que explodiram os softwares proprietários, eram dados preços cada vez mais altos por softwares que eram vitais para os computadores pessoais. Sendo assim, o conceito de software livre se opõe à essa realidade, que contraria o propósito de que o software em si deve almejar o lucro. Geralmente é anexada uma licença de software livre, seguindo os princípios que serão declarados posteriormente nesta seção, tendo, por obrigação, o código fonte do programa disponível.

Para melhorar o entendimento, é feita uma comparação do "Software Livre" com o conceito de "liberdade de expressão", e não em "cerveja grátis". Mostrando a importância que é a comunidade usufruir dos softwares livres bem como cooperar com a manutenção, divulgação e treinamento do mesmo. Hoje em dia ainda, infelizmente, muitos usuários só utilizam dos softwares sem interagir com a comunidade, promovendo as alterações propostas pela comunidade.

A essência do Software Livre é baseada nas chamadas quatro liberdades, definidas pela FSF são, respectivamente:

- A liberdade de executar o programa, para qualquer propósito;
- A liberdade de estudar como o programa funciona, e adaptá-lo para as suas necessidades. Aceso ao código-fonte é um pré-requisito para esta liberdade;
- A liberdade de redistribuir cópias de modo que você possa ajudar ao seu próximo;
- A liberdade de aperfeiçoar o programa, e liberar os seus aperfeiçoamentos, de modo que toda a comunidade se beneficie. Acesso ao código-fonte é um pré-requisito para esta liberdade.

Só pode ser considerado um Software Livre se o usuário deste possuir as liberdades listadas acima, inclusive revender, oferecer treinamento e cobrar pelo uso deste. Somente não será permitido se o proprietário deste quiser tornar o software proprietário não permitindo que os demais usuários tenha acesso conforme as quatro liberdades.

A interpretação da definição do Software Livre fica a critério do usuários, devendo estas, serem analisadas de forma cautelosa. A liberação de uma licença deve ser garantida pela sociedade e as restrições impostas pelo principal responsável por aquele software devem passar por um processo de análise da comunidade para que este esteja de acordo com a ideologia de Software Livre.

### 2.2.1. Desenvolvimento em Software Livre

Mais conhecido como Desenvolvimento Colaborativo, sendo este considerado pela Comunidade mundial de Software Livre como forma de trabalho revolucionária, sendo este não originado propriamente nela. Na prática o objetivo é o mesmo, mas as abordagens são distintas, como por exemplo o não reconhecimento de hierarquias e instituições dos anarquistas [STALLMAN], como a utilização não autorizada e de livre uso dos bens simbólicos da cultura hacker [LÉVY]; [LEMOS] ou como a permuta, a bricolagem dos pioneiros da micro-informática [LEMOS].

Existem várias equipes de desenvolvimento espalhadas pelo mundo, alguns bem consolidados, desenvolvendo para o GNU-Linux, como Debian e Gentoo. Estes que possuem todo um gerenciamento voltado para este fim, modelo hierárquico próprio, congressos mundiais próprios e uma postura bem definida em relação aos conceitos do Software Livre.

O modelo de gerenciamento para Software Livre sugerido por Linus Torvalds parte do princípio "*release early, release often*" (do inglês: libere cedo e freqüentemente), delegando tudo que pode e esteja sempre aberto a discussões. A comunidade Linux parece assemelhar-se a um grande e barulhento bazar de diferentes agendas e aproximações (adequadamente simbolizada pelos repositórios do Linux, que aceitaria submissões de qualquer pessoa) de onde um sistema coerente e estável poderia aparentemente emergir

somente por uma sucessão de milagres [RAYMOND]. Para Linus Torvalds, o liberar significa discutir com a comunidade e deixe com que eles o ajudem com ideias, desenvolvimento, etc.

Para Raymond, o desenvolvimento de software livre funcionaria perfeitamente, desde que este seja uma promessa aceitável, pois é dessa ideia que a comunidade vai cooperar com *brainstorming* de ideias ou sugestão de co-desenvolvimento. Tendo assim, um desenvolvimento robusto em cima da promessa proposta. "*Given enough eyeballs, all bugs are shallow*." (do inglês: Quanto mais olhos, mais os problemas ficam visíveis), em outras palavras, se um software tiver vários usuários e desenvolvedores, provavelmente vai ter sucesso no desenvolvimento proposto visto que qualquer problema vai ser analisado rapidamente e corrigido.

### 2.2.2. Compartilhando com a comunidade

Segundo Raymond, para se obter excelência no desenvolvimento do software junto à comunidade, é preciso que a ideia seja proposta de forma que seja feita uma aposta, por parte da comunidade, em cima desta. E também é necessário que esse software já tenha sido iniciado de alguma forma, pois é em cima de uma proposta palpável que os demais membros da comunidade vão focar sua atenção naquela abordagem.

O processo básico de compartilhamento de um Software Livre com a comunidade funciona da seguinte forma:

- O responsável inicial divulga para a comunidade;

- O responsável publica o projeto em algum local de hospedagem livre, como um FTP por exemplo;

- O responsável disponibiliza todas as informações, procedimentos e bibliotecas para que os co-desenvolvedores possam trabalhar em cima daquela proposta;

- Encaminhar o programa para a obtenção de uma licença *copyleft*;

- A comunidade começa a trabalhar no projeto como desenvolvedores e co-desenvolvedores.

### 2.2.3. QUALIPSO

O Qualipso (Quality Platform for Open Source) é uma solução tecnológica que engloba diferentes níveis de negócio envolvendo a qualidade do processo. Esse projeto tem por objetivo implementar tecnologias, procedimentos, leis e políticas visando a potencialização das práticas de desenvolvimento do Software Livre, tornando este um artefato mais usual para empresas usuárias deste. Esse projeto foi globalizado com o intuito de ter um maior financiamento por parte das grandes indústrias, academia e governo, tornando assim, o projeto não só um benefício para a comunidade do software livre, bem como um investimento político. Os países colaboradores são de diferentes origens: França, Itália, Brasil, Espanha, China, Alemanha e Escócia. O último país a entrar nessa colaboração foi a Índia, que é país referência em desenvolvimento de software mundial, possuindo uma grande abundância de profissionais capacitados.

O projeto visa melhorar a qualidade do Software Livre de forma global, buscando formas de trabalho que sejam adequadas para quaisquer contextos. Desde a definição de processos de desenvolvimento, modelos de negócio até a implementação de ferramentas para avaliar a qualidade do sistema em termos de interoperabilidade, robustez e escalabilidade. Outro foco do projeto é manter uma rede de profissionais capacitados e interessados em manter a melhoria do produto final de forma que este seja um processo contínuo.

O que torna o projeto interessante é a proposta de implantar uma padronização no aparentemente caótico modelo de desenvolvimento colaborativo, buscando aliar processos formais num contexto onde a informalidade predomina. Apesar da abordagem atípica, o projeto tem conquistado seu espaço no cenário mundial, desenvolvendo práticas como o OMM (Open Source Maturity Model), adequando os processos tradicionais do CMMI ao contexto do Software Livre ou até centros especializados que oferecem treinamentos e workshops para diversas áreas e contextos dentro desse modelo.

## 2.3. SOFTWARE PUBLICO

O conceito do SPB (Software Público Brasileiro) tem por sua essência definir uma forma de trabalho, uma abordagem e o padrão do desenvolvimento de software na Administração Pública e a rede de parceiros da sociedade. Essa forma de trabalho compreende na interação dos órgãos públicos e nas demais esferas de poder, e destes com as empresas e a sociedade. Um dos objetivos é a diminuição dos gastos com software proprietário e outros custos inerentes a não utilização do software livre (e afins), aprimoramento dos aplicativos já utilizados e melhorar o atendimento à população, além de aproximar a sociedade em forma de colaboração.

Já houve a disponibilização de algumas soluções desenvolvidas pelos órgãos responsáveis como o CACIC, que é um coletor automático de informações computacionais e o Ginga, que é um *middleware* para TV Digital, bem como softwares na área da saúde, educação, meio ambiente, gerenciamento de contratos, entre muitos outros.

O portal do Software Público no Brasil é pioneiro, sendo esta sugerida pela primeira vez no FISL (Fórum Internacional de Software Livre). A idéia do portal já tinha surgido na década de 90 mas o portal entrou em vigor só em 2007. Essa proposta já foi estudada para ser implantada em outros países como o Paraguai, por exemplo.

Houve uma facilidade para a introdução de novas ferramentas e soluções nos órgãos federais com a iniciativa do Portal do Software Público Brasileiro, promovendo assim a integração entre as demais entidades e oferece uma gama de serviços públicos tecnológicos para sociedade.

Algumas iniciativas como o 5CQualiBR tem por objetivo a estruturação dos conceitos para obter uma maior qualidade nos processos e produtos desenvolvidos no portal, tendo por base o investimento da comunidade com informações relevantes, cursos, treinamentos e boas práticas.

### 2.3.1. 5CQualiBR

É um projeto que tem como objetivo geral aumentar a Qualidade dos softwares disponíveis no portal do Software Público Brasileiro (SPB) e fortalecer a qualidade dos relacionamentos entre os participantes deste portal. Através de uma ótima mais ambiciosa é possível afirmar que seu propósito poderá, em última análise, aumentar a qualidade da indústria de software Brasileira. Em termos práticos, busca-se por meio do 5CQualiBR promover uma dinâmica de construção de conhecimentos associáveis à qualidade dos

softwares disponíveis no SPB. As premissas para esta construção dinâmica de conhecimentos são a Cooperação e o Compartilhamento de informações e ideias. O meio ambiente para cooperação e o compartilhamento é a Comunidade e, por fim, o "mastro" de sustentação desta dinâmica é a Confiança entre os participantes e a confiança no resultado desta interação coletiva.

O 5CQualiBR será instrumentalizado por meio de um site que hospedará comunidades interessadas em participar da construção de conhecimentos, sobretudo, no sentido de contribuir com o SPB. Cada comunidade neste site irá dedicar-se a temas específicos do conhecimento que inicialmente serão: Interoperabilidade de software; Teste de Software, Qualidade dos produtos de software, Qualidade do processo de desenvolvimento de software, Qualidade de prestação de serviço; Sistema de gestão da produção colaborativa no SPB; Disseminação e sustentabilidade do SPB.

#### 2.3.1.1. Testes de Software no 5CQualiBR

Um subprojeto chamado "Critérios e Procedimentos de Teste de Software para SPB", que tem por objetivo, juntamente com as comunidades do SPB, definir padrões de teste de software. A estrutura tecnológica de teste de software é um conjunto de documentos que abordam os conhecimentos em testes de software com objetivo de apoiar os diversos usuários do SPB numa atividade de teste.

Os documentos e processos descritos no vetor de Testes de Software devem ter a clareza de forma que atenda às necessidades dos usuários, que são dos mais variados nesse contexto, sendo estes:

- Empresas de desenvolvimento de Software;
- Empresas usuárias de Software;
- Entidades governamentais;
- Universidades e Instituições de Ensino e Pesquisa;
- Comunidades de Teste de Software.

# 3. Estado da Arte

## 3.1. Test-Driven Development (TDD)

Test-Driven Development (TDD) ou em português Desenvolvimento dirigido por testes é uma prática de desenvolvimento de software que tem sido utilizada esporadicamente por décadas. Com esta prática, um engenheiro de software realiza um ciclo curto entre a codificação de testes unitários e o desenvolvimento de código para passar por estes testes. O desenvolvimento dirigido por testes recentemente re-emergiu como uma prática vital para o desenvolvimento das metodologias ágeis (Cockburn 2001), em particular a Extreme Programming (XP; Beck 2005). No entanto, pouca evidência empírica suporta ou refuta a larga utilidade dessa prática em um contexto industrial.

O TDD consiste em testes de unidade automatizados, onde são definidos os requisitos do código antes da sua implementação. Os testes contêm asserções que podem ser verdadeiro ou falso. Passando pelos testes, será confirmado se o comportamento do código está correto, permitindo que seja dada continuidade ao desenvolvimento do mesmo.

Com a utilização do desenvolvimento dirigido a testes há uma mudança no paradigma de criação de softwares, já que os programadores passam a codificar pensando em como testá-los. Utilizados de forma correta, o TDD permite a geração de um design avançado da aplicação.

Para o desenvolvimento de um TDD consideramos uma tarefa sendo uma pequena parte de um requisito que pode ser implementado em alguns dias ou menos (Beck e Fowler, 2001). Os engenheiros de software desenvolvem o código através de iterações rápidas com os seguintes procedimentos:

**Ciclos curtos (minute-by-minute):**

- Escrever alguns novos casos de testes automatizados para a tarefa.

- Implementar códigos que deverão passar com sucesso pelos novos casos de testes de unidade.

- Re-executar os casos de testes para garantir que o novo código passe com sucesso.

**Ciclos por tarefa:**

- Integrar o código e os testes no código existente.

- Re-executar todos os casos de teste para garantir o novo código não afete nenhum caso de teste que rodava anteriormente.

- Se necessário, consertar eventuais problemas de implementação do código ou dos testes.

- Re-executar todos os testes para garantir que o código refatorado não pare em nenhum caso de teste que passava anteriormente.

No TDD, o desenvolvimento do código depende do desenvolvedor, pois ele faz constantes alterações no código, na sua estrutura e cria novas funcionalidades. Uma nova funcionalidade não é considerada devidamente finalizada, até que os seus casos de testes sejam criados e executados corretamente.

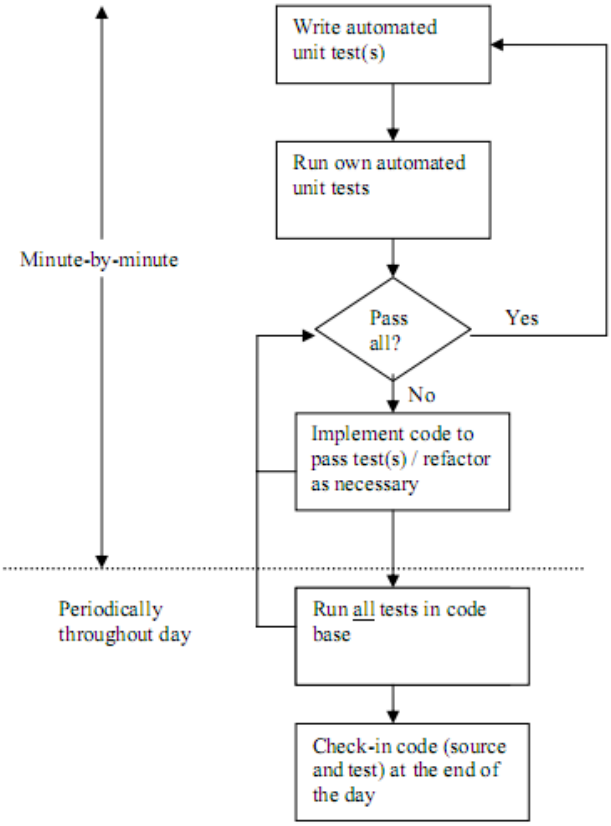



**Figura 4 – Ciclo do TDD**

**Pontos fortes do TDD:**

- O TDD é considerado muitas vezes como um processo de design ao invés de um processo de testes. Kent Beck, responsável pela inclusão do TDD no XP e Ward Cunningham, um dos primeiros defensores de TDD num contexto XP, disse: "Implementação de pré-testes não é uma técnica de teste" (Beck, 2001; Note: a prática de pré-testes eventualmente veio a ser conhecido como TDD). Eles afirmam que TDD é uma técnica de análise e design e que: "código de pré-teste tende a ser mais coesos e menos acoplados do que códigos em que o teste não faz parte do ciclo de codificação" (Beck, 2001).

- A pequena granularidade do ciclo de testes dá um contínuo feedback ao desenvolvedor. Com TDD, falhas e/ou defeitos são identificados muito cedo e rapidamente o novo código é adicionado ao sistema. Consequentemente, a fonte do problema é também mais fácil de ser determinado, pois os problemas surgem aos poucos. É notável que a eficiência de remoção de falhas e defeitos e a correspondente redução no tempo de depuração e manutenção compensam o tempo adicional gasto escrevendo e executando casos de teste.

- Os casos de testes de unidade automatizados escritos com TDD são recursos valiosos para o projeto. Posteriormente, quando o código é aumentado, modificado ou atualizado, a execução dos testes de unidade automatizados, podem ser utilizados para identificar defeitos recentemente criados. Por exemplo, em testes de regressão.

- Infelizmente, manutenções de código e "pequenas alterações" podem ser quase 40 vezes mais propensas a erros do que novas implementações [HUMPHREY] e, muitas vezes, novas falhas são criadas durante a fase de depuração e manutenção. A facilidade de execução dos casos de testes automatizados, depois de que qualquer alteração é realizada, devem também permitir uma integração mais fácil de novas funcionalidades e, portanto, reduzir a probabilidade de correções e alterações introduzirem novos defeitos. Os casos de teste TDD são essencialmente testes de regressão de alta granularidade e baixo nível. Com a constante execução dos casos de testes automatizados, pode-se descobrir quase de imediato se uma mudança gerou algum problema no código, ao invés de uma descoberta tardia, o que eventualmente poderia gerar mais riscos e prejuízos ao projeto.

**Pontos fracos do TDD:**

- Existe certa dificuldade em utilizar o TDD onde é necessário testar uma funcionalidade por inteiro. Por exemplo, as interfaces de usuário, programas que trabalham

com banco de dados, e algumas funcionalidades que dependem de configurações específicas de rede.

- Os testes tornam-se parte do custo de manutenção do projeto.

-Testes mal escritos, são caros para manter. Existe o risco de testes que geram regularmente falsas falhas e são ignorados, quando geram uma falha real, podem ser ignoradas.

- Apoio dos gestores. Sem a organização inteira estar envolvida no processo e desenvolvimento de TDD, a gestão pode acreditar que escrever teste é desperdício de tempo.

-Testes de unidade criados em um ambiente de desenvolvimento dirigido por testes são normalmente criados pelo desenvolvedor, que também irá escrever o código que está sendo testado. Os testes podem, portanto, partilhar os mesmos pontos cegos com o código: Se, por exemplo, um desenvolvedor não perceber que determinados parâmetros de entrada devem ser verificados, provavelmente nem o teste, nem o código irão verificar esses parâmetros de entrada. Se o desenvolvedor não interpreta corretamente a especificação dos requisitos a serem desenvolvidos, tanto os testes quanto código estarão errados.

**Estudo de caso:**

MSN da Microsoft:

Este estudo de caso foi realizado medindo o desenvolvimento de um aplicativo *web services* na divisão MSN da Microsoft [BHAT],. Os fatores de contexto relacionados são apresentados na Tabela 1. O projeto durante seu ciclo de vida de desenvolvimento envolveram entre 5 e 8 desenvolvedores.

| **Descrição da Métrica** | **Valores** |
|---|---|
| Defeitos/mil linhas de código (utilizando TDD). | X |
| Defeitos/mil linhas de código comparado com uma equipe na organização, mas sem a utilização do TDD. | 4.2X |
| Aumento no tempo de codificação da funcionalidade, por causa do TDD (%) [estimado | 15% |

| | |
|---|---|
| pelo gerente de desenvolvimento] | |

**Tabela 1 – Medidas de comparação de utilização do TDD**

Os resultados das medidas apresentadas na Tabela 1 indicam que há um aumento de 4,2 vezes a qualidade do código desenvolvido utilizando TDD comparado ao projeto utilizando uma abordagem não TDD na mesma hierarquia organizacional. Além disso, o projeto que foi desenvolvido utilizando TDD levou 15% mais tempo do que o projeto não TDD.

