

**LIGA DE ENSINO DO RIO GRANDE DO NORTE
FACULDADE NATALENSE PARA O DESENVOLVIMENTO
DO RIO GRANDE DO NORTE**

JOSE ANCHIETA SOARES DE PAIVA

**TESTE DE SOFTWARE: METODOLOGIAS,
APLICAÇÕES E FERRAMENTAS**

**NATAL/RN
2008**

JOSÉ ANCHIETA SOARES DE PAIVA

TESTE DE SOFTWARE: METODOLOGIAS, APLICAÇÕES E FERRAMENTAS

Monografia apresentada ao Curso de Graduação em Sistemas de Informação da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte como requisito obrigatório para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Gleydson Lima, Msc.

NATAL/RN
2008

f81.55.04
J6f/1100m
19 20 11

Catálogo na Publicação – Biblioteca da FARN
Setor de Processos Técnicos

Paiva, José Anchieta Soares de

Teste de software: metodologias, aplicações e ferramentas / José
Anchieta Soares de Paiva. – Natal: FARN, 2008.

53f.

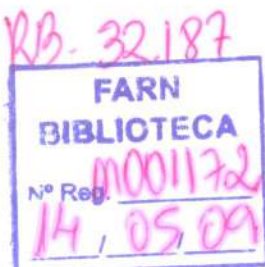
Orientador: Gleydson Lima (Msc.)

Monografia (Bacharelado em Sistemas de Informação) – Faculdade
Natalense para o Desenvolvimento do Rio Grande do Norte.

1. Teste de Software – Monografia. 2. Extremme Programming XP –
Monografia. 3. Ferramentas de Teste – Monografia I. Título.

RN/FARN/BC

CDU 004.4



004.4
P168t
2008
ex 01

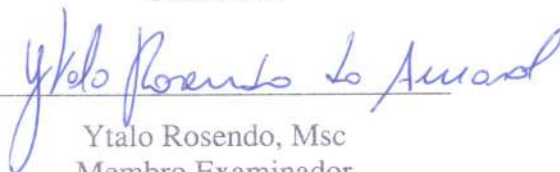
JOSÉ ANCHIETA SOARES DE PAIVA

**TESTE DE SOFTWARE: METODOLOGIAS,
APLICAÇÕES E FERRAMENTAS**

Monografia apresentada ao Curso de Graduação em Sistemas de Informação da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte, aprovada em ____ de ____ de ____, pela banca examinadora composta pelos seguintes membros:



Gleydson Lima, Msc
Orientador



Ytalo Rosendo, Msc
Membro Examinador



Josué Vitor, Msc
Membro Examinador

FARN

Aos meus pais, por todo apoio recebido em todos os momentos da minha vida.

A minha esposa que foi compreensiva, companheira e me apoiou durante toda essa nova etapa.

AGRADECIMENTOS

Agradeço a princípio a Deus, que me permitiu a inteligência. Ao meu orientador, pelas orientações e principalmente pela compreensão não somente como professor, mas também como amigo.

A Faculdade Natalense para o Desenvolvimento do RN, em especial ao reitor Daladier da Cunha Lima por todas as oportunidades concedidas pelo mesmo.

RESUMO

O objetivo deste trabalho foi apresentar o conceito de teste de software, definindo seus tipos, métodos e classificações em categorias, bem como discutir as técnicas, metodologias e ferramentas disponíveis no mercado de forma a auxiliar aqueles que procuram produzir software com mais qualidade. Por meio de pesquisa bibliográfica foram tratados os principais modelos de teste, técnicas de desenvolvimento orientadas a teste, TDD, XP, RUP, além da apresentação de diversas ferramentas de teste, divididas por categorias e especificações, apresentando particularidades sobre o que cada uma pode oferecer, além de abordar os dois universos dos softwares, sendo os softwares comerciais e os softwares livres.

Palavras-chave: Metodologia; Teste; Testes unitários; Testes funcionais, Software; Ferramentas.

ABSTRACT

The purpose of this study was to present the concept of testing software, defining their types, methods and classificações into categories and discuss the techniques, methodologies and tools available on the market in order to assist those who seek to produce software with more quality. Through literature search were treated the main types of testing, aimed at developing techniques for testing, TDD, XP, RUP, in addition to the provision of various tools of testing, divided by categories and specifications, submit specific about what each can offer In addition to addressing the two worlds of software and the commercial software and free software.

Keywords: Methodology, Testing, Testing Unit, Functional Testing, Software, Tools.

LISTA DE QUADROS

Quadro 1	Categorias de teste de software	16
Quadro 2	Tipos de Teste RUP	23
Quadro 3	Teste funcional	24
Quadro 4	Teste de volume	24
Quadro 5	Teste de Segurança	25
Quadro 6	Teste de acessibilidade	25
Quadro 7	Teste de usabilidade	26
Quadro 8	Teste de stress	27
Quadro 9	Teste de regressão	27
Quadro 10	Teste de integridade	28
Quadro 11	Teste de desempenho	29
Quadro 12	Teste de carga	30
Quadro 13	Teste de instalação	31
Quadro 14	Teste de configuração	31
Quadro 15	Classe de teste em java	44

LISTAS DE ILUSTRAÇÕES

Figura 1	Defeito x erro x falha	14
Figura 2	Técnica de Caixa-Preta (Funcional)	15
Figura 3	Metodologia Waterfall	18
Figura 4	Modelo SawTooth	18
Figura 5	Modelo Shark Tooth	19
Figura 6	Spiral Model	20
Figura 7	Modelo V	21
Figura 8	Modelo W	21
Figura 9	Modelo <i>Butterfly</i> (Borboleta)	22
Figura 10	Modelo de desenvolvimento orientado a testes no XP	33
Figura 11	Fornecedores mais importantes (Ferramentas Comerciais)	38
Figura 12	Ferramentas Open Source	39
Figura 13	Ferramentas Open Source, Performance, Carga e Stress	40
Figura 14	Arquitetura JUnit	41
Figura 15	Metódos JUnit	43

LISTA DE ABREVEATURAS, SIGLAS E SIMBOLOS

IEEE	Institute of Electrical and Electronics Engineers
RUP	Rational Unified Process
TDD	Test-Driven Development
XP	Extremme Programming

SUMÁRIO

1 INTRODUÇÃO.....	11
2 REFERENCIAL TEÓRICO.....	12
2.1 TESTE DE SOFTWARE.....	12
2.1.1 Conceitos.....	13
2.1.2 Níveis de Teste.....	15
2.1.3 Metodologias de Teste.....	17
2.2 RUP.....	22
2.3 EXTREMME PROGRAMMING XP.....	32
2.4 TDD (TEST-DRIVEN DEVELOPMENT).....	35
2.5 FERRAMENTAS DE TESTES AUTOMÁTICOS.....	36
2.6 JUNIT.....	40
3 METODOLOGIA.....	45
4 ANÁLISE DE RESULTADO.....	46
5 CONCLUSÃO.....	49
REFERENCIAS.....	50

1 INTRODUÇÃO

O desenvolvimento de software envolve uma série de atividades tais como, técnicas, métodos e ferramentas utilizadas, mesmo assim erros ainda podem acontecer, com o passar do tempo atividades denominadas de garantia da qualidade de software tem sido introduzidas no processo de desenvolvimento, entre elas Verificação, Validação e Teste, tendo como objetivo minimizar a ocorrência de erros riscos associados, dentre elas a atividade de teste é uma das mais utilizadas, constituindo-se em um dos elementos para fornecer evidências da confiabilidade do software em complemento a outras atividades, como por exemplo, o uso de revisões e de técnicas formais e rigorosas de especificação e de verificação. O teste consiste uma análise dinâmica do produto e é fator de alta relevância para identificação e eliminação de erros que persistem.

O teste de software envolve basicamente quatro etapas: planejamento, projeto, execução e avaliação de resultados, etapas essas que ao longo do processo de desenvolvimento concretizam-se em três fases de teste: unidade, integração e sistema. O teste de unidade concentrador de esforços concentra esforços na menor unidade do projeto de software, ou seja, testando cada módulo do software, separadamente, já o teste de integração é aplicado durante a integração da estrutura do programa, visando encontrar erros nas interfaces entre os módulos, sendo o último teste de sistema, realizado após a integração visando identificar erros de funções e características de desempenho que não estejam de acordo com a especificação.

Assim, o teste de software tem evoluído a ponto de ter se tornado fundamental o uso de ferramentas de teste para o suporte a atividade, uma vez que essa atividade é muito propensa a erros, além de improdutiva quando aplicada manualmente, assim a disponibilidade de ferramentas de testes automatizados proporciona maior qualidade e produtividade para a área. Técnicas e metodologias estão surgindo baseadas no teste de software, abordados processos tais como RUP, metodologias e práticas como XP e TDD para demonstrar o processo de teste de software de uma maneira diferenciada, sendo ele agora um dos atores principais no processo de desenvolvimento e garantia da qualidade.

Tendo como objetivo geral apresentar as metodologias, ferramentas e aplicações da área de teste de software. Com relação aos objetivos específicos classificar e conceituar as ferramentas da área de teste mais importantes n mercado, sendo elas comerciais ou open-source .

Foram abordados a partir do capítulo dois os conceitos mais importantes da área de teste, além das metodologias e ferramentas, sendo apresentada a metodologia no capítulo seguinte, exibição dos resultados no capítulo quatro finalizando com a conclusão.

2 REFERENCIAL TEÓRICO

2.1 TESTE DE SOFTWARE

O teste é um pré-requisito necessário para a implementação bem sucedida de um produto de software, mas com o ensaio de tecnologias disponíveis atualmente, é muitas vezes considerado como sendo difícil, enfadonho, consumidor de tempo e inadequado (LI; WU, 2004). Podem ser definidos como o funcionamento de uma aplicação em condições controladas, a fim de avaliar a sua qualidade (ASH, 2003).

Eles são importantes porque contribuem substancialmente para garantir que um software faça tudo aquilo que é suposto fazer, bem como garantir que ele não faça nada mais do que aquilo que é suposto a fazer (MCGREGOR; SYKES, 2001).

O processo de teste de software agora se trata de um grande diferencial no que se diz respeito a ajudar na melhoria da qualidade de software e o produto final. Testes também se tornam importantes dentro da área de qualidade de software, mas não devem parar de ser feitos, muito menos de serem enxergados como o “resto” dentro de um projeto, pois ele passa a ser um projeto de outro projeto e agora tem que ser trabalho antes, durante e depois (MOLINARI, 2003).

Testes exigem recursos consideráveis e a efetiva utilização desses recursos exige um bom planejamento e da boa gestão (MCGREGOR; SYKES, 2001).

Devem ser planejados segundo Molinari (2003) como um general organiza uma batalha, testando a aplicação como um ninja, ou seja, com precisão. Afinal mesmo que não esteja vendo os “bugs”, eles farão parte da aplicação, e estarão esperando, apenas aguardando o momento certo de aparecer.

Para Myers; Beizer (apud TOSETTO, 2004) um bom planejamento é necessário para que se obtenham bons casos de teste; caso contrário, pode-se obter um conjunto de casos que não encontrará erros (o verdadeiro objetivo da atividade), ou ter um número excessivo de casos, o que resultaria em elevação dos custos do processo de desenvolvimento e em uma

relação custo-benefício não compensador. É recomendado que as técnicas sejam combinadas no planejamento de um teste rigoroso de um programa, visto que as mesmas não são excludentes, mas sim, complementares, ao mesmo tempo em que possuem diferentes forças e fraquezas.

