Teste de Software – Uma Introdução e Exemplos

João Rotta Neto
Maria Cristina Nunes dos Santos

Sumário

Este artigo apresenta uma breve introdução aos Testes de Software, relaciona e descreve as principais técnicas de testes e aponta algumas situações que provocam defeitos em software. Trechos de códigos são apresentados para exemplificar e analisar as características e uso de cada Teste.

This paper demonstrates a brief introduction to Software Testing, it lists and describes the main techniques of tests and relates some error prone situations. Code snippets are provided in order to analise and exemplify the characteristics and use of each Test.

Palavras Chave

Teste, Estrutural, Cobertura, Funcional

Índice

1. Introdução

Teste de software é o processo de execução de software de maneira controlada visando responder a questão:

"O software se comporta conforme especificado ?"

Durante os testes asseguramos se o software foi implementado corretamente, isto é, **verificamos** o software; também asseguramos se ele foi construído de acordo com seus requisitos, ou seja, **validamos** o software.
O propósito do teste de software não é provar que um programa está bem escrito mas sim descobrir todos os defeitos que eventualmente existem nele [GIANTURCO]. Através dos testes não comprovamos a ausência de erros ou defeitos, apenas mostramos os existentes. Um bom teste é aquele que exercita diversas porções do software fazendo-o falhar. Assim, dizemos que o teste de software é uma tarefa destrutiva e não construtiva. Igualmente importante a examinar se um software realiza as tarefas para as quais foi projetado é descobrir o que ele faz não intencionalmente [RABIN].
Sabemos que o custo de alteração dos softwares aumenta ao longo do tempo, portanto, quanto mais tarde um defeito for encontrado, maior será o custo da sua correção. Daí a importância da realização de testes desde o início de um projeto.

2. Processo de Teste

Podemos visualizar um programa como sendo uma função que descreve uma relação entre um elemento de entrada (elemento do domínio) e um elemento de saída. O Processo de Teste é utilizado para assegurar que o programa realiza fielmente esta função. Os elementos essenciais de um teste são:

- programa na sua forma executável ou código fonte;
- uma descrição do comportamento esperado do programa;
- uma descrição do seu domínio funcional (entradas)

O Processo de Teste consiste então em obter um valor válido do seu domínio funcional (ou inválido caso estivermos testando a robustez do programa), determinar o comportamento esperado do programa para o valor escolhido, executar o programa, observar seu comportamento e, finalmente, comparar este comportamento com o esperado. Se o comportamento atual não coincidir com o esperado, dizemos que este teste descobriu (no sentido de tornar visível) um erro.

A completa verificação de um programa poderia ser obtida através de um Teste Exaustivo, isto é, a aplicação do Processo de Teste para todos os valores do domínio. No entanto, geralmente os domínios funcionais são infinitos ou, quando finitos, são suficientemente grandes tornando inviável a quantidade de testes necessária para cobri-los. Adrion [ADRION], muito sensato, diz que todos os programas que podem ser exaustivamente testados podem ser substituídos por uma *look-up table*.

```
int FuncaoA(char ch);
int FuncaoB(int x, int y);
```

Considerando `char` com 1 byte e `int` com 4 bytes, no exemplo ao lado o domínio de entrada da `FuncaoA` apresenta 256 elementos e o da `FuncaoB` 18.446.744.073.709.551.616 elementos.

Desta forma, uma atividade importante nos testes é encontrar um conjunto de dados de teste que seja grande o suficiente para cobrir uma porção significativa do domínio e ainda pequeno suficiente para que seja viável a aplicação do Processo de Teste em todos os seu elementos.

3. Técnicas de Teste

Basicamente existem duas técnicas de testes de software – a chamada Teste Estrutural ou *White-Box* ou ainda *Glass-Box* e o Teste Funcional ou *Black-Box.*
No Teste Estrutural os dados de teste exercitam a lógica interna dos componentes de software e comparam o comportamento do programa contra a intenção aparente do seu código fonte.
Na outra técnica, o Teste Funcional ou *Black-Box*, os dados de teste são utilizados para comparar o funcionamento do sistema contra seus requisitos não levando em consideração como os programas operam internamente.

3.1 Testes Estruturais

Analisando a estrutura interna do software podemos derivar casos de teste que:

- garantam que todos os caminhos independentes dentro de um módulo sejam exercitados pelo menos uma vez;
- exercitem todas as decisões lógicas para valores *falsos* ou *verdadeiros*;
- executem todos os laços em suas fronteiras e dentro de seus limites operacionais;
- exercitem as estruturas de dados internas para garantir a sua validade.

