CLAUDINEI DE OLIVEIRA DOS SANTOS

COMPARAÇÃO DE FERRAMENTAS PARA AUTOMATIZAÇÃO DE TESTE EM DESENVOLVIMENTO ÁGIL

ASSIS

2009

COMPARAÇÃO DE FERRAMENTAS PARA AUTOMATIZAÇÃO DE TESTE EM DESENVOLVIMENTO ÁGIL

CLAUDINEI DE OLIVEIRA DOS SANTOS

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Graduação, analisado pela seguinte comissão examinadora:

Orientador: José Augusto Fabri.

Analisador (1):

Analisador (2):

ASSIS

2009

CLAUDINEI DE OLIVEIRA DOS SANTOS

COMPARAÇÃO DE FERRAMENTAS PARA AUTOMATIZAÇÃO DE TESTE EM DESENVOLVIMENTO ÁGIL

Trabalho de Conclusão de Curso apresentado ao
Instituto Municipal de Ensino Superior de Assis,
como requisito do Curso de Graduação, analisado
pela seguinte comissão examinadora:

Orientador: José Augusto Fabri.

Área de Concentração: Ferramentas para automatização de teste de software.

ASSIS
2009

DEDICATÓRIA

Dedico este trabalho a Deus em primeiro lugar, aos meus pais que não mediram esforços financeiros para que meu sonho de concluir um curso superior tornasse realidade, também dedico este trabalho a mim mesmo que por ter me superado mais uma vez pela força e capacidade de enfrentar os obstáculos com a guia de Deus. Fico muito feliz em poder escrever estas palavras e agradecer de coração a todos que acreditaram em mim.

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter me dado forças para chegar onde estou hoje, e sabedoria para que nos momentos de fraqueza não desistisse dos meus sonhos.

Fica também meus agradecimentos ao meu orientador e professor José Augusto Fabri, que sempre esteve preocupado com o andamento deste trabalho, sanando minhas dúvidas e prepondo novas idéias para um trabalho de qualidade.

Também quero agradecer meus pais Elza e Orides e minha irmã Sonia, que não pouparam esforços para que eu chegasse até aqui, sempre me incentivando nos momentos felizes e tristes, assim me dando a oportunidade de estar escrevendo este trabalho de conclusão de curso, contribuindo com um dos meus milhares de sonhos que não me cansarei até realizá-los.

Agradeço a minha namorada Regiane que sempre procurou me entender, nos momentos em que estava ausente, me dando força carinho e atenção e sendo uma grande amiga até os dias de hoje.

Aos meus amigos de classe e da vida que também contribuíram para que este momento se tornasse uma realidade palpável, a todos meu agradecimento de coração.

Contudo deixo meus agradecimentos a todos que fizeram parte da minha vida não só acadêmica, mas também da minha vida pessoal, que grande parte do que sou hoje todos são presentes, também agradeço meu avô Efraim e minha avó Tereza que muito me ensinou.

RESUMO

Este trabalho tem como objetivo abordar o uso de Desenvolvimento Guiado por Teste (TDD), em desenvolvimento ágil de software, bem como abordar características e técnicas de Extreme Programming mais conhecidas como (XP). Também mostrar ferramentas que facilitam tais operações dentro desta técnica (TDD), onde será apresentado testes laboratoriais entre duas excelentes ferramentas, JUnit e TestNG para a automatização dos mesmos.

Aplicar critérios comparativos como a avaliação heurística proposta por Nielsen[1994], para extrair dados qualitativos de ambas ferramentas, também aplicar critérios quantitativos como:

Teste de desempenho que avalia o quanto a ferramenta é rápida na execução de seus casos de teste.

Complexidade do código que avalia o quanto a ferramenta é capaz de otimizar a escrita de códigos.

Palavras-Chave: TDD, XP, Metodologia Ágeis de Desenvolvimento, JUnit , TestNG.

ABSTRACT

This work has as objective board the Test Driven Development (TDD) in agile software development and to discuss features and techniques of Extreme Programming best known as (XP). Also show tools that facilitate such operations in this technique (TDD), where laboratory tests will be presented between two excellent tools, TestNG and JUnit to automate them.

Apply benchmarks such as heuristic evaluation proposed by Nielsen [1994], to extract qualitative data from both tools, as well as quantitative criteria apply: Performance test that assesses how quickly the tool is in the execution of its test cases.
Complexity of code that evaluates how the tool is able to optimize the writing of codes.

Keywords: TDD, XP, Agile Software Development, JUnit, TestNG.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

INTRODUÇÃO

Para melhorar o desenvolvimento de software com mais qualidade e segurança, as empresas estão apostando em metodologias ágeis de desenvolvimento, que oferecem mais segurança e qualidade no produto final, gerando credibilidade com seus clientes, que por sua vez ficam satisfeitos por terem um produto entregue no prazo determinado junto ao contratante, além do cliente ter acesso a “realises” (pequenas partes funcionais do projeto) prontas ao longo do projeto, oferecendo uma visão mais detalhada de como está ficando o sistema, assim dando o poder do cliente interferir quando algo não o agrada ou não está como foi definido. Dentro deste contexto vamos abordar características e técnicas de Extreme Programming (XP), que tem como objetivo propor técnicas que garante a qualidade e segurança do código desenvolvido, assim modelando um novo conceito no desenvolvimento de software, ao contrário dos métodos tradicionais como modelo “Cascata”, XP propõe mais agilidade, conforto e comunicação face-a-face no desenvolvimento do projeto. O XP utiliza uma grande quantidade de regras, objetivos e ações a serem seguida. Uma destas ações é considerado (a) fundamental para o sucesso da adoção do XP que é chamado de Development Guided by Test, que em português se diz Desenvolvimento Guiado por Teste ou simplesmente (TDD), onde especifica que todo teste deve ser escrito antes da implementação de suas respectivas classes. XP Também sugere o comprimento de algumas premissas que são elas: Comunicação, Simplicidade, Feedback e Coragem.

O conteúdo deste trabalho tem como objetivo oferece uma visão sobre Metodologias Ágeis de Desenvolvimento e como empregar algumas de suas técnicas, englobando as práticas de XP assim como um de seus princípios primordiais o Desenvolvimento Guiado por Teste (TDD), também oferecer dados comparativos de duas diferentes ferramentas para a automatização de testes unitários. Os testes serão realizados em laboratório com a finalidade de observar pontos fracos e fortes de ambas as ferramentas. Gerando informações para possíveis conclusões de qual ferramenta utilizar, e observar pontos onde uma é mais vantajosa que a outra e de como ambas tratam o mesmo roteiro de teste unitário.

1. DESENVOLVIMENTO ÁGIL

Segundo TELES (2004), desenvolvimento Ágil define um conjunto de metodologias ágeis de desenvolvimento e também estabelece uma estrutura conceitual para ser aplicada em engenharia de software. No desenvolvimento ágil de software as características ágeis tenta prevenir problemas futuros que poderiam afetar o projeto se forem descobertos no status final (período de entrega para o cliente), o desenvolvimento ágil presa pelos seus princípios que define que o sistema em si deve se apresentado em pequenas iterações (apresentar funcionalidade funcionando ao cliente) em curto espaço de tempo entre 1 a 4 semanas, dando mais segurança no desenvolvimento onde se pode ter uma aceitação ou rejeição do cliente, existem muitas características em desenvolvimento ágil, mas um de seus principais paradigmas é a comunicação e de preferência face a face, que permite captar com mais detalhe os requisitos que o cliente deseja que o sistema tenha.

Alguns dos princípios de um Desenvolvimento Ágil são definidos por:

- Garantir a satisfação do cliente.
- Gerar software funcional para o cliente dentro de semana e não de meses.
- Simplicidade.
- Cliente presente, menos contratos e documentos.
- União constante entre cliente e desenvolvedor, gerando a confiança.

Metodologias ágeis pregam estes itens acima dentre outros, de que se empregados corretamente refletem na satisfação do cliente. Para garantir que estes passos sejam cumpridos dentro do escopo ágil de desenvolvimento, a empresa tem que estar estruturada e educada com estes princípios. Segundo Castro [2006:21], retrata que, tudo se iniciou com um grupo de engenheiro de software, que se reuniram e descobriram que seus projetos de sucesso eram cruzados em alguns pontos idênticos, quando eles utilizavam os seguintes métodos:

- Indivíduos e iterações entre eles **mais** que processos e ferramentas.

- Software em funcionamento **mais** que documentações abrangentes.
- Colaboração com o cliente **mais** que negociações de contratos.
- Responder a mudanças **mais** que seguir um plano.

Apos levantar estes tópicos que se refletiam em ambos os casos de sucesso de cada engenheiro, que sempre firmaram conhecer o lado direito, mas seguiram o lado esquerdo dos tópicos citados acima, com isso surgiu o manifesto ágil mais conhecido hoje como desenvolvimento ágil.

Com base nas informações encontradas no texto acima, definindo que os projetos que utilizam desenvolvimento ágil adotem um processo iterativo em espiral como mostra a **figura 1**. Onde cada setor da espiral corresponde a ciclos de desenvolvimento que determina um tempo fixo de uma a quatro semanas.

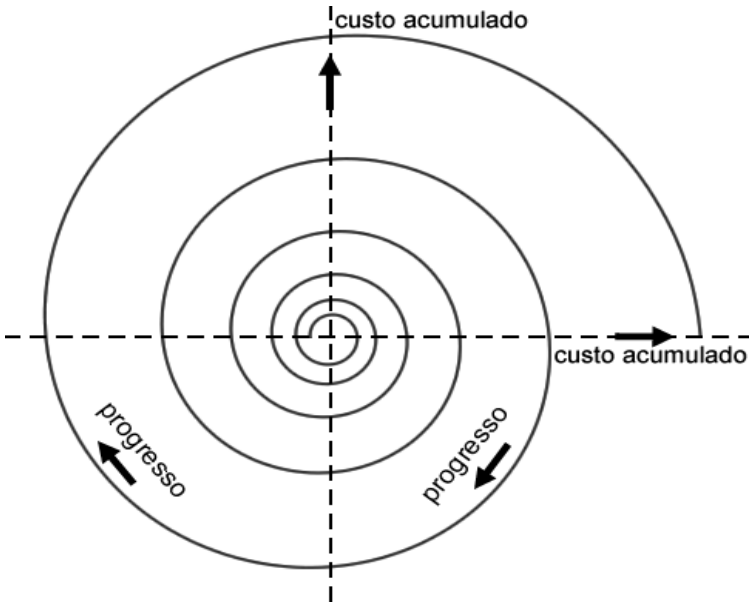

Figura 1. Desenvolvimento Iterativo em espiral (Castro [2006:22]).

Apresentado na **figura 1** pode se perceber que o processo de evolução do software é de uma forma incremental onde cada setor da espiral corresponde a um determinado período de tempo para entregar uma parte funcional para o cliente, facilitando os feedback com mais freqüência por parte do cliente.

2. EXTREME PROGRAMMING.

Segundo Teles (2004), Extreme Programming ou XP é um processo de desenvolvimento de software que agrega características ágeis, para um bom funcionamento das etapas que são estabelecidas para o desenvolvimento do software. Dentro deste escopo, XP aborda técnicas que fazem dos projetos de software um verdadeiro sucesso, quando for bem empregado e seguindo os princípios e práticas do XP. Para que tais processos sejam aceitos, existem algumas limitações para tornar a adoção de XP com mais confiança e qualidade, os requisitos para uma melhor adoção seria:

- Projetos onde requisitos não são especificados de uma forma coerente pelo cliente ou projetos que mudam com freqüência.
- Projetos que utilizam orientação a objeto.
- Um quadro de no máximo 12 (dose) desenvolvedores.
- Desenvolvimento evolucionário, aonde os projetos vão ganhando novas funcionalidades ao decorrer do tempo.

Seguindo estas configurações o projeto tem grande chance de se adequar em técnicas ágeis como XP.

Dentro do XP destacam 4 (quatro) valores essenciais como TELES (2004):

- Feedback.
- Comunicação.
- Simplicidade
- Coragem.

Para aderir uma metodologia como XP o conhecimento de tais questões são indispensável ter o conhecimento: O que é feedback? Para que tanta comunicação? Porque simplicidade? Coragem do que? . Para exemplificar segue a descrição de cada questão proposta por XP:

- **Feedback** é um mecanismo utilizado de cliente para desenvolvedor, onde o cliente logo após ver o resultado de uma release (pequena parte já pronta do sistema) dispara sugestões ao desenvolvedor sobre aquela determinada funcionalidade que lhe foi entregue.
- **Comunicação** é um artefato que dever ser abusado entre os desenvolvedores para que se criem um ambiente onde todos estão “falando a mesma língua”, sendo que está comunicação deve ocorrer face-a-face entre cliente desenvolvedor e entre os próprios desenvolvedores gerando um entendimento de todas as partes de cliente para desenvolvedor e de desenvolvedor para desenvolvedor.
- **Simplicidade** é a forma de como deve ser implementada as funcionalidades de um software, de uma forma clara e suficiente para que atenda os requisitos proposto pelo cliente. Se preocupar com os problemas presentes e os futuros deixa para resolver no futuro é uma boa técnica para aplicar a simplicidade nos códigos desenvolvidos e entregar com mais freqüências partes funcionais para o cliente.
- **Coragem** tem como princípios fazer com que a equipe acredite nas técnicas de XP, e alterar o código do sistema constantemente mesmo sabendo dos riscos que podem ocorrer pelo fato de estarem utilizando casos de testes para cada nova funcionalidade inserida no código.

Seguindo estes valores o XP também sugere uma seqüência de boas praticas para melhorar o planejamento e gerenciamento no decorrer do desenvolvimento do software, que são divididas em três camadas como mostra a **figura 2**.