2.1.1 Conceitos

Molinari (2003) classifica alguns termos para falha de software, sendo eles:

- Defeito, *bug*, problema, erro, incidência, anomalia, variação, falha, falta, inconsistência e característica.

Nota-se que dentre todos existe algo em comum, como nos exemplos:

- Falha, falta e defeito, representam de forma semelhante “o software falhou em não funcionar corretamente ou em não fazer a coisa correta”.
- Anomalia, incidência e variação representam um “funcionamento eventual ou involuntário do software”.
- Problema, erro e “bug” são termos genéricos.

Para Dias Neto (2007) dentro da área de teste de software existem alguns conceitos relacionados e que antes do aprofundamento devem ser estudados. Também deve-se diferenciar defeitos, erros e falhas. As definições usadas seguem a terminologia padrão para Engenharia de Software do IEEE – Institute of Electrical and Electronics Engineers – (IEEE 610, 1990). Seguem as definições:

- **Defeito** é um ato inconsistente cometido por um indivíduo ao tentar entender uma determinada informação, resolver um problema ou utilizar um método ou uma ferramenta. Por exemplo, uma instrução ou comando incorreto.
- **Erro** é uma manifestação concreta de um defeito num artefato de software. Diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução de um programa constitui um erro.
- **Falha** é o comportamento operacional do software diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversos erros e alguns erros podem nunca causar uma falha. Dias Neto (2007) apresenta Figura 1 representando o universo dos conceitos.

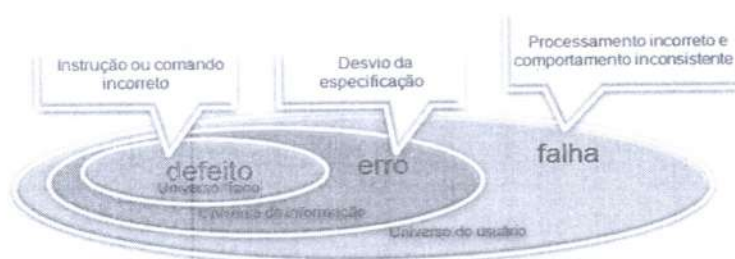


Figura 1: Defeito x erro x falha
 Fonte: Dias Neto (2007)

Molinari (2003) explica que os conceitos abaixo são de extrema importância, que caso os entenda terá condições de usar qualquer ferramenta de automação de testes ou não de forma clara e objetiva, bastando apenas entender os recursos de cada uma, pra ele são considerados “sete conceitos mágicos de teste” simples, mas de extrema estratégia pelo fato da fronteira entre eles ser muito sutil:

- **Planejamento de testes:** Trata-se do próprio processo de planejamento.
- **Plano de Testes:** Artefato principal, oriundo do fruto do planejamento.
- **Requerimento de Teste:** É o que se quer testar na aplicação, ou seja, uma meta ou a parte exata do que vai ser testado.
- **Casos de Teste:** descreve uma condição particular a ser testada e é composto por valores de entrada, restrições para a sua execução e um resultado ou comportamento esperado (CRAIG; JASKIEL, apud DIAS NETO, 2007).
- **Procedimento de Teste:** É uma descrição dos passos necessários para executar um caso (ou um grupo de casos) de teste (CRAIG; JASKIEL, apud DIAS NETO, 2007).
- **Script de Teste:** É a implementação do procedimento de teste usando uma ferramenta automatizada, ou seja, na prática é o fruto de uma gravação da execução do procedimento, já o script manual significa o script de teste não gravado (MOLINARI, 2003).
- **Ponto de Verificação:** É um “check” (opção) que se faz dentro do script automatizado para saber se tudo ocorreu bem e o teste foi bem sucedido (DIAS NETO, 2007).

2.1.2 Níveis de Teste

A atividade de Teste de Software conforme Molinari (2003) pode ser vista como se fossem 3 dimensões:

- **Primeira dimensão:** Trata do estado do teste, podendo se considerar três tipos básicos:
 - **Teste de unidade:** É o teste em nível de componente ou classe, cujo objetivo é um pedaço (unidade) do código (MOLINARI, 2003).
 - **Teste de integração:** Segundo Amaral (2006) é uma atividade sistemática aplicada durante a integração da estrutura do programa, visando descobrir erros associados às interfaces entre os módulos, ou seja, constrói a estrutura do programa determinada pelo projeto a partir dos módulos testados no nível de unidade.
 - **Teste de sistemas:** Para (MYERS apud TOSETTO, 2004) testa o sistema como um todo, comparando o mesmo com seus objetivos originais.
- **Segunda dimensão:** Técnica do teste, para (GRAF; SERRANO, 2006) existem basicamente duas maneiras de construir testes, vejamos os tipos de teste da segunda dimensão:
 - **Teste de Caixa-Preta:** Para ASH (2003) é um tipo de testes que se tem idéia do código está em uma caixa preta, não sendo possível ver o interior muito menos como funciona, apenas o modo como é capaz de usá-lo como um usuário final. Dados de entrada são fornecidos, o teste é executado e o resultado obtido é comparado a um resultado esperado previamente conhecido, a figura 2 demonstra o que se entende por caixa preta (DIAS NETO, 2007).

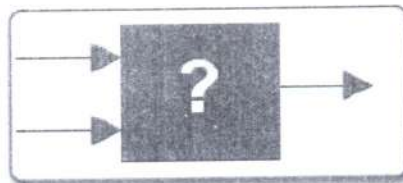


Figura 2: Técnica de Caixa-Preta (Funcional)
Fonte: Dias Neto (2007)

- **Teste de Caixa-Branca** (Também conhecida como técnica estrutural): Técnica de teste que avalia o comportamento interno do componente de software. Essa técnica trabalha diretamente sobre o código fonte do componente de software

para avaliar aspectos tais como: teste de condição, teste de fluxo de dados, teste de ciclos e teste de caminhos lógicos (PRESSMAN apud DIAS NETO, 2007).

- **Terceira Dimensão:** Metas do Teste (Tipo de teste), Segundo Bartié (2000) o quadro abaixo (quadro 1) classifica os tipos de teste da terceira dimensão.

Categoria	Objetivos	Requisitos	Resultados	Importância
Funcional	Garantir que os requisitos sejam implementados conforme especificado.	Profundo conhecimento das regras de negócio	Defeitos de aplicação e falhas no levantamento de requisitos	Essencial
Usabilidade (Interface)	Detectar problemas de interface que tornam o software pouco intuitivo.	Norma de padronização de interface	Defeitos de interface e navegação do software	Meio impacto
Volume e carga	Avaliar como o software reage a uma quantidade excessiva de transações.	Ambiente de software e hardware	Determinação da capacidade máxima de operação	Alto impacto, principalmente se o software for de missão crítica
Ambiente	Avaliar como o software reage a diversos hardwares e sistemas operacionais.	Ambiente de software e hardware	Determinação da configuração de hardware e software	Baixo impacto
Segurança	Detectar falhas de segurança que podem comprometer o sigilo e a fidelidade das informações	Normas de segurança	Falhas de segurança	Essencial
Compatibilidade	Avaliar a compatibilidade com outros componentes de software.	Componentes que possuem interface com o software testado	Defeitos de quebra de compatibilidade	Essencial
Performance	Determinar se o desempenho atende os requisitos	Ambiente de software e hardware	Falhas de performance	Médio impacto
Instalação	Validar o procedimento de instalação da aplicação.	Ambiente de software e hardware	Falhas de instalação da aplicação	Médio impacto
Confiabilidade e disponibilidade	Identificar interrupções no uso do software e tempo para resolução do problema.	Ambiente de software e hardware	Situações de interrupções de uso do software	Alto impacto
Recuperação	Avaliar o comportamento do software após uma condição anormal.	Ambiente de software e hardware	Falhas de recuperação e erros	Alto impacto
Contingência	Validar os procedimentos da contingência em uma situação prevista.	Ambiente de software e hardware	Defeitos de infraestrutura ou nos procedimentos de contingência.	Alto impacto

Quadro 1: Categorias de teste de software

Fonte: Bartié (2000)

2.1.3 Metodologias de Teste

Para Molinari (2003) um exemplo real de modelo e metodologia foi à estratégia que o campeão mundial ex-piloto de fórmula-1 Michael Schumacher, aplicando na prática metodologias e vários modelos que o levaram a ser campeão, tendo como fruto final à criação de um carro campeão. Desenvolvendo uma estratégia macro visando objetivos a longo prazo, um modelo, e a partir disso foi criada a implementação da estratégia anterior, sendo uma metodologia que sem, sombra de dúvida, tornou-se base vencedora para todas as transformações do carro e da fórmula-1 “idealizado” para realidade. Na área de teste existe um problema comum que é sempre confundir modelo com metodologia, portanto será diferenciado ambos mais à frente.

Segundo Molinari (2003, p. 114) abaixo a diferença entre modelo e metodologia:

- **Metodologia:** “conjunto de técnicas, métodos e estratégias voltadas para a criação e/ou execução de algo, que em geral pode ser um ou mais produtos e/ou serviços”.
- **Modelo:** “conjunto de técnicas que representam, ou que tenta representar, algo da realidade ou do conceito”.

No contexto de testes de softwares, é possível encontrar diversos modelos que às vezes se autodenominam metodologias de testes que visam otimizar, criar ou representar o processo de teste de software. Tem-se como exemplo comum o “V-Model” e o “W-Model”, pois ambos tentam ser modelo e metodologia ao mesmo tempo. Quando se fala em modelo, refere-se a um nível de testes que inclui quem faz o que e as áreas envolvidas dentro da empresa, já a metodologia está num nível abaixo, visando mostrar tudo em detalhes (MOLINARI, 2003).

Referente a área de testes de software foi representado modelos que visam representar e orientar o processo de teste, lembrando que testes agora tem que ser vistos como um projeto, que geralmente faz parte de outro projeto abaixo os principais modelos (MOLINARI, 2003).

O modelo Waterfall (cascata) é a mais antiga metodologia de desenvolvimento de software. Segue uma progressão lógica da definição de requisitos de concepção do sistema, a construção do sistema o ensaio do sistema, e colocar o sistema em funcionamento. É usado principalmente para o desenvolvimento de sistemas de processamento batch (em lote) que fazer. O modelo cascata assume que, no final da fase requisitos, as exigências em matéria de desenvolvimento são conhecidas (PERRY, 2006).

A figura 3 mostra claramente o modelo em forma de cascata, ordenando em forma de progressão.



Figura 3: Metodologia Waterfall
Fonte: Molinari (2003)

Modelo Sawtooth que mostra a interação entre usuário e desenvolvedor, tendo o segundo um enfoque simples, semelhante a um pedreiro que faz uma obra numa casa, ou seja, fazendo sob encomenda, conforme foi pedido. Também pode ser chamado de “Serra” ou “Visão Dentária” facilmente visível na figura 4, devido ao fato de lembrar a parte inferior de uma arcada dentária cujos dentes são pontiagudos (MOLINARI, 2003).