3.1.1 Cobertura das Linhas

Cobertura das Linhas ou Cobertura das Declarações (*Statement Coverage*) é o mais simples dos testes de Cobertura. Ele mede se todas as linhas ou declarações foram executadas. Este teste pode ser aplicado diretamente sobre o código objeto de um programa não havendo a necessidade de processamento nem alteração do código fonte. *Profilers* normalmente medem a Cobertura das Linhas. Por outro lado, este teste apresenta algumas desvantagens exemplificadas a seguir:

```
int a, b, c;
a = 4;
b = 0;
if (condição)
{
    b = 2;
}
c = a / b;
```

No exemplo ao lado, se `condição` for `TRUE`, o código apresentará 100% de Cobertura das Linhas, executará com sucesso mas, ainda assim, apresenta um erro.
Se alguma vez condição for `FALSE`, o código irá falhar (*divisão por zero*).

```
if (condição)
{
    declaração01;
    declaração02;
    ...
    declaração98;
    declaração99;
}
else
{
    // esta linha apresenta
    // um erro
    declaração100;
}
```

Neste outro exemplo, se `condição` for `TRUE`, obtemos 99% de Cobertura das Linhas. Valor que pode ser suficiente para o teste em questão. No entanto a única que não foi exercitada, a `declaração100`, apresenta um erro.

3.1.2 Cobertura das Decisões

Cobertura das Decisões mede se as expressões nas estruturas de controle de fluxo são testadas para `TRUE` e `FALSE`. A expressão deve ser avaliada como um todo não importando se apresenta operadores lógicos `AND` ou `OR`. A vantagem desta medida é a simplicidade, no entanto ela pode ser não efetiva em linguagens que apresentam curto circuito (*short circuit*) de operadores.

```
if (A() && (B() || C()))
    declaração1;
else
    declaração2;
```

Este teste pode considerar o código ao lado completamente testado sem nunca ter exercitado a função `C()`.
A expressão ao lado é considerada `TRUE` se `A()` retornar `TRUE` e `B()` retornar `TRUE`, e `FALSE` se apenas `A()` retornar `FALSE`.

3.1.3 Cobertura das Condições

Cobertura das Condições é similar à Cobertura das Decisões porém verifica todas as sub-expressões. Porém este teste é vulnerável quando definimos variáveis auxiliares para simplificar o código. Existem também situações que não se pode obter 100% de cobertura devido à combinações de operadores inviáveis.

```
int flag;
flag = A() && (B() || C());
if (flag)
    declaração1;
else
    declaração2;
```

Se re-escrevermos o código do exemplo anterior utilizando uma variável auxiliar `flag` o teste de Cobertura das Condições funcionará como Cobertura das Decisões.

```
if ((valor == 1) || (valor == 2))
{
    declaração1;
}
```

Neste exemplo nunca testaremos `TRUE` e `TRUE` pois as condições são mutuamente exclusivas, ou seja, nunca atingiremos 100% de Cobertura.

```
int a, b;
if (F(a && b))
{
    declaração1;
}

boolean Funcao(int a)
{
    return false;
}
```

Cobertura das Condições não garante a Cobertura das Decisões. No exemplo ao lado, podemos testar todas as combinações de `a` e `b` e, mesmo assim, `declaração1` nunca será executada.

3.1.4 Cobertura dos Caminhos

Este teste mede se todos os caminhos de cada função foram executados pelo menos uma vez. Caminho é a seqüência única de decisões do ponto de entrada de uma função até sua saída. Se um módulo contém um *loop*, então existem distintos caminhos para zero, uma ou várias iterações do *loop*. A vantagem da Cobertura dos Caminhos é que ela exige um teste detalhado. As desvantagens são que o número de caminhos cresce exponencialmente com o número de decisões e que alguns caminhos são impossível de se exercitar devido à dependência dos dados.

```
if (a) { }
if (b) { }
if (c) { }
```

3 condições
$2^3 = 8$ caminhos
Acrescentando uma condição, dobramos o número de caminhos. Não raro, temos 10 condições num trecho de código, o que leva a 1024 caminhos !

```
if (condição1)
{
    declaração1;
}
declaração2;
if (condição1)
{
    declaração3;
}
```

O exemplo ao lado apresenta 4 caminhos possíveis mas somente 2 viáveis.

3.1.5 Cobertura LCSAJ

LCASJ (*Long Code Sequence And Jump*) é uma variação da Cobertura dos Caminhos que considera somente sub-caminhos no código fonte que são executados em seqüência. A vantagem desta medida é que ela é mais detalhada que a Cobertura das Decisões e evita a complexidade da Cobertura dos Caminhos.

3.1.6 Outros Testes Estruturais

3.1.6.1 Cobertura do Fluxo de Dados

Cobertura do Fluxo de Dados também é uma variação da Cobertura dos Caminhos que considera somente sub-caminhos entre a declaração ou atribuição de uma variável até seu uso. A vantagem desta medida é que os caminhos a serem exercitados estão diretamente relacionados com a maneira na qual o programa manipula seus dados.