Alguns pontos a serem questionados:

- Desenvolvedores que utilizaram TDD poderiam estar mais motivados a produzir códigos com mais qualidade, pois estavam testando um novo processo. Por outro lado, todos os envolvidos eram profissionais que tinham que finalizar suas atividades.

-O projeto desenvolvido utilizando TDD pode ter sido mais fácil de desenvolver do que o não TDD. Esse fato pode ser minimizado pelo fato de terem sido gerados pela mesma organização e mesmo gerente. É certo que não existiu uma grande diferença na complexidade do projeto TDD e do não TDD.

Ver Apêndice D – Como funciona a aplicação de um TDD.

## 3.2. Descartando os métodos tradicionais

Dentre vários motivos pelos quais os métodos tradicionais são descartados num projeto de Software Livre, podemos destacar a necessidade de documentação do mesmo. Em processos tradicionais de Engenharia de Software, na fase de concepção do projeto, os produtos são os documentos de requisitos de negócio, especificação, análise de sistemas e outros.

Essa forma de trabalho é muito útil quando se existe uma relação de cliente-empresa e esta precisa ter sua comunicação nivelada a ponto que não se tenha nenhum ruído que possa afetar a satisfação do mesmo. Todas os níveis do processo tradicional (utilizando metodologias do tipo cascata, por exemplo) são extremamente necessários visto que a informação não pode se perder no decorrer do projeto.

Porém, foi visto nos capítulos anteriores que a forma de desenvolvimento colaborativo é, de certa forma, caótica, é inviável a utilização de processos da Engenharia de Software tradicional para contextualização das atividades de desenvolvimento em Software Livre.

Projetos como o QUALIPSO ou o 5CQualiBR são extremamente dependentes da própria comunidade se dar conta da necessidade de uma padronização objetivando o aumento de qualidade do processo e no produto como um todo. Mas isso exige também um nível de maturidade da própria comunidade. A realidade do Software Livre é que, se o usuário desenvolvedor (podendo este ser uma empresa por exemplo) tiver interesse de adequar o programa às suas necessidades, ele o fará, sem seguir processos, metodologias ou procedimentos.

# 4. Proposta

Basicamente a proposta deste trabalho é a criação de uma comunidade de Analistas de Testes responsáveis pela aplicação de Testes de Regressão no sistema em vigor. Os usuários da comunidade teriam por obrigação indicar quais eram os pontos críticos do software e os que são passíveis a serem afetados pelas alterações em vigor. É inviável ter um acompanhamento durante o processo de desenvolvimento de um Software Livre, justamente pela forma de trabalho. Em outras palavras, as atividades de testes tradicionais como revisão e aplicação de casos de teste, caem por água abaixo. Tendo isso em vista, a única oportunidade de testar o sistema é de fato quando ele já foi desenvolvido, ou pelo menos parte dele.

## 4.1. Técnicas de modelagem de casos de teste

**Para que os casos de teste automatizados pela ferramenta possam ter uma maior cobertura do código com o mínimo esforço, é necessário que sejam aplicadas algumas técnicas durante a criação do *script* de testes. Esse script é criado a partir de um software conhecido como *Selenium* (http://seleniumhq.org/), sendo este, uma ferramenta livre do grupo Apache.**

Algumas dessas técnicas como Análise de Extremidades, Partição de Equivalência, devem ser aplicadas durante a scriptação, mas outras técnicas como Tabelas de Decisão, Grafos de Transição de Estados, Tabelas de Pares ou Checklists devem ser aplicadas antes de criar os scripts, com o objetivo de poder prever testes mais robustos (APÊNDICE C – Técnicas de Modelagem de Casos de Teste).

## 4.2. Comunidades de Software Livre

Os repositórios de Softwares Livres mais conhecidos são o Sourceforge e o próprio site do projeto GNU (**G**NU **N**ão é **U**nix). Basta acessá-los e procurar o tipo de software desejado, usufruir bem como colaborar com os demais usuários e desenvolvedores das comunidades de Software Livre.

O Sourceforge (do slogan "*Find and develop open source software*", no inglês significa "Procure e desenvolva software livre") tem uma característica bem interessante e diferenciada das demais propostas de repositórios, além de ser um repositório comum de software único e exclusivamente livres. Os softwares podem ser procurados pelo seu nome ou pela classe a qual ele pertence, através de filtros, de maneira customizada. Cada software possui sua comunidade própria, opção de suporte a dúvidas pela própria comunidade e contato com seus criadores, para os interessados em colaborar.

O projeto GNU foi uma iniciativa de Richard Stallman em 1984 da criação de um sistema operacional compatível com o UNIX que deu início aos projetos de Software Livre, originando, através de uma colaboração de Linus Torvalds, o núcleo desse sistema operacional que mais tarde foi chamado de Linux (seu nome que é a aglutinação de Linus com UNIX).

Dados extraídos no site da IBM (www.ibm.com) sobre as comunidades de Software Livre relatam que 90% dos membros dessas comunidades são vistos apenas como *observadores*, estes acompanham, lêem tudo mas não contribuem com nada; 9% são chamados de *editores*, são aqueles membros que eventualmente tiram dúvidas e participam de forma parcialmente ativa; e apenas 1% destes membros são os chamados *criadores*, são aqueles que têm a preocupação em gerar material e contribuir de fato com a criação, melhoria e propagação dos Softwares Livres.

### 4.2.1. Imersão na Comunidade de Software Livre

As comunidades de Software Livre são bem abertas aos interessados, e estas tem aumentado bastante a cada ano. Os participantes têm observado cada vez mais a necessidade de colaboração para a melhoria e no aumento da qualidade dos Softwares Livres.

Para participar é, de certa forma, simples. O usuário deve definir seu perfil de envolvimento, se é um observador, editor ou criador. Depois disso, basta se inscrever nas listas de discussão, entrar em contato com os *criadores* da comunidade, identificar-se como um desenvolvedor e/ou testador para que os responsáveis possam sugerir uma forma de envolvimento ou relatar as dificuldades ou diretrizes do projeto livre.

### 4.2.2. Testes em Comunidades de Software Livre

Hoje o que existe de Testes em Softwares Livres é a interação dos *criadores* com os *editores*. No caso do portal Sourceforge, cada software possui uma seção chamada *Tracker*, onde os colaboradores postam inconsistências no sistema conforme os padrões de mercado. Sendo assim, é perceptível que a equipe responsável pelo desenvolvimento do Sourceforge tem a preocupação de manter uma comunicação eficiente entre os colaboradores (que, conforme o modelo de desenvolvimento tradicional, seria a figura do Analista de Testes e do Desenvolvedor).

Ver exemplo no Apêndice F – Exemplo Clam Win Free Antivirus.

Existe também a utilização do TDD (já discutido na seção 5.1) durante o desenvolvimento, mas isso é uma decisão que deve ser feita pelo criador do software durante a definição da arquitetura do mesmo, para que um certo nível de qualidade possa ser esperado no software em questão. A utilização de técnicas de testes unitários (conforme descrito na seção 4.1 deste documento) diminui o índice de bugs encontrados no sistema mas não previne que esses estejam completamente ausentes da aplicação.

TDD não previne que bugs estejam ausentes na aplicação porque não considera o ‘fator humano’. Além de ser uma técnica de desenvolvimento e não de testes, sua abordagem é focada em encontrar bugs no código criando-se testes de regressão de alta granularidade e baixo nível, não sendo levada em conta a perspectiva de testes na visão do usuário. Assim, existe uma dificuldade em utilizar TDD quando é necessário testar uma funcionalidade por inteiro e os fluxos de execução mais complexos bem como os fluxos de negócio que o usuário está habituado a realizar no sistema.

Como o TDD é comumento aplicado pela comunidade de software livre como uma técnica normal de desenvolvimento, a comunidade de testes poderia aplicar os testes baseando-se então na visão do usuário para que sejam cobertas as duas abordagens.

### 4.2.3. Imersão de Testes na Comunidade de Software Livre

A abordagem de imersão dos profissionais de Testes na comunidade de Software Livre é mais comercial do que técnica propriamente dita. Hoje existem listas de discussão como a DFTestes do Google Groups ou a própria lista dos profissionais que estão acerca da certificação BSTQB do Yahoo Groups, comunidades no LinkedIn, Orkut, Facebook e outras formas de comunicação na web, como Twitter e Blogs famosos na área de Testes de Software como Test Expert[31], TESTAVO[30], Sem Bugs[32], QualidadeBR[33], entre outros.

Hoje, segundo artigo postado no blog Test Expert, existe um déficit de profissionais na área de Testes que conheçam linguagens de programação, isso se dá pelo fato desses profissionais focarem em atividades de testes manuais durante sua carreira como profissional de testes e perderem contato com linguagens de programação que provavelmente foram aprendidas durante faculdade, cursos técnicos, entre outros.

Por esse motivo, o profissional não tem a oportunidade de utilizar ferramentas de Automação de Casos de Teste, e não consegue vaga em empresas que precisam de profissionais com estes conhecimentos.

Sendo assim, se sua empresa participar na colaboração de algum Software Livre, esse profissional terá a oportunidade de utilizar (ou melhorar) seus conhecimentos técnicos nessas ferramentas de Automação de Testes. Portanto, é necessário que haja uma oportunidade ou uma abordagem ideal para que esses testadores imirjam no contexto do Software Livre.

### 4.2.4. Criação e Fomento da Comunidade de Testes em Software Livre

O princípio básico, como falado anteriormente, é a cooperação entre os profissionais de teste de software mas, acima disto, é também a cooperação entre as comunidades de desenvolvimento com a comunidade de teste de software.

Para fomentar essa idéia, em primeiro lugar é necessário que sejam reunidos profissionais com desejo de compartilhar conhecimento e com paixão por idéias e iniciativas novas.

Ao mesmo tempo, é necessário moderar as comunidades de teste criadas para que cresçam em número sem perder o propósito inicial. Isto porque algumas comunidades, com o passar do tempo, se tornam extremamente políticas e se perdem em discussões intermináveis, muitas vezes até mesmo agressivas. No meio destas discussões seus membros acabam por ficarem perdidos por não conseguirem expor suas idéias. É necessário evitar este tipo de afastamento, por isso é tão importante que exista uma figura que modere e articule todas as iniciativas e bloqueie qualquer iniciativa que não seja a de puramente compartilhar conhecimento e idéias. Criando-se assim um meio propício ao desenvolvimento e crescimento das comunidades de teste de forma saudável.

Também é necessário que profissionais renomados sejam convidados a participar das comunidades de teste e expor suas idéias, se sintam livres para comentar, escrever, discutir e, principalmente, orientar os membros das comunidades em como realizar suas tarefas de teste.

Enfim, não existe ‘fórmula mágica’, para se fomentar uma comunidade de teste de software, mas é preciso que estas comunidades sejam criadas baseando-se nos princípios primordiais das comunidades de software livre e que também seja criado um código de conduta para a participação destes usuários na comunidade sem burocratizar ou impedir o direito de expressão de cada membro da comunidade.

## 4.3. Como testar um Software Livre?

Sendo este o objetivo principal deste projeto de conclusão de curso, foi-se feito um estudo onde a única forma existente e viável de aplicar testes no contexto do Software Livre era em nível de código, aplicando técnicas como o TDD (Test-Driven Development, apresentado na seção 5.1) e outros.

Portanto, conforme os estudos já apresentados nas seções anteriores deste documento, foi observado que, conforme os padrões de mercado, é possível executar fases de teste onde a interação com Analista e/ou Desenvolvedor não é obrigatória. São testes baseados na perspectiva do usuário e no comportamento normal deste em relação ao sistema.

É possível criar um Caso de Teste apenas a partir da descrição de uma funcionalidade mapeada pelo usuário ou por membros do time que estejam visando o ponto de visto do usuário. O nível de detalhamento deste artefato, segundo [Rex Black, 2010], é feito conforme o nível de detalhamento do artefato em questão. Mas como a criação dos Casos de Teste é feita a partir do próprio sistema, este deve ter uma maior riqueza de detalhes, de forma que não se tenha dúvidas de como o testador (ou em outros casos, o usuário) deve executar o software.

Como seria o esforço dessa comunidade de testadores? Primeiramente em conjunto com os usuários e desenvolvedores do software em questão, dever-se-ia mapear todas as funcionalidades e, a partir destas, descrever todos os casos de teste (conforme já descrito na seção 4.1.2.4) utilizando as técnicas de testes de software (descritas na seção 8.0) com objetivo de minimizar o esforço na criação dos Casos de Teste com uma maior abrangência do código em questão. Além disso, existe a questão de possíveis falhas na definição e descrição dos Casos de Teste. Este tipo de falha se detectada na execução dos testes pode acarretar um re-trabalho considerável e prejudicar na confiança da comunidade de desenvolvedores com a comunidade de testes. Para resolver tal situação deve-se levar em conta um moderador, uma pessoa responsável por revisar os Casos de Teste e aprová-los ou não. Esta pessoa deve ter conhecimento considerável na aplicação e devolver o Caso de Teste ao testador caso este tenha cometido algum erro de especificação. Este tipo de solução pode prevenir com que erros na especificação gerem erros na execução dos testes, aumentando assim a confiabilidade dos testes.

Todavia, suponha-se que sejam mapeados 1500 casos de teste para um software de médio porte em questão e toda vez que um usuário/desenvolvedor faça uma alteração, a comunidade de usuários e testadores retestaria todos os casos de teste novamente. Nesse caso, supõe-se que cada caso de teste demore cinco minutos para ser executado, numa conta simples, demoraria 125 horas para que todos os casos de teste fossem executados. Se estes fossem executados de forma colaborativa, paralela e sincronizada por 100 usuários, demoraria 1 hora e 15 minutos para que o software fosse retestado por completo.

Na teoria funciona, na prática, não. É inviável, para não dizer impossível, reunir 100 testadores/usuários que estejam dispostos a trabalhar em paralelo e de forma sincronizada, num software de médio porte de uso razoável. Ou seja, a única opção acessível para possibilitar a execução de Testes de Regressão na aplicação seria Automatizar os Casos de Teste.

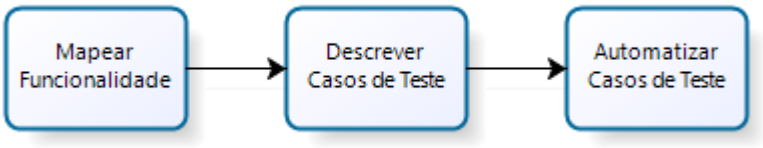



**Figura 12 – Fluxograma para automatização de Casos de Teste**

A aplicação de Testes de Regressão, que conforme já especificado na seção 4.1 é o tipo de teste feito quando alguma alteração é feita no código já existente e estável, é viável no contexto de Software Livre.

Voltando nos capítulos anteriores, segundo [BINDER], existem cinco abordagens selecionáveis para aplicação de Testes de Regressão. Por questões de praticidade, a abordagem que é sugerida para Softwares Livres é a do Reteste total. Por que as demais abordagens não são aplicáveis?

Re-teste baseado em casos de uso de maior risco – Embora a comunidade de Software Livre tenha um grande interesse em manter os níveis de qualidade nos software por eles desenvolvidos, é trabalhoso depender de membros que tenham um conhecimento técnico e de negócio do software. Ou seja, esses membros providos de capacidade para descrever quais são os “casos de uso” (sim, entre aspas, pois não existe caso de uso em Software Livre), cenários ou qualquer definição de módulos que possuem um maior risco para o software em estudo, ficariam sobrecarregados. Mas nem isso, essa forma de trabalho iria demandar certo gerenciamento por parte de alguns membros.

Reteste por perfil – Inviável definir quais são os perfis operacionais dos requisitos dos casos de uso. Mesmo que essa abordagem seja passível a estudo, cairia na mesma questão do item anterior (Reteste baseado em casos de uso de maior risco).

Reteste de segmentos modificados – Não é aplicável, pois exigiria que o desenvolvedor descrevesse quais classes, módulos e partes do software que foram alteradas naquela versão que está sendo disponibilizada.

Reteste com firewall – Não aplicável.

É visível que a abordagem mais trabalhosa é do Reteste total. Mas como tudo em Software Livre funciona de forma colaborativa, a comunidade de testadores também deve ter um esforço grande para manter a qualidade do software em questão.

### 4.3.1. Automação de Casos de Teste

Automação de Casos de Teste, segundo a Wikipédia, é o uso de software para o controle da execução do teste de software, a comparação dos resultados esperados com os resultados reais, a configuração das pré-condições de teste e outras funções de controle e relatório de teste. De forma geral, a automação de teste é começada a partir de um processo manual de teste já estabelecido e formalizado.

Para automatizar Casos de Teste é necessário utilizar um software de forma que descreva os passos exatos de uma atividade no sistema (mais comumente chamada de caso de teste). A descrição desses passos é feita através de um script criado nessa ferramenta, que pode ser das seguintes formas:

- Leitura dos campos na tela, a ferramenta faz a leitura do código da aplicação, identificando quais são os campos e métodos referentes na tela;

- Por referência, durante a criação do script, o programador identifica a posição (por coordenada XY) do elemento na tela e qual interação o script terá com aquele elemento.

Lembrando que, é necessário que o código esteja estável e funcionando para que os scripts sejam criados. Apesar de existir uma proposta de mercado chamada *Right from the Start* (na tradução literal, "desde o começo") que tem como premissa básica, padronizar a criação de documentos de requisitos e especificação técnica de modo que os scripts de casos de teste automatizados sejam criados antes do sistema ser desenvolvido, ou seja, o desenvolvimento do software é criado em paralelo com o desenvolvimento dos scripts de automação dos casos de teste.

Dentro dos parâmetros normais de teste, existe também a possiblidade de algum erro de especifição nos testes ocasionar um erro na geração dos scripst e conseqüente erro na execução dos testes, sendo que o código pode estar funcionando corretamente. Para tratar isto, é necessário que a figura do moderador retorne e dê o aval se é um bug de teste ou um bug no desenvolvimento. Este tipo de problema, se resolvido sem moderação, pode ocasionar numa discussão interminável entre membros da comunidade de desenvolvimento e membros da comunidade de testes.