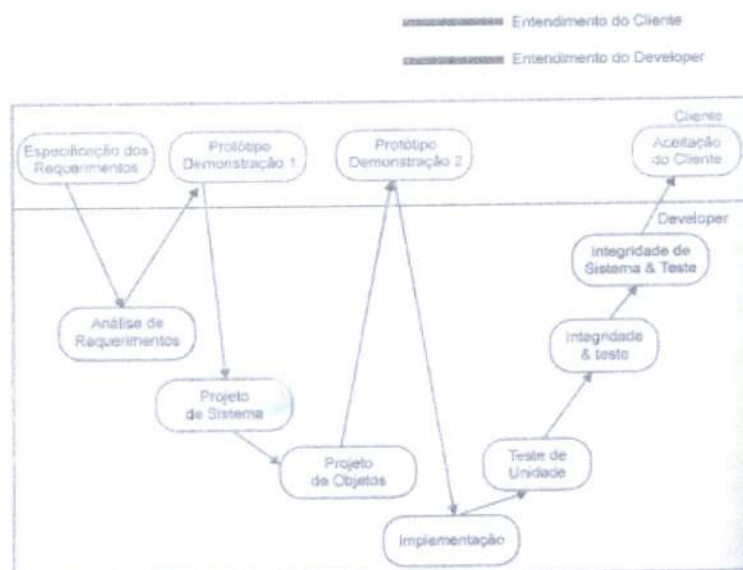


Figura 4: Modelo SawTooth
Fonte: Molinari (2003)

O modelo SharkTooth é semelhante ao SawTooth, porém é diferenciado pelo fato de contar com a participação de um “gerente” que entra como revisor no processo, buscando minimizar o “gap” (lacunas) de padrões de desenvolvimento e qualidade. A diferença dele com o SawTooth é que quem fará a integração do sistema e o respectivo teste é o novo elemento “gerente”, ou seja, alguém mais experiente. Modelo também chamado de “boca de tubarão” ou “teste tubarão” porque parece com o “sorriso de um tubarão” que através da figura 5 pode ser visualizado com clareza (MOLINARI, 2003).

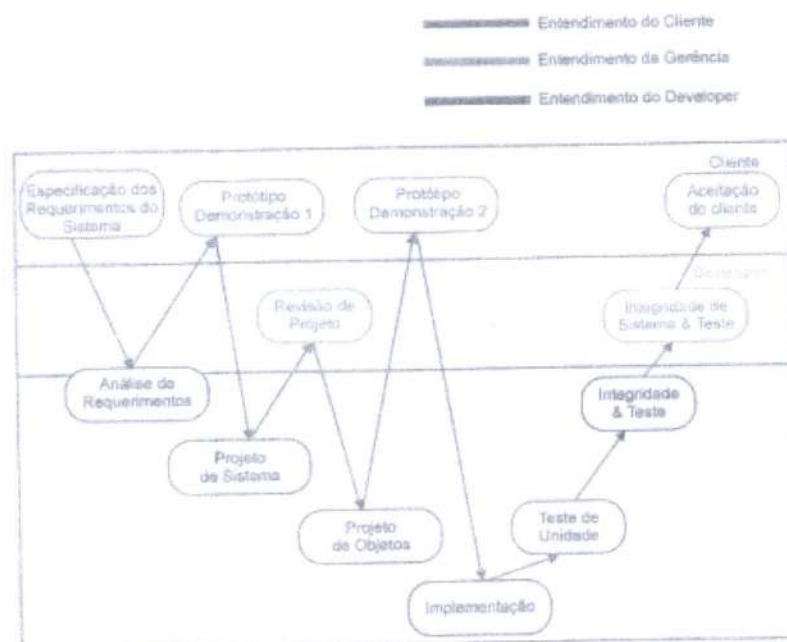


Figura 5: Modelo Shark Tooth
Fonte: Molinari (2003)

O modelo Espiral é baseado na necessidade de iterar (repetir), contém tantas iterações sejam necessárias para levar a bom termo um produto. Cada iteração exige que os participantes do plano definam seu ciclo de vida, protótipo, análise de riscos, escrever requisitos, construir modelos, desenhos detalhados, código, unidade, sistema de testes e instalação (VINAY, 2008), podendo ser claramente observados na Figura 6.

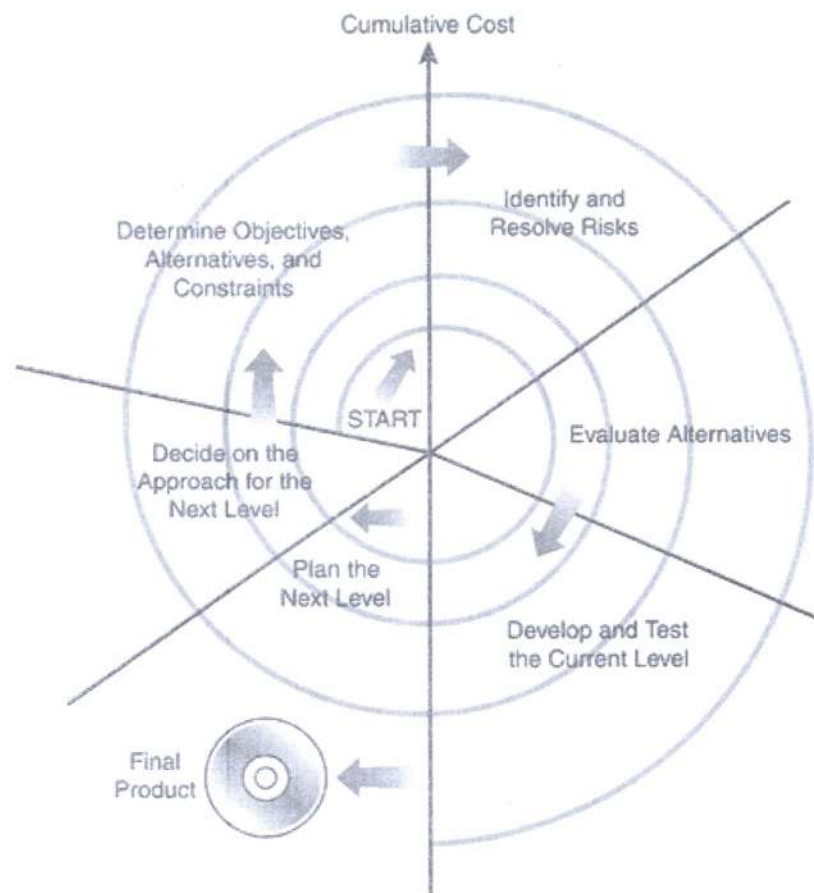


Figura 6: Spiral Model
Fonte: Patton (2005)

Modelo V pode-se dizer que a maioria dos modelos de processos conecta-se graficamente a ele, que se tornou um dos mais populares. É conhecido como modelo V, pois descreve graficamente as fases individualmente em forma de "V". Neste caso "V" vem de verificação e validação. Modelo facilmente compreendido, pela sua simplicidade, pois as atividades são ordenadas numa forma sequencial em níveis de abstração de modo que as conexões com as fases de desenvolvimento tornam-se bastante claras (MOLINARI, 2003).

Perry (2006) explica que o modelo V desenvolve dois processos, um para a construção do sistema e outro para testar, conforme podemos ver através da Figura 7, eles se inter-relacionando em V os dois processos.

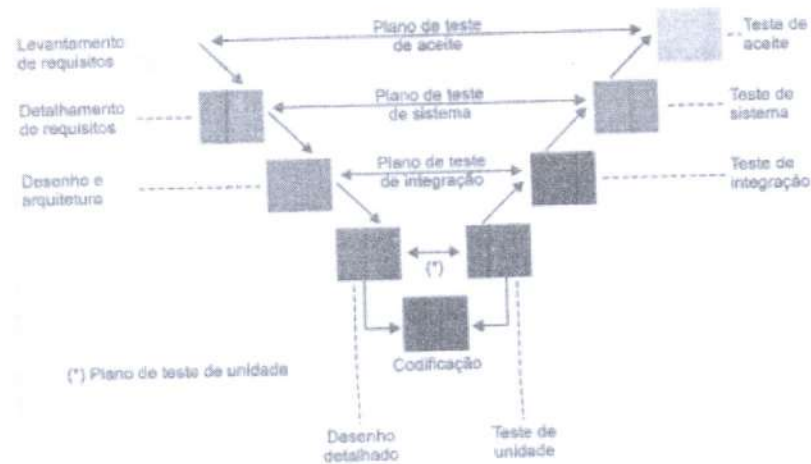


Figura 7: Modelo V
Fonte: Molinari (2003)

Molinari (2003) explica que o modelo W baseia-se no modelo “V”, tendo como diferença que para cada etapa do modelo teríamos uma etapa de “testes” que correspondesse a cada fase, tornando possível eliminar os “bugs” desde o início, sendo do lado esquerdo um planejamento de cada fase e no lado direito haveria um teste de cada fase, através da figura 8 notamos a relação com o W. Tendo como característica primordial diminuir custos, partindo do princípio que quanto mais se teste, mais cedo os problemas são encontrados, ficando os problemas minimizados, mas não resolvidos, tendo como vantagem a atividade de teste se tornar bem clara.

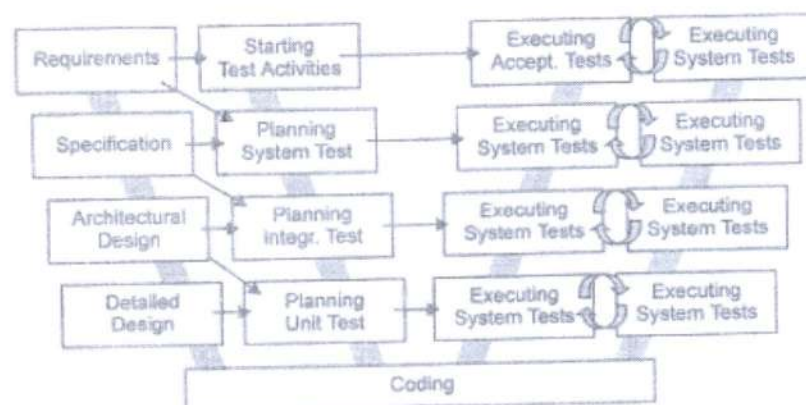


Figura 8: Modelo W
Fonte: Molinari (2003)

Conforme Molinari (2006), o modelo *butterfly* (borboleta) também se baseia no modelo “V”, tentando linearizar uma tarefa que não seja linear: desenvolvimento de software. A visão do modelo borboleta é que cada subfase representaria um componente de um único ser mutável e de vida curta, daí a origem do “borboleta” bem representada na figura 9, sendo a

asa da esquerda a etapa de análise, já a da direita corresponde análise e a parte central a execução em si dos testes.

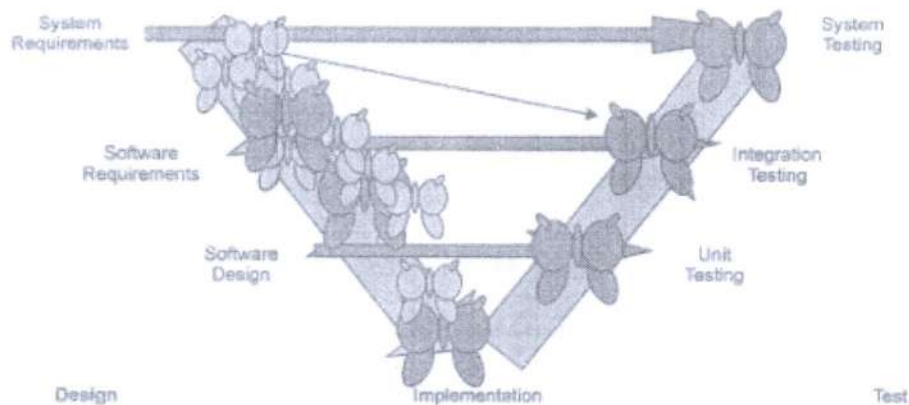


Figura 9: Modelo *Butterfly* (Borboleta)
Fonte: Molinari (2003)

2.2 RUP

Rational Unified Process (também chamado de processo RUP) é um processo de engenharia de software. Oferecendo uma abordagem baseada em disciplinas para atribuição de tarefas e responsabilidades dentro de uma organização de desenvolvimento. Tendo como meta garantir a produção de software de alta qualidade que atenda às necessidades dos usuários dentro de um cronograma e de um orçamento previsível (RATIONAL, 2002).