Caminhos para a variável `i`

```
int i = 0;
declaração1;
declaração2;
if (condição1)
{
    ...
}
declaração3;
if (condição2)
{
    ...
    i++;
}
else
{

}
declaração4;
if (condição3)
{
    ...
    Função1(i);
}
```

Caminho 1 quando `condição2` for `TRUE`.

Caminho 2 quando `condição2` for `FALSE`.

3.1.6.2 Cobertura das Corridas

Esta medida reporta se múltiplas *threads* executam o mesmo trecho de código simultaneamente (concorrente) ajudando detectar falha no sincronismo do acesso à recursos. Sincronização de *threads* é um assunto delicado e sua implementação, se não for cuidadosa, é forte candidata a apresentar bugs intermitentes.

3.1.6.3 ASSERT

Uma definição de ASSERT é:

ASSERT(*booleanExpression* **)**
Parâmetros
booleanExpression
Especifica uma expressão (incluindo valores de *pointers*) que podem ser diferentes de zero ou zero.

Assert avalia seu argumento. Se o resultado for 0, a macro imprime uma mensagem de diagnóstico e aborta o programa. Se a condição for diferente de zero, não faz nada. [MSDN]

```
char *CopiaBuffer(char *pSrc, char *pDest, unsigned int tam)
{
    // tanto o pointer source quanto o destination devem
    // ser diferentes de NULL

    ASSERT (pSrc != NULL && pDest != NULL);
    while (tam-- > 0)
        *pDest++ = *pSrc++;
    return pDest;
}
```

No exemplo acima, a chamada à `ASSERT` garante que os dois *pointers* passados como parâmetros são válidos.

Embora o uso de ASSERTs caracterizem uma boa prática de programação, a Programação Defensiva, seu uso deve ser cauteloso pois ele pode esconder bugs. Ainda no exemplo acima, o bug de fato está na função que chama a `CopiaBuffer` com algum de seus parâmetros `NULL`.

3.1.6.4 *Walkthroughs* e *Pair Programming*

Walkthroughs e *Pair Programming* (aqui traduzido por Programação em Dupla) são práticas que, através da comunicação e contato entre os desenvolvedores, acabam por testar os programas desenvolvidos. A primeira sugere uma avaliação e, consequentemente, um teste assim que o programa é codificado e a segunda sugere a avaliação e teste durante a codificação.

Durante um *walkthrough* um desenvolvedor, por exemplo, fornece os dados de teste e conduz o time de desenvolvimento através de uma simulação manual do sistema. Os dados de teste são caminhados através do sistema com seus resultados intermediários mantidos num quadro ou num papel. Os dados de teste devem ser simples devido a restrições da simulação feita por pessoas. O *walkthrough* incentiva a discussão do sistema e não somente completar a sua simulação. Muitos erros são descobertos através do questionamento das decisões que o desenvolvedor tomou e não pelo dados de teste. [ADRION]

Steve McConnell [MCCONNELL] define livremente *walkthroughs* como qualquer reunião na qual dois ou mais desenvolvedores revisam o trabalho técnico com o propósito de aumentar sua

qualidade. *Walkthroughs* podem apontar problemas nos requisitos ainda na fase de especificação do sistema ou podem ser utilizados para a revisão de código onde o autor do código apresenta algumas listagens a dois ou mais revisores. Este prática tem se mostrado bastante efetiva para a localização de defeitos em programas.

Kent Beck [BECK] define Programação em Dupla como o diálogo entre duas pessoas tentando simultaneamente programar (analisar, projetar e testar) e entender juntas como programar melhor. É a conversa em vários níveis assistida e focada num computador.
A Programação em Dupla permite que o software vá sendo testado durante seu desenvolvimento pois um programador pode escrever a aplicação enquanto o outro desenvolve os testes para validar o trabalho do primeiro. Interessante notar que os códigos gerados para a aplicação e para os testes são normalmente semelhantes pois tentam resolver o mesmo problema com enfoques diferentes. Esta prática, além de promover um 'crescimento' dos programadores, também gera um grande conjunto de testes que poderão ser utilizados caso os requisitos do sistema sofram alterações e nos Testes de Regressão.

3.2 Testes Funcionais

Os testes funcionais procuram descobrir erros nas seguintes categorias:

- funções incorretas ou ausentes;
- erros de interface;
- erros nas estruturas de dados ou no acesso a banco de dados externos;
- erros de desempenho;
- erros de inicialização e término.

Os testes funcionais, normalmente aplicados durante as últimas etapas da atividade de teste, são projetados para responder às seguintes perguntas:

- Quais entradas constituirão bons casos de teste ?
- O sistema é particularmente sensível a certos valores de entrada ?
- Como as fronteiras dos dados de entrada são isoladas ?
- Quais volumes de dados o sistema pode tolerar ?