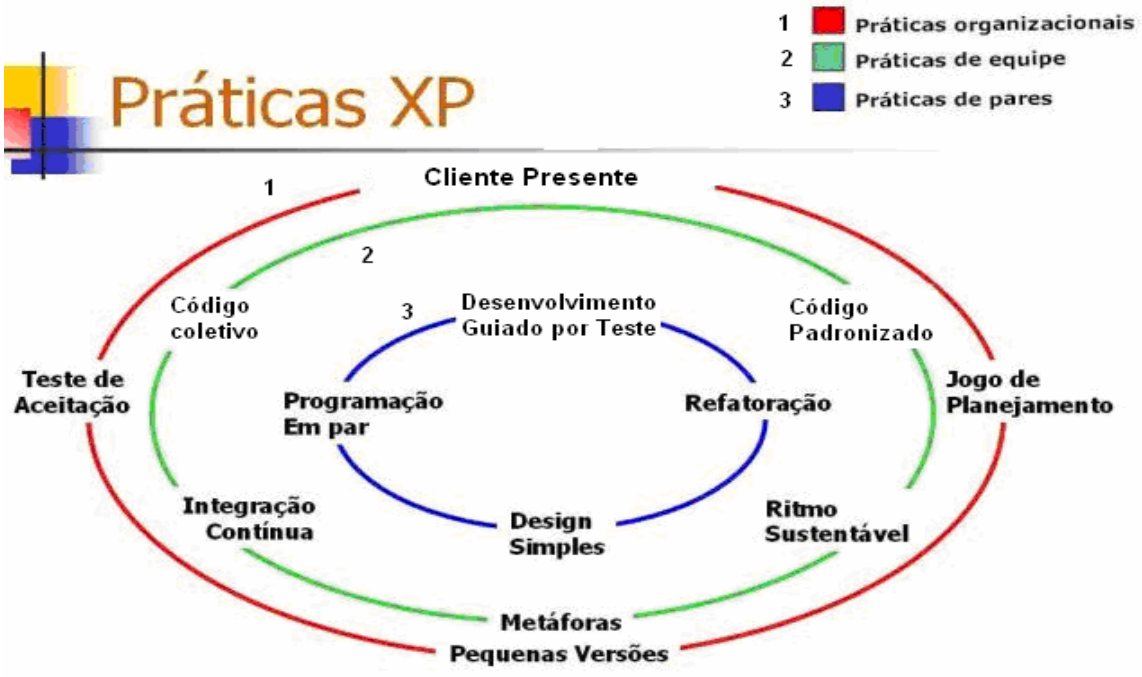

Figura 2. Práticas utilizadas em XP (Adaptado de Medeiros [2009]).

2.1 PRÁTICAS ORGANIZACIONAIS.

- **Cliente Presente** é um mecanismo que ajuda a equipe de desenvolvimento receber feedback constante e com uma comunicação mais eficaz com a técnica face-a-aface e também gerando mais simplicidade e uma aceitação mais rápida por parte do cliente.
- **Jogo de Planejamento** também é uma prática bastante utilizada, principalmente depois de um release ou iteração que é apresentada para o cliente, onde toda a equipe junto com o cliente realiza uma reunião para definirem os próximos passos, que por sua fez e definido pelo cliente em pequenos cartões o que deverá ser entregue nos próximos releases ou iterações, o conteúdo destes cartões é chamado de historias.
- **Pequenas Versões** é um grande técnica para ter um retorno com mais velocidade e eficiência do cliente, as pequenas versões gera um feedback de aceitação ou de aprimoramento, sendo possível corrigir com mais facilidade

um erro de uma parte do código que foi recentemente implementado do que depois de um longo tempo como é feito em modelos tradicionais de software.

2.2 PRÁTICAS DE EQUIPE.

- **Código Coletivo** uma técnica muito utilizada dentro de XP, diferente de outras técnicas onde define que cada desenvolvedor é responsável por uma única tarefa, com isso o código se torna “proprietário” de quem o criou, já com XP o código produzido todos tem acesso e podem alterar quando achar necessário, assim o código fica mais seguro por estar sendo revisado por toda a equipe e gera mais confiança na eliminação de erros.
- **Código Padronizado** seguindo a risca, este tópico garante que a equipe tenha uma comunicação não só face-a-face, mas também na escrita dos códigos, com este padrão definido no início do projeto fica mais fácil a leitura dos códigos desenvolvidos entre as equipes de desenvolvimento.
- **Integração Continua** é viável para garantir uma qualidade no desenvolvimento do software e gerar feedback com mais rapidez, este procedimento deve ser realizado diversas vezes ao dia, diferentes de outras técnicas em que a integração e feita a longo prazo, proporcionando um desconforto maior quando for necessário alguma alteração no código, somando estas interações temos a release como mostra a **figura 3**.

Figura 3. Modelo de um Projeto na forma Incremental (Medeiros [2009]).

- **Ritmo Sustentável** gera para a equipe de desenvolvimento um conforto maior no seu desempenho do trabalho, o ritmo sustentável prega que a equipe como um toldo não excede em seus horários de trabalho assim a equipe sempre esta disposta e mais atenta, com isso gerando menos probabilidade de erros e estresse, ganhando tempo com menos retrabalho que poderia ser gerado sem estes fatores em prática.
- **Metáforas** são utilizadas junto com ritmo sustentável, onde o uso é para explicar de uma forma mais clara e eficaz determinado problema que gera durante o desenvolvimento, e também para explicar partes técnicas ao cliente que for leigo no assunto, com esta prática os desenvolvedores estão sempre exercitando suas criatividades e se interagindo mais sobre o assunto.

2.3 PRÁTICAS DE PARES.

- **Programação em Par** uma das técnicas mais questionada do XP, onde gera desconforto em nível de desenvolvedor para desenvolvedor, que diz não se sentirem bem por estar expondo seu código a outro, mas para quebrar este "tabu" XP afirma que um código desenvolvido a dois, um condutor (pessoa que coloca mão na massa) e um navegador (pessoa que fica o tempo todo revisando o código escrito), a probabilidade de gerar erro futuros é muito menor, e também é uma forma de trocarem experiências e trocar idéias para solucionar diferentes problemas, assim podendo nivelar todos em um mesmo patamar.
- **Desenvolvimento Guiado por Teste** uma técnica não utilizada por desenvolvedores que seguem o modelo tradicional de desenvolvimento de software, onde declaram ser uma atividade muito "chata" e não necessária para um funcionamento correto do software. Para XP esta é uma técnica rigorosamente seguida de uma forma natural pelos desenvolvedores, que sempre criam testes automatizados para assegurar a segurança dos códigos escritos, caso futuramente apareça alguma necessidade de alterações em

um determinado trecho do código. Para este caso é abordado dois tipos de testes um chamado de teste unitário onde o desenvolvedor cria pequenos testes no código protegendo classes e métodos de alteração indesejada que venha a causar erro “bugs” de aceitação, onde é testado o conjunto de todos os testes unitários e é esperado que o retorno satisfaça a história (definida pelo cliente), e seus valores de aceitação.

- **Refactoring** é uma técnica que anda paralela a código coletivo, onde todos têm a obrigação de consertar o código caso o desenvolvedor vem a encontrar duplicidade no mesmo ou com uma dificuldade de leitura, assim visando erros futuros e empregando técnicas ágeis.

- **Design Simples** uma técnica que ajuda o desenvolvedor focar naquilo que foi pedido pelo cliente, nada de tentar “encher o balão com a boca”, alem de perder tempo pode trazer conseqüências mais a frente, então a lógica é programe o necessário pra que as requisições do cliente sejam aceitas do modo mais simples possível.

Para demonstrar como uma estória (Cartão de funcionalidades) se comporta em um projeto de software utilizando Extreme Programming XP a **figura 4** demonstra de uma forma clara cada participante e seus respectivos papéis dentro do grupo.

Figura 4. Ciclo de Vida de uma Estória em XP (Medeiros [2009]).

3. TESTE DE SOFTWARE.

De acordo com Maldonado (et. al.2006), teste de software é uma tarefa que exige muito conhecimento e habilidade para criar ciclos e rotinas para testar determinada parte de um código, sendo assim os testes de unidade é um bom começo para uma evolução, até chegar aos testes de integração onde é centralizando todos os testes de unidade gerando um único teste para validação do software e garantir uma maior confiabilidade no código.

No esquema abaixo como mostra a **figura 5** temos uma modelagem de como os testes estão sendo processados, temos um conjunto de elementos a ser testado representado por CoT (Conjunto Teste), este conjunto de teste é aplicado a uma series de caso de teste representado por CaT (Conjunto Caso Teste), que se relaciona com o programa P, para gerar uma saída, logo após ter entrado com o conjunto de teste CoT para que o testador TS (Testador Software) possa estar conferindo se a saída é o esperado pela especificação gerada pelo cliente, representada por EC (Especificação Cliente), no ultimo estagio representado por D (Decisão), pode ser aceito dois tipo de resposta sucesso ou falha, caso retorne falso o código deve ser depurado e corrigido pela equipe de desenvolvimento, senão começa o mesmo ciclo com novo conjunto de dados(especificação) .

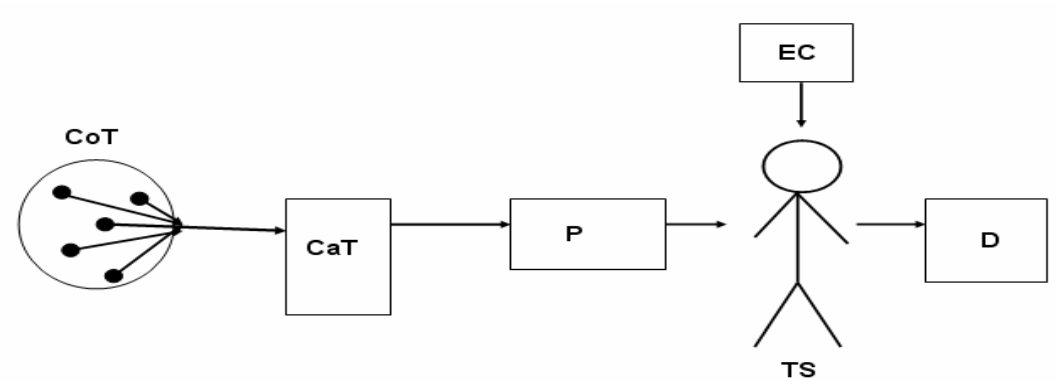

Figura 5. Cenário de um processo de teste (Adaptado de Maldonado[2006:3]).

3.1 DEFEITO, ERRO E FALHA.

Para reforçar o entendimento de teste de software aplicado em uma empresa, Claudio [2008:54-59], cita alguns dos métodos mais utilizados, e define que teste é uma rotina para definir o grau de confiabilidade na execução de um sistema, além de ser uma tarefa onde sua principal atividade é apresentar o maior número de erros possível para se obter um maior nível de confiabilidade, fica definido e diferenciado três conceitos básicos para um entendimento mais solido entre: defeitos, erros e falhas.

- **Defeitos** são cometidos por pessoas que acham que são entendidas de um determinado assunto e acabam determinando caminhos que deveriam ser seguidos de uma forma diferente, ou até mesmo a utilização de ferramentas e métodos inadequados para uma situação.
- **Erros** é uma ação visível da ocorrência de um determinado defeito, erro de um software pode ser um resultado que não coincide com os valores de entrada, ou não é coerente com a especificação definida pelo cliente.
- **Falhas** ocorrem devido a erros inesperados onde o sistema não se apresenta com um comportamento esperado pelo usuário.

A **figura 6** mostra de uma forma mais específica onde cada conceito citado é aplicado e como eles são classificados. Os defeitos ocorrem na parte física (no próprio sistema), que é causado por intervenção humana, com isso gerando erro pelo fato de não atender as necessidades que o cliente especificou, junto com os erros vem às falhas, que por sua vez afeta o comportamento final do sistema, e às vezes inviabilizando o uso do mesmo.

Figura 6. Defeito x Erro x Falha (Claudio [2008:55]).

3.2 TIPOS DE TESTES.

Dentro deste contexto algumas técnicas se destacam entre as mais utilizadas para o desenvolvimento de teste de software, com isso se faz necessário entender algumas das técnicas mais utilizadas no desenvolvimento de software entre elas estão:

Teste Funcional ou mais conhecido como teste caixa preta (Black Box) a **figura 7** enfatiza que os testes devem ser visto de um ponto mais alto, focando com teste no software apenas usando a interface do cliente, onde o testador entra com valores e espera um resultado que esteja presente na definição do cliente, através de uma lista, mais conhecido como (Check List).

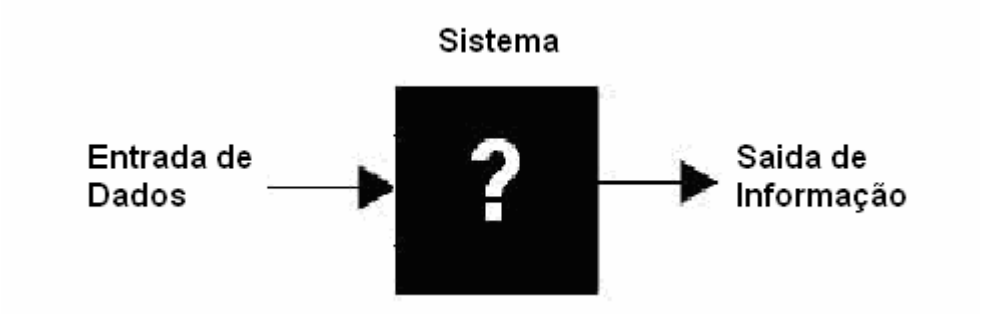

Figura 7 Teste Funcional (Adaptado de Claudio [2008:57]).