Durante o ciclo de vida do projeto, devem existir pontos de verificação visando à qualidade do produto. O desenvolvimento e a verificação criam um processo repetitivo (iterativo e incremental) onde o produto vai sendo avaliado e, caso necessário, melhorado, até estar maturado, pronto para ser disponibilizado ao usuário final. Durante estas fases, nos pontos de verificação, podem ser realizados testes, para garantir a conformidade do produto (DATASUS, 2007).

Conforme Datasus (2007) “o RUP categoriza qualidade utilizando o modelo **FURPS** (*Functionality, Usability, Reliability, Performance e Supportability*). O Quadro 2 apresenta os Tipos de Teste, agrupados por Dimensões de Qualidade, e as características correspondentes a cada grupo”: (grifo do autor).

Categoria	Tipos de Teste	Requisitos
Funcionalidade	<u>Teste Funcional</u> <u>Teste de Volume</u> <u>Teste de Segurança</u> <u>Teste de Acessibilidade</u>	Conjunto de características; Capacidade; Segurança.
Usabilidade	<u>Teste de Usabilidade</u>	Fator humano e Estética; Consistência da Interface do usuário; Help online e assistentes.
Confiabilidade	<u>Teste de Estresse</u> <u>Teste de Regressão</u> <u>Teste de Integridade</u> <u>Teste de Estrutura</u>	Frequência e severidade de falhas; Confiança nos resultados; Recuperação de falhas; Tempo entre falhas (MTBF).
Desempenho	<u>Teste de Desempenho</u> <u>Teste de Carga</u> <u>Teste de Contenção</u> <u>Teste de Perfil de Desempenho</u>	Vazão; Eficiência; Tempo de Resposta; Consumo de recursos.
Suportabilidade	<u>Teste de Instalação</u> <u>Teste de Configuração</u>	Facilidade de instalação; Requisitos de configuração; Requisitos de adaptabilidade; Requisitos de Compatibilidade.

Quadro 2: Tipos de Teste RUP
Fonte: Datasus (2007)

Rational (2002) classifica os tipos de teste existentes no modelo **FURPS**, conforme detalhamento abaixo:

Teste de função: Testes destinados a validar as funções do objetivo do teste conforme o esperado, fornecendo os serviços, métodos ou casos de uso necessários. Esse teste é implementado e executado em diferentes objetivos do teste, como unidades, unidades integradas, aplicativos e sistemas.

Objetivo do Teste	Assegurar a funcionalidade do sistema, incluindo entrada de dados, processamento e resposta.
Técnica/Método	Executar cada caso de uso, fluxo de caso de uso, ou função, utilizando dados válidos e inválidos para verificar o seguinte: Os resultados esperados ocorrem quando dados válidos são usados. As mensagens de erro de avisos apropriadas são exibidas quando dados inválidos são utilizados. Cada regra de negócio é corretamente aplicada.
Observações	

Quadro 3: Teste funcional
Fonte: Datasus (2007)

Teste de volume: Teste destinado a verificar a capacidade do objetivo do teste de lidar com um grande volume de dados, como entrada e saída ou residente no banco de dados. O teste de volume abrange estratégias de teste, como, por exemplo, a entrada de dados do volume máximo de dados em cada campo ou a criação de consultas que retornem todo o conteúdo do banco de dados ou que tenham tantas restrições que nenhum dado seja retornado.

Objetivo do Teste	Verificar se o sistema funciona com sucesso sob os seguintes cenários de grande volume: Número máximo (real ou fisicamente capaz) de clientes conectados, ou simulados, todos realizando a mesma função de pior desempenho. Tamanho máximo alcançado do banco de dados, real ou adaptável, e múltiplas consultas ou transações executadas simultaneamente.
Técnica/ Método	Utilizar testes desenvolvidos para Teste de Desempenho ou Testes de Carga. Múltiplos clientes devem ser utilizados, executando os mesmos testes ou testes complementares para produzir a transação de pior desempenho ou transações mistas (umas de alta e outras de baixa desempenho). Definir um limite para o tamanho do banco de dados (real, adaptável, ou preenchido com dados representativos) e múltiplos clientes para executar consultas e reportar transações simultaneamente por longos períodos.
Observações	

Quadro 4: Teste de Volume
Fonte: Datasus (2007)

Teste de segurança: Testes destinados a garantir que o objetivo do teste e os dados (ou sistemas) possam ser acessados apenas por determinados atores. Esse teste é implementado e executado em vários objetos de teste. DATASUS (2007) ressalta que o teste segurança tem seu foco em duas áreas principais de segurança:

- Segurança em nível de aplicação, incluindo acesso aos dados ou às funções do negócio;
- Segurança em nível de sistema, incluindo o acesso ao sistema feito localmente ou remotamente.

Objetivo do Teste	Verificar as restrições de acesso em nível de aplicação e sistema.
Técnica/Método	Segurança em nível de aplicação: identificar e listar cada tipo de usuário e as funções ou dados aos quais cada tipo tem permissão. Criar testes para cada tipo de usuário e verificar cada permissão criando transações específicas para cada um dos tipos. Modificar o tipo de usuário e executar novamente os testes para os mesmos usuários. Em cada caso, verificar aquelas funções adicionais ou se os dados estão corretamente disponíveis ou negados.
Observações	

Quadro 5: Teste de segurança
Fonte: Datasus (2007)

Teste de acessibilidade: Testes destinados a verificar se a interface do usuário fornecesse o acesso apropriado às funções do sistema, bem como navegação adequada, também garante que os objetos dentro da interface do usuário funcionem de acordo com os padrões definidos pelo cliente.

Objetivo do Teste	Verificar se a navegação através do sistema reflete corretamente as funções e requisitos de negócio. Objetos e características da janela, tais como: menus, tamanho, posição, estado e foco conforme os padrões.
Técnica/Método	Criar ou modificar testes para cada janela, verificando corretamente a navegação e os estados de objeto para cada janela do sistema e objetos. Fazer navegação janela-a-janela, campo-a-campo, e uso dos métodos de acesso (teclas de tabulação, movimentos de mouse, teclas de atalho).
Observações	

Quadro 6: Teste de acessibilidade
Fonte: Datasus (2007)

Teste de usabilidade: Testes que enfatizam:

- fatores humanos,
- estética,
- consistência na interface do usuário,
- ajuda on-line e contextual,
- assistentes e agentes,
- documentação do usuário e material de treinamento.

Objetivo do Teste	Verificar: A facilidade de operação do sistema pelo usuário; A facilidade de entendimento das funções do sistema pelo usuário, através da utilização de manuais, help on-line, agentes e assistentes eletrônicos, etc.
Técnica/Método	Executar diversas operações do sistema, utilizando manuais e help on-line.
Observações	

Quadro 7: Teste de Usabilidade
Fonte: Datasus (2007)

Teste de stress: Tipo de teste de confiabilidade destinado a avaliar como o sistema responde em condições anormais. O stress no sistema pode abranger cargas de trabalho extremas, memória insuficiente, hardware e serviços indisponíveis ou recursos compartilhados limitados. Normalmente, esses testes são executados para compreender melhor como e em quais áreas o sistema será dividido, para que os planos de contingência e a manutenção de atualização possam ser planejados e orçados com bastante antecedência.

Objetivo do Teste	Verificar se o sistema funciona corretamente e sem erros sob as seguintes condições de estresse: Pouca ou nenhuma memória disponível no servidor (RAM e HD). Número máximo real, ou fisicamente suportado, de clientes conectados ou simulado. Múltiplos usuários realizando as mesmas transações contra os mesmos dados ou contas. Volume de transação limite ou transações mistas (umas de alta e outras de baixa desempenho).
Técnica/Método	Usar testes desenvolvidos para Desempenho ou Testes de Carga. Para testar recursos limitados, os testes devem ser executados em uma máquina simples, e as memórias RAM e do drive de disco no Servidor devem ser reduzidas ou limitadas. Para Testes de Estresse permanentes, múltiplos clientes devem ser usados, executando os mesmos testes ou testes complementares, para produzir volumes de transação limite ou de transações mistas (umas de alta e outras de baixa desempenho).
Observações	Para estressar a rede podem ser necessárias ferramentas para carregá-la com mensagens ou pacotes. O armazenamento persistente utilizado pelo sistema deve ser reduzido temporariamente para restringir o espaço disponível para o crescimento do banco de dados.

Quadro 8: Teste de Stress
Fonte: Datasus (2007)

Teste de regressão: Testes de um software que foi modificado ou de iterações anteriores. Tem como propósito garantir que qualquer falha tenha sido reparada e que nenhuma operação que funcionava anteriormente tenha falhado após os reparos, ou seja, que as novas características adicionadas não criaram problemas com as versões anteriores ou com outros sistemas.

Objetivo do Teste	Verificar se as alterações realizadas geraram alguma inconsistência no aplicativo ou em outros sistemas.
Técnica/Método	Os scripts desenvolvidos anteriormente são executados progressivamente para verificar que nenhuma falha foi introduzida após as mudanças.
Observações	

Quadro 9: Teste de regressão
Fonte: Datasus (2007)

Teste de integridade: Testes destinados a avaliar a robustez do objetivo do teste (resistência a falhas) e a compatibilidade técnica em relação à linguagem, sintaxe e utilização de recursos. Esse teste é implementado e executado em vários objetivos do teste, como unidades e unidades integradas.

Objetivo do Teste	Executar os métodos de acesso à base de dados e regras de negócio independente da interface do usuário, de forma que seja possível observar e registrar o comportamento funcional incorreto ou a corrupção de dados.
Técnica/Método	Executar os métodos de acesso e processos, utilizando em cada um, dados válidos e inválidos ou solicitações de dados. Examinar o banco de dados para garantir que os dados tenham sido povoados como planejado e que todos os eventos ocorreram apropriadamente. Revisar os dados retornados para garantir que estão corretos.
Observações	Os testes podem exigir a disponibilização do ambiente específico de drivers ou do sistema gerenciador de banco de dados, para que os dados sejam diretamente incluídos na base. Caso existam processos que dêem suporte a este teste, os mesmos devem ser invocados manualmente.

Quadro 10 Teste de integridade
Fonte: Datasus (2007)

Teste de estrutura: Testes usados em 2 contextos principais, mas com naturezas bem distintas, tendo em comum idéia e conceito praticamente iguais, destinados a avaliar a adequação do objetivo do teste em relação a seu design e sua formação.

- **Teste de estrutura de Itens Internos do Código.**
Relaciona-se ao teste da estrutura interna do código-fonte do software. Em geral, essa forma de teste de estrutura é realizada como um teste "estático", em oposição a um teste "dinâmico", no qual o software em si não é executado para realizar o teste. As ferramentas de diagnóstico analisam sintaticamente o código-fonte, procurando pontos fracos e erros estruturais e, normalmente, fornecendo uma lista que permita tomar uma ação corretiva subsequente. Em geral, esse é um tipo de teste e de avaliação realizado por desenvolvedores, e não por testadores do sistema.
- **Teste de Estrutura de Sites da Web**

Em geral, esse teste é realizado em aplicativos habilitados para a Web, garantindo que todos os links estejam conectados, que o conteúdo apropriado seja exibido e que não haja conteúdo órfão:

Teste de desempenho: O teste de desempenho mede e avalia o tempo de resposta, o número de transações e outros requisitos sensíveis ao tempo, compara o desempenho de um objetivo do teste (novo ou desconhecido) a um sistema e uma carga de trabalho de referência conhecido.