Existem também algumas iniciativas (nesse caso, privada), como a da empresa Aspire Systems, que prevê o Retorno do Investimento (do inglês, *ROI – Return of Investiment*) de uma ferramenta de automação de casos de teste numa empresa. São feitos alguns cálculos nos detalhes do produto, tecnologia, tamanho do time de testes, quantidade e duração das entregas do projeto, parâmetros da execução de testes (como número de casos de

testes e outros) e custos em geral da equipe de testes. São apresentadas conforme a figura abaixo:

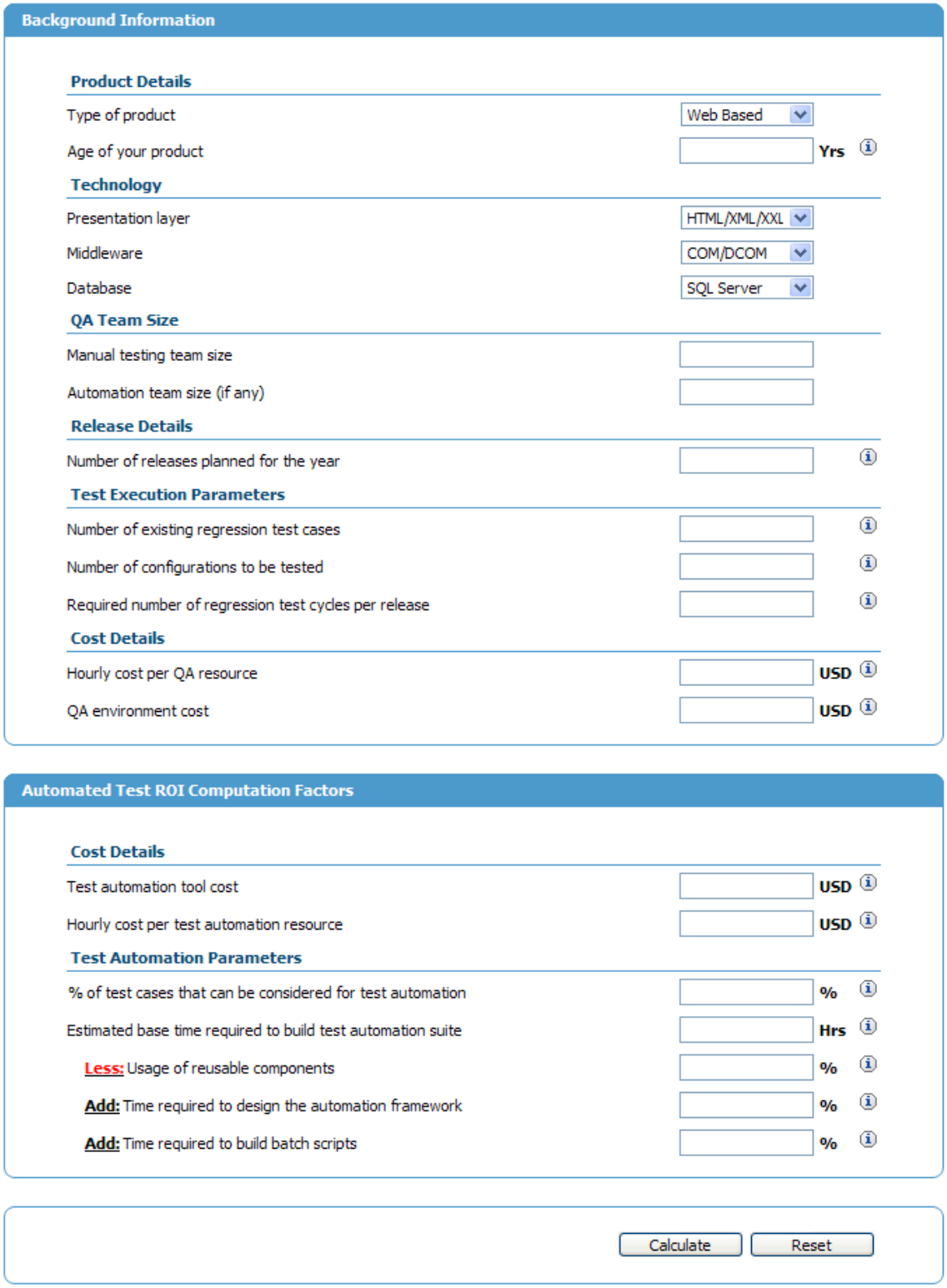



**Figura 13 – Visualização da Ferramenta da Aspire Systems**

Em outras palavras, conforme pesquisa feita no blog TESTAVO[30], é desaconselhável automatizar casos de teste em projetos curtos (ou de poucas entregas) ou módulos/projetos que sofrem mudanças constantes. Isso se dá ao fato de que teria muito retrabalho para manter os scripts atualizados. Mas mesmo que os gestores do projeto optem por automatizar os testes, é aconselhável que parte deles seja manual (aproximadamente 75%).

### 4.3.2. Prós e Contras da Automação de Casos de Teste

Conforme descrito na seção anterior, existem situações específicas nas quais um projeto deve optar por automatizar parte dos seus casos de teste, sendo assim, é importante levar em consideração quais são os pontos positivos e negativos quando o projeto toma essa decisão, diz Caetano, criador do blog TESTEXPERT[31].

Pontos positivos:

- Tempo de execução: Esse é o grande motivo pelo qual existe um encantamento por parte dos gestores pela automação de testes, o tempo de execução de um projeto é muito curto, se limitando a "um clique", rodar os scripts e avaliar quais obtiveram sucesso e quais não funcionaram – encontrando assim, um erro;

- Elegibilidade: É simples de identificar o que está sendo feito no script, uma vez que está descrito o caminho que ele percorreu no sistema e os dados utilizados nesse caminho;

- Foco: O script faz exatamente o que foi descrito, existe uma maior garantia que o script está implementado corretamente uma vez que o caminho percorrido é padrão;

- Modularização (reutilizável): O script também pode ser criado orientado a objetos, reutilizando os módulos já criados em novos scripts. Por exemplo, modularizando o caso de teste "Cadastro de clientes", o script ficaria dessa forma: Login/Cadastro/Cliente; reutilizando assim, os módulos Login/Cadastro para o script "Cadastro de Empresas", ficando dessa forma: Login/Cadastro/Empresa.

Pontos negativos:

- Tempo de Desenvolvimento: Em projetos tradicionais de desenvolvimento de software, principalmente quando estes são criados sob demanda, o tempo de desenvolvimento é um fator crítico na entrega dos projetos. Como diz o ditado: tempo é dinheiro.

- Confiabilidade: Pelo fato do script funcionar de forma linear, diferentemente de um Analista de Testes experiente, o script não vai explorar as demais funcionalidades da tela forçando a mesma para encontrar um defeito.

- Custo: O custo aumenta drasticamente, uma vez que, conforme estudos, o tempo de desenvolvimento de um script é 2/3 do tempo de desenvolvimento do mesmo software, tornando o projeto aproximadamente 65% mais longo. Gerando assim, um custo adicional para o projeto. Ou seja, para que uma empresa opte por utilizar ferramentas de automação no seu contexto, é preciso que faça um estudo de viabilidade de aplicação de uma ferramenta em específico, caso contrário, o custo do projeto pode ultrapassar o esperado.

### 4.3.3. Ferramentas para Automação de Casos de Teste

Conforme dito nas seções anteriores, são utilizadas ferramentas para a criação desses scripts de automação de casos de teste. Portanto, é preciso levar em consideração que, por melhores que sejam as ferramentas, hoje ainda não é possível encontrar uma ferramenta que atenda 100% as expectativas dos projetos. Sendo assim, durante o estudo de viabilidade, é preciso avaliar se a ferramenta é útil em determinado contexto.

Um dos critérios para a escolha da ferramenta de automação de Casos de Testes é o fato de ela ser baseada na visão do usuário. Outro critério importante é o fato de a ferramenta também ser Livre, para que possa estar de acordo com os princípios da comunidade de Software Livre. É necessário também que seja levado em consideração o nível de aceitação da ferramenta no mercado de software livre. Baseando-se neste critério. dentre as principais ferramentas do mercado, é importante destacar as seguintes ferramentas:

- *Badboy*: É uma ferramenta *free*, fácil utilização, porém limitada em relação a recursos tecnológicos. É fácil de utilizar em páginas web simples como CRUD's (*Create Read Update and Delete*, do inglês: Criar, Ler, Atualizar e Apagar). Porém, esta ferramenta não possui muita aceitação no mercado de ferramentas de automação para projetos de grande porte.

- *HP-Mercury Quicktest* Pro: É uma ferramenta privada de um custo elevado. Todavia, a HP-Mercury é responsável pelas ferramentas de ponta de Testes de Software, obtendo outras iniciativas como o *Loadrunner* (para Testes de Desempenho) e o *Quality Center* (para Gestão de Testes);

- *IBM Rational Robot*: É uma ferramenta privada de um custo elevado e de qualidade inferior ao HP-Mercury Quicktest Pro;

- *AutomatedQA TestComplete*: É uma ferramenta privada, porém de baixo custo. Atende a projetos de médio porte e está em constante atualização. Bom custo benefício.

- *Selenium*: É uma ferramenta livre, que possui colaboração de grandes *players* do mercado como a equipe do Google. É de grande aceitação pela comunidade de software livre, está de acordo com os princípios da comunidade e os cenários são baseados na iteração entre usuário e aplicação. Logo atende todos os requisitos necessários para a escolha da ferramenta. É a ferramenta objeto deste estudo. Será detalhada nas seções seguintes.

## 4.4. Sistema Livre

O osCommerce 3 é um sistema de loja online gratuito e open source que é disponibilizado segundo a licença GNU. Possui um vasto conjunto de funcionalidades e permite a administração do sistema sem custos, taxas ou limitações.

Está em funcionamento, aproximadamente, desde 2001 e já foi utilizado por mais de 12.700 lojas cadastrada na seção "lojas online" do site e mais milhares de lojas online pelo mundo.

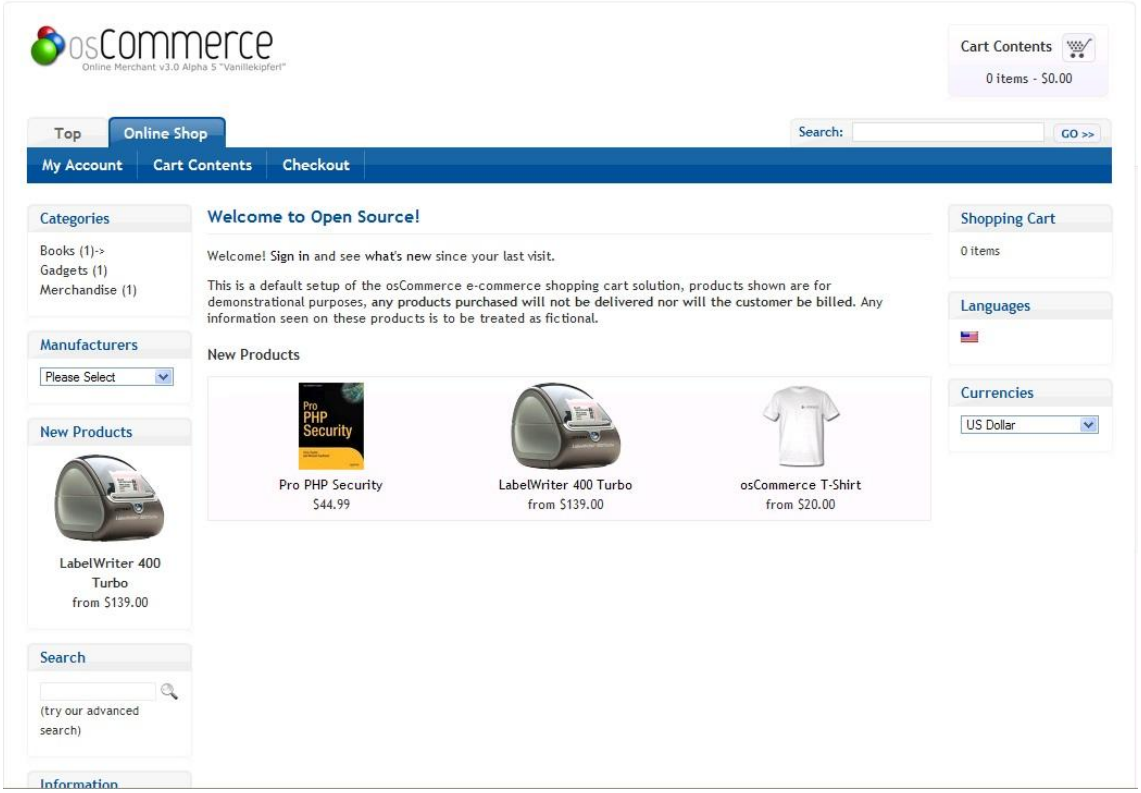



**Figura 14 – Visualização da Ferramenta osCommerce**

### 4.4.1. Critérios de Escolha

O osCommerce possui uma comunidade muito ativa que consiste em mais de 239 mil proprietários de lojas e desenvolvedores que criam e atualizam seus módulos diariamente e dão suporte uns aos outros.

Entretanto, não existe uma metodologia de testes que consiga diagnosticar falhas entre as diversas atualizações e instalações dos módulos.

### 4.4.2. Instalação

O osCommerce 3 necessita dos seguinte pré-requisitos:

- PHP v5.2+ (com extensão MySQLi)
- MySQL v4.1.13+ ou v5.0.7+

O download do código pode ser feito a partir deste link: http://www.oscommerce.com/solutions/downloads

Após baixar o sistema e instalar as dependências, ele deve ser extraído e copiado para o diretório raiz onde o servidor PHP lê os documentos, geralmente conhecido como "Document Root". Para iniciar a instalação os seguintes passos devem ser seguidos:

1) Abrir no navegador o endereço de onde o osCommerce foi copiado. Neste caso http://localhost/loja.

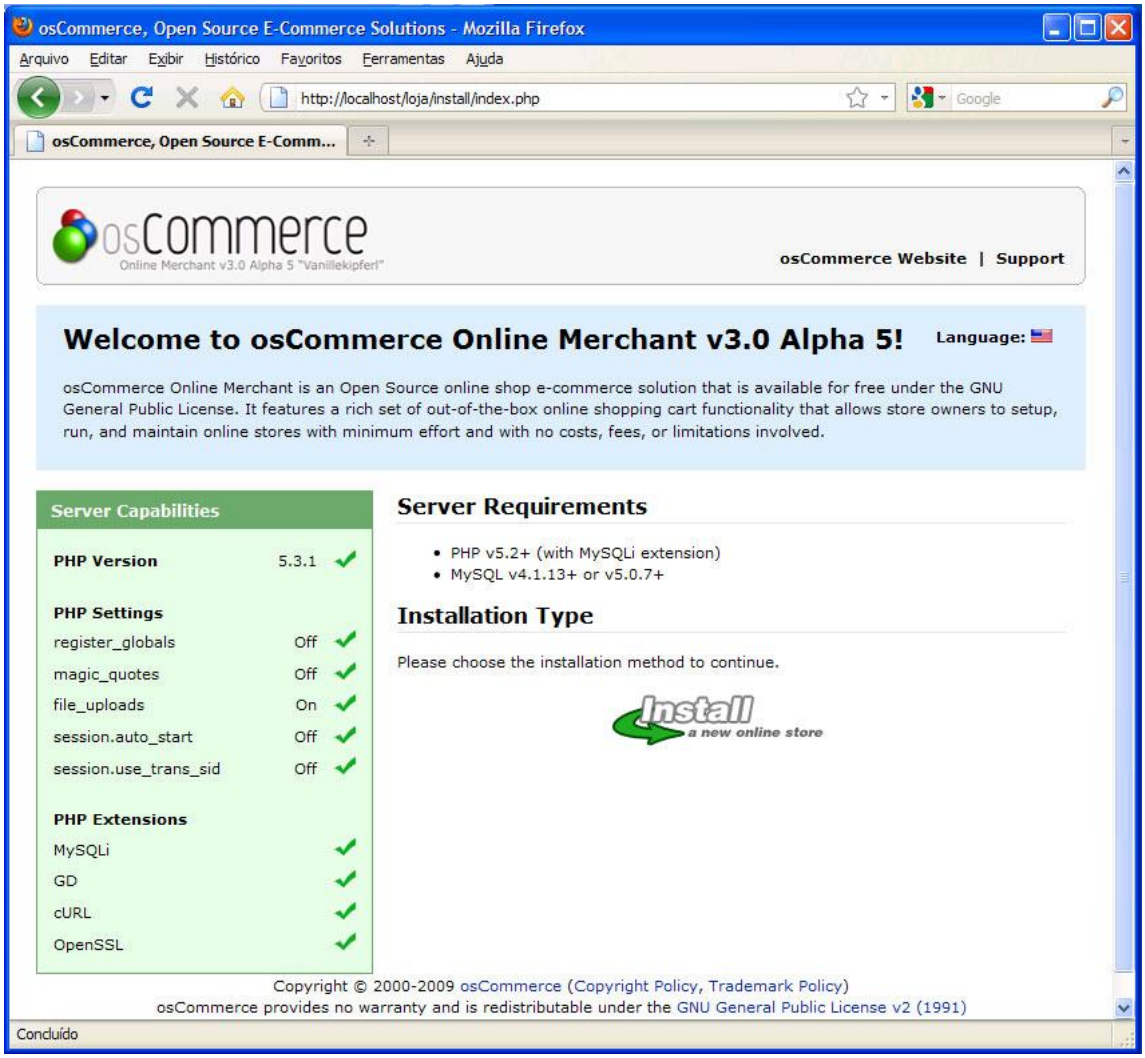



**Figura 15 – Instalação da Ferramenta osCommerce**

2) Se todos os requisitos para o funcionamento correto do sistema forem atendidos (ver na tabela "Server Capabilities"), clicar em "Install".

3) Nesta tela colocamos o endereço onde o banco de dados está instalado.

Para configurar a base de dados, deve-se tê-la criado anteriormente. Aqui foi dado o nome de "loja".