Teste Estrutural conhecido como teste caixa branca (White Box) a **figura 8** define que os testes devem ser aplicado minuciosamente direto no código fonte, onde é verificado por testadores com um nível mais elevado de conhecimento na linguagem que foi desenvolvida, assim muitas vezes o próprio programador pode fazer está verificação, desde que conheça técnicas de teste.

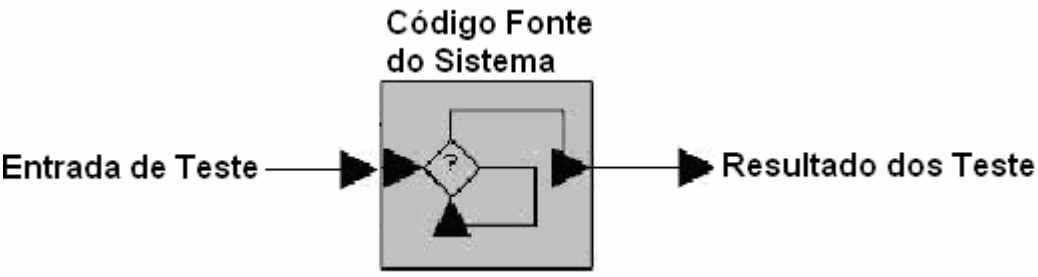

Figura 8. Teste Estrutural (Adaptado de Claudio [2008:57]).

Teste Unitário uma técnica bastante utilizada quando nos deparamos de como testar as classes separadamente uma das outras, e indiferente se uma se relaciona com outra. Seguindo este ponto de vista teste unitário nos deixa muito claro o que devemos testar e o como devemos testar, além de ser uma prática muito importante quando se desenvolve aplicando técnicas ágeis.

Teste de Integração é utilizado quando nos demos conta que todos os testes de unidade já foram realizados, então chegamos a um ponto onde todas as classes e testes irão interagir entre si, assim podemos verificar o comportamento quando integramos tudo e os resultados exibidos pelos testes, para uma conclusão se ocorreu conforme os testes recomendavam.

Teste de Aceitação é realizado junto com o cliente que por sua vez definiu os mesmos junto a um analista de teste, neste ponto o cliente segue um roteiro pré definido por ele ou (ele + analista de teste), e então vai averiguando se tudo que foi descrito está realmente o satisfazendo, e é neste ponto em que o cliente gera feedback para os desenvolvedores sobre aquela determinada funcionalidade que foi implementada assim utilizando este teste para ver se o software realmente funciona.

Teste de Estresse também considerado um dos testes mais importante, é sempre realizado quando todos os outros testes já foram pré aprovado pelo cliente, o sistema é submetido onde ele realmente vai ser executado (ambiente de produção),

ejetando a uma grande carga de dados e de ações executadas ao mesmo tempo com a finalidade de mostrar o quanto o sistema e resistente, a quantidade de processos que executar sem retornar algum tipo de erro (memória, banco de dados, desempenho) e o quanto tempo ele pode agüentar tamanha pressão de dados e execuções simultâneas.

Teste de Software é abordado em todos os projetos de sistema, visando assegurar que uma grande variedade de teste seja aplicada aos seus respectivos módulos para garantir uma qualidade a mais ao software livrando de erros “Bugs” que possa vir a interferir no seu funcionamento do software como um todo.

Em desenvolvimento ágil teste é considerado como um dos requisitos primordiais, quando se utiliza desenvolvimento guiado por teste que será o tema do próximo capítulo, que estuda as complexidade e aplicações junto a XP.

Se empregado de forma correta os testes podem prevenir vários aspectos indesejáveis no processo de desenvolvimento, principalmente quando se trata de custo, quando os testes são deixados para última hora e tratados de forma sem a importância que deveria o software terá grande probabilidade de gerar erro.

Os testes citados acima são uns dos mais utilizados em desenvolvimento de software, junto a estes padrões que vem acompanhado de outros tipos de teste tais como: Teste de Compatibilidade, Teste de conformidade, Teste Funcional, Teste de Carregamento, Teste de Desempenho, Teste de Regressão, Teste Smoke.

4. DESENVOLVIMENTO GUIADO POR TESTE (TDD).

De acordo com Sanchez [2006] que define sobre TDD, no início de seu artigo cita que desenvolvimento guiado por teste é uma técnica que aplica uma maneira contraditória, de como os desenvolvedores estão acostumados a desenvolver códigos, uma comparação simples que afirma está verdade é que quando se pensa em um projeto de software, e de quais são os caminhos ou tarefas a ser executada, as atividades de teste sempre vem em último plano, dando mais oportunidade para que problemas sejam descobertos na fase final do projeto, sendo assim o retrabalho de retomar a linha de raciocino seja mais demorada e mais custosa, aumentando ainda mais o custo do sistema e estourando o prazo de entrega. Para contornar este tipo de problema XP prega que todo o código (Classe e Métodos) a ser desenvolvido deve estar resguardado por um conjunto de teste que garanta que se algum erro for causado por um defeito de programação ou lógica do negócio o mesmo seja reportado imediatamente para o desenvolvedor tomar as devidas providências, e corrigindo o erro com menos tempo do que se fosse para ser corrigido depois de 1 mês, isto define que TDD é realmente uma prática que se torna tão essencial quanto o próprio desenvolvimento do projeto, e indispensável quando se utiliza metodologia ágil como extreme programming XP.

4.1 BENEFÍCIOS.

- **Simplicidade** com o uso de TDD o desenvolvedor sempre vai se preocupar com aquilo que o método deve realmente fazer, com este objetivo em prática o código fica mais limpo e entendível por todos que o observarem.
- **Confiança** quando protegemos nosso código com os testes e de como deverá se comportar, isso gera mais confiança para realizar alterações no código sem ter medo que ocorra um erro e o mesmo seja descoberto em longo prazo, sendo assim podendo corrigi-lo imediatamente e gerando mais um caso de teste para assegurar que o mesmo não volte a ocorrer.

- **Documentação** se definir bem os casos de teste, o mesmo pode ser utilizado como uma documentação não só para o usuário final, mas também para um novo membro do grupo caso alguém vier a sair e também usar está documentação para conhecer as regras do sistema, de uma maneira em que a necessidade de olhar documentos e mais documentos para estar a par de como tudo funciona seja irrelevante.
- **Refactoring** é uma técnica onde o desenvolvedor tem a oportunidade de aperfeiçoar o código escrito, com os testes validados a possibilidade de existir código duplicados ou desnecessário é grande, então aplica se o “refatoring” no código deixando menos poluído e enxuto.
- **Maior Valor Agregado ao Produto** aplicando TDD com todos seus princípios o código fica com menor probabilidade de instabilidade ou de apresentar erro, quando for submetido ao ambiente de produção, quando o cliente sabe que seu projeto foi desenvolvido baseado nos critérios de testes que foi definido, a aceitação é maior e a confiança do uso contínuo prevalece ao produto.

A grande preocupação da técnica TDD é com novos fragmentos de código inserido no projeto, devem ser automatizados e ser executados para verificar se o novo fragmento carrega algum tipo de erro “bugs” e venha a interferir no projeto final, alertando o desenvolvedor quando houver tais ocorrências.

4.2 REGRAS.

TDD segue um padrão que sempre o leva a um conjunto de regra simples e práticas. Como é descrito abaixo:

1. **Escreva um Teste que Falha:** neste princípio a regra é pensar somente de como os teste devem ser implementados e não de como as classes e os métodos de uma estória (especificação) funciona, dividir bem os teste implementando é uma boa técnica, ao invés de tentar resolver tudo em um

único método de teste, procurar descrever bem de como cada método de teste funciona, para quando voltar a ler os métodos somente pelo nome já se saber o que é esperado do método.

2. **Faça o Teste Passar:** todas as classes de testes implementadas e com seus métodos e regras de validação bem especificada, de acordo com a definição imposta pelo cliente, inicia a codificação da classe que vai descrever a estória (especificação) do projeto fazendo com que os testes implementados anteriormente seja executados de forma que todos seja realizado com sucesso, nesta implementação definimos somente as classes, métodos e códigos para que a classe de teste seja válida, sem se preocupar com a quantidade de código necessário para a validação, neste ponto o importante é resolver o problema com base nos testes pré definidos.
3. **Refatore:** neste ponto a visão fica mais focada em como eliminar códigos duplicados, o importante é realizar sempre as alterações com o suporte da classe de teste e sempre os executando, caso uma determinada alteração reflita no mau funcionamento dos testes realizados o desenvolvedor seja notificado com um alerta de que a implementação que foi inserida contém erros “bugs”, quando isso ocorrer não é necessário o pânico pelo simples motivo de saber que o código está protegido pelos testes e sempre vão assegurar que pelo mínimo de erro introduzido ao código o desenvolvedor sempre será notificado, lembrando que para isso ocorrer às validações devem estar implementadas na classe de teste e com uma regra de negócio bem definida e empregada nas classes de teste.

Na **Figura 9** é representado de uma forma mais objetiva e clara de como todos estes processos se integram, para que possam ser aderidas de forma objetiva as técnicas de TDD em desenvolvimento ágil XP.

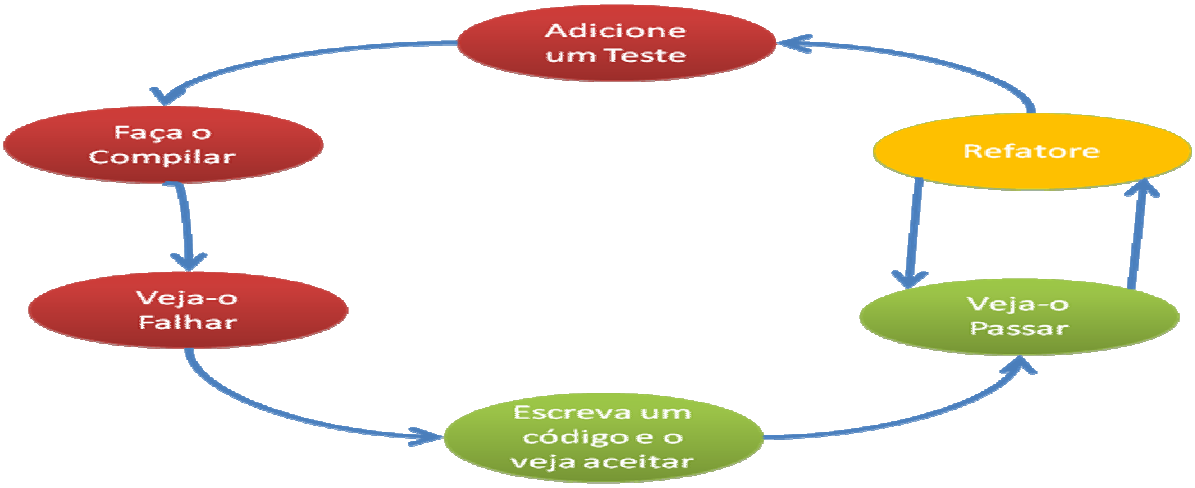

Figura 9. Ciclo de vida no processo em TDD.

Seguindo o esquema demonstrado na **figura 9**, Pinto (2009) descreve que o ciclo de vida de um processo em TDD se resume em 3 critérios básicos e práticos sendo:“Não escreva código de negócios, antes de escrever um teste que falhe, Escreva **somente o suficiente** no teste para demonstrar uma falha, Escreva **somente o suficiente** no código para passar no teste” Pinto (2009).

Extreme programming XP, considera que as práticas abordadas por TDD devem ver utilizadas como um processo indispensável na sua adoção, seguindo a risca o desenvolvimento guiado por teste, para que no final possa desfrutar de todos os benefícios extraídos desta união de XP / TDD.

TDD vem demonstrando a cada caso de sucesso a sua importância dentro do escopo no desenvolvimento de software, assim se firmando cada vez mais e evoluindo a cada dia em seus conceitos, de como cada teste possa ser implementado por todos que desejam aderir técnicas ágeis utilizando XP.

4.3 PROCESSO.

Depois de realizar várias iterações entre estes processos Pinto (2009) representa todo o processo de TDD em um único esquema mostrado na **figura 10**.

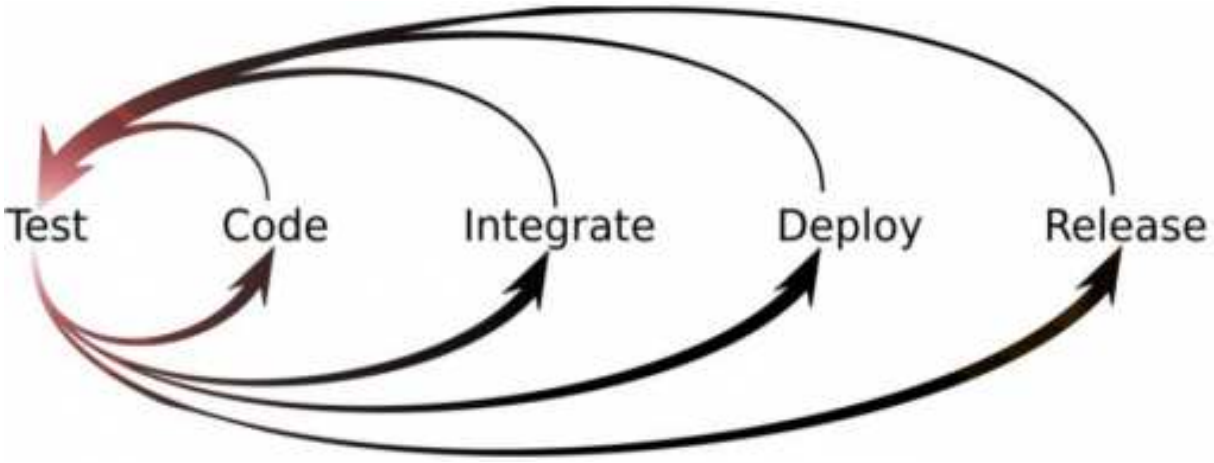

Figura 10. Esquema de um processo completo aplicando TDD (Pinto [2009]).