Objetivo do Teste	Verificar comportamento do sistema para funções de transações ou de negócio designadas sob as seguintes condições: Carga normal de trabalho. Carga limite de trabalho.
Técnica/Método	Usar Procedimentos de Teste desenvolvidos para teste funcional ou de ciclo de negócio. Modificar arquivos de dados para aumentar o número de transações ou os scripts para aumentar o número de iterações em que cada transação ocorre. Os scripts devem ser executados em uma máquina (melhor caso para verificar usuário simples, transações simples) e ser repetido com múltiplos clientes (virtual ou real, veja observações abaixo).
Observações	Um teste de desempenho incluir uma carga de trabalho no servidor. Existem muitos métodos que podem ser usados para se fazer isso, incluindo: Transações realizadas diretamente para o servidor, normalmente na forma de SQL. Criação de carga de usuário "virtual" para simular muitos clientes, normalmente muitas centenas. Ferramentas de Emulação de Terminal Remoto são usadas para carregar a rede com sobrecarga de mensagens. Utilização de múltiplos clientes físicos, cada um executando scripts de teste para aumentar a carga do sistema. Testes de desempenho devem ser realizados em uma máquina dedicada ou em um tempo dedicado. Isso permite controle total e medida precisa. Os bancos de dados usados para testes de desempenho devem ter o tamanho real ou aproximado.

Quadro 11: Teste de desempenho

Fonte: Datasus(2007)

Teste de carga: Tipo de teste de desempenho usado para validar e avaliar a aceitabilidade dos limites operacionais de um sistema de acordo com cargas de trabalho variáveis, enquanto o sistema em teste permanece constante. Em algumas variáveis, a carga de trabalho permanece constante e a configuração do sistema em teste é que varia. Geralmente, as medições são tomadas com base na taxa de transferência de dados da carga de trabalho e no tempo de resposta da transação alinhado. O teste de carga submete o sistema à variação de carga de trabalho para medir e avaliar os comportamentos de desempenho e a sua habilidade de continuar funcionando apropriadamente sob cargas de trabalho diferentes. Também avalia as características de desempenho, assim como tempos de resposta, taxas de transações e outros casos sensíveis ao tempo.

Objetivo do Teste	Verificar o comportamento do sistema sob condições de carga de trabalho variante.
Técnica/Método	Usar testes desenvolvidos para Teste Funcional ou de Ciclo Negócio. Modificar arquivos de dados para aumentar o número de transações ou os testes para aumentar o número de vezes que cada transação ocorre. Variar o número de usuários conectados.
Observações	Os testes de carga devem ser realizados em uma máquina dedicada, simulando condições reais de trabalho, ou em um tempo dedicado. Isto permite controle total e medida precisa. Os bancos de dados usados para teste de carga devem ter o tamanho real ou aproximado.

Quadro 12: Teste de carga
Fonte: Datasus (2007)

Teste de contenção: Testes destinados a verificar se os objetivos do teste podem lidar de forma aceitável com as demandas múltiplas de vários atores no mesmo recurso, (registros de dados, memória, etc.).

Teste Perfil do Desempenho: Teste em que o perfil de andamento do objetivo do teste é monitorado (inclusive fluxo de execução, acesso a dados e chamadas de função e de sistema), a fim de identificar e lidar com gargalos de desempenho e processos ineficientes.

Teste de Instalação: Teste destinado a garantir que o objetivo do teste seja instalado conforme o esperado em diferentes configurações de hardware e/ou software e sob diferentes condições (como no caso de espaço insuficiente em disco ou interrupção de energia), verificando que, uma vez instalado, o software funciona corretamente. Esse teste é implementado e executado em aplicativos e sistemas.

Objetivo do Teste	Verificar se o software é instalado corretamente em cada configuração de hardware exigida, sob as seguintes condições: Nova instalação, uma nova máquina, nunca tendo sido instalado o sistema antes. Atualização, máquina tendo previamente instalado o sistema, mesma versão Atualização, máquina tendo previamente instalado o sistema, versão mais velha.
Técnica/Método	Realizar a instalação. Utilizar um conjunto predeterminado de scripts de testes funcionais.
Observações	

Quadro 13: Teste de instalação
Fonte: Datasus (2007)

Teste de Configuração: Teste destinado a garantir que o objetivo do teste funcione conforme o esperado em diferentes configurações de hardware e/ou software, as especificações de hardware e software podem mudar e esta mudança pode afetar o sistema a ser desenvolvido. Esse teste também pode ser implementado como um teste de desempenho do sistema.

Objetivo do Teste	Verificar se o alvo de teste funciona corretamente nas configurações de hardware e software exigidos.
Técnica/Método	Usar scripts de testes funcionais. Abrir e fechar vários softwares relacionados que não sejam alvos de teste (por exemplo, Microsoft® Excel e/ ou Microsoft® Word), antes ou durante a execução dos testes. Repetir os processos acima, minimizando a memória convencional disponível na estação de trabalho cliente, quando aplicável Efetuar testes em todos os ambientes definidos em Especificações Suplementares.
Observações	Quais aplicações são tipicamente utilizadas? Quais dados as aplicações estão administrando; por exemplo, uma grande planilha eletrônica aberta no Excel ou um documento de 100 páginas no Word? O sistema operacional de rede, os servidores da rede, bancos de dados, etc. devem também ser mencionados como parte desse teste.

Quadro 14: Teste de Configuração
Fonte: Datasus (2007)

2.3 EXTREMME PROGRAMMING (XP)

Beck (1999) explica que Extreme Programming (XP) foi concebido e desenvolvido para atender às necessidades específicas de desenvolvimento de software conduzido por equipes pequenas, face à vaga e evolução das exigências. Esta nova metodologia leve convencional enfrenta muitos desafios fundamentais, incluindo os de longa data, pressuposto de que o custo de mudar de uma peça de software necessariamente aumenta drasticamente ao longo do tempo. XP reconhece que os projetos têm de trabalhar para conseguir essa redução de custo de explorar as economias depois de terem sido obtidos. Os fundamentos do XP incluem:

- A distinção entre as decisões que vierem a ser feita por interesses comerciais e as que devem ser feitas por partes interessadas no projeto.
- Escrevendo testes unitários antes de programação e manutenção de todos os ensaios publicados o tempo todo.
- Integrar e testar o conjunto do sistema - várias vezes ao dia.
- Produzindo todos os softwares em pares, dois programadores em uma tela.
- Começando projetos com um design simples que evolui constantemente a necessária flexibilidade para adicionar e remover complexidades desnecessárias.
- Colocar um sistema mínimo em produção crescendo rapidamente e que, independentemente da direção provar mais valioso.

Jeffries (2001) conceitua *Extreme Programming* como uma disciplina de desenvolvimento de software com base em valores de simplicidade, comunicação, *feedback*, e coragem. Ela funciona com foco em agilidade de equipes e qualidade de projetos, na presença de práticas simples, com feedback suficiente para permitir que a equipe veja onde estão e possam sintonizar as práticas à sua situação única.

Dentro dos princípios ou práticas da XP o teste vem primeiro, os programadores continuamente escrevem unidades de teste e os executam para que o processo de desenvolvimento continue. Em cada ciclo os programadores apresentam uma versão testável do software ao cliente. Os clientes escrevem testes para validar o sistema. O feedback que no XP nada mais é que o contato incessante com o cliente a respeito do projeto. O teste também tem papel nesse princípio, pois informações sobre o código são dadas por testes periódicos, destacando tanto os erros individuais quanto do software de maneira integrada (VIANNA; DESCHAMPS, 2008).

Para Molinari (2003) fica a dúvida se a revolução de testes de software chegou ao mundo dos programadores, ou se o processo de testes evoluiu tanto ao ponto de se pregar sua própria extinção silenciosa, ele enfoca o impacto causado pela Extreme Programming (XP) em testes de software, bem como eles se relacionam. A responsabilidade de realizar os testes de software na (XP) fica a cargo do programador, já os testes de aceitação são responsabilidades do “consumidor” (que pode ser qualquer um na prática) desde que tenha o poder de validar o negócio em mãos. Este tipo de desenvolvimento segundo alguns autores inclui testes de unidade e de aceitação (usuário). As vantagens abordadas no ponto de vista da XP para com o teste de software vão desde que o teste de unidade que fica melhor sendo feito pelo programador, mas não podendo esquecer os demais tipos de teste, tais como caixa-preta.

Equipes XP utilizam uma técnica conhecida como desenvolvimento orientado a testes. Ela consiste em escrever um mecanismo de teste automatizado antes de codificar cada história e cada método do sistema, conforme (BECK, 1999). A figura 10 exemplifica claramente o desenvolvimento orientado a testes no XP.

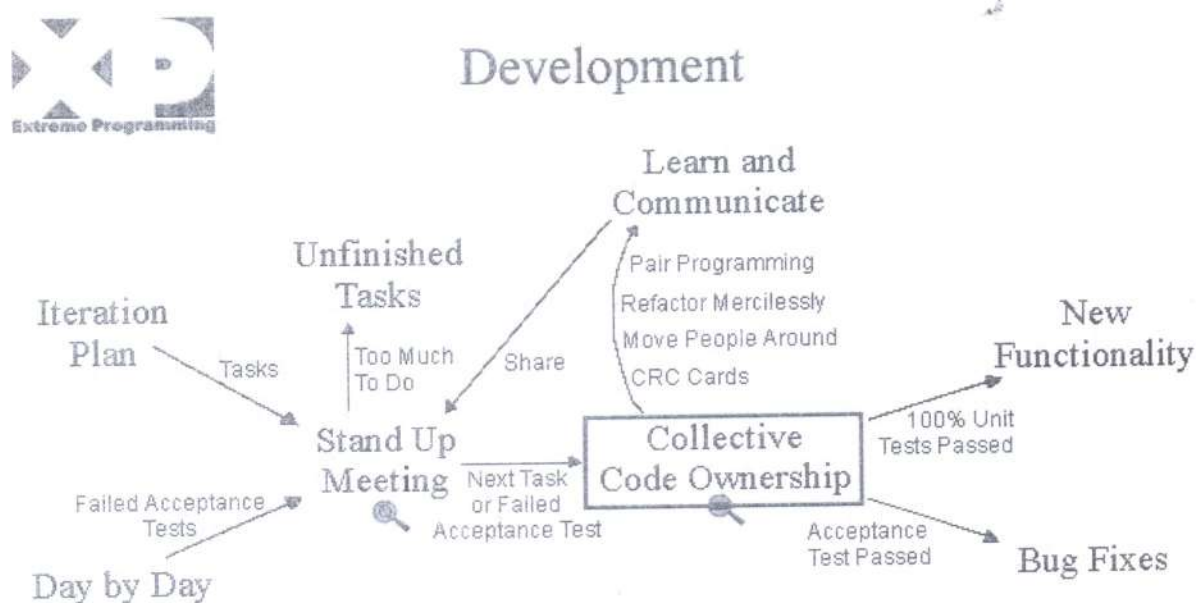


Figura 10: Modelo de desenvolvimento orientado a testes no XP
Fonte: Jeffreis (2008)

O XP se concentra, sobretudo em dois tipos de testes: o teste de unidade e o teste de aceitação. O primeiro tenta assegurar que cada componente do sistema funcione corretamente. O segundo procura testar a interação entre os componentes, as quais dão origem às funcionalidades (BECK, 1999).