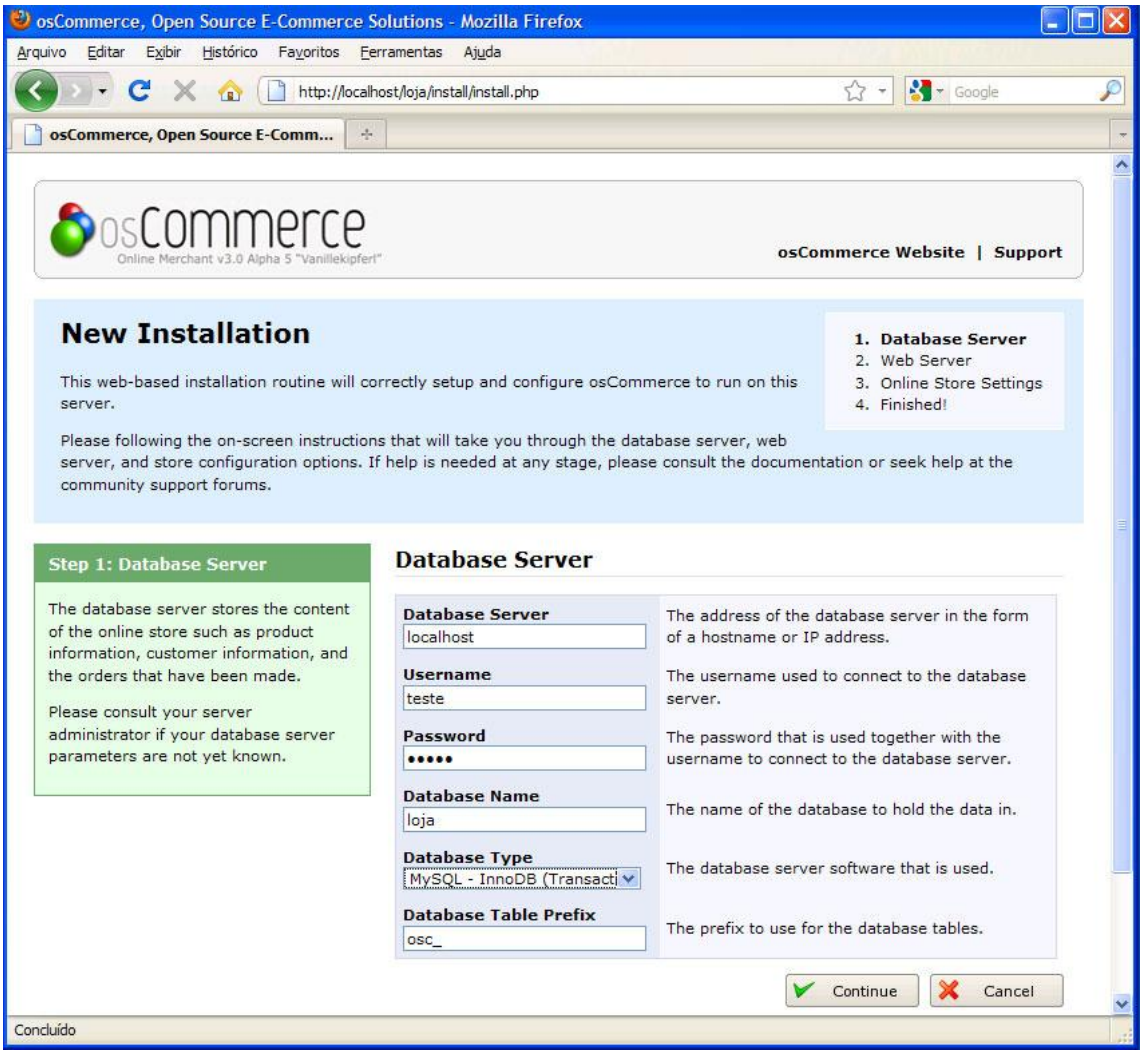



**Figura 16 – Instalação da base de dados da Ferramenta osCommerce**

4) Se a base foi instalada com sucesso, a instalação irá para esta página. Basta confirmar as informações.

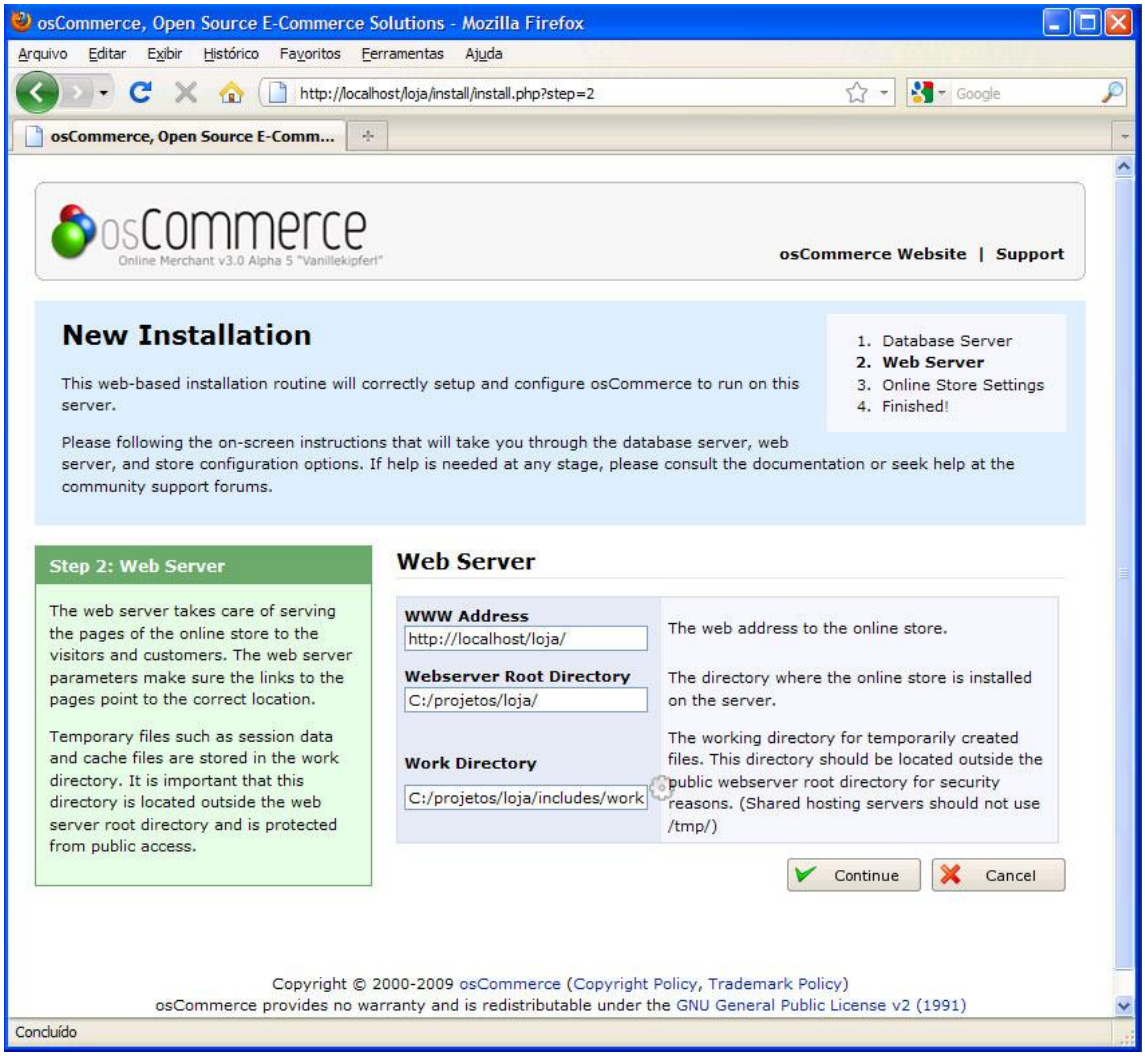



**Figura 17 – Instalação da base de dados da Ferramenta osCommerce**

5) Nesta etapa deve-se configurar os dados da loja e do administrador e apertar o botão "continue".

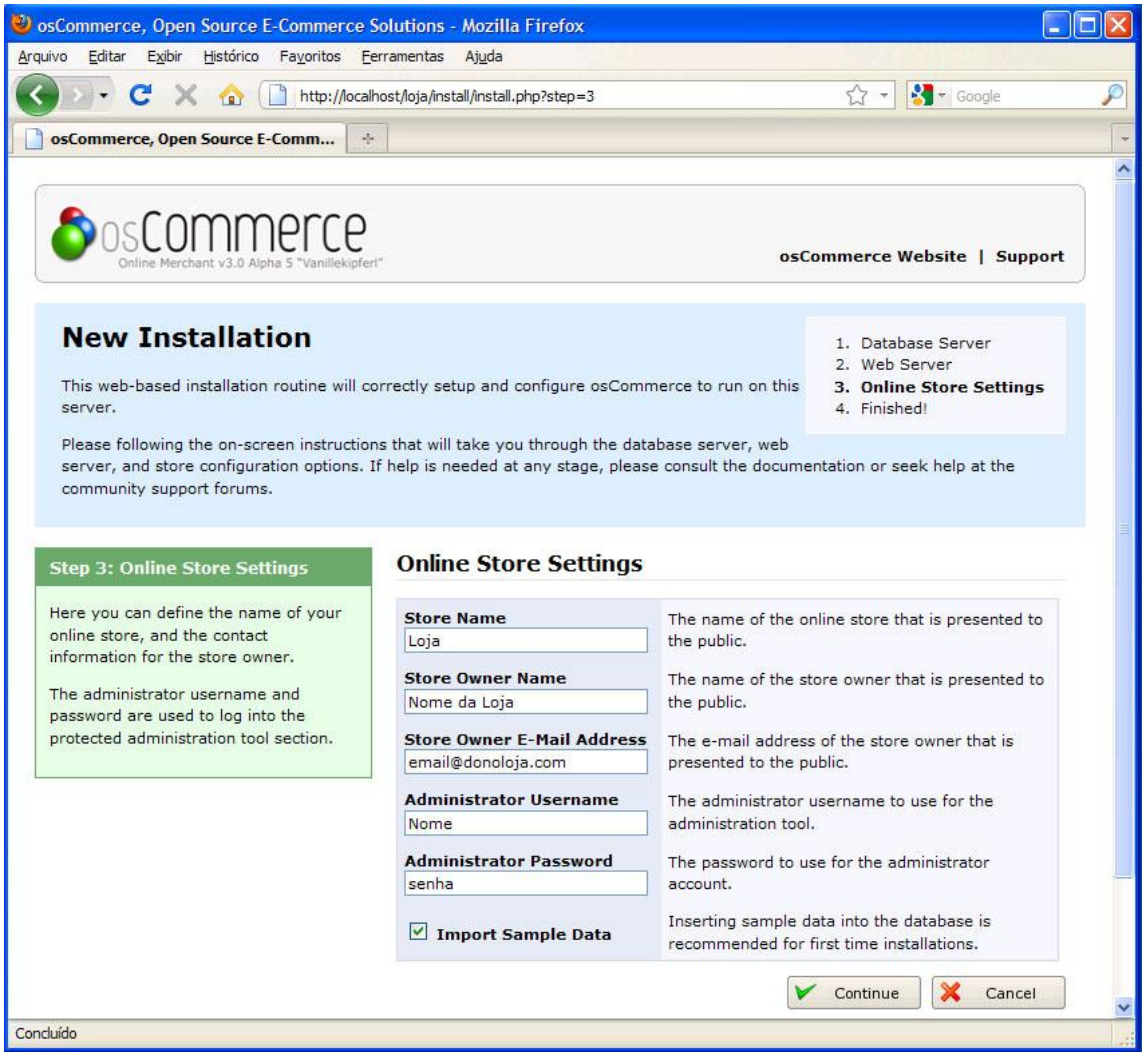



**Figura 18 – Configuração da Ferramenta osCommerce**

6) Se o sistema foi instalado com sucesso, esta tela deverá ser exibida:

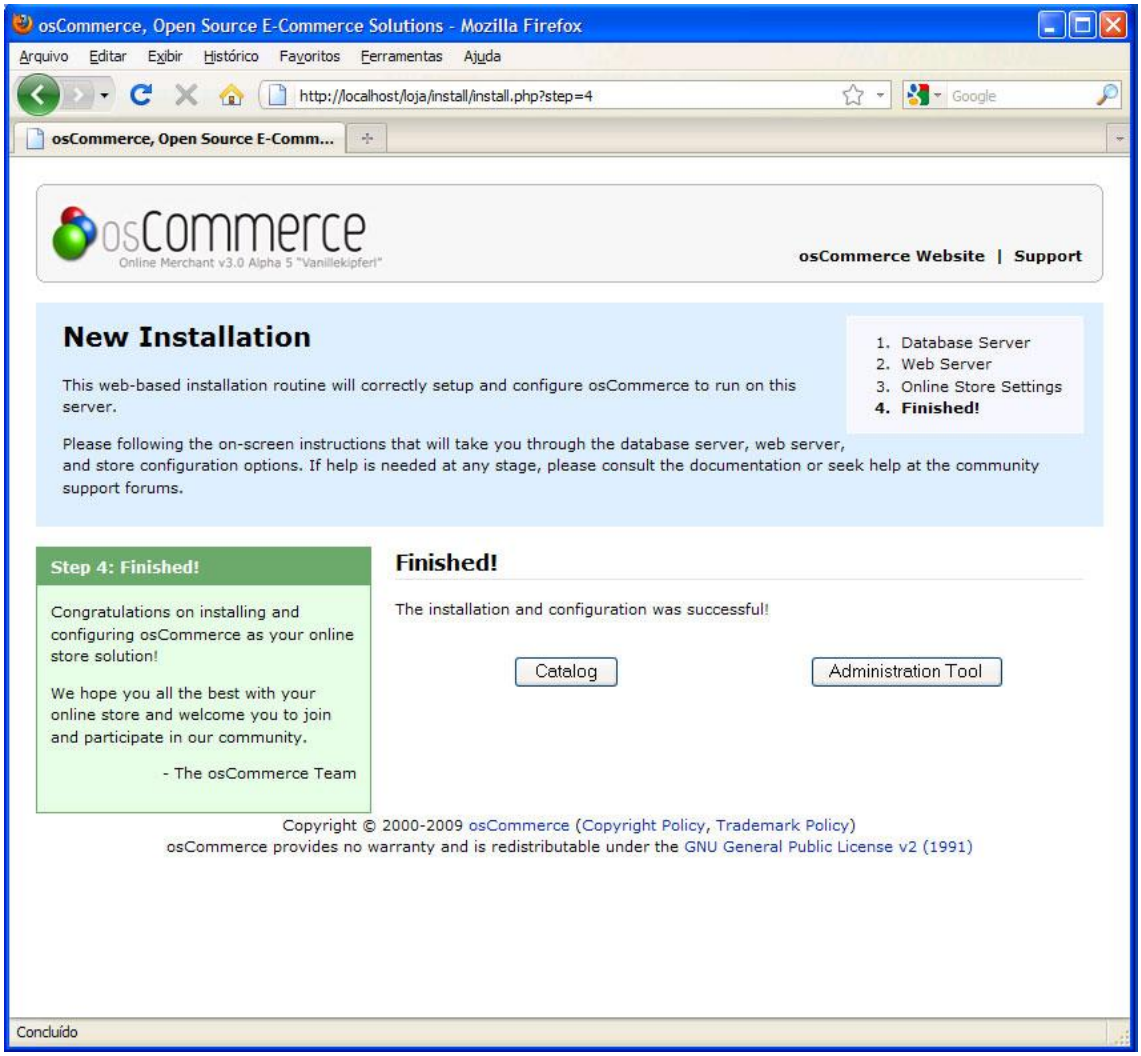



**Figura 19 – Término da Configuração da Ferramenta osCommerce**

7) Por questões de segurança o diretório "install" deve ser removido da estrutura de pastas e o arquivo "config.php" deve ser alterado para "somente leitura".

### 4.4.3. Selenium

Selenium é um framework livre de automação de casos de teste que auxilia, através de scripts, na execução de testes funcionais a partir da interface de uma aplicação web.

Funciona nos principais browser's, plataformas e linguagens, como iOS, LINUX e Microsoft, Opera, Safari, Firefox, Internet Explorer e Chrome, PHP, Rails, JAVA, .NET.

Foi criado pela ThoughtWorks, quando houve a necessidade de testar um sistema Web de rastreamento de despesas, mas hoje recebe contribuição de desenvolvimento do Google e outras grandes corporações.

O Selenium é dividido em três principais ferramentas, cada um contendo uma função específica para auxiliar no teste automatizado de sistemas web:

#### 4.4.3.1. Selenium IDE

É um add-on que fornece um ambiente de desenvolvimento integrado para a construção e execução de casos de teste, individuais ou múltiplos testes. O Selenium-IDE tem um recurso de gravação que armazena as ações do usuário em forma de um script.

#### 4.4.3.2. Como funciona o Selenium:

O Selenium grava as ações do usuário e gera um script que pode ser executado automaticamente. Com um conjunto de testes, é criado um Test Case, onde é possível que todos os testes sejam executado de forma sequencial.

É sempre aconselhável dividir as ações em pequenas etapas. Por exemplo, criar em um script o cadastro do usuário e no outro o login no sistema. Em alguns sistemas mais complexos as ações podem ser tratadas por uma única ação executada na tela.

Por ser uma ferramenta que funciona com base na interface do sistema, os resultados dos casos de testes devem ser mapeados com base em uma resposta na tela. Por exemplo, um clique no botão "OK", tem que resultar em uma mensagem "OK apertado com sucesso". Assim, é possível saber se o caso de teste foi executado com sucesso, bastando apenas registrar a asserção de "texto presente" "OK apertado com sucesso".

Ao rodar o script, o Selenium fará a ação "Apertar Botão OK" e esperará como resultado o texto "OK apertado com sucesso". Se o texto for mostrado na tela, o script apresentará um resultado sem erros. Caso não apareça, neste caso, será mostrado um erro.

#### 4.4.3.3. Instalações do Selenium IDE

Pré-requisito:

- Firefox 3.0;

Para instalar o Selenium-IDE, deve-se ir ao endereço http://seleniumhq.org/download/ e escolher a última versão do Selenium-IDE. Ao clicar no link, o abrirá uma janela de confirmação de instalação do plugin. Ao confirmar, ele será instalado automaticamente.

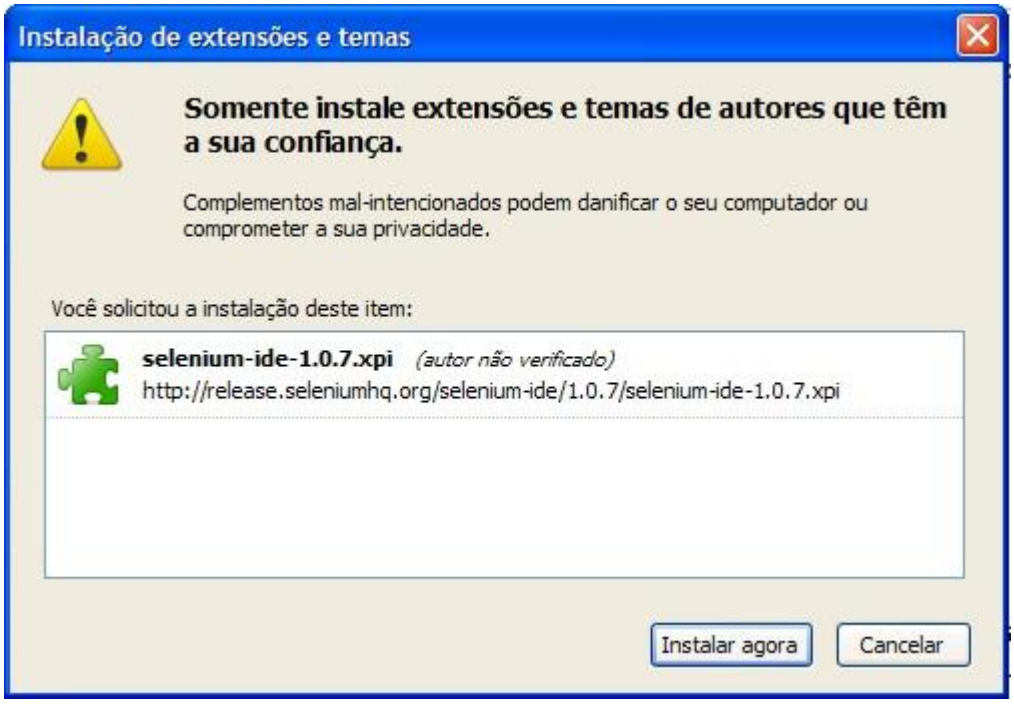



**Figura 20 – Instalação do Selenium**

Após a instalação será necessário reiniciar o Firefox.

Quando o Firefox for aberto novamente, no menu "Ferramentas" irá aparecer o item "Selenium IDE", que deverá ser executado para abrir a interface do Selenium.