Na **figura 10** é descrito uma seqüência completa de um projeto implementado usando técnicas de TDD onde cada módulo (Test, Code, Integrate, Deploy, Release) estão em constante integração junto à fase de teste com um único objetivo gerar códigos mais robustos, eficazes e seguros.

TDD é uma técnica baseada em desenvolver testes e depois codificar, para realizar testes de unidade TDD utiliza ferramentas para a automatização dos mesmos, e assegurar que sejam realizados com mais eficiência e segurança. Para abranger o quanto é grande o território de TDD além dos testes de unidade citado acima e no **Capítulo 3**.

Contudo XP e TDD sempre serão aliados muitos fortes para definir o sucesso e o fracasso de cada projeto, a adoção deverá ser completa e aplicada paralelamente a cada conceito definido em ambas as técnicas.

Para realizar testes unitários utilizando TDD, é importante ter algumas sacadas “cartas nas mangas” tipo, saber o que testar e de como testar algumas dicas pode

ser citadas para alcançar o objetivo com uma velocidade maior, as práticas sugeridas são:

- Teste valores que são representados por negativo e positivo, valor grandes também como o próprio zero, estes podem ser um dos primeiro casos para assegurar uma maior qualidade nos casos de teste.
- Teste de Valor Limite assegura que os números mais próximos aos valores válidos, sejam submetidos aos testes para garantir que tal limite não seja quebrado, e seu algoritmo esteja delimitando de forma correta aos seus respectivos valores.
- Teste de String muito grande ou contendo valores nulos, até mesmo muito pequenos ou vazios também pode ser uma boa técnica.

Seguindo estes exemplos dentre outros, as classes de teste ficam mais solidas e robustas em relação à execução dos testes, deixando suas respectivas classes com mais segurança e confiáveis.

5. FERRAMENTAS DE AUTOMATIZAÇÃO DE TESTE

As ferramentas para a automatização de testes veio propor para o desenvolvedor uma maneira simples e eficaz de trabalhar com testes. Em metodologias ágeis com a empregabilidade de técnicas como TDD, o uso de ferramentas para avançar com o processo da criação de testes vem sendo freqüentemente utilizada, o objetivo é ganhar tempo e confiabilidade no desenvolvimento dos casos de teste.

Automatizar os testes de software tem uma grande vantagem, como citado nos **Capítulos 3 e 4,** os testes que são executados com mais freqüência são de unidade e aceitação, que são consideráveis uma atividade importante para garantir que o projeto seja entregue ao cliente com muito pouco a ocorrência de erros “bugs” que possa estar vindo a atrapalhar ou aumentar o custo no desenvolvimento, relembrando:

Teste de Unidade: é onde se cria uma classe de teste específica para cada unidade (classe) correspondente, lembrando que os testes devem ser implementado antes da própria classe, com o único objetivo de forçar o desenvolvedor a pensar de como implementar os testes e quais os resultados esperados.

Teste de Aceitação: também conhecido como teste de caixa preta, onde o cliente é o operador e submete o sistema a uma série de informações e recolhe os resultados, comparando com os esperados validando aquela etapa, este processo é realizado até que o cliente esteja satisfeito e seguro de que o sistema esteja funcionando corretamente, lembrando que está técnica pode ser empregada a cada entrega do produto para o cliente “realises”, com a finalidade de gerar feedback aos desenvolvedores.

Para que os testes tivessem uma melhor qualidade e autonomia, foram criadas ferramentas que auxiliassem no desenvolvimento do mesmo, desde então foram surgindo vários tipos de ferramentas automatizadas, cada uma com a sua peculiaridade em um tipo de teste, na **figura 11** é demonstrado algumas das ferramentas mais utilizadas nos dias de hoje.

Figura 11. Ferramentas para desenvolvimento de teste de software.

Neste conjunto de ferramentas representada na **figura 11** alguns objetivos entre elas são, possibilitar a criação de testes automatizados para que o desenvolvedor possa proteger de alguma maneira cada funcionalidade do projeto, não só em nível de código, mas também em nível de sistema.

Em relação às ferramentas JUnit e TestNG que serão citadas e utilizadas neste trabalho, para realização dos testes unitários, comparar e exercitar pontos em que ambas são utilizadas no mesmo contexto, recolhendo informações que relatam as críticas, vantagens e desvantagens que uma venha ter diferente da outra.

Partindo deste ponto de vista de como as ferramentas de automatização são importantes, e quais serão utilizada neste processo de comparação é importante citar características e funcionalidades das ferramentas JUnit e TestNG.

5.1 JUNIT v4.5.

JUnit é um framework open-source criado por Eric Gamma e Kent Beck, a proposta de Gama e Beck era criar uma ferramenta que auxiliasse na produção de testes, e aumentaria a produtividade em relação a codificação dos testes.

Com a grande gama de tipos de teste, junit se destaca com a criação de teste unitário ou conhecido também com teste de unidade, onde é desenvolvidos pequenos testes para cobrir uma determinada parte do código com o único objetivo de validar cada método que seja considerado a se submeter a uma validação.

Com a execução da classe de teste os resultados são reportados rapidamente para o desenvolvedor, que verificando já sabe se ocorreu algum tipo de erro “bug” durante a execução da classe de teste.

Estes resultados são mostrados de uma forma gráfica, quando ocorrer a barra vermelha **Figuras 12** os testes por algum motivo parou de funcionar na **Figura 13** ocorreu a barra verde significando que os testes estão sendo satisfatórios.

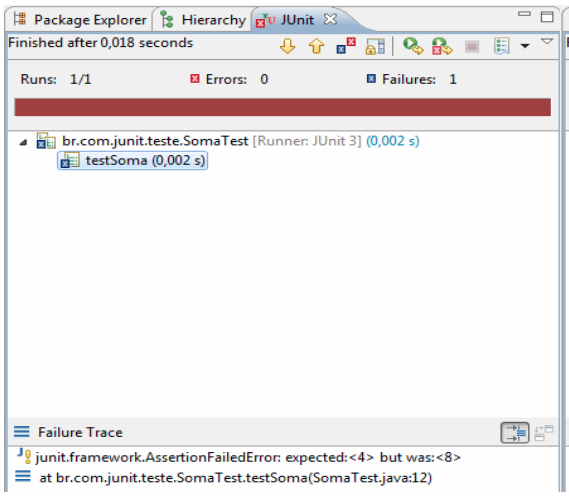

Figura 12. JUnit caso de erro.

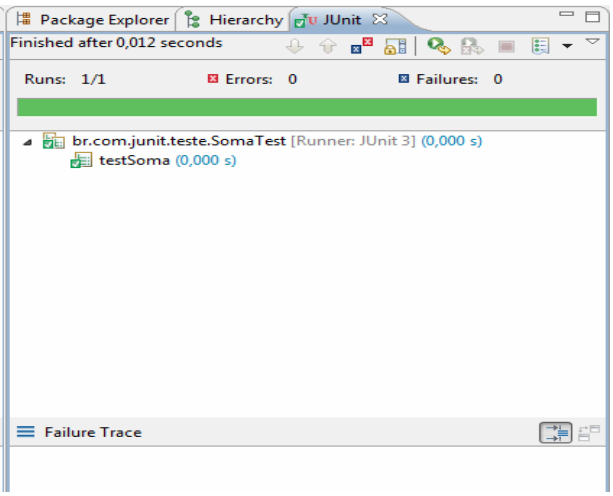

Figura 13. JUnit caso de sucesso.

Nas outras demais ferramentas que realizam teste de unidade segue a mesma linha de raciocino vermelho “erro” **figura 12** e verde “sucesso” **figura 13** como na ferramenta TestNG que será abordada no próximo tópico.

Com a utilização do JUnit, uma série de benefícios ocorre pela adoção, alguns dos principais são:

- Automatizar e criar com facilidade código de testes.
- Uma das ferramentas mais populares entre os desenvolvedores de teste unitário e totalmente open-source.
- Executa testes paralelamente ao desenvolvimento da aplicação e expõe os resultados de sucesso ou erro imediatamente.
- JUnit é orientado a objeto.

Para entender melhor como o framework JUnit é organizado dentro de sua API, a **figura 14** demonstra de uma forma visual de como suas classes estão organizadas e estruturadas.

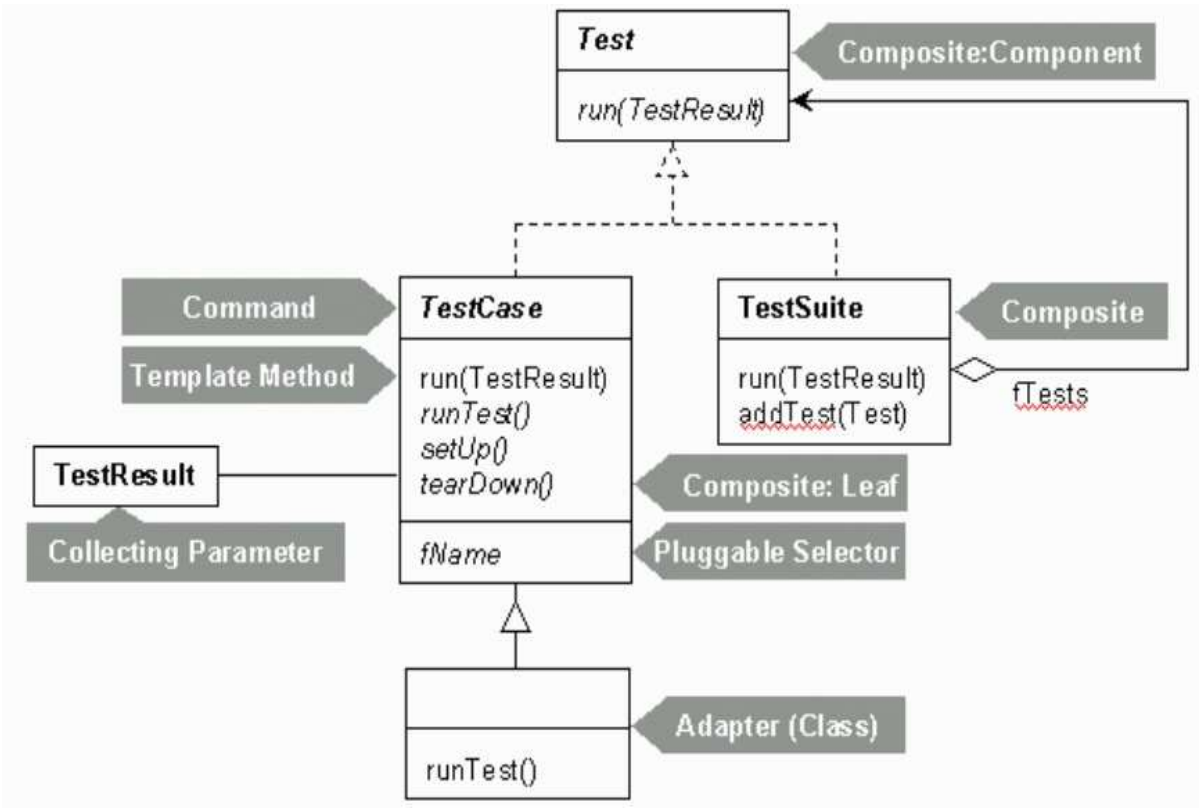

Figura 14. Estrutura da API JUnit (Cook’s Tour [2009]).

Colocando em evidência de uma forma explicativa de como estas classes se comportam dentro da API do JUnit.

A classe Test:

- Com o método run(TestResult) tem o objetivo de coletar as informações de um caso de teste executado.

A classe TestCase utiliza os métodos:

- Run(TestResult) que tem como objetivo passar um objeto do tipo TestResult quando o método Run for executado para cada caso de teste.
- RunTest() tem como objetivo controlar execuções de um teste em particular.
- Setup() usado logo no inicio do teste, sua responsabilidade é iniciar todas as verificações de alguma dependência para a execução dos testes Ex: instanciar uma classe, abrir conexão com banco de dados e etc.
- TearDown() freqüentemente aplicado no final dos testes com o efeito de desfazer “limpar” tudo que foi feito naquele escopo de teste.

A classe TestSuite utiliza os métodos:

- Run(TestResult) usando também para passar um objeto do tipo TestResult para cada caso de teste que executem o método Run.
- AddTest(Test) que possibilita a inserção de novos teste em uma determinada rotina.

Com estas definições podemos observar o quanto complexo uma ferramenta que se propôs a facilitar a injeção de independência na escrita de teste, seja tão robusta e completa e altamente mutável a cada situação dos testes.

A Classe TestCase tem como objetivo realizar a execução de vários teste separadamente um do outro, simplificando o método run() será chamado para cada método que for testado.

A Classe TestSuite tem como objetivo executar vários métodos de teste em um único teste armazenando os resultados no objeto TestResult, para isso os seguintes

métodos se fazem necessários, addTest() para adcionar uma lista de testes e o método run() que será responsável por executá-los.