Testes de Unidade no XP:

- São escritos pelos desenvolvedores enquanto codificam o sistema.
- Devem ser automatizados e rápidos de executar
- Devem ser criados para todas as classes do sistema.
- São escritos antes da implementação (TDD).
- São um meio de comunicação entre os desenvolvedores,
- São a fonte de coragem para realizar a fatoração.

Originalmente os testes de aceitação são chamados de teste de funcionalidade, no XP são tratados da seguinte maneira:

- São escritos pelo cliente.
- São escritos no momento da escrita da história.
- O cliente sabe quando uma funcionalidade é terminada.
- Para cada história deve existir pelo menos um teste de aceitação.
- Uma história só é declarada terminada quando seus testes de aceitação passarem pelo usuário.

Durante uma iteração dos usuários histórias são selecionados para os testes, o cliente especifica cenários de teste quando a história do usuário for executada corretamente. Uma história pode ter uma aceitação ou muitos testes. Durante uma iteração dos usuários histórias são selecionados para os testes, o cliente especifica cenários de teste quando a história do usuário for executada corretamente. Uma história pode ter uma aceitação ou muitos testes.

Ensaio de aceitação são testes sistema de caixa preta, cada teste de aceitação representa algum resultado esperado a partir do sistema, os clientes são responsáveis por verificar a exatidão dos testes e rever os resultados para decidir a partir das falhas as que são de alta prioridade.

Testes de aceitação devem ser automatizadas para que eles possam ser executados com frequência.

2.4 TEST-DRIVEN DEVELOPMENT (TDD)

Para Santos (2008) Test-Driven Development (TDD) é uma metodologia que utiliza os testes para auxiliar a tomada de decisão por parte dos desenvolvedores na hora certa. Deixando claro que TDD não é somente testes, mas sim como utilizar os testes para desenvolver software de uma maneira simples e incremental, pois além de maximizar a qualidade e o design do software, também simplifica o processo de desenvolvimento.

Beck (2002) enfatiza que o código tem que ser testável para que o desenvolvedor tenha o controle das mudanças, porque mudanças podem ser sinônimos de novo bugs. TDD trata-se de uma abordagem evolutiva para o desenvolvimento que combina, primeiro deve-se testar antes de escrever o código, evitando erros futuros e efetuando um ganho em produção, em outras palavras é uma maneira de pensar através do seu projeto antes de escrever seu código funcional, escrevendo dessa forma um código limpo e que funciona.

Para Vanderlei (2008) no método TDD tudo deve ser testado sempre e não fazê-lo em qualquer situação é alarmante. Para adicionar uma funcionalidade a um sistema que não foi desenvolvido sob o conceito de testar antes de programar, antes mesmo de se desenvolver o caso de teste da nova funcionalidade, é necessário escrever casos de teste para o que se espera que o sistema já faça. Os resultados desses testes serão utilizados para avaliar o impacto da alteração que começará a ser desenvolvida por meio de novos casos de teste. Outra situação importante é a correção de *bugs*, para a qual também se deve escrever um caso de teste para demonstrá-lo antes de iniciar a correção do código.

O desenvolvimento de sistemas pela abordagem TDD geralmente é auxiliado pelo uso de ferramentas de teste de unidades que estão disponíveis no mercado para diversas plataformas de desenvolvimento. Estas ferramentas em geral disponibilizam bibliotecas de classes e métodos que auxiliam a construção de rotinas de teste. O princípio básico do teste de unidade é permitir o teste das menores unidades de código possíveis comparando os resultados de cada operação com os valores esperados (HUNT apud VANDERLEI, 2000).

2.5 FERRAMENTAS DE TESTES AUTOMÁTICOS

Rational (2002) aborda que ferramentas de automatização de teste estão sendo lançadas cada vez mais no mercado para automatizar as atividades em teste. Existem várias ferramentas de automatização, sendo improvável que uma única ferramenta seja capaz de automatizar todas as atividades de teste. Boa parte das ferramentas enfoca uma determinada atividade ou grupo de atividades, e algumas só abordam um aspecto de uma atividade.

Ao avaliar as diversas ferramentas de automatização de teste, é importante conhecer o tipo de ferramenta que está sendo avaliada, suas limitações e quais atividades são abordadas e automatizadas por ela.

Geralmente, as ferramentas de teste são avaliadas e adquiridas com base nas seguintes categorias:

- **Função**

As ferramentas de teste podem ser caracterizadas pela função que executam. Estas são as designações de funções comuns para ferramentas:

Ferramentas de aquisição de dados que adquirem os dados a serem usados nas atividades de teste. É possível adquirir dados através da conversão, extração, transformação ou captura dos dados existentes, ou através da geração a partir de casos de uso ou especificações suplementares.

Ferramentas de medição estática que analisam as informações contidas nos modelos de design, no código-fonte ou em outras fontes fixas. A análise produz informações sobre o fluxo lógico, o fluxo de dados ou as métricas de qualidade, como complexidade, capacidade de manutenção ou linhas de código.

Ferramentas de medição dinâmicas que executam uma análise durante a execução do código. As medições incluem a operação do código em tempo de execução, como desempenho e detecção de erros na memória.

Simuladores ou Geradores que executam atividades que não estão ou não podem estar disponíveis para fins de teste, por questão de tempo, custo ou segurança.

Ferramentas de gerenciamento de teste que auxiliam no planejamento, design, implementação, execução, avaliação e gerenciamento dos artefatos ou atividades de teste.

- **Caixa branca versus Caixa Preta**

As ferramentas de teste costumam ser caracterizadas como de caixa branca ou de caixa preta, com base na maneira como são usadas ou na tecnologia/conhecimento necessários para usá-las.

As ferramentas de caixa branca utilizam o conhecimento do código, dos modelos de design ou de outros materiais de origem para implementar e executar os testes.

As ferramentas de caixa preta só utilizam os casos de uso ou a descrição funcional do objetivo do teste. Enquanto as ferramentas de caixa branca têm conhecimento de como o objetivo do teste processa a solicitação, as ferramentas de caixa preta utilizam as condições de entrada e saída para avaliar o teste.

- **Especialização**

Além das amplas classificações de ferramentas apresentadas anteriormente, elas também podem ser classificadas por especialização:

As ferramentas de registro/reprodução combinam a aquisição de dados com a medição dinâmica. Os dados de teste são adquiridos durante o registro de eventos (na implementação do teste). Mais tarde, durante a execução do teste, os dados são usados para "reproduzir" o script de teste usado para avaliar a execução do objetivo do teste.

As ferramentas de medição de qualidade são ferramentas de medição estáticas que executam uma análise estática dos modelos de design ou do código-fonte para estabelecer um conjunto de parâmetros que descreva a qualidade do objetivo do teste. Os parâmetros podem indicar confiabilidade, complexidade, capacidade de manutenção ou outras medidas de qualidade.

As ferramentas de monitoramento de cobertura indicam a abrangência do teste, identificando quanto do objetivo do teste foi coberto, em alguma dimensão, durante o teste. As classes comuns de cobertura incluem casos de uso baseados em requisitos, ramificação ou nó lógicos (baseados em código), estado dos dados e pontos de função.

Os geradores de caso de teste automatizam a geração dos dados de teste. Eles usam uma especificação formal das entradas de dados do objetivo do teste ou os modelos de design/código-fonte para produzir dados de teste que testem as entradas nominais, entradas de erro e casos de limite e fronteira.

As ferramentas de comparação comparam os resultados do teste com os resultados de referência e identificam as diferenças. Os comparadores diferem quanto

à especificidade para determinados formatos de dados. Por exemplo, os comparadores podem ser baseados em pixels (para comparar imagens de bitmap) ou em objetos (para comparar propriedades ou dados do objeto).

Os extratores de dados fornecem entradas para casos de teste provenientes de fontes existentes. As fontes incluem bancos de dados, streams de dados em um sistema de comunicação, relatórios ou modelos de design/código-fonte.

Molinari (2003) explica que existem duas concorrências dentro das ferramentas de teste, uma comercial focando as funcionalidades e outra acadêmica focando a forma básica da concepção.

Nogueira (2008) explana sobre o mercado de ferramentas disponíveis no mercado apresentando as ferramentas comerciais mais importantes conforme figura a seguir:

Fornecedores comerciais mais importantes

Ferramenta	Tipo de licença	
	Per named user	Floating
AutomatedQA TestComplete Enterprise	US\$ 1.999	US\$ 4.499
IBM Rational Functional Tester	US\$ 7.139	US\$ 13.780
Borland SilkTest	US\$ 7.500	US\$ 15.000
Mercury Quick Test Pro	US\$ 10.957	US\$ 20.766

Figura 10: Fornecedores mais importantes (Ferramentas Comerciais)
Fonte: Nogueira(2008)

Para Nogueira (2008) as ferramentas OpenSource (Código aberto) mais importantes do mercado podem ser retratadas conforme as figuras a seguir:

Fornecedores free/openSource mais importantes

Ferramentas de Automação Funcional

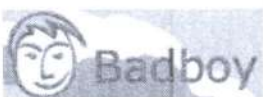
			
Nome	BadBoy	Selenium	Watir
Site	http://www.badboy.com.au/	http://selenium.openqa.org/	http://wtr.rubyforge.org
Tecnologia	Web	Web	Web
Finalidade	Automação por capture/replay na Web com suporte a Ajax	Suite de ferramentas para automação de páginas Web via capture/replay	Automação de testes para páginas Web via programação (Ruby)

Figura 11: Ferramentas Open Source
Fonte: Nogueira(2008)

Tratando ainda de ferramentas código aberto, podem destacar algumas para testes de performance, carga e stress, conforme ilustração seguinte:

Fornecedores free/openSource mais importantes

Ferramentas de Performance, Carga e Stress

			
Nome	JMeter	OpenSTA	WebLOAD
Site	http://jakarta.apache.org/jmeter/	http://www.opensta.org/	http://www.webload.org/
Tecnologia	Java	C++	C++
Finalidade	Performance para conteúdo estático e dinâmico para Java	Performance baseados em arquitetura CORBA	Performance para aplicações Web

Figura 12: Ferramentas Open Source, Performance, Carga e Stress
Fonte: Nogueira (2008)

2.6 JUNIT

O JUnit é um framework código aberto, criado por Eric Gamma e Kent Beck, com desenvolvimento orientado para teste em qualquer código baseado em Java (BECK, 2004). Para Serra (2008) com o JUnit, o programador terá em suas mãos uma ferramenta poderosa que pode ajudar a eliminar boa parte dos erros de seu código, de uma maneira muito fácil.

Oliveira (2005) ressalta que o teste unitário ou de unidade é a modalidade de teste que pode ser considerada como estágio inicial da cadeia de testes, pois testa o menor dos componentes de um sistema de maneira isolada, o JUnit possibilita essa criação e execução de testes unitários em Java, através da criação de classes de teste.

Para Beck (2004) JUnit automatiza os testes de unidade, reduzindo o esforço exigido para testar código com frequência, ao mesmo tempo desenvolver.

Vantagens do uso do JUnit (VICENTE DE NETO, 2008):

- Criação rápida do código de teste, aumentando a qualidade do desenvolvimento e teste.
- Muito utilizado pelos desenvolvedores da comunidade código-aberto, havendo bastante material livre disponível.
- Depois de escritos, os testes são executados rapidamente sem necessidade de interrupção do processo de desenvolvimento.
- JUnit checa e fornece uma resposta imediata dos testes.
- Livre e orientado a objetos.