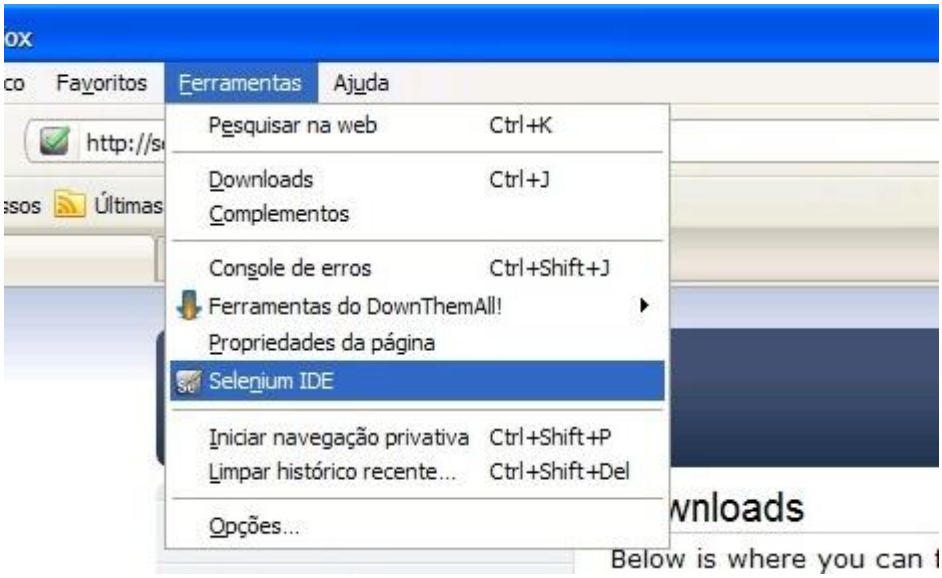



**Figura 21 – Execução do Selenium IDE**

Ao selecionar, teremos a janela do Selenium:

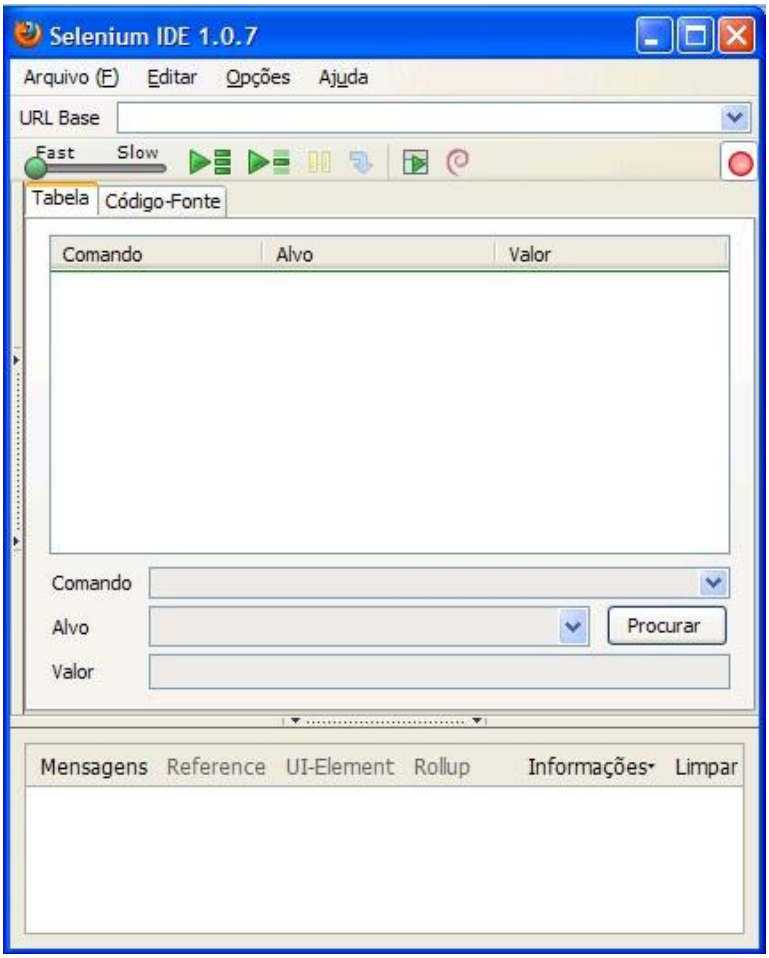



**Figura 22 – Execução do Selenium no Firefox**

#### 4.4.3.4. Exemplo de um Caso de Teste

**Caso de teste "login":**

a) Pré-condições:

- o usuário não poderá estar logado no sistema.

b) Ações do Usuário:

- clica no botão "My Account".
- preenche o campo "E-Mail Address" com informações de uma conta já cadastrada anteriormente.
- preenche o campo "Password" com a senha utilizada no cadastro.
- clica no botão "Sign In".

c) Ações do sistema:

- Vai para a tela da conta do usuário.

d) Resultado:

- usuário logado no sistema.

#### 4.4.3.5. Exemplo de um Caso de Teste Automatizado no Selenium

Para criar um caso de testes para o Selenium, é preciso adaptar o caso de teste "login" adicionando algumas asserções:

Validação Selenium:

- verifica a existência do texto: "View or change my account information".
- verifica o título da página: "Open Source: My Account".

Para criar o script do caso de teste "login", o Selenium-IDE deve ser aberto e iniciado o modo de gravação. Depois, devem ser seguidos os passos descritos no caso de teste.

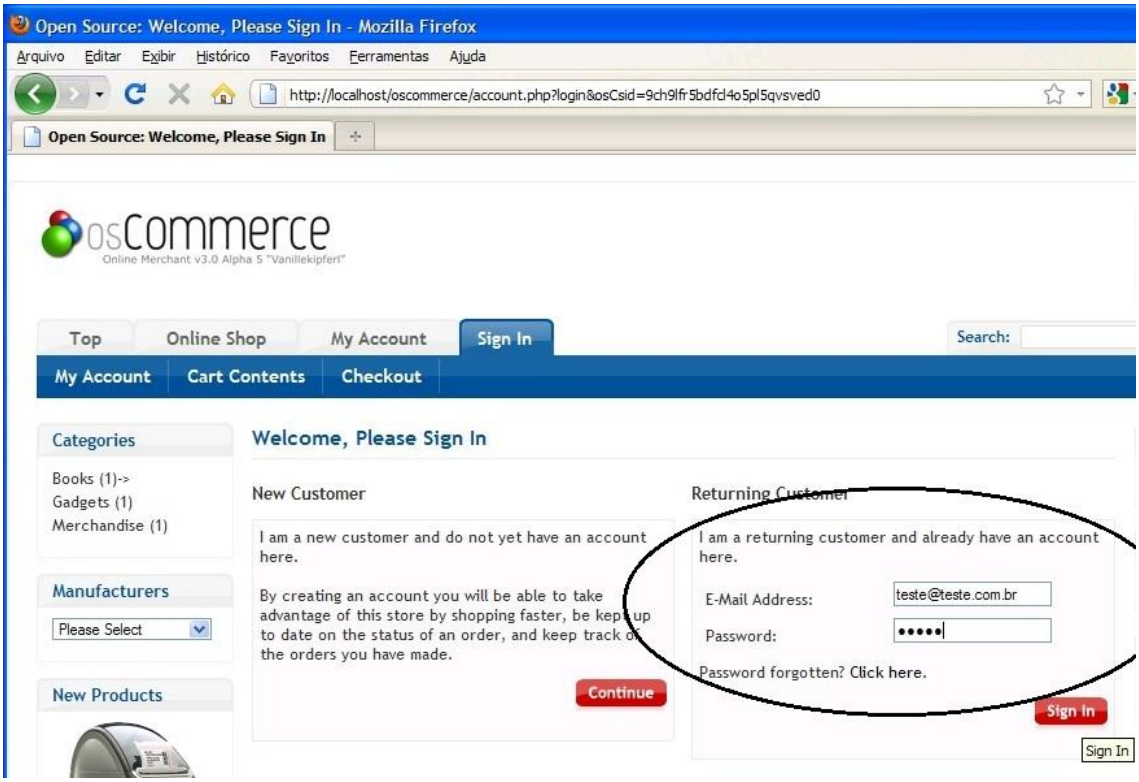



**Figura 23 – Login na Ferramenta osCommerce**

Na figura acima está destacada a interação do usuário, que é o preenchimento dos campos.

Adicionando as asserções:

As asserções são parâmetros adicionados para que o Selenium consiga testar a sua veracidade, no caso do Selenium, a presença do mesmo na tela.

Para adicionar uma asserção basta selecionar o texto, clicar com o botão direito em cima e selecionar a opção "assertTextPresent".

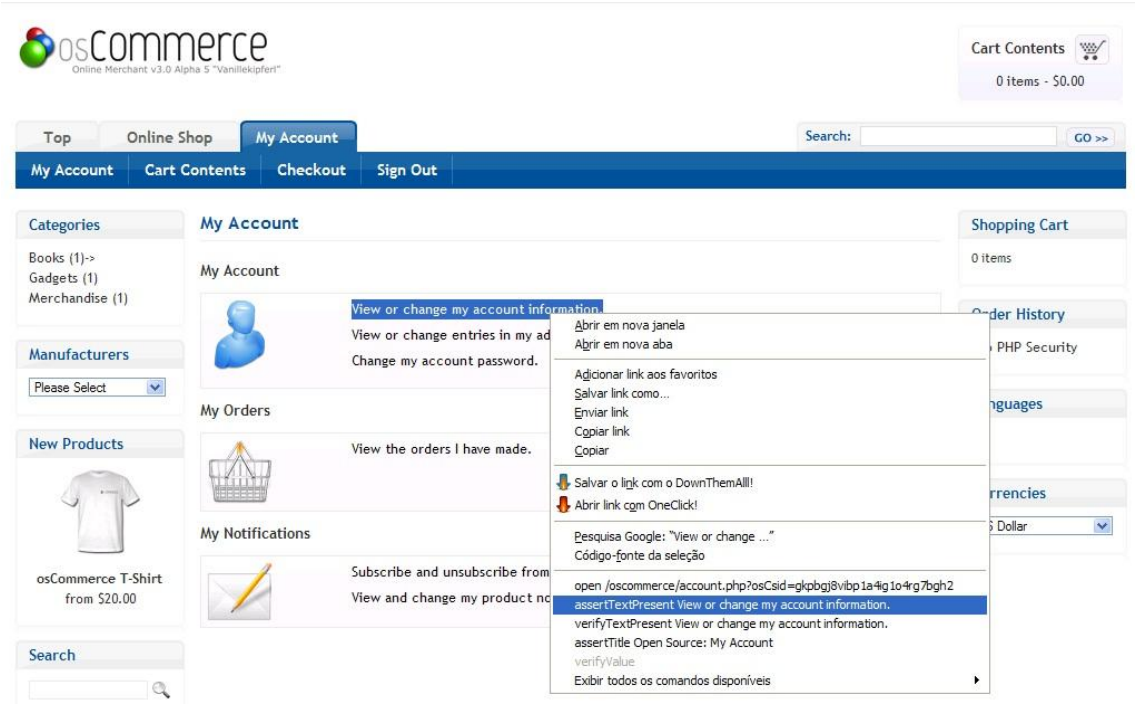



**Figura 24 – Execução da Ferramenta osCommerce**

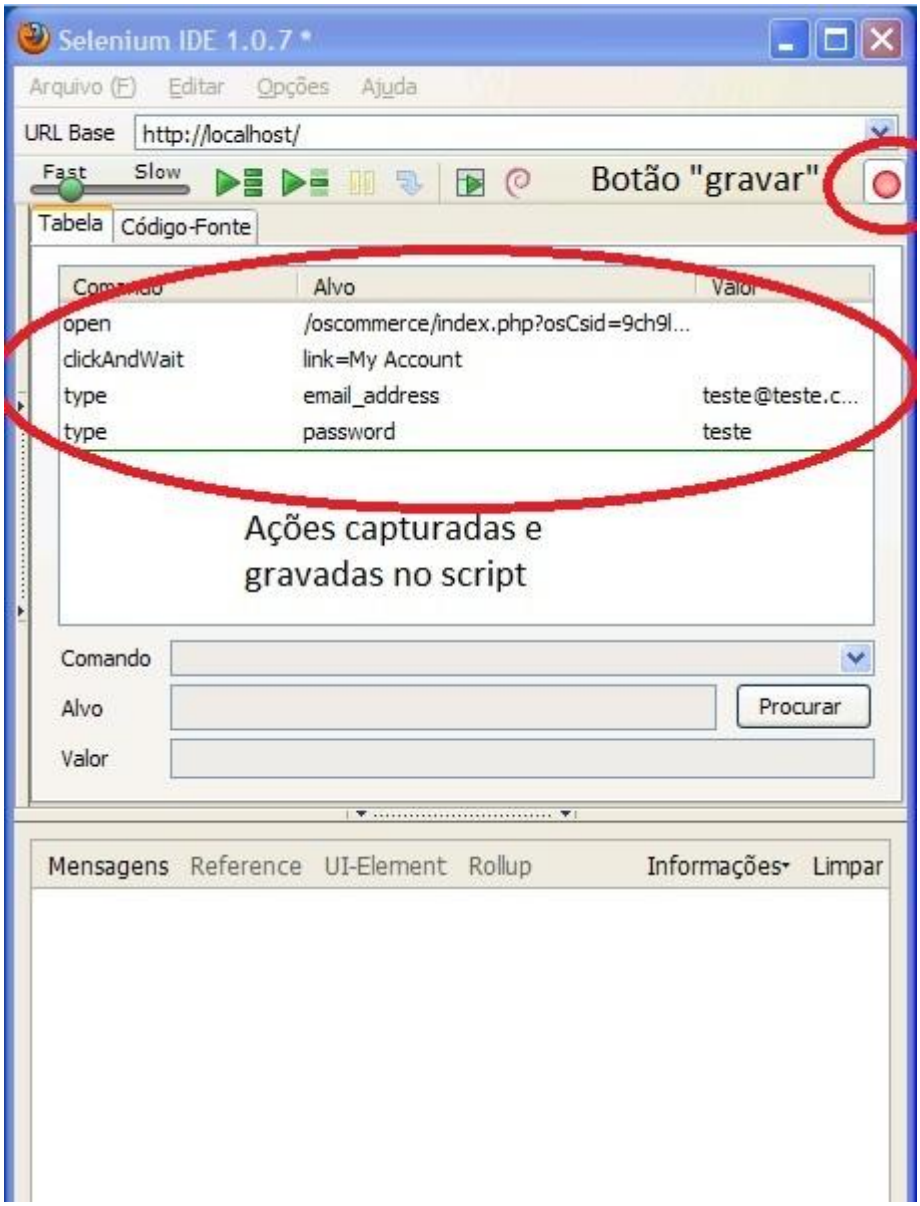



**Figura 25 – Execução do Selenium na Ferramenta osCommerce**

Na figura acima estão destacados o botão "gravar" e a lista de ações capturadas pelo Selenium.

Para salvar o teste, é preciso clicar em "Arquivo" e selecionar a opção "Salvar teste".

### Código-fonte gerado

O Selenium permite que seja exportado o código-fonte do script em diversas linguagens, para que seja utilizado no próprio sistema que está sendo testado: HTML, JAVA, C#, PERL, PHP, Pyton e Ruby.

A partir do script anteriormente criado foram gerados os seguintes códigos:

**HTML:**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head profile="http://selenium-ide.openqa.org/profiles/test-case">
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<link rel="selenium.base" href="" />
<title>login</title>
</head>
<body>
<table cellpadding="1" cellspacing="1" border="1">
<thead>
<tr><td rowspan="1" colspan="3">login</td></tr>
</thead><tbody>
<tr>
        <td>open</td>
        <td>/oscommerce/account.php?login&amp;osCsid=gkpbgj8vibp1a4ig1o4rg7bgh2</td
>
        <td></td>
</tr>
<tr>
        <td>click</td>
        <td>//div[@id='header-nav-lower']/ul/li[1]/a</td>
        <td></td>
</tr>
<tr>
        <td>type</td>
        <td>email_address</td>
        <td>teste@teste.com.br</td>
</tr>
```

```
<tr>
	<td>type</td>
	<td>password</td>
	<td>teste</td>
</tr>
<tr>
	<td>click</td>
	<td>//div[@id='main-column']/div[1]/form/div/p[3]/input</td>
	<td></td>
</tr>
<tr>
	<td>assertTextPresent</td>
	<td>View or change my account information.</td>
	<td></td>
</tr>
<tr>
	<td>assertTitle</td>
	<td>Open Source: My Account</td>
	<td></td>
</tr>
</tbody></table>
</body>
</html>
```

**JAVA (JUNIT)**

```
package com.example.tests;

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

public class login extends SeleneseTestCase {
	public void setUp() throws Exception {
```

```
        setUp("http://change-this-to-the-site-you-are-testing/", "*chrome");
    }
    public void testLogin() throws Exception {

    selenium.open("/oscommerce/account.php?login&osCsid=gkpbgj8vibp1a4ig1o4rg7bgh2");
        selenium.click("//div[@id='header-nav-lower']/ul/li[1]/a");
        selenium.type("email_address", "teste@teste.com.br");
        selenium.type("password", "teste");
        selenium.click("//div[@id='main-column']/div[1]/form/div/p[3]/input");
        assertTrue(selenium.isTextPresent("View or change my account information."));
        assertEquals("Open Source: My Account", selenium.getTitle());
    }
}
```

**C#**

```
using System;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading;
using NUnit.Framework;
using Selenium;

namespace SeleniumTests
{
    [TestFixture]
    public class login
    {
        private ISelenium selenium;
        private StringBuilder verificationErrors;

        [SetUp]
```

```
        public void SetupTest()
        {
            selenium = new DefaultSelenium("localhost", 4444, "*chrome",
"http://change-this-to-the-site-you-are-testing/");
            selenium.Start();
            verificationErrors = new StringBuilder();
        }

        [TearDown]
        public void TeardownTest()
        {
            try
            {
                selenium.Stop();
            }
            catch (Exception)
            {
                // Ignore errors if unable to close the browser
            }
            Assert.AreEqual("", verificationErrors.ToString());
        }

        [Test]
        public void TheLoginTest()
        {

    selenium.Open("/oscommerce/account.php?login&osCsid=gkpbgj8vibp1a4ig1o4rg7b
gh2");
            selenium.Click("//div[@id='header-nav-lower']/ul/li[1]/a");
            selenium.Type("email_address", "teste@teste.com.br");
            selenium.Type("password", "teste");
            selenium.Click("//div[@id='main-
column']/div[1]/form/div/p[3]/input");
```

```
                    Assert.IsTrue(selenium.IsTextPresent("View or change my account
information."));
                    Assert.AreEqual("Open Source: My Account", selenium.GetTitle());
                }
        }
}
```

**PERL**

```
use strict;
use warnings;
use Time::HiRes qw(sleep);
use Test::WWW::Selenium;
use Test::More "no_plan";
use Test::Exception;

my $sel = Test::WWW::Selenium->new( host => "localhost",
                    port => 4444,
                    browser => "*chrome",
                    browser_url => "http://change-this-to-the-site-you-are-testing/" );