Também são utilizados métodos dentro da classe TestCase() para a validação dos resultados esperados por cada caso de teste, um dos principais e mais utilizados além do setUp() e tearDown() também temos destaques para:

- assertEquals() utilizado pra comparar se uma determinada resposta retornada pela execução do caso de teste Ex: assertEquals(esperado,retornado).
- assertFalse() testa um valor do tipo booleano para verificar se o mesmo é falso Ex: assertFalse(métodoQueRetornaUmBooleano).
- assertTrue() testa um valor do tipo booleano para verificar se o mesmo é true Ex: assertTrue(métodoQueRetornaUmBooleano).
- assertNotNull() testa se um objeto não está nulo Ex: assertNotNull(objeto).

Para começar a desenvolver rotinas de teste, tente realizar somente o necessário, estude o problema para ter uma visão mais apurada, por exemplo, se for preciso efetuar 3 casos de teste não fique procurando outros, essa técnica é muito vantajosa quando aplicado de forma coerente.

Na **Figura 15** é descrito um processo de como fica organizado os casos de teste.

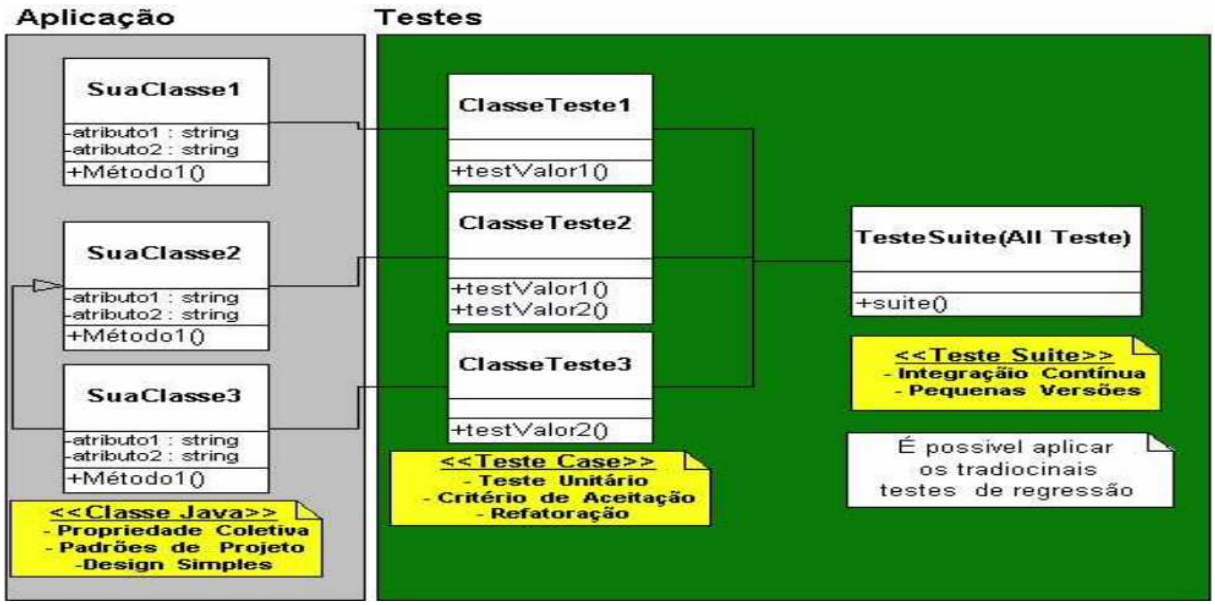

Figura 15. Organização dos testes e práticas de XP (Medeiros [2009]).

5.1 TESTNG v5.9.

TestNG é um framework licenciado e distribuído sob os termos da licença Apache versão 2.0, projetado e “inspirado” no JUnit, mas com suas particularidade especifica, foi desenvolvido por Cédric Beust e Alexandru Popescu, o uso de anotação (Annotations) é uma característica muito forte deste framework, TestNG foi o primeiro a adotar tais técnicas, garantindo uma maior flexibilidade no momento em que as classes de testes são desenvolvidas, fornecendo uma gama de facilidade através de anotação, que foi incorporada ao JUnit a partir da v 4.x.

Um dos aspectos parecidos com o JUnit é a forma de como a resposta é reportada para o desenvolvedor, na **figura 16** e **figura 17** também demonstram caso de erro e de sucesso na verificação e validação dos testes.

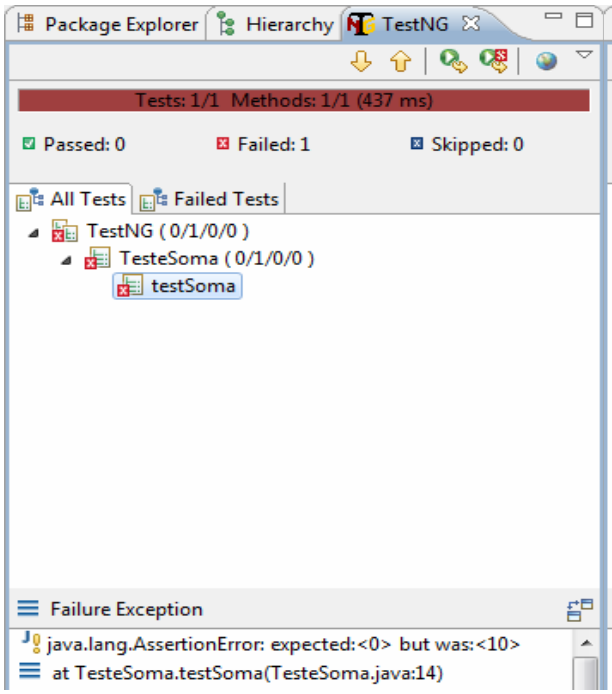

Figura 16. Teste falhou.

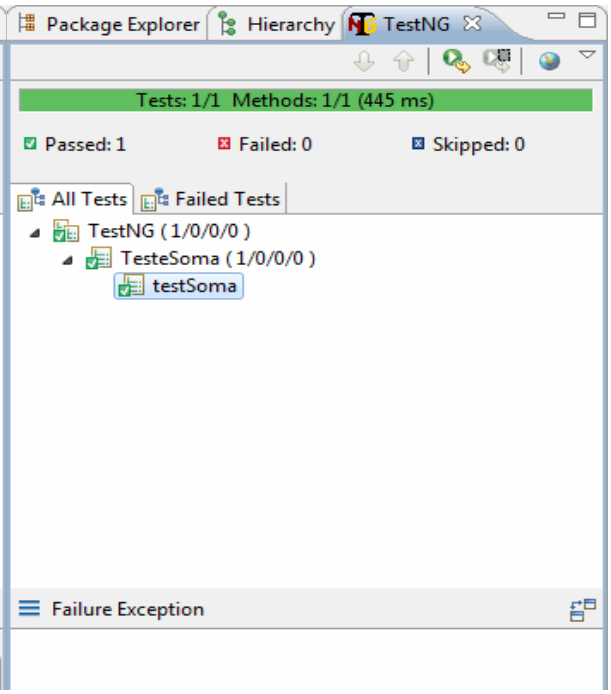

Figura 17. Teste aceito.

Como podemos perceber temos uma característica muito semelhante ao JUnit, mas o framework TestNG não se delimita somente nestes exemplos de visualizações, também disponibiliza uma interface web, que possibilita a navegação entre as classes de testes e métodos executados. Que também é uma das principais característica que TestNG oferece ao desenvolvedor.

Para exemplificar, quando executamos os teste utilizando TestNG ele automaticamente cria uma estrutura através das <tags> de anotação, que possibilita a criação de uma interface web para melhorar a navegação, a geração desta interface web é muito importante porque pode ser visualizada como uma boa documentação, na **Figura 18** característica deste esquema de navegação via web.

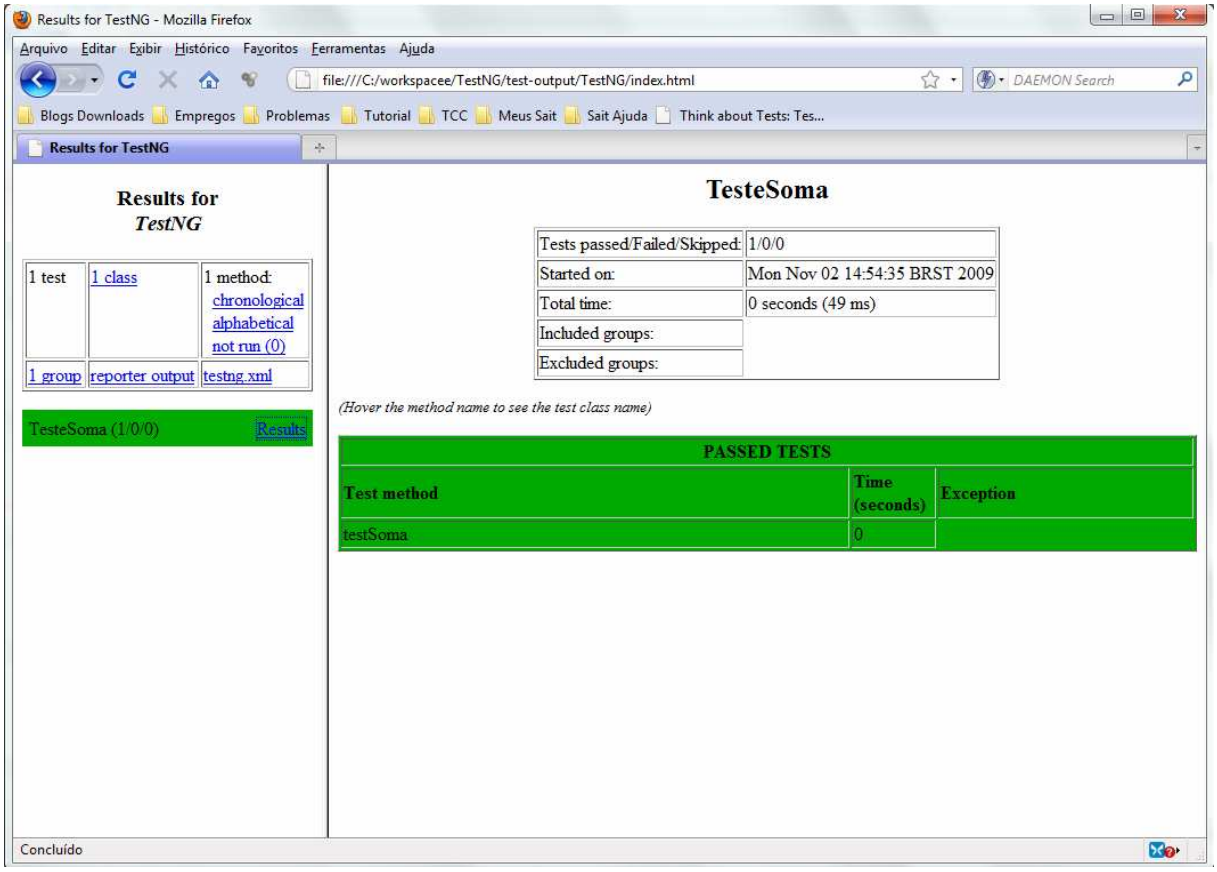

Figura 18. Opção via web para navegar nos testes realizados.

Aprofundando um pouco mais nas características de TestNG, se observa algumas características semelhantes ao JUnit .A página gerada pelo TestNG é criada no momento da execução dos teste, eles são atualizados freqüentemente e possibilitando a alteração da pagina no momento de sua execução, se for o caso acrescentar ou retirar trechos deste código através de um arquivo do tipo XML manualmente.

Com está variedade de opção para informar o desenvolvedor de tudo que se passa no código, TestNG desfruta de uma quantidade enorme de recursos para tornar este processo ainda mais agradável principalmente as anotações.

Na **Figura 19** segue o modelo da API do TestNG de como está estruturada uma grande ferramenta de teste, algumas das principais características serão abordadas ao longo deste capitulo.

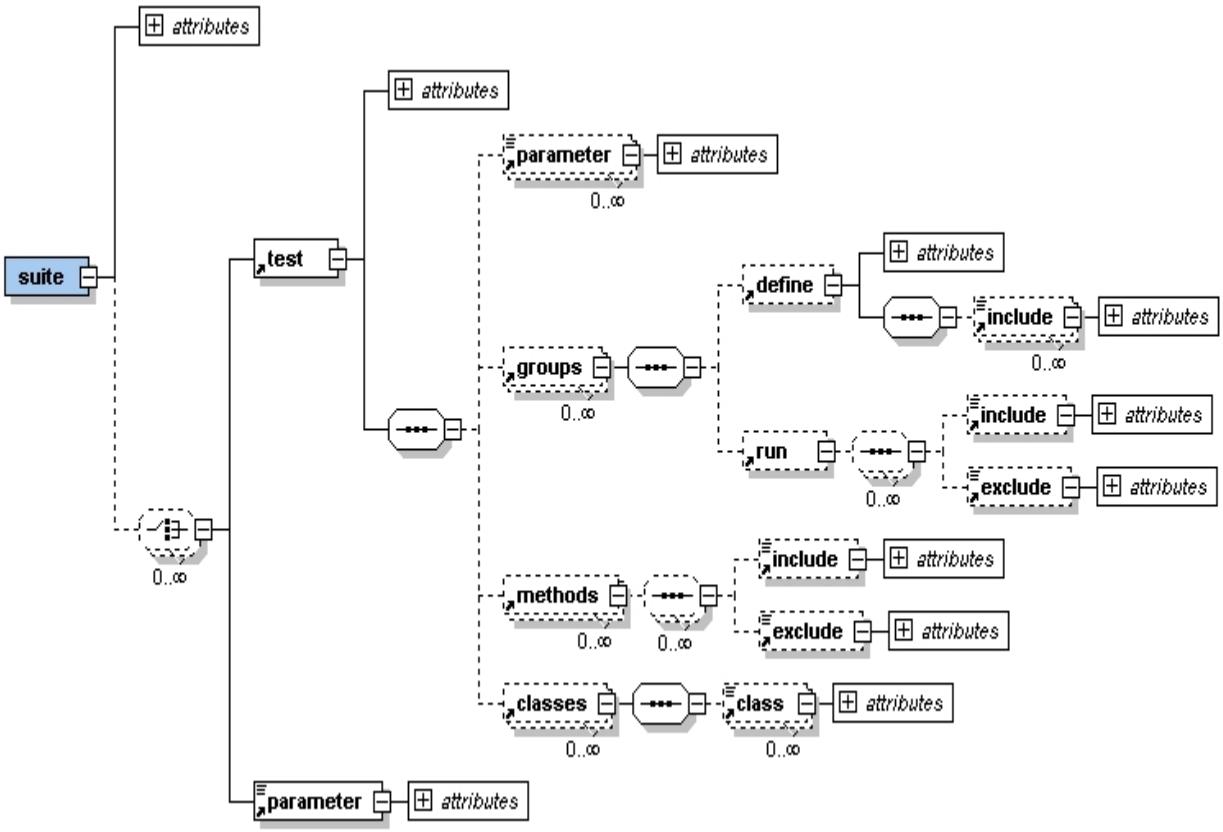

Figura 19. API da Ferramenta TestNG (Suit TestNG [2009]).