Beck (2004) ressalta que utilizando JUnit, o programador terá uma ferramenta muito poderosa que o ajudará a eliminar todos (ou quase todos) os *bugs* de seu código de maneira muito “familiar”. Ou seja, os programadores gostam de programar, então criou-se, através do Java, uma forma interessante onde fosse possível a criação de programas que realizassem os testes pelo programador. E é utilizando esse conceito que JUnit permite deixar a fase de teste do desenvolvimento de sistemas bem mais interessante. A seguir figura demonstra a arquitetura do JUnit, indispensável para o entendimento, pois demonstra como as classes se organizam na arquitetura da API (VICENTE DE NETO, 2008):

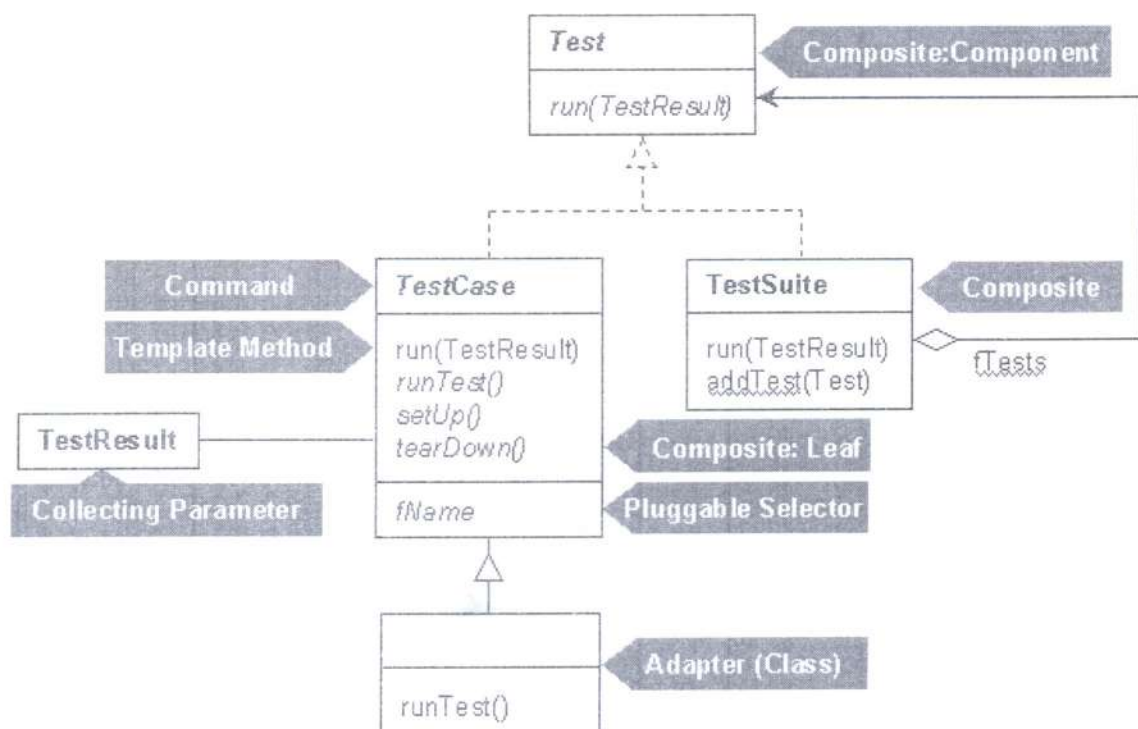


Figura 16: Arquitetura Junit
Fonte: Lopes (2008)

Lopes (2008) detalha cada uma das principais classes e seus respectivos métodos, começando pela classe `TestCase`, contendo os métodos abaixo:

- **command** – O padrão (pattern) permite encapsular um pedido (de teste) como objeto e fornece um método `run()`.
- **run()** – Cria um contexto (método `setUp()`); em seguida executa o código usando um contexto e verifica o resultado (método `runTest()`); e por fim, limpa o contexto (método `tearDown()`).
- **setUp()** – Método chamado antes de cada método, pode ser utilizado para abrir uma conexão de banco de dados.
- **tearDown()** – Método chamado depois de cada método de teste, usado para desfazer o que `setUp()` fez, por exemplo fechar uma conexão de banco de dados.
- **runTest()** – Método responsável por controlar a execução de um teste particular.

Em seguida trataremos da Classe `TestSuite`:

- **composite** – O padrão (pattern) permite tratar objetos individuais e composições de objetos de forma uniforme. Este padrão requer os seguintes participantes:
- **addTest()** – Método responsável por adicionar um novo teste.

Ao executar o JUnit todas as suas expectativas são validadas através dos métodos chamados `assertXxx`, onde `Xxx` pode ser `True`, `False`, `Equals`, ou outra condição. Além disso, pode ser passada para esses métodos alguma mensagem específica caso o teste falhe. O JUnit registra todo o caminho percorrido durante as falhas obtidas nos métodos `assertXxx` e relata após a execução de todos os testes. Abaixo segue alguns dos métodos de assertivas que existem no JUnit com a descrição da assinatura e a operação de cada um, conforme figura abaixo:

Para ver a lista completa dos métodos de assertivas consulte <http://www.junit.org/junit/javadoc/index.htm>

Algumas observações a respeito da criação de testes com JUnit devem ser vistas, tais como toda classe de teste deve ser obrigatoriamente uma subclasse da classe `TestCase`, que está declarada no pacote `junit.framework` localizado no arquivo `junit.jar`. O nome da classe de teste pode ser qualquer um (é aconselhável começar com uma palavra que indique que é uma classe de teste seguido do nome da classe que se deseja testar), mas a convenção de nomes usada para seus métodos (unidades de teste) deve seguir o padrão `testNomeDoMetodo()`, e

este por sua vez, deve ser um método público sem argumentos. Seguindo estas regras o JUnit conseguirá testar classes automaticamente sem que, para isso, o programador tenha fazê-lo.

Método	Descrição	Teste passa se
<code>assertEquals(a,b)</code>	Compara dois valores	<code>a.equals(b)</code>
<code>assertFalse(a)</code>	Avalia uma expressão booleana	<code>a == false</code>
<code>assertTrue(a)</code>		<code>a == true</code>
<code>assertNotNull(a)</code>	Compara uma variável com nulo	<code>a != null</code>
<code>assertNull(a)</code>		<code>a == null</code>
<code>assertNotSame(a,b)</code>	Compara dois objetos	<code>a == b</code>
<code>assertSame(a,b)</code>		<code>a != b</code>
<code>fail()</code>	Causa uma falha no teste atual	

Figura 17: Métodos JUnit

Fonte: Lopes (2008)

Um exemplo do uso do JUnit pode ser visto abaixo, comentários detalham cada tipo de teste executados na classe TestPessoa:

Exemplo de uma classe de teste em Java

```
01  import junit.framework.TestCase;
02  /**
03   * Testes de unidade para a classe { @link Pessoa}
04   */
05  public class TestPessoa extends TestCase {
06      /**
07       * Um teste de unidade para verificar se o nome está
08       * formatado corretamente
09       */
10      public void testObterNomeCompleto() {
11          Pessoa p = new Pessoa("Fulano", "Tal");
12          assertEquals("Fulano Tal", p.getNomeCompleto());
13      }
14      /**
15       * Um teste de unidade para verificar se os nulos são
16       * tratados corretamente
17       */
18      public void testNomeEhNulo() {
19          Pessoa p = new Pessoa(null, "Tal");
20          assertEquals("? Tal", p.getNomeCompleto());
21          // Este código só é executado se passar pelo
22          // assertEquals anterior.
23          p = new Pessoa("Fulano", null);
24          assertEquals("Fulano ?", p.getNomeCompleto());
25      }
26  }
```

Quadro 18: Classe de Teste
Fonte: Dados da pesquisa

3 METODOLOGIA

Pelo fato da pesquisa visar instruir e abordar os principais conceitos de teste de software, além de comparar modelos, ferramentas e padrões ela foi executada no ponto de vista de sua natureza como uma pesquisa aplicada, pois gerou conhecimentos para aplicações práticas, soluções de problemas e principalmente uma documentação que servirá como guia para quem desejar trabalhar na área.

Na sua abordagem de pesquisa, ela é qualitativa, pois se relaciona com o mundo real, não sendo utilizadas técnicas estatísticas. A análise indutiva dos seus dados e o processo é o foco da pesquisa.

Como seu objetivo visa explicar e aprofundar os conhecimentos, procurar e identificar fatores está inclusa em pesquisa explicativa.

Em relação aos procedimentos técnicos, é uma pesquisa tanto bibliográfica, pois como a maioria dos projetos, foi elaborada a partir de materiais já publicados, principalmente de livros, também estuda profundamente ferramentas de teste.

A elaboração, seguida pela coleta de dados, foi feita através de livros, sites, artigos e trabalhos já publicados a respeito de testes de software. Os testes unitários foram realizados ao final do projeto.

4 ANÁLISE DE RESULTADO

Após a realização da pesquisa, foi constatado que existe uma quantidade enorme de ferramentas de teste de software, estando as mesmas separadas por tipo e categoria, o referido estudo demonstra as principais ferramentas trabalhadas na pesquisa.

Entre os principais fornecedores comerciais do mercado destacam-se:

- **TestComplete** da empresa americana AutomatedQA que oferece suporte nativo para a automação de sistemas desenvolvidos em várias tecnologias, tais como .NET, Java, Visual Basic, Borland Delphi, WEB, PowerBuilder, FoxPro, entre outros. Além disso, o TestComplete oferece recursos para a realização de testes de desempenho para aplicações WEB e testes funcionais de WEB Services, Unidade avançada de testes de apoio, sendo possível estender os projetos TestComplete com testes unitários com o criado MSTest, JUnit, NUnit ou DUnit testes, suporte para testes manuais, TestComplete mostra uma janela exploração texto que descreve ações do usuário para ser executado. Após realizar as ações, um testador pode clicar no botão Sucesso ou Falha para confirmar se o teste falhou ou foi executada com sucesso, sendo os resultados dos testes registrados na execução do teste log.
- **Rational Functional Tester** da IBM ferramenta de teste automatizado orientada a objetos que testa aplicativos Windows, .NET, Java, HTML, Siebel, SAP, AJAX, PowerBuilder e Flex, e permite a gravação de scripts confiáveis e consistentes que podem ser reproduzidos para validar novas construções de um aplicativo de teste. O Rational Functional Tester é executado em sistemas operacionais Microsoft Windows 2000, Windows XP, Windows Vista e Linux. Está disponível em dois ambientes de desenvolvimento integrado e em duas linguagens de script. O Rational Functional Tester, Java Scripting, utiliza a linguagem Java e o IBM Rational Software Delivery Platform. O Rational Functional Tester, VB.NET Scripting, utiliza a linguagem VB.NET e o ambiente de desenvolvimento Microsoft Visual Studio .NET.
- **Borland SilkTest** da Borland ferramenta de testes de software automatizados funcionais e de regressão, tendo como características principais: Criação de relatórios de qualidade e suporte a decisão baseados na web, gerenciamentos de ambiente de teste e de problemas baseada na web, registrando defeitos,

controlando status e monitorando a resolução. Planejamento dos testes, suporte a diversos tipos de testes tais como unitários, funcionais e de performance.