$sel->open_ok("/oscommerce/account.php?login&osCsid=gkpbgj8vibp1a4ig1o4rg7bgh2");
$sel->click_ok("//div[\@id='header-nav-lower']/ul/li[1]/a");
$sel->type_ok("email_address", "teste\@teste.com.br");
$sel->type_ok("password", "teste");
$sel->click_ok("//div[\@id='main-column']/div[1]/form/div/p[3]/input");
$sel->is_text_present_ok("View or change my account information.");
$sel->title_is("Open Source: My Account");
```

**PHP**

```
<?php
```

```
require_once 'PHPUnit/Extensions/SeleniumTestCase.php';

class Example extends PHPUnit_Extensions_SeleniumTestCase
{
  protected function setUp()
  {
    $this->setBrowser("*chrome");
    $this->setBrowserUrl("http://change-this-to-the-site-you-are-testing/");
  }

  public function testMyTestCase()
  {
    $this->open("/oscommerce/account.php?login&osCsid=gkpbgj8vibp1a4ig1o4rg7bgh2");
    $this->click("//div[@id='header-nav-lower']/ul/li[1]/a");
    $this->type("email_address", "teste@teste.com.br");
    $this->type("password", "teste");
    $this->click("//div[@id='main-column']/div[1]/form/div/p[3]/input");
    $this->assertTrue($this->isTextPresent("View or change my account information."));
    $this->assertEquals("Open Source: My Account", $this->getTitle());
  }
}
?>
```

**Pyton**

```
from selenium import selenium
import unittest, time, re

class login(unittest.TestCase):
    def setUp(self):
        self.verificationErrors = []
        self.selenium = selenium("localhost", 4444, "*chrome", "http://change-this-to-the-site-
you-are-testing/")
```

```
        self.selenium.start()

    def test_login(self):
        sel = self.selenium
        sel.open("/oscommerce/account.php?login&osCsid=gkpbgj8vibp1a4ig1o4rg7bgh2")
        sel.click("//div[@id='header-nav-lower']/ul/li[1]/a")
        sel.type("email_address", "teste@teste.com.br")
        sel.type("password", "teste")
        sel.click("//div[@id='main-column']/div[1]/form/div/p[3]/input")
        self.failUnless(sel.is_text_present("View or change my account information."))
        self.assertEqual("Open Source: My Account", sel.get_title())

    def tearDown(self):
        self.selenium.stop()
        self.assertEqual([], self.verificationErrors)

if __name__ == "__main__":
    unittest.main()
```

**Ruby**

```
require "test/unit"
require "rubygems"
gem "selenium-client"
require "selenium/client"

class login < Test::Unit::TestCase

  def setup
    @verification_errors = []
    @selenium = Selenium::Client::Driver.new \
      :host => "localhost",
      :port => 4444,
```

```
    :browser => "*chrome",
    :url => "http://change-this-to-the-site-you-are-testing/",
    :timeout_in_second => 60

  @selenium.start_new_browser_session
 end

 def teardown
  @selenium.close_current_browser_session
  assert_equal [], @verification_errors
 end