Seguindo o esquema da **figura 19** e os conceitos de testes de unidade as seguintes anotações são os mais populares dentre uma vasta e complexa estrutura de métodos e anotação, entre elas estão:

- @BeforeSuite o método que recebe esta anotação é executado antes que todos os outros teste seja executado.
- @AfterSuite é utilizado para dizer que o método que leva está anotação será executado depois que todos os outros testes forem executados.
- @BeforeTest específica que o método com está anotação será executado antes dos testes, muito utilizado para os métodos SetUp(),

quando é preciso iniciar uma classe ou método antes que ele seja testado.

- @AfterTest ao contrário do @BeforeTest sua função é especificar que o determinado método vai ser executado depois que os outros testes estiverem terminados.
- @BeforeGroups garante que os métodos que esteja demarcados por essa anotação é executado antes de qualquer outro, como o objetivo de prever se algum outro método depende da execução de um determinado grupo.
- @AfterGroups executa o bloco com está anotação quando todos os outros grupos já foram executados.
- @BeforeClass o método anotado será o primeiro a ser executado na classe atual.
- @AfterClass indica que o método anotado será executado depois que todos os métodos da classe atual ser executados.
- @BeforeMethod o método com esta anotação será executado antes de qualquer outro método.
- @AfterMethod indica que o método será iniciado depois que todos os outros métodos pertencente a classe serem executados.

Estes foram algumas das principais anotações (Annotations) que descreve o comportamento de possibilidades para executar os métodos de teste, algumas outras anotações se fazem importantes e não foi citada para não alongar muito a parte técnica do documento, mas fica disponível um link para que possam acessar o manual, [Manual TestNG, 2009].

Para refletir sobre uma anotação, que se faz muito importante quando não se tem uma base de dados com informações suficiente para a execução dos teste proposto, a utilização da anotação @DataProvider se faz muito importante.

- @DataProvider muito utilizado quando surge a necessidade de dados que não estão disponíveis no momento da execução dos teste na **Figura 20** é exemplificada de uma forma mais clara a utilização desta anotação.

Exemplo da utilização da anotação @DataProvider **Figura 20**, simples porém eficaz para o entendimento desta estrutura.

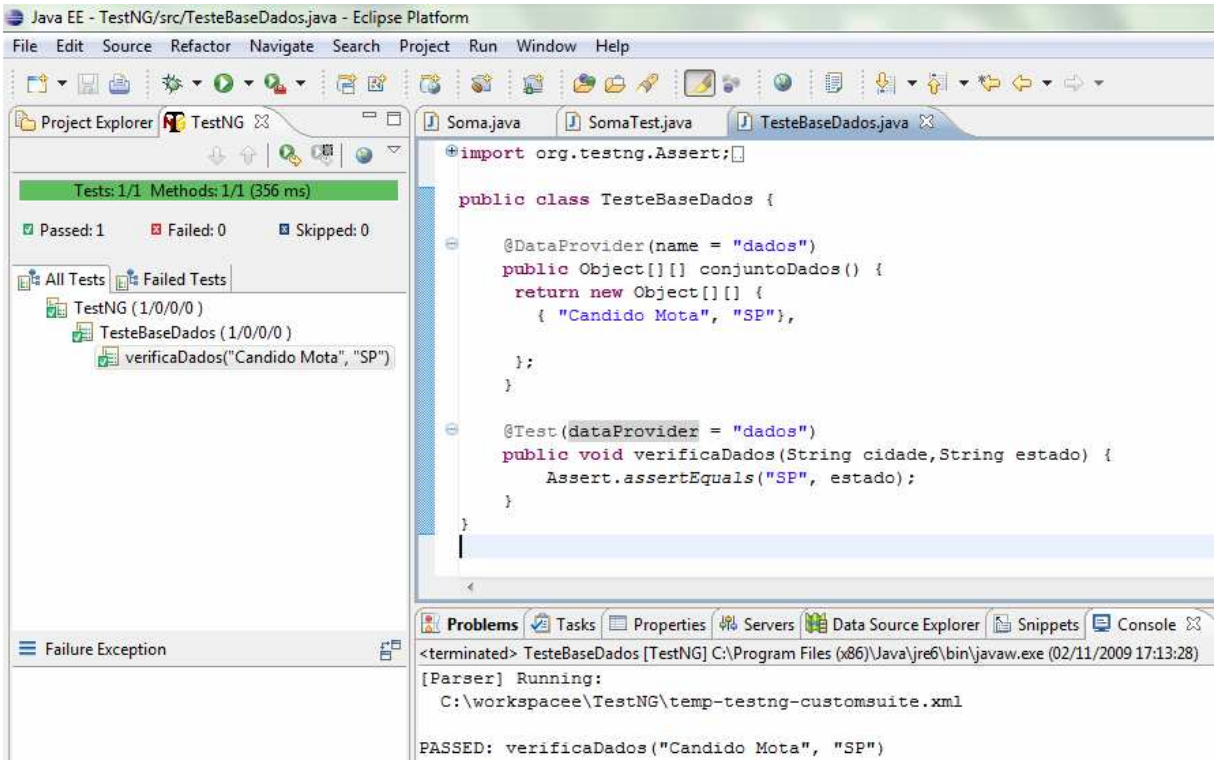

Figura 20. Utilizando anotação @DataProvider.

As anotações citadas acima é algumas dos destaques, além destas anotações temos que relevar outras anotações como: @Test, @Configuration e outras propriedades que seguem como padrão para deixar os testes mais flexíveis utilizando a passagem de parâmetros e configuração dos métodos de testes.

6. CRITÉRIOS COMPARATIVOS.

Utilizando os casos de teste implementados no **Capítulo 5**, será definido um conjunto de critérios para a avaliação destas ferramentas JUnit e TestNG, estes critérios abordam as ferramentas de maneira qualitativa e quantitativa. O primeiro critério é a utilização de um questionário de avaliação Heurística (qualitativa) proposto por Nielsen[1994] com o objetivo de avaliar a usabilidade de uma determinada ferramenta, e definir questões simples e eficazes no nível de usuário. Nielsen[1994] apresenta 10 itens que compõem o questionário heurístico.

1 – Visibilidade:

- A ferramenta informa o usuário sobre o que está ocorrendo?

- O feedback está apropriado?

2 – Coerência:

- A ferramenta usa convenções do mundo real ao invés de termos orientados ao sistema?

- A ferramenta usa a linguagem do usuário?

3 – Controle e Liberdade:

- A ferramenta oferece formas fáceis para sair de situações indesejáveis?

4 – Consistência e Padronização:

- A ferramenta segue convenções de plataforma?

5 – Prevenção de Erros:

- Previne a ocorrência de problemas?

- A ferramenta dificulta o surgimento de erros por parte do usuário?

6 – Reconhecer é melhor que lembrar:

- É necessário lembrar uma informação que já foi tratada em outro local?

- Os objetos, ações, e opções estão visíveis e acessíveis?

7 – Flexibilidade e eficiência de uso:

- A ferramenta permite aceleradores para usuários experientes?

- A ferramenta permite que os usuários criem comandos (teclas de atalho, p.ex.) para ações freqüentes?

8 – Estética e Design suficiente:

- A ferramenta apresenta informação irrelevante?

- A ferramenta organiza as informações de forma clara e elegante?

9 – Recuperação de erros:

- As mensagens são simples, estão na linguagem do usuário?

- As mensagens indicam o problema claramente?

- As mensagens sugerem uma solução ou como evitar o erro?

10 – Documentação e Ajuda:

- É de fácil acesso?

- Está focalizada na tarefa do usuário?

- Apresenta clareza nos passos a serem seguidos?

- É sucinta?

Para responder estas questões, Nielsen [1994] propôs uma tabela de 2 colunas, considerando os campos, [Problemas Detectados] que descreve qual o problema encontrado para resolver uma determinada questão e [Grau de Severidade] como está representado na **Tabela 1.**

Gra	Níveis de Severidade
0	Não concordo que seja um problema.
1	Problema de facilidade.
2	Problema pequeno de usabilidade.
3	Problema grande é importante consertar.
4	Problema catastrófico é imperativo consertar.

Tabela 1. Grau de Severidade

Aplicando o questionário heurístico o próximo passo é aplicar algumas técnicas (quantitativa) que permite o analista recolher informação com mais precisão, definindo o nível de comparação entre valores apresentados por ambas as ferramentas citada no **Capítulo 5**, uma técnica que não deve ser ignorada é de comparar onde uma é mais rápida que a outra, usando **teste de velocidade**.

Para aplicar está técnica foi desenvolvido 5 passos para o levantamento destes requisitos, tais como:

1 - Sistema operacional em que a ferramenta está instalada.

2 - Configuração técnica do equipamento no nível de hardware (processador, memória (RAM) e tamanho/velocidade do HD), da máquina onde o sistema operacional está instalado.

3 - Ferramentas utilizadas para a execução dos testes.

3 - IDE utilizada para o desenvolvimento dos testes.

4 - Uma quantidade de teste maior ou igual a 5 que force vários caminho para verificar o tempo que cada ferramenta leva exercitar os caminhos propostos.

5 - Dados que exiba o tempo gasto por cada ferramenta em questão, levando em conta a variação em que os processos da própria máquina onde o ambiente foi configurado ficam variando freqüentemente a carga de uso do processador e memória.

Com as duas avaliações citadas, pode ter mais uma opção a ser explorada, que são os casos de complexidade dos códigos (levando em consideração a quantidade de linhas necessárias para executar um mesmo código utilizando diferentes ferramentas), com está questão em pauta podemos considerar as 3 questões citadas abaixo a serem respondidas para obter tais resultados.

1 - Ferramenta utilizada para a execução do código.

2 - Exemplos que demonstrem os mesmos problemas codificados em diferentes ferramentas.

3 - Numero de linhas que foi utilizado para codificar o mesmo problema para cada ferramenta.

Neste capítulo foram abordadas 3 casos de comparação para serem aplicadas nas ferramentas JUnit e TestNG, mas existem muitas outras formas de avaliação, com tanto foi somente estás consideradas para suprir e exemplificar pontos comparativos entre ambas as ferramentas.

7. COMPARANDO JUNIT COM TESTNG.

Aplicando a avaliação heurística citada no **Capítulo 6**, proposta por Nielsen [1994], para auxiliar na obtenção de dados para avaliar as ferramentas de automatização de teste citadas no **Capítulo 5**.

Avaliação Heurística aplicada na ferramenta JUnit.

Heurística	Problemas detectados	Grau de Severidade
1 – **Visibilidade**: -A ferramenta informa o usuário sobre o que está ocorrendo? -O *feedback* está apropriado?	- Nenhum problema detectado a ferramenta descreve muito bem o que está acontecendo. -Os erros são apresentados na hora que foi localizado.	0
2 – **Coerência** - A ferramenta usa convenções do mundo real ao invés de termos orientados ao sistema? - A ferramenta usa a linguagem do usuário?	- Na declaração dos testes pode ocorrer um vago esquecimento de qual comando é pra ser usado. - A linguagem é simples, mas não muito intuitiva.	2
3 – **Controle e Liberdade**: - A ferramenta oferece formas fáceis para sair de situações indesejáveis?	-Nenhum problema detectado a ferramenta oferece todo controle sobre os testes.	0
4 **– Consistência e Padronização**: - A ferramenta segue convenções de plataforma?	- Nenhum desvio foi detectado, a ferramenta oferece padronização.	0

5 – **Prevenção de Erros**: -Previne a ocorrência de problemas? - A ferramenta dificulta o surgimento de erros por parte do	- Nenhum problema detectado com a utilização desta ferramenta. - Sim a ferramenta utiliza constantemente um verificador validando sua sintaxe.	0
6 – **Reconhecer é melhor que lembrar**: -É necessário lembrar de uma informação que já foi tratada em outro local? -Os objetos, ações, e opções estão visíveis e acessíveis?	- Se for necessário para o bom funcionamento dos testes e até mesmo corrigir alguns problemas de dependências, a ferramenta atende muito bem. - Se forem declarados de forma correta não haverá problema.	1
7 – **Flexibilidade e eficiência de uso**: -A ferramenta permite aceleradores para usuários experientes? -A ferramenta permite que os usuários criem comandos (teclas de atalho, p.ex.) para ações	-Sim usando teclas de atalho, teclas de alto completar ajudam muito. - Ferramenta não provê a criação de novas teclas de atalhos.	2
8 – **Estética e Design suficiente**: -A ferramenta apresenta informação irrelevante? - A ferramenta organiza a informação de forma clara e	- A ferramenta provê um conjunto de dados de saída ideal para uma análise e facilmente descobrir o causador. - A ferramenta poderia ter uma forma mais elegante para visualizar os problemas.	3
9 – **Recuperação de erros**: -As mensagens são simples, estão na linguagem do usuário? -As mensagens indicam o problema claramente? -As mensagens sugerem uma solução ou como evitar o erro?	- Poderia ser reportado ao usuário em uma língua mais formal e clara. - Porem para quem possui muita prática os erros são fáceis de serem interpretados. - Não, mas as mensagens poderiam expressar melhor de onde estão os erros.	2
10 – **Documentação e Ajuda**: -É de fácil acesso? -Está focalizada na tarefa do usuário? -Apresenta clareza nos passos a serem seguidos? -É sucinta?	- Sim a documentação pode ser acessada pela web no endereço <http://junit.sourceforge.net/doc/ > - Seu foco é muito preciso e claros para quem saiba um pouco de inglês. - Sua resposta é clara e sucinta.	2

Tabela 2. Avaliação Heurística para JUnit.