- **Quick Test Pro** ferramenta de teste da Mercury que tem como características realizar testes de unidade e integração, usando como método caixa preta, abrangendo as categorias de teste funcional, volume carga, suportando testes a aplicações web e desktop, através de scripts de teste, tendo como resultado geração de arquivo de log.

Entre as ferramentas open-source destacam-se:

- **Badboy** ferramenta para teste de aplicações web desenvolvida em C++, tem como características o suporte a testes do tipo capture-replay, além de testar a medição de desempenho e carga de sites, com ela é possível alterar parâmetros das paginas testadas, efetuar asserções por texto ou por html, tais como cor, javascript e etc. Apresenta como vantagens primeiramente o fato de ser freeware dispõe de uma boa usabilidade, documentação e é extensível (plugins).
- **Selenium-IDE** é uma ferramenta de testes funcionais para aplicações web que implementa o JSFTR (JavaScript Functional Test Runner), que é um framework que executa na página a ser testada, a fim de automatizar esta tarefa. A ferramenta se apresenta como uma extensão do Firefox e depende do navegador para funcionar. Também é disponibilizado o núcleo da Selenium para ser executado independentemente de plataforma e do Firefox. Neste caso, não é disponibilizada uma interface para configuração e execução dos testes, somente o JSFTR.
- **Watir** que significa "Web Application Testing in Ruby", ferramenta que executa programaticamente interações com um browser da mesma maneira que as pessoas interagem, desenvolvida em Ruby linguagem open source focada em alta produtividade e simplicidade, através do Watir é possível simular nos navegadores cliques em links, botões de confirmação, preenchimento de formulário e etc.
- **JMeter** ferramenta do grupo apache, usada para testar o desempenho tanto em recursos estáticos e dinâmicos como os arquivos estáticos, Java Servlets, scripts CGI, objetos Java, base de dados servidores de FTP, além disso pode

usado para simular uma carga pesada em um servidor, rede ou objeto para testar a sua força ou na análise de desempenho global no âmbito de diferentes tipos de carga, tendo como foco principal (testes de performance, carga e stress).

- **OpenSTA** Ferramenta distribuída de forma popular, software de fonte aberta que realiza scripts de teste HTTP e HTTPS, carga pesada, com medições de desempenho, oferece também módulo de coleta de dados com temporizadores, desempenho do Windows e estatísticas.
- **WebLoad** Ferramenta de testes de carga e performance, suporta os protocolos HTTP/HTTPS, incluindo um gravador baseado em proxy para capturar tráfegos registrados no HTTP/HTTPS, suporte para executar sessões de teste de linha de comando a partir de uma interface (e não a partir do console), apresenta relatório de teste de análise para um aprofundamento do desempenho, com estatísticas para identificar gargalos.

5 CONCLUSÃO

A área de teste de software está em expansão, através da pesquisa conclui-se que existem diversas metodologias e ferramentas para serem trabalhadas, mas vale salientar que o trabalho de escolha da melhor depende do ponto de vista do profissional de teste, pelo fato de agregar qualidade ao software. Infelizmente o material disponível em português ainda é escasso, mas isso abre um campo de pesquisa enorme, no qual a qualificação pode fazer toda diferença, através de uma boa certificação e da maturidade profissional para se adquirir e planejar a melhor estrutura de teste surgem os profissionais de visão que avaliam de acordo com a necessidade todas as ferramentas disponíveis no mercado, mas algumas já se destacam, tal como JUnit na área de testes unitários principalmente pelo fato de ser software livre Java , bem como Selenium nos testes de automação funcional de páginas web, além das gigantes comerciais TestComplete da AutomatedQA, Funcional Tester da IBM e a Quick Test Pro da Mercury.

REFERENCIAS

AMARAL, Leonardo Albernaz. **Diminuição da intrusão do teste de software em programas paralelos**. 2006. Dissertação (Mestrado em Ciências da Computação) – Pontifícia Universidade Católica do Rio Grande do Sul – PUC, Porto Alegre, 2006. Disponível em: <http://tede.pucrs.br/tde_busca/arquivo.php?codArquivo=442>. Acesso em: 09 maio 2008.

ASH, Lydia. **The web testing companion: The insider's guide to efficient and effective tests**. Indianapolis, Wiley, 2003.

BARTIÉ, Alexandre. **Garantia da qualidade de software: adquirindo maturidade organizacional**. Rio de Janeiro: Campos, 2002.

BECK, Kent. **Extreme programming explained**. [s.l] Addison-Wesley, 1999.

_____. **JUnit pocket guide**. [s.l] O'Reily, 2004.

_____. **Test-driven development by example**. [s.l] Addison-Wesley, 2002.

DATASUS. **Processo de Teste de Software (PTS)**. 2007. Disponível em: <<http://pts.datasus.gov.br/documentos/SiteProcessoTesteSoftware.pdf>>. Acesso em: 01 jun. 2008.

DIAS NETO, Arilo Cláudio. Introdução a Teste de Software. **Revista Engenharia de Software Magazine**, Rio de Janeiro, n. 01, 2007. Disponível em: <http://kameha.devmedia.com.br/site/engsoft/es_alta.zip>. Acesso em: 09 jun. 2008.

DIAS NETO, Arilo Cláudio; TRAVASSOS, Guilherme Horta. Maraká: Infra-estrutura computacional para apoiar o planejamento e controle dos testes de software. **Anais... V Simpósio Brasileiro de Qualidade de Software – COOPE/Sistemas**, UFRJ - Rio de Janeiro, 2006. Disponível em: <<http://www.sbc.org.br/bibliotecadigital/download.php?paper=865>>. Acesso em: 02 jun. 2003.

GRAF, Clayton; SERRANO, Adriano dos Santos. Teste de Software. **Revista de divulgação técnico-científica do ICPG**, Instituto Catarinense de Pós-Graduação Blumenal/SC, v. 3, n. 09, 2006. Disponível em: <http://www.icpg.com.br/hp/revista/download.exec.php?rpa_chave=bb1f2d7c7ef2d5dbeba7>. Acesso em: 12 jun. 2008.

HUNT, Andrew; THOMAS, David. **Pragmatic unit testing in Java with Junit**. Pragmatic Programmers, Llc [s.l] 2003.

JEFFREIS, Ron. **What is extreme programming**. 2008. Disponível em: <<http://www.xprogramming.com/xpmag/whatisxp.htm>>. Acesso em: 07 maio 2008.

LI, Kang; WU, Menqi. **Effective software test automation: Developing an automated software testing tool**. [s.l] Sybex, 2004.

LOPES, Fred. **JUnit: framework de testes unitários**. 2008. Disponível em: <<http://www.dimap.ufrn.br/~jair/ES/slides/JUnit.pdf>>. Acesso em: 01/11/08.

MCGREGOR, John D; SYKES, David A. **Practical guide to testing object-oriented software**. [s.l] Pearson, 2001.

MOLINARI, Leonardo. **Testes de software: produzindo sistemas melhores e mais confiáveis**. São Paulo: Érica, 2003.

NOGUEIRA, Elias. Automação de Testes: Mitos e Verdades. **Anais...1º Seminário Catarinense de Qualidade e Teste de Software**, SENAI/Ctai, Florianópolis, SC, 2008. Disponível em: <http://scqts.com.br/apresentacoes/Elias_Nogueira.pdf>. Acesso em 10/11/08.

OLIVEIRA, Eric C M. **Java: Testes Unitários e JUnit**. Artigo, 2005. Disponível em: <<http://www.linhadecodigo.com.br/Artigo.aspx?id=576>>. Acesso em: 10 jun. 2008.

PATTON, Ron. **Software testing**. [s.l] Sams, 2005.

PERRY, Willian E. **Effective methods for software testing third edition**. Indiana, Wiley, 2006.

RATIONAL, Unified Process. **Rational unified process**. 2002. Disponível em: <<http://www.wthreex.com/rup/>>. Acesso em: 10 jun. 2008.

SANTOS, Wagner Roberto dos. **TDD – Test Driven Development**. Artigo.[s.d] Disponível em: <<http://netfeijao.blogspot.com/2008/01/tdd-test-driven-development.html>>. Acesso em: 10 jun. 2008.

SERRA, Paulo Roberto Furtado; SERRA, Antonio de Barros; MOURA, Hélio Augusto Sabóia. **Junit** - Uma Solução Java para a realização de testes de programação. Fortaleza, FLF/Faculdade Lourenço Filho, 2003. Disponível em: <<http://www.flf.edu.br/midias/FLF.EDU/25.pdf>>. Acesso em 20/11/08.

TONIAZZO, José Carlos. **Extreme programming**: Uma abordagem em testes de software utilizando XUnit. Monografia (Pós-Graduação em Engenharia de Software) – Universidade Comunitária Regional de Chapecó, Chapecó - SC, 2007. Disponível em: <http://zetoniazzo.files.wordpress.com/2007/08/monografia_jose_toniazzo.pdf>. Acesso em: 12 jun. 2008.

TOSETTO, Mauro. **Ambiente organizacional para teste de software**. Monografia (Análise de Sistemas) – Universidade do Vale do Rio dos Sinos, São Leopoldo Rio Grande do Sul, 2004. Disponível em: <http://www.inf.unisinos.br/alunos/arquivos/TC_Mauro_Tosetto.pdf>. Acesso em: 15 jun. 2008.

USP. **Utilizando o JUnit**. [s.d] Disponível em <<http://clinmaldb.usp.br/LABD/downloads/Junit.ppt>> Acesso em 03 dez. 2008.

VANDERLEY, Celso Arruda. **Análise comparativa entre os métodos de desenvolvimento de software MDD e TDD**, Artigo Acadêmico, Instituto de Pesquisas Tecnológicas, Mestrando em Engenharia da Computação 2008. Disponível em: <http://celsoarruda.com/index_files/artigoMDD_TDD.pdf>. Acesso em: 10 jun. 2008.

VASCONCELOS, Alexandre Marcos Lins; SOARES, Elifrancis Rodrigues. Adaptando o Rup para análise de cobertura de código. **Anais... I Simpósio Brasileiro de Teste de Software**, Centro de Informática – Universidade Federal de Pernambuco (UFPE) Recife, PE – Brasil, 2007. Disponível em: <<http://www.unibratrec.com.br/sbts/diretorio/Adaptando.pdf>>. Acesso em: 15 jun. 2008.

VIANNA, Leonardo Mello; DESCHAMPS, Alexandro. XP – Extreme Programming. Artigo, **Ápice Engenharia de Software**, 2008. Disponível em: <[http://www.apicesoft.com/common/articles/Apice%20Engenharia%20de%20Software%20-%20XP%20\(Extreme%20Programming\)%20\(Leonardo%20Mello%20Viana\)%20-%20Fevereiro%20de%202008.pdf](http://www.apicesoft.com/common/articles/Apice%20Engenharia%20de%20Software%20-%20XP%20(Extreme%20Programming)%20(Leonardo%20Mello%20Viana)%20-%20Fevereiro%20de%202008.pdf)>. Acesso em: 12 jun. 2008.

VICENTE DE NETO, Aristides. **Criando testes com JUnit**. Artigo, 2008. Disponível em: <http://www.javafree.org/dependencias/tutoriais/testes_junit.pdf>. Acesso em: 01 jun. 2008.

VINAY, Petter Farrel. **Manage software testing**. Boca Raton, FL Auerback, 2008.

ZEZO. **Ferramentas de teste:** Selenium, 2007. Disponível em:
<<http://www.testexpert.com.br/?q=node/591> >. Acesso em: 23 out. 2008.