 def test_login
  @selenium.open
"/oscommerce/account.php?login&osCsid=gkpbgj8vibp1a4ig1o4rg7bgh2"
  @selenium.click "//div[@id='header-nav-lower']/ul/li[1]/a"
  @selenium.type "email_address", "teste@teste.com.br"
  @selenium.type "password", "teste"
  @selenium.click "//div[@id='main-column']/div[1]/form/div/p[3]/input"
  assert @selenium.is_text_present("View or change my account information.")
  assert_equal "Open Source: My Account", @selenium.get_title
 end
end
```

**Testando o script**

Para testar o script deve-se abrir o teste salvo anteriormente, clicando em "Arquivo", opção "Abrir" e escolher o arquivo de teste.

Para executá-lo, deve-se clicar no ícone "Executar do começo".

O Selenium repetirá todos os passos do script gravado anteriormente e marcará com verde cada passo executado com sucesso e de vermelho caso alguma ação apresente problema.

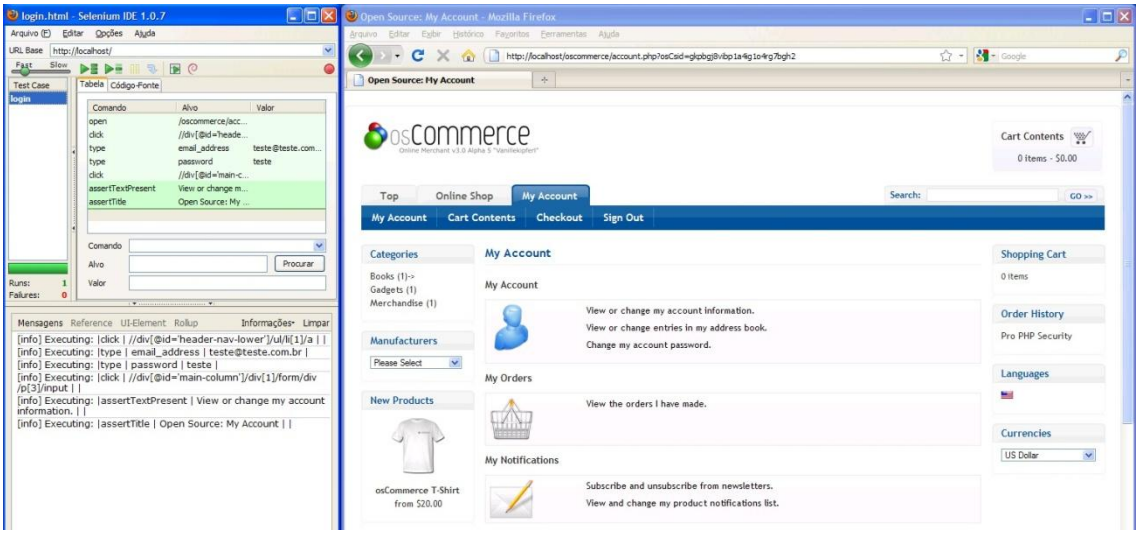



**Figura 26 – Execução do Test Case na Ferramenta osCommerce**

Na imagem acima, podemos ver a lista de ações em verde (todas executadas com sucesso) e à direita a tela "My Account", conforme o teste case gerado.

Caso não seja cumprido o pré-requisito "o usuário não poderá estar logado no sistema", o resultado do script será o seguinte:

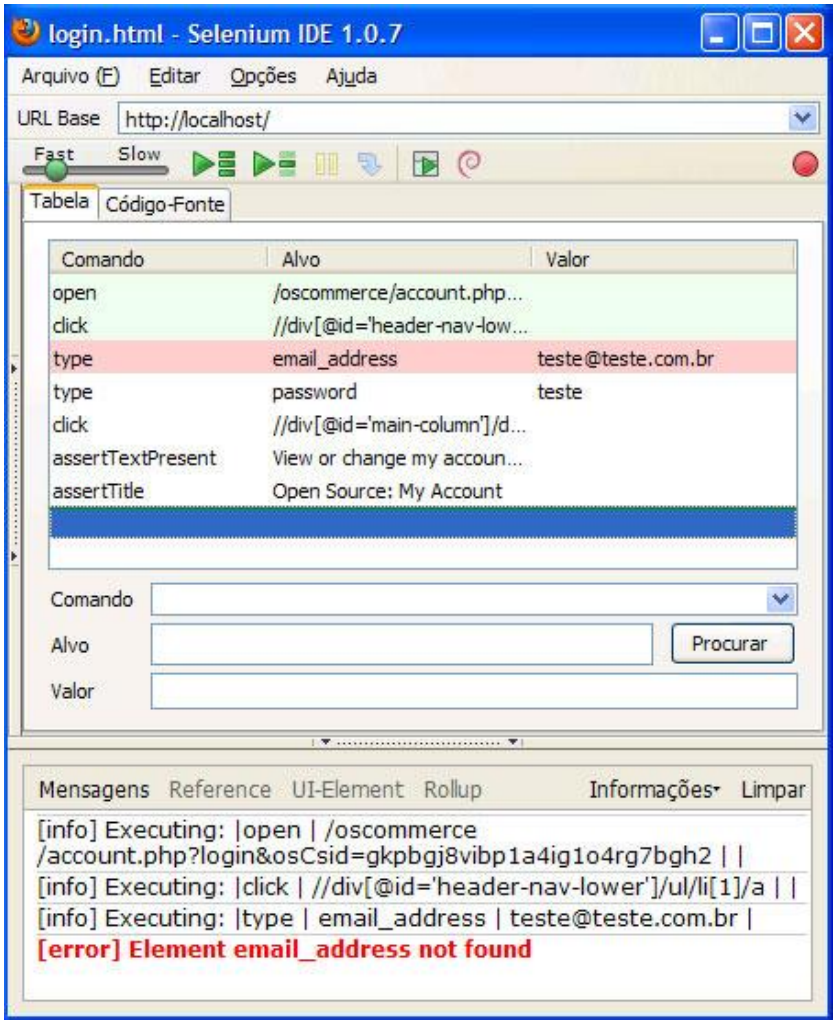



**Figura 27 – Resultado da Execução do Test Case na Ferramenta osCommerce**

O erro gerado foi causado pelo fato de que, quando executada a ação "clica no botão "My Account"", o campo "email_address" não apareceu, pois como o usuário já estava logado, a página solicitada mostrava as informações do usuário que efetuou o login.

### 4.5. Avaliação da Proposta

Para avaliar se o método proposto é adequado para a comunidade de software livre, é preciso não apenas estar de acordo com os princípios. É necessário também verificar sua validade junto à comunidade. Para tal foi criado um questionário que possa coletar as primeiras informações de *feedback* junto à comunidade de software livre.

O erro gerado foi causado pelo fato de que, quando executada a ação "clica no botão "My Account"", o campo "email_address" não apareceu, pois como o usuário já estava logado, a página solicitada mostrava as informações do usuário que efetuou o login.

# 5. Conclusão

Para aplicação do modelo proposto no objeto deste trabalho, é fundamental que haja uma forte partição tanto da comunidade de Testes de Software como dos próprios colaboradores (editores e criadores), estes últimos que promoveriam a adesão dos interessados em perpetuar a qualidade do software em questão.

Visando entender melhor qual a lógica proposta no decorrer deste projeto, é importante levar em consideração um importante ponto falho no modelo atual:

- Segundo a citação "Given enough eyeballs, all bugs are shallow." [RAYMOND], é importante levar em consideração que existe sim uma enorme gama de testadores naturais em cada software, isso é evidente. O ponto falho está justamente no tempo que esse software fica disponível para uso (ou segundo o modelo tradicional, em produção). Os usuários precisam escolher o Software Livre, implantá-lo na sua empresa para depois de implantado e em uso, descobrir que possuem falhas nas suas funções vitais, de modo que não consigam proceder o uso daquele software. Gerando um desconforto e uma desconfiança por parte dos usuários tradicionais nesses modelos de softwares propostos.

Os mais conservadores, ainda não optam por utilizar os softwares livres por questões de confiabilidade nas aplicações utilizadas. Somente os softwares com maior credibilidade no mercado como o BROffice, LINUX, Apache entre outros que possuem tal credibilidade devido ao grande número de usuários e colaboradores.

Todavia, o modelo proposto, sugere que:

- A execução dos testes sejam mais rápidas (pelo fato dos testes serem automatizados, somente o tempo de criação dos scripts é demorado);

- As falhas no software são encontradas antes deles serem colocados em produção;

- Maior confiabilidade no software, uma vez que, a cada nova versão, o desenvolvedor executaria os testes automatizados antes de fechar o pacote e disponibilizando este para a comunidade;

- Utilizando relatórios de execução dos casos de teste automatizados, pode haver uma espécie de certificado de garantia de qualidade dos softwares que possuem pelo menos parte dos seus casos de teste automatizados;

Evitando que essa proposta seja taxada como impossível, impraticável ou quaisquer outros adjetivos pejorativos, é importante levar em consideração que ter 100% dos casos de teste automatizados em qualquer projeto, na prática, é utopia. Todavia, qualquer iniciativa que visa melhorar a qualidade dos softwares desenvolvidos no contexto de Software Livre, é uma iniciativa bem vinda.

## REFERÊNCIAS


**[**MYERS**]**, Glenford. **J. The Art of Software Testing.** Wiley; 3 edition (November 8, 2011)

[GRAHAM], Dorothy; BLACK, Rex; EVANS, Isabel; VEENENDAL, Erik Van. **Foundations of Software Testing: ISTQB Certification. Intl Thomson Business Pr;** Revised edition (January 28, 2008)

[SPILLNER], Andreas; LINZ, Tilo; SCHAEFER, Hans. **Software Testing Foundations: A Study Guide for the Certified Tester Exam.** Rocky Nook; 3 edition (February 4, 2011)

[BLACK], Rex. **Advanced Software Testing - Vol. 1: Guide to the ISTQB Advanced Certification as an Advanced Test Analyst.** Rocky Nook; 1 edition (October 15, 2008)

**[**PRESSMAN**]**, Roger S. **Engenharia de Software.** McGraw-Hill; 6ª edição (2006)

[BEIZER], Boris. **Software Testing Techiniques. Hardcover, subsequent edition (1990).**

[BINDER], Robert V. **Testing Object-Oriented Systems: Models, Patterns, and Tools.** Addison-Wesley Professional (November 7, 1999)

[STALLMAN] Richard M., Joshua Gay, Lawrence Lessig. **Free Software, Free Society: Selected Essays of Richard M. Stallman [Paperback].** Createspace (30 Dec 2009)

[LÉVY] Wiseman, J Goron. **Guidelines for the use of personal data in system testing [Paperback].** BSI British Standards Institution; 1 edition (22 Sep 2003)

[LEMOS] R. De, A. Saeed, T. Anderson. **Requirements Specification and Verification for Safety- Critical Systems: a Train Set Example (Technical report series) [Paperback].** University of Newcastle upon Tyne, Computing Laboratory (1991)

[RAYMOND] McLeod Jr., Gerald D. Everett. **Software Testing: Testing Across the Entire Software Development Life Cycle [Hardcover].** Wiley-Blackwell (7 Aug 2007)

[HUMPHREY] Watts S. **Introduction to the Team Software Process (SEI Series in Software Engineering) [Hardcover].** Addison Wesley; 1 edition (24 Aug 1999)

[BHAT], Thirumalesh; Nagappan, Nachiappan. Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies. Table 3: Project A- Outcome Measures, p. 05

[1] http://www.qualitytest.com.br/teste_de_software.php Acessado em 05/08/2009

[2] http://pt.wikipedia.org/wiki/Software_livre Acessado em 05/08/2009

[3 http://www.softwarelivre.gov.br/ Acessado em 05/08/2009

[4] http://pt.wikibooks.org/wiki/Software_Livre/Hist%C3%B3ria_do_Software_Livre Acessado em 05/08/2009

[5] http://en.wikipedia.org/wiki/Linus%27_Law Acessado em 07/08/2009

[6] http://catb.org/esr/writings/cathedral-bazaar/cathedral-bazaar/ar01s04.html Acessado em 07/08/2009

[7] http://www.devmedia.com.br/articles/viewcomp.asp?comp=8035 Acessado em 27/11/2009

[8] http://www.neodream.com.br/5cqualibr/flash/ Acessado em 27/11/2009

[9] http://www.softwarepublico.gov.br/5cqualibr/ Acessado em 27/11/2009

[10] http://www.softwarepublico.gov.br/ Acessado em 27/11/2009

[11] http://www.softwarelivre.gov.br/ Acessado em 27/11/2009

[12] http://softwarelivre.org/ Acessado em 27/11/2009

[13] Revista InfoBrasil. Ano II. No 7

[14] http://andrefaria.com/2008/07/12/test-driven-development/ Acessado em 27/11/2009

[15] http://www.cem.itesm.mx/dacs/publicaciones/logos/anteriores/n53/cshwingel.html Acessado em 29/11/2009

[16] http://pt.wikipedia.org/wiki/Test_Driven_Development Acessado em 02/12/2009

[17] http://pt.wikipedia.org/wiki/Modelo_V Acessado em 03/05/2010

[18] http://pt.wikipedia.org/wiki/Teste_de_unidade Acessado em 03/05/2010

[19] http://www.ic.unicamp.br/~eliane/Cursos/SeminarioTestes/Teste_Regressao.ppt Acessado em 03/05/2010

[20] http://www.devmedia.com.br/articles/viewcomp.asp?comp=10025

[21] http://research.microsoft.com/en-us/projects/esm/nagappan_tdd.pdf

[22] http://www.agile2007.org/downloads/proceedings/006_On the Sustained Use_860.pdf

[23] http://www.rsinet.com.br/?q=node/20

[24] http://seleniumhq.org/ Acessado em 03/10/2010

[25] http://pt.wikipedia.org/wiki/Projeto_GNU Acessado em 12/10/2010

[26] http://oscommerce.com/ Acessado em 16/10/2010

[27] http://www.ibm.com/developerworks/br/opensource/library/os-community/ Acessado em 18/10/2010

[28] http://pt.wikipedia.org/wiki/Automação_de_teste Acessado em 19/10/2010

[29] http://br-linux.org/faq-softwarelivre/ Acessado em 19/05/2011

[30] http://testavo.blogspot.com/ Acessado em 20/05/2011

[31] http://www.testexpert.com.br/ Acessado em 21/05/2011

[32] http://sembugs.blogspot.com/ Acessado em 21/05/2011

[33] http://qualidadebr.wordpress.com/ Acessado em 23/05/2011

[34] http://www.york.ac.uk/depts/maths/tables/orthogonal.htm Acessado em 23/05/2011

# APÊNDICE A – Ciclo de Vida do Processo de Testes

## 1. Descrição das atividades

Cada atividade definida no processo, em determinada fase, será descrita da seguinte forma:

1) Nome da atividade
2) Objetivo
3) Artefatos de entrada
4) Artefatos de saída
5) Tarefas a serem realizadas em cada atividade
6) Atores envolvidos na atividade e suas responsabilidades
7) Referências a padrões de documentação e construção que devem ser utilizados na realização das tarefas, quando houver.
8) Referências a documentos de procedimentos e orientações relacionados à execução de algumas tarefas, quando houver.

É importante observar que, como os projetos estão todos relacionados e têm como objetivo comum a informatização de atividades em projetos de desenvolvimento de software padrão. As atividades definidas nas próximas seções são fruto da pesquisa das referências de inúmeros autores (dentre eles Myers e Beizer) consagrados no mercado de Testes de Software.

### 1.1. Fase de planejamento

Depois de realizadas as atividades de levantamento de necessidade e planejamento do projeto, é quando as atividades de testes têm início. Assim que o documento de requisitos é definido e revisado, é criado um documento de Estratégia de Testes, identificando as principais necessidades do sistema, quais características funcionais e não funcionais do

software devem ser atendidas, ou seja, como o software deve ser testado. Com base no plano de projeto, já refinado, é criado um documento de Plano de Testes, detalhando as atividades de testes no decorrer do projeto, ou seja, o que deve ser testado. Com base neste planejamento é feita uma estimativa inicial dos recursos necessários ao projeto e um cronograma inicial é elaborado.

Para facilitar o entendimento, na utilização dos nomes e a documentação sugerida para as metodologias mais utilizadas no mercado (modelo cascata, RUP, SCRUM, XP, etc.), é importante levar em consideração que os itens sublinhados (como, o que) são a forma mais simples de perceber que a criação dos documentos é mera formalidade. O que realmente importa para a equipe de testes num projeto em específico é fornecer um direcionamento das atividades, pois é nessa etapa onde as atividades são definidas.

Esta fase é executada uma única vez e suas saídas são revisadas a cada nova iteração durante a execução do projeto. As fases seguintes de preparação e execução são realizadas para cada iteração planejada.

### 1.1.1. Atividade: Elaborar a Estratégia de Testes

**1) Objetivo**

Delimitar o escopo do sistema a ser testado

**2) Artefatos de entrada padrão**

Declaração de escopo

Documento de definição de requisitos funcional e não funcional aprovado pelo solicitante.

Documentação referente aos projetos/sistemas relacionados

**3) Artefatos de saída padrão**

Documento de Estratégia de Testes

**4) Tarefas a serem realizadas**

a) Definir a missão e o objetivo dos testes.

b) Fases de teste envolvidas: Testes unitários, de integração, de desempenho, de sistema, de aceitação, de regressão, etc.

**5) Atores envolvidos na atividade e suas responsabilidades**

Responsável pelas atividades de teste - executa

Gerente do projeto – acompanha

Analista de Sistemas – acompanha

**6) Procedimentos e Orientações**

Definir a missão do projeto está geralmente associada a definição do objetivo do mesmo. Pode-se declarar simplesmente "achar o maior número de defeitos", bem como pode ser "promover a integração com o sistema legado" ou "dar prioridade aos módulos de maior risco para o projeto", etc. Pode estar explícito bem como pode precisar de um maior entendimento do negócio, isso dependerá de cada projeto.

O objetivo do teste está intimamente associado às atividades que serão executadas no decorrer do projeto de testes, como por exemplo, casos de teste baseados em risco, baseados na especificação, ou até, não criar casos de teste, somente executar testes exploratórios, tudo depende.

### 1.1.2. Atividade: Revisão da documentação

**1) Objetivo**

Refinar o documento e nivelar o entendimento do deste pelos demais membros da equipe

**2) Artefatos de entrada padrão**

Quaisquer documentos que exijam um maior nível de detalhamento ou entendimento

**3) Artefatos de saída padrão**

Documento em questão refinado

Ata da reunião de revisão

**4) Tarefas a serem realizadas**

a) Planejamento: Autor seleciona os participantes e define funções (descritas no item 5)

b) *Kick-off*: Distribuição dos documentos e explicação dos objetivos, processos e documentos.

c) Preparação individual: Cada participante revisa a documentação, levanta possíveis dúvidas ou comentários.

d) Reunião de revisão: Moderador conduz a reunião de revisão. Revisores levantam suas dúvidas durante a reunião, tem uma breve explicação. Gravador anota todas as dúvidas e comentários para serem corrigidos no documento original.

e) Retrabalho: Autor do documento corrige o documento de forma que ele fique mais claro para os demais participantes e envolvidos no projeto.

f) *Follow-up*: Atividades relacionadas a finalização dessa revisão (adicionar ao repositório de dados, enviar por email, etc.)

**5) Atores envolvidos na atividade e suas responsabilidades**

Gerente - acompanha

Autor – autor do documento e responsável pelo planejamento e kick-off

Moderador – conduz a reunião

Revisores – quaisquer envolvidos no projeto

Gravador – anota as dúvidas, sugestões e alterações do documento na reunião

**6) Procedimentos e Orientações**

O objetivo de cada revisão deve ser pré-definido

As pessoas corretas devem estar envolvidas

Defeitos e incoerências encontrados são bem vindos.

Tornar esse processo uma experiência positiva para o autor (melhora da documentação)

### 1.1.3. Atividade: Elaborar Plano de Testes

**1) Objetivo**

Elaborar o Plano de Testes identificando as atividades de testes relacionadas àquele projeto. Define com e o que deve ser testado no projeto.

**2) Artefatos de entrada padrão**

Estratégia de Testes

Plano do projeto

Documento de definição de requisitos funcional e não funcional aprovado pelo solicitante.

Documentação referente aos projetos/sistemas relacionados

**3) Artefatos de saída padrão**

Plano de testes

**4) Tarefas a serem realizadas**

a) Itens de Teste: Especificação de requerimentos, modelagem de dados, esquema do banco de dados, etc.

b) Funcionalidades a serem testadas: É o que deve ser testado pela perspectiva do usuário.

c) Funcionalidades que não devem ser testadas: Pode ser postergado para a próxima iteração, baixo risco (já pode ter sido testada anteriormente e considerada estável), etc.

d) Escopo:

a. Uso de alguma ferramenta especial?
b. Como as métricas vão ser coletadas?
c. Como o ambiente de testes deve ser configurado?
d. Software? Hardware? Combinação de SW, HW e software de terceiros?
e. Quais serão as regras para testes de regressão?

e) Critérios de sucesso/falha: Número mínimo de defeitos encontrados, ou defeitos de alta severidade, execução de todos os casos de teste, etc.

f) Ambiente de Testes: VPN? Dados para teste? Ferramentas? Capacidade de hardware? Restrição de uso durante os testes?

g) Responsabilidades: Responsáveis por modelar os testes, por avaliar os riscos da aplicação, por avaliar quais são as funcionalidades que devem e as que não devem ser testadas.

Observação: Boa prática é selecionar por função na empresa e não por nome.

h) Necessidade de pessoal ou treinamento: Treinamento num produto ou ferramenta específica, necessidade de um profissional com conhecimento específico

i) Cronograma: Definição das atividades do projeto

j) Riscos e contingências: Explicar o que deve ser feito caso , por exemplo, alguma das situações venham a acontecer:

a. Falta de ferramentas específicas;
b. Entrega tardia da aplicação para testes;
c. Mudança nos requerimentos originais, etc.

**5) Atores envolvidos na atividade e suas responsabilidades**

Analista de Testes – executa

Analista de Sistemas – acompanha e aprova

Gerente do projeto- aprova

## 1.2. Fase de preparação

Nesta fase, para cada nova funcionalidade ou alteração numa já existente, essas etapas devem ser seguidas para que o projeto de testes tenha êxito, levando em consideração a constante revisão desses itens.

**Preparação dos casos de teste**

A preparação dos casos de teste envolve as atividades de refinamento dos itens levantados no Plano de Testes, elaboração destes com o mesmo nível de detalhamento definido nas etapas de análise referentes a esta. Definição do objetivo do teste, pré-condições, dados de teste e pós-condições, idealmente o nível de detalhamento dos documentos de

análise devem permitir que o responsável crie os casos de teste baseando-se numa execução passo a passo do documento de casos de teste.

Os casos de teste devem ser definidos de modo que sejam suficientes para a execução do mesmo. Evitando que o usuário tenha que acessar outros documentos para obter êxito nas suas atividades de teste. Caso seja necessário, deve-se referenciar outros documentos durante a criação do caso de teste, mas sempre levando em consideração que esta referência pode se perder num dado espaço de tempo.

**Preparação do ambiente de testes**

São as atividades de preparação do ambiente de testes, estas se resumem a manter um ambiente estável e que, idealmente, simule o ambiente de produção. É nessa etapa onde os dados de teste são criados para que não atrase a execução.

O marco final da fase de desenvolvimento é o ambiente estável, carregado com os dados necessários e homologado para dar início às atividades de execução dos casos de teste.

### 1.2.1. Atividade: Elaborar Casos de Teste

**1) Objetivo**

Garantir que a funcionalidade atende aos requisitos.

**2) Artefatos de entrada padrão**

Todos os artefatos de requisitos

Todos os artefatos da especificação técnica

Documento de arquitetura

Documento de Plano de Testes

Documento de casos de uso

Outras aplicações ou partes da aplicação já construídas que são necessárias para a criação dos testes.

**3) Artefatos de saída padrão**

Roteiros de teste criados na ferramenta de Gestão de Testes (Testlink)

**4) Tarefas a serem realizadas**

a) Refinamento dos itens de teste definidos no Plano de Testes

b) Detalhamento dos casos de uso

c) Definir o sumário de cada roteiro de teste

a. Resumo: Explicação sucinta sobre o objetivo do roteiro.

Exemplo: Cadastrar usuários com idade entre 18 e 25 anos no sistema acadêmico

b. Pré-condições: Condições em que o sistema deve se encontrar ao iniciar o caso de teste.

Exemplo: Executar um caso de teste anteriormente, ter um usuário pré-cadastrado no sistema (se esse roteiro tiver por objetivo, por exemplo, a edição dos dados de um usuário), etc.

c. Dados de teste: Todos os dados que devem estar pré-carregados no sistema para dar início à execução do caso de teste.

Exemplo: Usuário com 18 anos, terceiro grau incompleto, renda entre 1 e 3 salários mínimos.

d. Pós-condições: Situação em que o sistema deve se encontrar ao finalizar o caso de teste.

Exemplo: Habilitação do cadastro de usuários entre 26 e 40 anos.

d) Descrever os passos (bem como os resultados esperados para cada passo) que o usuário deve seguir para que tenha êxito na execução do roteiro.

Observação: Ver na seção 4.1.3 do Apêndice um exemplo de caso de uso e como detalhar um caso de teste.

e) Aplicar as técnicas de modelagem de casos de teste

**5) Atores envolvidos na atividade e suas responsabilidades**

Analista de Testes – executa

Analista de Sistemas – acompanha e aprova

Gerente do projeto – aprova

**6) Procedimentos e Orientações**

O objetivo do caso de teste é achar *bugs* (embora muitos acreditem que o objetivo é validar os fluxos)

Utilização de uma ferramenta de Gestão de Casos de Teste para facilitar o acesso e manutenção: *Testlink*

Manter um padrão:

- Identificador único;
- Padrão de texto, de botões, campos, identificadores, etc.
- Repetir passos anteriores para analisar outros caminhos/opções

### 1.2.2. Atividade: Preparar o ambiente de testes

**1) Objetivo**

Oferecer um ambiente com infra-estrutura similar ao de produção, com bases de dados reduzidas, descaracterizadas e íntegras, utilizando processos automáticos de carga e validação de informações. O resultado é a realização de testes integrados e com total visualização dos processos de negócio da empresa.

**2) Artefatos de entrada padrão**

Plano de testes

Sistema em desenvolvimento

Acesso ao Banco de Dados

Quaisquer outras ferramentas periféricas ao sistema que necessitem estar configuradas/preparadas para a execução dos casos de teste

**3) Artefatos de saída padrão**

Sistema pronto para ser testado

**4) Tarefas a serem realizadas**

a) Preparar a infra-estrutura tecnológica do ambiente de testes e, através de ferramentas, controlar a utilização de seus recursos.

b) Estabelecer infra-estrutura tecnológica de hardware, software e *testware*.

c) Definir o tipo de ambiente (se é sistema por demanda ou ativo)

d) Definir as políticas, normas e padrões do ambiente e dos processos.

e) Estabelecer as regras e critérios para criação da Massa de Dados de Teste.

a. Se os dados vão ser carregados manualmente

b. Se os dados vão ser carregados automaticamente (através de uma ferramenta de automação dos dados do ambiente)

c. Se os dados vão ser carregados diretamente no Banco de Dados (verificar se estes precisam estar associados a Regras de Negócio interdependentes ou se podem ser simplesmente adicionados sem afetar a estrutura do ambiente).

**5) Atores envolvidos na atividade e suas responsabilidades**

Administrador do Banco de Dados – executa

Analista de Testes – executa

Desenvolvedor – acompanha

Gerente – acompanha

**6) Procedimentos e Orientações**

Deve, idealmente, haver um ambiente exclusivo e dedicado para testes.

**Definição de requisitos de infra-estrutura:**

*Hardware*: CPU e armazenamento

*Software*: Básico e produtos

*Testware*: Recursos humanos com especialização e foco em testes, e ferramentas de gerenciamento, documentação, automação, produtividade e geração de massa de testes.

Segurança: Segregação física ou lógica

Metodologia de teste: Processos estruturados

**Tipo de ambiente:**

Ativo: permanentemente disponível; sincronismo com a produção; processamento diário; controle rígido dos acessos, testes integrados.

Por demanda: gerado pela necessidade; sem sincronismo; execução sob demanda; flexibilidade de acessos; testes integrados.

**Normatização:**

Definição de normas de controle de configuração e versão, gestão de mudanças e problemas.

Definição de padrões de Nomenclaturas e Codificação (identificador de ambientes, estruturas de processamento e automação de processamento).

**Massa de dados de teste:**

Definição da origem dos dados: laboratório, produção ou simulado

Definição das ferramentas de produtividade

## 1.3. Fase de execução

Esta fase se resume à execução dos casos de teste. Nesta fase o Analista de Testes deve seguir os casos de teste definidos na fase anterior seguindo os objetivos pré-definidos na Estratégia dos Testes. Principalmente durante a execução dos casos de teste é que deve haver um controle sobre a execução dos casos de teste. Nesta fase é onde os defeitos devem ser reportados ao desenvolvedor. Os retestes também devem ser previstos nesta fase.

O final desta fase é quando o sistema é encaminhado estável (com os defeitos reportados já corrigidos) para o Analista de Sistemas efetuar a homologação.

### 1.3.1. Atividade: Executar Casos de Teste

**1) Objetivo**

Manter um nível de qualidade no sistema desenvolvido.

**2) Artefatos de entrada padrão**

Casos de teste

Protótipo de telas

Quaisquer artefatos que possam ajudar na execução dos casos de teste

**3) Artefatos de saída padrão**

Sistema testado, homologado e aprovado pelo Analista de Testes

**4) Tarefas a serem realizadas**

a) Executar os casos de teste

b) Reportar ao desenvolvedor os defeitos encontrados

**5) Atores envolvidos na atividade e suas responsabilidades**

Analista de Testes – executa

Gerente – acompanha

**6) Procedimentos e Orientações**

Deve estar sincronizado com as atividades de desenvolvimento

Assim que houver uma entrega de uma aplicação do sistema, os testes devem ser aplicados.

Utilização de técnicas de testes funcionais para melhor abrangência dos mesmos (ver seção 8 deste documento).

Registrar os resultados reais e comparar com os resultados esperados para verificar se o sistema está de acordo com o planejado.

### 1.3.2. Atividade: Reportar Defeitos

**1) Objetivo**

Melhorar o nível de comunicação entre o desenvolvedor e o analista de testes.

**2) Artefatos de entrada padrão**

Casos de teste

**3) Artefatos de saída padrão**

Ticket gerado no Mantis

**4) Tarefas a serem realizadas**

Durante a execução dos casos de teste deve-se reportar um defeito quando este for encontrado.

**5) Atores envolvidos na atividade e suas responsabilidades**

Analista de Testes – reporta

Desenvolvedor – corrige

Gerente – acompanha

**6) Procedimentos e Orientações**

Encontre o defeito e reproduza-o

Reporte a reprodução fiel do defeito encontrado

Para um sistema, um erro só é um erro quando pode ser reproduzido

Não utilizar termos ofensivos ao reportar um defeito

## APÊNDICE B – Criação de um Caso de Teste a partir de um Caso de Uso

Na criação dos casos de testes, o responsável deve aplicar as técnicas de modelagem de casos de teste (descritas na seção 8 deste documento) para ter uma maior cobertura do código com o mínimo de esforço durante a execução dos mesmos.

**Exemplo de Caso de Uso:**

| **Fluxo normal** | **Fluxos alternativos** |
|---|---|
| 1. Cliente coloca um ou mais itens no carrinho de compras<br>2. Cliente seleciona *checkout*<br>3. Sistema coleta endereço, informações de pagamento e de envio do cliente<br>4. Sistema apresenta todas as informações do usuário<br>5. Usuário confirma a ordem de pedido para o envio do Sistema | 1. Cliente seleciona *checkout* com o carrinho de compras vazio; Sistema retorna mensagem de erro<br>2. Cliente provê endereço, informações de pagamento ou de envio do cliente inválidos<br>3. Sistema abandona a transação antes ou durante o *checkout*; Sistema executa *logout* do cliente depois de 10 minutos inativo |

**Exemplo de Caso de Teste (baseado no Caso de Uso da seção– 4.1.3.1):**

Este Caso de Teste deverá ser inserido na ferramenta de Gestão de Testes no momento em que for modelar o mesmo.

| Passos | Resultado esperado |
|---|---|
| Passo 1 – Colocar um item no carrinho | Passo 1 – Item no carrinho |
| Passo 2 – Clicar em <Checkout> | Passo 2 – Tela de *checkout* |
| Passo 3 – Colocar valores válidos para endereço, utilize Mastercard para pagamento e método de entrega válido e clicar em <Enviar> | Passo 3 – A tela aparece corretamente e os inputs válidos são aceitos |
| Passo 4 – Verificar a ordem de envio | Passo 4 – Aparece como digitado na tela |
| Passo 5 – Confirmar a ordem de envio | Passo 5 – A ordem de envio está cadastrada corretamente no sistema |
| Passo 6 – Repetir os passos 1-5 mas colocar 2 itens no carrinho e escolher VISA como forma de pagamento | Passo 6 – Conforme feito nos passos 1-5 |

# APÊNDICE C – Técnicas de Modelagem de Casos de Teste

**Para que os casos de teste automatizados pela ferramenta possam ter uma maior cobertura do código com o mínimo esforço, é necessário que sejam aplicadas algumas técnicas durante a criação do *script* de testes. Esse script é criado a partir de um software conhecido como *Selenium* (http://seleniumhq.org/), sendo este, uma ferramenta livre do grupo Apache.**

Algumas dessas técnicas como Análise de Extremidades, Partição de Equivalência, devem ser aplicadas durante a scriptação, mas outras técnicas como Tabelas de Decisão, Grafos de Transição de Estados, Tabelas de Pares ou Checklists devem ser aplicadas antes de criar os scripts, com o objetivo de poder prever testes mais robustos.

## 5.1.1. Análise de Extremidades

Essa é a técnica mais antiga de Testes de Software, citada pela primeira vez por Glenford J. Myers no livro *Art of Software Testing*, Análise de Extremidades (ou Análise dos Valores Extremos ou *Boundary Value Analysis*) é basicamente testar os valores das extremidades, onde geralmente os erros são encontrados.

Essa técnica só é aplicável quando o sistema possui alguma relação de "maior que" ou "menor que" entre os valores declarados no sistema.

Por exemplo, digamos que um sistema para público adolescente só permite cadastrar idades entre 12 e 19 anos. A técnica prevê que existem valores válidos e inválidos nessa faixa etária. Por exemplo, 12, 15 e 19 são valores válidos, e 0, 11 e 20 são valores inválidos. Veja a imagem a seguir:

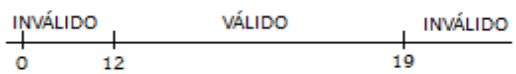



**Figura 5 – Exemplo aplicação Análise de extremidades**

O que explica o principal motivo dessa técnica é a comum confusão feita por desenvolvedores na utilização de sintaxes como: “maior que ou igual a”, “maior que”, “menor que ou igual a” e “menor que”.

Por exemplo, se o desenvolvedor digitar: “IF 12 <= valor <= 19 THEN aceita” é diferente de “IF 12 < valor < 19 THEN aceita”, no primeiro exemplo o sistema aceitaria os valores 12 e 19, que são válidos, no segundo exemplo, não. Esse é um exemplo trivial, mas em sistemas mais complexos isso pode se tornar uma iminente confusão.

### 5.1.2. Partição de equivalência

Conceitualmente, partição de equivalência significa dividir as entradas em grupos (ou Subclasses ou *Equivalence Partitioning*) que sejam tratados da mesma forma no sistema, exibindo um comportamento similar, essa técnica foi igualmente citada no livro lançado em 1979 por Myers, sendo esta, uma subdivisão da técnica citada no item acima.

A idéia é separar os itens de teste em subclasses que possuam as mesmas características e sejam disjuntos, ou seja, nenhum dos grupos selecionados podem ter um valor de entrada que pertença a ambos os grupos.

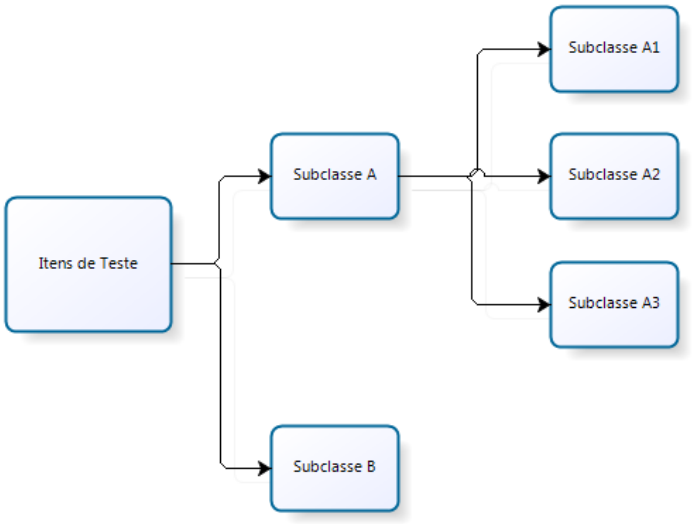



**Figura 6 – Exemplo de Partição por Equivalência**

Idealmente, todos os valores devem ser testados, mas não necessariamente eles devem ser combinados. Imagine que vamos dividir os itens de teste em dois grupos principais: classes de conexão (X) e browser (Y), são duas conexões X1 (*wireless*) e X2 (ADSL) utilizando três browsers Y1 (Internet Explorer), Y2 (Firefox) e Y3 (Chrome). Com três casos de testes (representado pelas três colunas) essas opções podem ser eficientemente testadas, veja a figura a seguir:

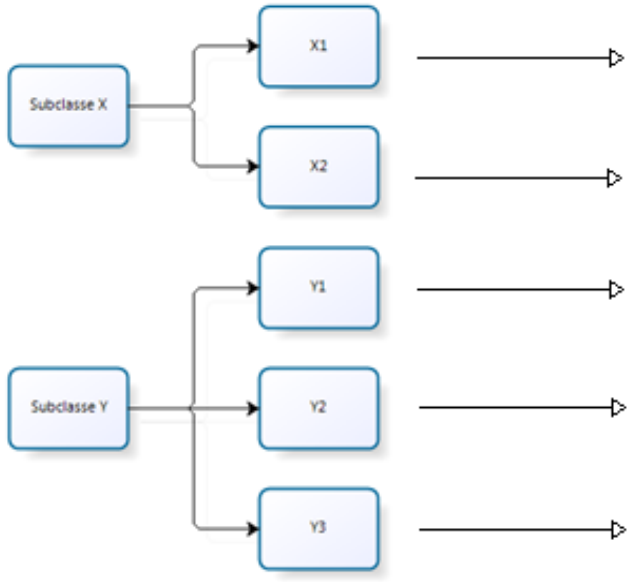
| | TC1 | TC2 | TC3 |
|---|---|---|---|
|  | **X** | | **X** |
| | | **X** | |
| | **X** | | |
| | | **X** | |
| | | | **X** |

**Figura 7 – Exemplo de derivação de *Test Cases***

Perceba que foram necessários três casos de teste para testar todas as classes de equivalência, ao invés de testar a combinação entre estes, o que resultaria em seis casos de teste. É importante salientar que essa técnica (bem como todas as outras) aplica os testes em áreas de maior cobertura do código, é possível que exista uma combinação de browser + conexão, não coberta pelos itens testados anteriormente, que gere um defeito. Estatisticamente isso é uma exceção.

### 5.1.3. Tabelas de decisão

Tabelas de decisão (ou *Decision Tables*) é uma maneira de modelar os testes baseados nas regras de negócio, esta é objeto de estudo de certificações como o ISTQB, mas foi citada nas referências da certificação por James Whittaker, através da sua coletânea *How to break a software* (Web, Security, e outros) como forma de testar todas as possibilidades num conjunto de questões a serem respondidas. O objetivo principal dessa técnica é verificar se as possíveis combinações do sistema estão sendo manipuladas de acordo com o previsto. Em outras palavras, definir quais casos de teste devem ser projetados para executar uma combinação de entradas (*inputs*) e/ou estímulos

descritos na tabela de decisão. É uma técnica que tem por objetivo auxiliar na criação de casos de teste.

Para obter um melhor entendimento de como as tabelas de decisão funcionam, é importante mostrar na prática quando e como elas devem ser utilizadas. Por exemplo, imagina-se uma empresa que faz o processamento de cartões de crédito para validação dos titulares e vendedores (empresas de cartão como Mastercard, VISA, etc.), algumas perguntas devem ser respondidas ao definir as ações que devem ser tomadas de acordo com cada perfil.

– O nome da pessoa e as outras informações sobre cartão de crédito estão corretas?

– O cartão ainda está ativo ou foi cancelado?

– A pessoa possui  limite ou acima dele?

– A transação foi feita de um endereço válido ou suspeito?

Essas perguntas devem ser respondidas e devem responder a algumas questões importantes:

– A transação deve ser aprovada?

– Devemos ligar para o titular? (Por exemplo, avisá-lo que a transação foi feita de um endereço suspeito)

– Ligar para a empresa do cartão? (Por exemplo, perguntar a eles se o cartão cancelado foi confiscado)

| Condições | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Conta existe? | S | S | S | S | S | S | S | S | N | N | N | N | N | N | N | N |
| Conta ativa? | S | S | S | S | N | N | N | N | S | S | S | S | N | N | N | N |
| Possui limite? | S | S | N | N | S | S | N | N | S | S | N | N | S | S | N | N |
| Endereço existente? | S | N | S | N | S | N | S | N | S | N | S | N | S | N | S | N |
| **Ações** | | | | | | | | | | | | | | | | |
| Aprovado? | S | N | N | N | N | N | N | N | N | N | N | N | N | N | N | N |
| Ligar para o titular? | N | S | S | S | N | S | S | S | N | N | N | N | N | N | N | N |
| Ligar para o vendedor? | N | N | N | N | S | S | S | S | S | S | S | S | S | S | S | S |

**Figura 8 – Exemplo de Tabela de Decisão**

Perceba que o total de casos de testes foram estabelecidos de acordo com a quantidade de condições, por exemplo, na primeira linha foram preenchidos oito vezes com S e oito vezes com N, na segunda linha, de quatro em quatro, na terceira linha de dois em dois, e assim por diante. Entretanto, algumas transações podem não fazer

sentido, como por exemplo, a conta não existir e ser ativa, ou não existir e possuir limite, e por aí vai.

Para diminuir a quantidade de casos de testes, é necessário juntar as colunas que possuem valores em comum e não afetam nas ações tomadas. Por exemplo, perceba que sempre que a conta é inexistente (da coluna 9 à coluna 16), as ações são as mesmas, e por aí vai.

Colidindo as colunas da tabela, o resultado ficaria assim:

| **Condições** | 1 | 2 | 3 | 5 | 6 | 7 | 9 |
|---|---|---|---|---|---|---|---|
| Conta existe? | S | S | S | S | S | S | N |
| Conta ativa? | S | S | S | N | N | N | - |
| Possui limite? | S | S | N | S | S | N | - |
| Endereço válido? | S | N | - | S | S | - | - |
| **Ações** | | | | | | | |
| Aprovado? | S | N | N | N | N | N | N |
| Ligar para o titular? | N | S | S | N | S | S | N |
| Ligar para o vendedor? | N | N | N | S | S | S | S |

**Figura 9 – Exemplo 2 de Tabela de Decisão**

Nas condições que possuem hífen, significam que, independentemente da condição, a ação tomada é a mesma. Por exemplo, no caso de teste número 9, independentemente da conta estar ativa, possuir limite ou ter endereço válido ou inválido, a ação tomada é a mesma (não aprovado, não ligar para o titular e deve ligar para a empresa de cartões de crédito – vendedor)

### 5.1.4. Grafos de transição de estado

Rex Black (2005) define essa técnica (ou State Transition Diagram) para basicamente para validação de workflows e quaisquer outras formas de sistemas em que o estado pode variar devido a ações específicas dentro do próprio fluxo. O objetivo é para garantir que uma ação correta no sistema não vá levar o mesmo a um estado inválido.

Dividem-se os grafos de transição de estado em três elementos:

- Estado: É a situação em que o sistema se encontra por tempo indeterminado. Por exemplo: navegando no browser ou uma tela em específico do sistema (tela de *login*, tela de cadastro, etc).

- Evento: É o que acontece instantaneamente (ou por tempo determinado) que ocasiona numa transição. Por exemplo: clicar num link, efetuar *login*, etc.

- Ação: É a resposta do sistema durante uma transição. Uma ação é como um evento, acontece instantaneamente (ou por tempo determinado). Por exemplo: aparecer uma tela, disparar uma mensagem de erro, etc.

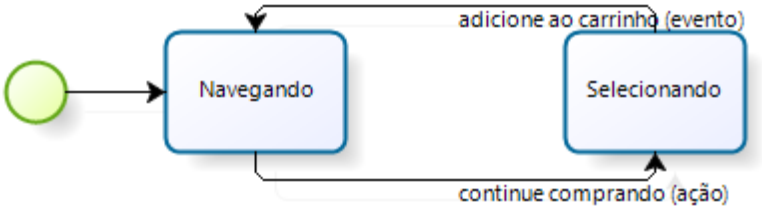



**Figura 10 – Exemplo de Grafo de Transição de Estados**

Nesse tipo de exemplo em específico, deve-se ter mais estados, como por exemplo, logando no sistema, efetuando a compra, enviando os dados de cadastro, etc. E mais ações e eventos, como abandonando a compra, logando corretamente ou incorretamente, confirmando dados de cadastro, etc.

Sendo assim, monta-se o diagrama com todos os estados e transições possíveis. Durante a criação ou execução dos testes, deve-se verificar se todos os estados são possíveis e se as transições (eventos e ações) levam para os estados corretos.

### 5.1.5. Tabelas de pares (ou Array Ortogonal)

Um estudo feito na universidade de YORK[34], definiu que a tabela de pares é criada quando precisa testar várias combinações de configurações. Sabe-se que é impraticável testar todas as possíveis combinações, levando isso em consideração, percebeu-se que a maior parte dos erros acontece em pares ou triplas. Essa técnica combina várias colunas de maneira que todos os dados se encontrem pelo menos uma vez, em pares.

Para melhor entendimento, veja a tabela a seguir.

| | **Fator** | |
|---|---|---|
| **Teste** | **1** | **2** |
| **1** | 0 | 0 |
| **2** | 0 | 1 |
| **3** | 1 | 0 |
| **4** | 1 | 1 |

**Tabela 2 – Exemplo de Array Ortogonal**

Cada coluna significa um item de configuração, e cada linha são os casos de teste, ou seja, a combinação de valores. Por exemplo: deve-se testar um sistema fator 1 seria o Sistema Operacional (onde os dois itens existentes são: Windows XP e Linux) e o fator 2 o Navegador (onde os dois itens são: Firefox e Internet Explorer). A tabela ficaria assim:

| | **Fator** | |
|---|---|---|
| **Teste** | **SO** | **Navegador** |
| **1** | Windows XP | Firefox |
| **2** | Windows XP | IE |
| **3** | Linux | Firefox |
| **4** | Linux | IE |

**Tabela 3 – Exemplo 2 de Array Ortogonal**

Nessa tabela, perceba que todas as combinações foram testadas, ou seja, não é útil para apenas dois fatores. Como funciona?

Todos os fatores da coluna um devem ser combinados com todos os fatores da coluna dois que devem ser combinados com a coluna três e assim por diante. A ilustração da tabela a seguir, mostra a eficiência dessa técnica, se fosse fazer um máximo de combinações, três fatores com dois itens cada, resultariam em seis casos de teste. Portanto, utilizando a técnica, esses valores diminuem para quatro. Veja como:

| | **Fator** | | |
|---|---|---|---|

| Teste | 1 | 2 | 3 |
|---|---|---|---|
| **1** | 0 | 0 | 0 |
| **2** | 0 | 1 | 1 |
| **3** | 1 | 0 | 1 |
| **4** | 1 | 1 | 0 |

**Tabela 4 – Exemplo 3 de Array Ortogonal**

Adaptando o exemplo anterior, supõe-se que o terceiro fator seria o tipo de conexão (banda larga e *wireless*), então, a tabela ficaria assim:

| | **Fator** | | |
|---|---|---|---|
| **Teste** | **SO** | **Navegador** | **Conexão** |
| **1** | Windows XP | Firefox | Banda larga |
| **2** | Windows XP | IE | *Wireless* |
| **3** | Linux | Firefox | *Wireless* |
| **4** | Linux | IE | Banda larga |

**Tabela 5 – Exemplo 4 de Array Ortogonal**

A pergunta é: está faltando algum valor? A resposta é não. Perceba que todos os valores da tabela combinam, em pares, com todos os valores da tabela:

- Windows XP combina com Firefox e IE (coluna dois) e com Banda larga e *Wireless* (coluna três). Mesma coisa para o Linux.

- Firefox combina com Windows XP e Linux(coluna um) e com Banda larga e *Wireless* (coluna três). Mesma coisa para o IE.

- Banda larga combina com Windows XP e Linux(coluna um) e com Firefox e IE (coluna dois). Mesma coisa para o *Wireless*.

Já existe uma forma de combinar esses fatores de maneira que não precise ficar gerando manualmente essa tabela. Existe uma técnica chamada *Taguchi Orthogonal Array* onde os valores já são pré-estabelecidos e só devem ser renomeados pelos valores em questão:

### 5.1.6. Checklists de testes

Essa técnica é considerada uma análise de risco em cima da estrutura do sistema. Basicamente o Analista de Sistemas ou o Analista de Testes fica responsável por criar uma série de itens que devem ser verificados quando o artefato (seja este um documento ou um módulo do sistema) for entregue [VEENENDAAL, 2007]. Conceitualmente, o responsável define uma lista de como proceder com os testes em alto nível e este é sempre descrito baseado por tema. Ou seja, cada sistema, possui um *checklist* único (podendo ser adaptável para sistemas semelhantes) para proceder com os testes.

Por exemplo, são muito comuns sistemas mais robustos possuírem um *checklist* de padrão de interface, qual deve ser a largura de certos campos, qual a cor utilizada em telas específicas, qual o padrão de alinhamento, etc.

Outro *checklist* bem comum é o de Usabilidade sugerido por Jakob Nielsen onde são listados conceitos básicos a seguir onde o sistema deve atender as características básicas de uma interação humana com o computador. São estes:

*1) Feedback*

2) Falar a linguagem do usuário

3) Saídas claramente demarcadas

4) Consistência

5) Prevenir erros

6) Minimizar a sobrecarga de memória do usuário

7) Atalhos

8) Diálogos simples e naturais

9) Mensagens de erro amigáveis

10) Ajuda e documentação

## APÊNDICE D – Funções e Responsabilidades do Processo de Testes

| Papel | Responsabilidades |
|---|---|
| Coordenador de Testes | Responsável por coordenar as atividades de teste<br>Responsável por seqüenciar a execução dos casos de teste |
| Arquiteto de Testes | Responsável pela preparação do ambiente de testes<br>Responsável pela criação dos dados de teste |
| Analista de Testes | Responsável por criar os casos de teste<br>Responsável pela aplicação das melhores técnicas de modelagem de casos de teste |
| Testador | Responsável por executar os casos de teste |
| Automatizador de Testes Funcionais | Responsável por criar os scripts de teste em ferramentas de automação (Selenium, Badboy, Quicktest Pro, Winrunner, etc.) |
| Automatizador de Testes de Desempenho | Responsável por criar os scripts de testes de desempenho (Loadrunner, Webload, JMeter, etc.) |

## APÊNDICE E – Como funciona a aplicação de um TDD

Teste de soma:

1) Design: criar um projeto para entender a funcionalidade a ser implementada. Deve-se listar todos os estados a serem testados (ver Apêndice A).