Avaliação Heurística aplicada na Ferramenta TestNG.

Heurística	Problemas detectados	Grau de Severidade
1 – **Visibilidade**: -A ferramenta informa o usuário sobre o que está ocorrendo? -O *feedback* está apropriado?	- A ferramenta se comunica muito bem com o utilizador, de uma forma escrita e ilustrativa através da web. - Gerando um rápido feedback.	0
2 – **Coerência** - A ferramenta usa convenções do mundo real ao invés de termos orientados ao sistema? - A ferramenta usa a linguagem do usuário?	- Sim, apesar de seguir um padrão convencional na linguagem e escrita através do idioma inglês, porém muito intuitiva.	0
3 – **Controle e Liberdade**: - A ferramenta oferece formas fáceis para sair de situações indesejáveis?	- Não foi detectada em nenhuma circunstância, em relação a situações indesejáveis.	0
4 **– Consistência e Padronização**: - A ferramenta segue convenções de plataforma?	- A ferramenta adota seus critérios para se adaptar a plataforma, utilizando plugins.	0
5 – **Prevenção de Erros**: -Previne a ocorrência de problemas? - A ferramenta dificulta o surgimento de erros por parte do usuário?	- Se praticada de uma forma correta os resultados serão os melhores. -Sim, se o usuário entrar com uma sintaxe errada a ferramenta assinala erro de sintaxe.	0
6 – **Reconhecer é melhor que lembrar**: -É necessário lembrar de uma informação que já foi tratada em outro local? -Os objetos, ações e opções estão visíveis e acessíveis?	- Se for necessário para refletir no bom funcionamento das classes de teste, é bom lembrar para não haver repetição de código. - Com o manual em mãos e de fácil acesso, cada instrução é bem explicada	0

7 – **Flexibilidade e eficiência de uso**: -A ferramenta permite aceleradores para usuários experientes? -A ferramenta permite que os usuários criem comandos (teclas de atalho, p.ex.) para ações frequentes?	- Com o alto complete, a ferramenta possibilita a agilidade no desenvolvimento, fornecendo facilidade de dedução de qual notação utilizar. - Com tanta ajuda a ferramenta é estática em relação à inserção de novas teclas de atalho.	1
8 – **Estética e Design suficiente**: -A ferramenta apresenta informação irrelevante? - A ferramenta organiza a informação de forma clara e elegante?	- A apresentação dos erros e seus causadores é mostrada de uma forma muito semelhante ao JUnit. - Diferente do JUnit, TestNG tem uma forma muito mais apresentável de mostrar seus erros, métodos e classes através de uma interface web.	0
9 – **Recuperação de erros**: -As mensagens são simples, estão na linguagem do usuário? -As mensagens indicam o problema claramente? -As mensagens sugerem uma solução ou como evitar o erro?	- Como JUnit, TestNG também tem suas formas próprias de mostrar suas mensagens assim o entendimento fica por conta de cada usuário da ferramenta. - Em relação as mensagens, a ferramenta apresenta de uma forma clara e intuitiva através da interface web, onde está os problemas e de como está estruturado a classe de teste.	2
10 – **Documentação e Ajuda**: -É de fácil acesso? -Está focalizada na tarefa do usuário? -Apresenta clareza nos passos a serem seguidos? -É sucinta?	- Apesar de a documentação estar em inglês, no endereço < http://testng.org/doc/ >, é de fácil acesso. - Encontrar dificuldade em aprender uma nova tecnologia faz parte do cotidiano dos desenvolvedores, mas com persistência se descobre o quanto fácil é o manuseio da ferramenta.	3

Tabela 3. Avaliação Heurística para TestNG.

Aplicada a avaliação heurística chega o momento de por em prática as técnicas de desenvolvimento para extrair mais algumas informações, o passo agora é priorizar quais são as velocidades que ambas as ferramentas executam uma seqüência de teste, conforme cada ferramenta utiliza sua sintaxe.

Utilizando os passos citados no **Capítulo 6**, sobre como recolher tais informações das ferramentas utilizadas neste trabalho, temos:

Teste de velocidade para a ferramenta JUnit.

1 – Informação sobre o sistema operacional.

Windows 7 v. 7100 Arquitetura 64 Bits.

2 – Configuração de Hardware onde se encontra o sistema operacional.

Processador: Intel(R) Core(TM)2 Duo CPU T6400 @ 2.00GHz 2.00GHz.

Memória(RAM): 3,00 GB.

Tamanho/Velocidade do HD: 250 GB 7200 RPM.

3 – IDE para a realização dos testes.

Eclipse Europa 3.3 mais detalhes < http://www.eclipse.org/platform >.

4 – Conjunto de Caso de Teste submetido à ferramenta. O caso de teste descreve uma seqüência de métodos para avaliar se um aluno tem condições de passar de ano, onde os seguintes requisitos são necessários:

O número de faltas não pode exceder mais que 75, neste caso o aluno está reprovado por falta. A média é calculada por 4 notas e não pode ser menor que 70, com uma exceção se for abaixo de 70 mas se estiver entre 70 e 40 o aluno vai direto para exame e se a nota do exame mais a média anterior dividido por 2 for maior ou igual a 50 retornar aluno aprovado caso contrário retornar aluno reprovado.

Na **figura 21** é apresenta como ficou estruturado o projeto para avaliar o aluno.

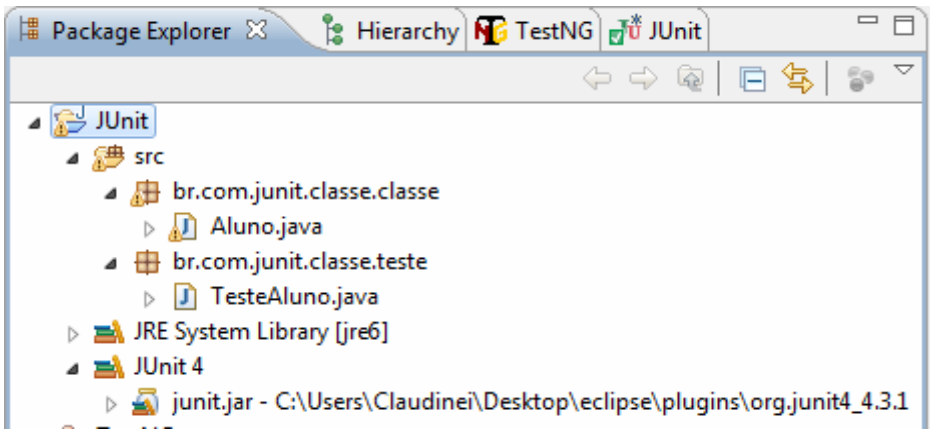

Figura 21. Estrutura de um projeto utilizando JUnit.

Na codificação abaixo é demonstrada a classe do tipo Aluno que será representada os dados do aluno, um método que avalia se o Aluno foi aprovado ou reprovado na disciplina, com os atributos verificadores que serão a base de dados para ser utilizado como critério de entrada (nota1, nota2, nota3, nota4, media, notaFinal, faltas).

```
package br.com.junit.classe.classe;

public class Aluno {
	private int faltas;
	private float nota1;
	private float nota2;
	private float nota3;
	private float nota4;
	private float media;
	private float notaFinal;

	// Os Métodos Gett e Sett não foram apresentados para poupar linhas
	// de codificação mas se fazem necessarios.

	public boolean calculaAprovacaoAluno(){
		if(faltas < 75){
			return false;
		}else{
		   media = (nota1 + nota2 + nota3 + nota4)/4;
		   if(media < 40){
			return false;
		   }else{
			if(media > 70){
				return true;
```

```
                    }else{
                        if(((media + notaFinal) / 2) > 50  ){
                            return true;
                        }else{
                            return false;
                        }
                    }
                }
            }
        }
}
```

Para exemplificar de como fica uma classe de teste desenvolvida nesta ferramenta JUnit a **figura 23** mostra de como é simples e fácil implementar os teste, lembrando que os casos de teste devem ser implementados antes mesmo da classe Aluno, assim o desenvolvedor aprende mais tentando criar os casos de testes, pelo fato de ter que realmente conhecer o problema para avaliar quais os pontos principais que deveria ser moldado nos casos de teste.

```
package br.com.junit.classe.teste;
import org.junit.Assert;
import org.junit.BeforeClass;
import org.junit.Ignore;
import org.junit.Test;
import br.com.junit.classe.classe.Aluno;

public class TesteAluno {
   private static Aluno aluno;
   @BeforeClass
   public static void inicializaClasseTeste(){
        aluno = new Aluno();
   }
   @Test
   public void testaAlunoRepovadoPorFalta(){
        aluno.setFaltas(74);
        Assert.assertEquals(false, aluno.calculaAprovacaoAluno());
   }
   @Test
   public void testaAlunoReprovadoPorMedia(){
        aluno.setFaltas(75);
        aluno.setNota1(40);
        aluno.setNota2(40);
        aluno.setNota3(40);
        aluno.setNota4(39);
        aluno.setNotaFinal(100);
        Assert.assertEquals(false, aluno.calculaAprovacaoAluno());
   }
```

```
    @Test
    public void testaAlunoAprovadoDireto(){
        aluno.setFaltas(75);
        aluno.setNota1(70);
        aluno.setNota2(70);
        aluno.setNota3(70);
        aluno.setNota4(71);
        Assert.assertEquals(true,aluno.calculaAprovacaoAluno());
    }
    @Test
    public void testaAlunoAprovadoComNotaFinal(){
        aluno.setFaltas(75);
        aluno.setNota1(40);
        aluno.setNota2(40);
        aluno.setNota3(40);
        aluno.setNota4(40);
        aluno.setNotaFinal(61);
        Assert.assertEquals(true,aluno.calculaAprovacaoAluno());
    }
    @Test
    public void testaAlunoReprovadoComNotaFinal(){
        Aluno aluno = new Aluno();
        aluno.setFaltas(75);
        aluno.setNota1(40);
        aluno.setNota2(40);
        aluno.setNota3(40);
        aluno.setNota4(40);
        aluno.setNotaFinal(60);
        Assert.assertEquals(false,aluno.calculaAprovacaoAluno());
    }
    // Exemplos A mais.
    // Teste que verifica o lancamento de uma ArithmeticException.
    @Test(expected = ArithmeticException.class)
    public void divisaoPorZero(){
        int i = 1/0;
    }
    // Desabilitar um caso de teste para não ser executado.
    @Ignore
    @Test
    public void testeDesabilitado(){
        System.out.println("Isto nao e para aparecer ...");
    }
} // fim da classe TesteAluno.
```

Usando este conjunto de casos de testes citados acima podemos executar e verificar o tempo que a ferramenta de teste JUnit leva para concluir o processo de execução de todos os testes.

Lembrando que o tempo de execução tem uma pequena variação pelo fato de no momento em que for submetido ao teste o sistema operacional pode ou não estar executando algum tipo de processo em background.

Então temos o tempo de duas execuções para exemplificar este caso de variação.

Tempo de Execução 1: 0,062 s. **Tempo de Execução 2:** 0,047 s.

Aplicando a última avaliação proposta neste trabalho referente a complexidade do código (nesta avaliação será considerado o número de linhas que foi utilizados para codificar os métodos dos casos de teste da **Classe TesteAluno**).

Para aplicar a avaliação de complexidade, avaliando as linhas utilizadas para a implementação dos casos de teste da Classe TesteAluno o questionário proposto no **Capítulo 6** sobre avaliação de complexidade fica estruturado da seguinte forma:

1 - Ferramenta utilizada para a execução do código.

A ferramenta utilizada nesta avaliação foi JUnit integrada com a IDE Eclipse3.4.

2 - Exemplos que demonstrem os mesmo problemas codificados em ferramentas diferentes.

Para poupar linhas de codificação repetidas usaremos como referência a classe que foi citada neste capítulo para a codificação da **Classe TesteAluno** utilizando a ferramenta JUnit.

3 - Número de linhas que foi utilizado para codificar o mesmo problema para cada ferramenta.

O número de linhas utilizada para codificar os métodos de teste utilizando a ferramenta JUnit foi:

Número de linhas utilizadas: 71 linhas.

Teste de velocidade para a ferramenta TestNG.

1 – Informação sobre o sistema operacional.

Windows 7 v. 7100 Arquitetura 64 Bits.

2 – Configuração de Hardware onde se encontra o sistema operacional.

Processador: Intel(R) Core(TM)2 Duo CPU T6400 @ 2.00GHz 2.00GHz.

Memória(RAM): 3,00 GB.

Tamanho/Velocidade do HD: 250 GB 7200 RPM.