```
/**
* Método de teste para o método de soma da classe SomaNumeros.
*
*/
public static void testeSomaNumeros() {
    //TODO: testa com um inteiro positivo e negativo
    //TODO: testa com um inteiro negativo e positivo
    //TODO: testa dois inteiros positivos
}
```

2) Criação do teste: implementar o código de teste conforme os estados identificados na etapa anterior, que testarão a função ainda não implementada.

```
/**
* Método de teste para a classe soma da classe SomaNumeros.
*
*/

public static void testeSomaNumeros () {
    int resultado;
    // testa com um inteiro positivo e negativo
    resultado = SomaNumeros.soma(2, -3);
    assertEquals("Valor esperado: -1, resultado: "+resultado, -1,
resultado);

    //TODO: testa com um inteiro negativo e positivo
    //TODO: testa dois inteiros positivos
}
```

3) Testar a aplicação: criar a classe para rodar o teste.

```
/**
* @author Luidi Andrade
*/
```

```
public class SomaNumeros {

/**
* Método que retorna a soma dos inteiros passados por *parâmetro.
*
* @param x
* @param y
*/
    public static int soma(int x, int y) {
            // TODO Auto-generated method stub
            return 0;
    }
}
```

4) Execução do teste: executando o teste neste momento, não irá passar, pois o método não foi implementado por completo.

5) Implementação do método: implementar o método com as funcionalidades requeridas para passar no teste, sem preocupações em otimizá-lo.

```
/**
* @author Luidi Andrade
*/
public class SomaNumeros {

/**
* Método que retorna a soma dos inteiros passados por *parâmetro.
*
* @param x
* @param y
*/
    public static int soma(int x, int y) {
            // TODO Auto-generated method stub
            return x+y;
    }
}
```

6) Re-executar o teste: se todos os testes passarem indicará que o código atende aos requisitos testados e que esta funcionalidade não afetou outras partes do sistema.

7) Refatorar o código: se houver necessidade de "limpar" o código, deve ser feito agora. O teste deve ser executado durante este processo para ter certeza de que as alterações não afetaram o sistema negativamente.

## APÊNDICE F – Exemplo Clam Win Free Antivirus

Esse é o exemplo da aba Tracker no software exemplo (ClamWin Free Antivirus).


**Figura 11 – Exemplo da aba Tracker no software Clam Win Free Antivirus**