3 – IDE para a realização dos Teste

Eclipse Europa 3.3 mais detalhes < http://www.eclipse.org/platform >.

4 – Conjunto de Caso de Teste submetido a ferramenta. O caso de teste descreve uma seqüência de métodos de teste para avaliar se um aluno tem condições de passar de ano onde os seguintes requisitos são necessários:

O número de faltas não pode exceder mais que 75 caso seja o aluno estará reprovado por falta. A média é calculada por 4 notas, e não pode ser menor que 70 com uma exceção se for abaixo de 70 e estiver entre 70 e 40 o aluno vai direto para exame e se a nota do exame somada com a média do aluno dividido por 2 for maior ou igual a 50 então retorna que o aluno foi aprovado, caso contrário informa que o aluno foi reprovado por não atingir uma nota suficiente no exame.

Na **figura 22** demonstra como ficou estruturado o projeto para avaliar o aluno.

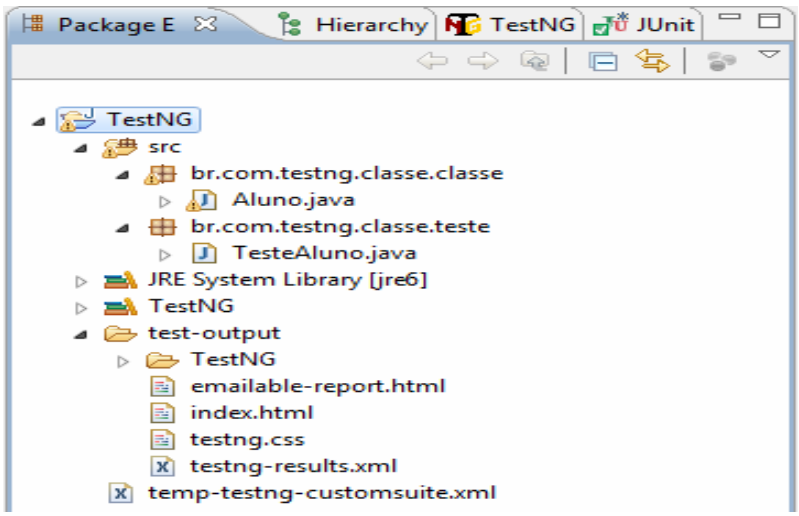

Figura 22. Estrutura do projeto utilizando TestNG.

Para evitar duplicidade de figuras iguais a mesma estrutura da classe de Aluno que seria mostrada logo abaixo, fica como referência a **Classe Aluno** codificada acima citada no tópico de que descreve os critérios para avaliar a velocidade da ferramenta JUnit (**página 40**) que é a mesma utilizada para a criação dos casos de teste com a ferramenta TestNG.

Neste momento também foi implementada uma classe de casos de teste utilizando a ferramenta TestNG, que por ser muito parecida com JUnit, mas com suas peculiaridades quando se escreve testes mais avançados.

Mas levando em consideração que tais casos de teste foram implementados com a finalidade de observar quanto tempo a ferramenta leva para executá-los, descrevemos abaixo a codificação destes casos de teste na **Classe TestaAluno**.

```
package br.com.testng.classe.teste;
import org.testng.Assert;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;
import br.com.testng.classe.classe.Aluno;

public class TesteAluno {
  public Aluno aluno;
  @BeforeClass
  public void inicializaClasseTeste(){
                aluno = new Aluno();
        }
  @Test(groups = "AprovacaoDoAluno")
  public void testaAlunoReprovadoPorFalta(){
          aluno.setFaltas(74);
          Assert.assertEquals(false, aluno.calculaAprovacaoAluno());
  }
  @Test
  public void testaAlunoReprovadoPorMedia(){
        aluno.setFaltas(75);
        aluno.setNota1(40);
        aluno.setNota2(40);
        aluno.setNota3(40);
        aluno.setNota4(39);
        aluno.setNotaFinal(100);
   Assert.assertEquals(false, aluno.calculaAprovacaoAluno());
  }
  @Test(groups = "AprovacaoDoAluno")
  public void testaAlunoAprovadoDireto(){
        aluno.setFaltas(75);
        aluno.setNota1(70);
        aluno.setNota2(70);
        aluno.setNota3(70);
        aluno.setNota4(71);
        Assert.assertEquals(true,aluno.calculaAprovacaoAluno());
  }
  @Test(groups = "AprovacaoDoAluno")
  public void testaAlunoAprovadoComNotaFinal(){
        aluno.setFaltas(75);
        aluno.setNota1(40);
        aluno.setNota2(40);
        aluno.setNota3(40);
        aluno.setNota4(40);
        aluno.setNotaFinal(61);
        Assert.assertEquals(true,aluno.calculaAprovacaoAluno());
  }
```

```
  @Test(groups = "AprovacaoDoAluno")
   public void testaAlunoReprovadoComNotaFinal(){
         Aluno aluno = new Aluno();
         aluno.setFaltas(75);
         aluno.setNota1(40);
         aluno.setNota2(40);
         aluno.setNota3(40);
         aluno.setNota4(40);
         aluno.setNotaFinal(60);
         Assert.assertEquals(false,aluno.calculaAprovacaoAluno());
   }
   // Mais Exemplos.
   // Teste que verifica se foi lancado uma ArithmeticExeption.
   @Test(expectedExceptions = ArithmeticException.class)
   public void divisaoPorZero(){
            int i = 1/0;
   }
   // Desabilitar um caso de teste para não ser executado.
   @Test(enabled = false)
   public void testeDesabilitado(){
            System.out.println("Isto nao e para aparecer ...");
   }
} // Fim da Classe TesteAluno.
```

Agora podemos notar tal semelhança nos casos de testes do JUnit com TestNG, usando este código e submetendo a execução podemos extrair um tempo que foi usado para a execução dos casos de teste.

Lembrando do que foi citado no tópico onde é executado os casos de teste com JUnit, o tempo pode variar dependendo do que o sistema operacional pode estar processando em background no momento da execução dos testes.

Levando estes fatores em consideração temos dois tempo para avaliar melhor esta questão de velocidade e variação que pode ocorrer durante a execução dos casos de teste.

Tempo de Execução 1: 406 milésimo de segundos.

Tempo de Execução 2: 390 milésimo de segundos.

Para finalizar este capítulo comparativo a última avaliação a ser aplicada leva em consideração a quantidade de linha que foi produzida para resolver o problema citado na avaliação de velocidade da ferramenta que foi a verificação de um aluno aprovado ou não em uma determinada disciplina, levando em consideração somente

a Classe TesteAluno que foi o principal modelo para comparar ambas as ferramenta, assim podemos observar com base na avaliação de complexidade citada no **Capítulo 6** que descreve as 3 questões abaixo.

1 - Ferramenta utilizada para a execução do código.

A ferramenta utilizada nesta avaliação foi TestNG integrada com a IDE Eclipse3.4.

2 - Exemplos que demonstrem os mesmo problemas codificados em ferramentas diferentes.

Para poupar linhas de codificação repetidas usaremos como referência a classe que foi citada neste capitulo para a codificação da **Classe TesteAluno** utilizando a ferramenta TestNG.

3 - Número de linhas que foi utilizado para codificar o mesmo problema para cada ferramenta.

O número de linhas utilizada para codificar os métodos de teste utilizando a ferramenta TestNG foi:

Número de linhas utilizadas: 69 linhas.

Para finalizar na **Tabela 4** é demonstrado todos os resultados das avaliações aplicadas para cada ferramenta abordada neste trabalho JUnit e TestNG.

JUnit		**TestNG**	
Avaliação Heurística		Avaliação Heurística	
Tempo de Execução dos Testes		Tempo de Execução dos Testes	
Tempo 1	0,062 s	Tempo 1	406 ms
Tempo 2	0,047 s	Tempo 2	390 ms
Linhas de Código utilizadas		Linhas de Código utilizadas	
Classe TestaAluno	71	Classe TestaAluno	69

Tabela 4. Tabela Comparativa das avaliações aplicadas em JUnit e TestNG.

8. CONCLUSÃO

Com a conclusão do trabalho, é importante estar fixado ao leitor, que e a utilização de metodologias ágeis como Extreme Programming (XP) e práticas como Desenvolvimento Guiado por Teste (TDD), é fundamental para a prevenção de erros oriundos que ocorrem na fase de desenvolvimento do software, refletindo na desconfiança do cliente com o sistema e o medo de alteração no código por parte do desenvolvedor.

Os dados extraídos pelas ferramentas JUnit e TestNG como foi abordado no **Capítulo 5 e 6**, fixa a utilização de cada ferramenta com suas peculiaridades, mas sempre focando nos mesmos casos de testes, oferece ao leitor uma quantidade de característica que as diferenciam não só em sua sintaxe mas também nos casos em que ambas interpreta os mesmo pontos com analogias diferentes.

As ferramentas que foram empregadas neste trabalho têm como único objetivo submetê-las a uma seqüência de validação de teste e extrair dados relevantes de ambas, para possíveis argumentos que decidem na hora de escolher entre uma ou a outra, quando uma é mais viável que a outra e sempre caracterizando os dois lados de ambas as ferramentas o bom/ruim, verificando onde estão às fraquezas entre uma e outra.

A utilização das ferramentas JUnit e TestNG tem como foco, levantar a teoria e aplicar na prática o aprendizado com a utilização de pequenos exemplos para distinguir a diferença de sintaxe, complexidade(avaliando o número de linhas codificadas) e desempenho de ambas.

Utilizando alguma destas ferramentas e seguindo o que foi discutido durante o trabalho, o próximo passo (um trabalho futuro) seria a empregabilidade destas técnicas no desenvolvimento de um software desde os levantamentos de requisitos até a entrega do sistema como um todo e conferindo os resultados no decorrer do projeto seguindo o que foi abordado e citado neste trabalho.

9. REFERÊNCIAS BIBLIOGRÁFICAS.

CASTRO, Vinicius Almeida Castro, Desenvolvimento Ágil Com Programação Extrema. 2006. 122 p. Dissertação (Trabalho Conclusão de Curso) - São Cristóvão – SE. Universidade Federal de Sergipe.

TELES, Vinícius Manhães, Extreme Programming, XP: Aprenda coma encantar seus usuários Desenvolvendo software com agilidade e alta qualidade. São Paulo: Editora Novatec, 2004.

MALDONADO, Jose Carlos; DELAMARO, Marcio Eduardo; JINO, Mario Introdução ao Teste de Software, 1. Ed. Rio de Janeiro: Editora Campus, 2006.

SOMMERVILLE, Ian. Engenharia de Software, 6. Ed. Tradução de André Mauricio de Andrade. São Paulo: Editora Pearson, 2003.

NIELSEN, J. Heuristic Evaluation. In: Usability Inspection Methods. John Wiley, New York, 1994.

10. REFERÊNCIAS ELETRÔNICAS.

KNIBERG, Henrik, Scrum e XP direto das Trincheiras, postado por infoq <http://infoq.com/br/minibooks/scrum-xp-from-the-trenches >. Acessado em: 4 Abr. 2009.

CLAUDIO, Arilo Dias Neto, Introdução a Teste de Software. In. Revista Engenharia de Software, Ed. 1, março, 2008. p. 54-59, Disponível em <http://www.devmedia.com.br/articles/viewcomp.asp?comp=8035> . Acessado em 27 Set. 2009.

ALENCAR, Roberto Luiz Sena,Test-Driver Development (TDD) – Desenvolvimento Guiado por Teste. Postado no site <HTTP://www.javafree.org>. Acessado em: 20 Set. 2009.

MUELLER, Rafael, Implantando Desenvolvimento Ágil na Empresa. Partes I, II, III, IV e V. Postado no site <http://queroseragil.wordpress.com/2008/09/10/implantando-desenvolvimento-agil-na-empresa/>. Acessado em: 11 Abr. 2009.

LEVISON, Mark, Tornando TDD Simples: Problemas e Soluções para Implementadores. Tradução Wagner R. Santos. Disponível em <http://www.infoq.com/br/articles/levison-TDD-adoption-strategy>. Acessado em: 4 Mar. 2009.

MEDEIROS, Manoel Pimentel. JUnit – Implementando Teste Unitários em Java. 2009. Disponível em <http://www.devmedia.com.br/articles/viewcomp.asp?comp=1432&hl=*Implementando*%20and%20*testes*%20and%20*unit%E1rios*%20and%20*em*%20and%20*Java*>. Acessado em 6 Mar. 2009.

MEDEIROS, Manoel Pimentel. Planejando seu Projeto com Extreme Programming. Parte I e II. 2009. Disponível em < http://www.devmedia.com.br/articles/viewcomp.asp?comp=4273>. Acessado em 6 Jun. 2009.

SANCHEZ, Ivan. Introdução ao Desenvolvimento Orientado a Teste (TDD). Disponível em <http://dojofloripa.wordpress.com/2006/11/07/introducao-ao-desenvolvimento-orientado-a-testes/>. Acessado em 5 Out. 2009.

Suit TestNG, TestNG: The next generation of unit testing, < http://www.javaworld.com/javaworld/jw-04-2005/jw-0404-testng.html >. Acessado em 30 Out. 2009.

PINTO, Gustavo H. L. TDD Desenvolvimento Guiado por Teste. Disponível em <http://amazontic.wordpress.com/2009/09/23/tdd-desenvolvimento-guiado-por-teste/>. Acessado em 30 Set. 2009.

Documentação TestNG, < http://testng.org/doc/documentation-main.html >. Acessado em 8 Out. 2009.

Cook’s Tour, JUnit a Cook’s Tour. <http://junit.sourceforge.net/doc/cookstour/cookstour.htm> Acessado em 20 Out. 2009.