3.2.1 Particionamento Equivalente

No Particionamento Equivalente examinamos a especificação do sistema para determinar as entradas (domínio) de cada módulo ou função do software. O domínio de entrada é então dividido em classes de equivalência ou partições nas quais seus valores são processados de forma similar. Os casos de testes são construídos com um valor de cada partição.

Vamos considerar, como exemplo, um sistema que processa um formulário de pesquisa onde o usuário informa o número de funcionários da sua empresa.

Especificação do Módulo: `Solicitar ao usuário o número de funcionários da sua empresa. O usuário deve informar um número. Se o número for negativo, assumir 0. Se o número for maior que 100, assumir 101. Senão armazena o número fornecido. Assumir o valor como inteiro.`

Da especificação obtemos as classes de equivalência.

`{} – não informou valor`	Sendo: `INT_MIN (-2147483647 - 1)` mínimo valor inteiro com sinal
`{valor não numérico}`	
`{INT_MIN até 0, 1 até 100, 101 até INT_MAX}`	`INT_MAX 2147483647` máximo valor inteiro com sinal

De onde podemos derivar os testes:

Teste	Dados de Teste
Teste 1 – a partir da classe de equivalência `{ }`	`não fornecer valor`
Teste 2 – a partir da classe de equivalência `{não numérico}`	`ABCD`
Teste 3 – a partir da classe de equivalência `INT_MIN até 0`	`-125`
Teste 4 – a partir da classe de equivalência `1 até 100`	`10`
Teste 5 – a partir da classe de equivalência `101 até INT_MAX`	`10246`

3.2.2 Análise de Fronteiras

O Teste de Análise de Fronteiras complementa o Particionamento Equivalente ao escolher valores para os testes nas extremidades das classes de equivalência. Pressman [PRESSMANa] recomenda que além dos valores das fronteiras, máximos e mínimos possíveis, sejam também testados valores imediatamente acima e abaixo deles.

Desta forma, assumindo o exemplo anterior, testaríamos mais os seguintes valores:

Valores de Teste
`INT_MIN, INT_MIN+1, -1, 0, 1`
`2,99,100,101`
`102,INT_MAX-1,INT_MAX`

A Análise de Fronteiras também deve ser utilizada em estruturas de dados com limites. Por exemplo, num array definido com 20 elementos, devemos testar o primeiro e o último elemento. Em linguagens que definem coleções *zero-based* onde para uma coleção com `N` elementos, o índice do primeiro elemento é `0` e o do último é `N-1`, a Análise de Fronteira se mostra muito útil para descobrir erros.

3.2.3 Experiência (*Error Guessing*)

Baseado na sua experiência o projetista dos testes 'advinha' os prováveis erros que podem ocorrer num determinado tipo de programa e projeta testes para descobri-los. Por exemplo, se recursos são alocados dinamicamente num programa, um bom trecho para procurar erros seria onde estes recursos são desalocados, isto é, se forem desalocados.
Esta técnica, quando utilizada por um engenheiro experiente, é bastante eficiente no projeto de casos de teste, por outro lado em mãos inexperientes pode ser desperdício de tempo [IPLa].

3.3 Testes Especializados

3.3.1 Regressão

Teste de Regressão é aquele que tenta descobrir falhas através da repetição de casos de teste utilizados anteriormente. Quando defeitos já corrigidos re-aparecem, dizemos que o software 'regride', daí o nome do teste – regressão.

O Teste de Regressão deve ser aplicado toda vez que o software sofra uma alteração seja para introdução uma nova funcionalidade ou correção de um defeito. Casos de testes deve ser bem documentados para facilitar sua aplicação em testes de regressão.

3.3.2 GUIs

As GUIs (*Graphics User Interface*), unânimes atualmente, trouxeram grandes benefícios em termos de usabilidade e padronização para o usuário final; por outro sua implementação é bastante complexa e extensa. Bradley [BRADLEY] sustenta que programas com GUI tem uma ordem de grandeza a mais de código a ser testado.

O teste de interfaces gráficas é praticamente inviável se não for feito com auxílio de uma ferramenta que grava (*capture*, *record*) as ações feitas manualmente por um usuário e depois as reproduz automaticamente (*replay*, *playback*). Existem basicamente dois tipos destas ferramentas: aquelas que gravam os movimentos do mouse em termos de suas coordenadas em tela e aquelas orientadas aos *widgets* que gravam quais componentes da interface foram manipulados. Embora o segundo tipo (orientado a *widgets*) seja um pouco mais ágil, para ambas ferramentas, a manutenção e reutilização dos casos de testes é difícil pois mudanças na interface são bastante comuns.

Uma forma de assegurar o teste da interface que vem se mostrando eficiente, embora trabalhosa, é incluir o teste como uma funcionalidade do aplicativo, onde o próprio aplicativo é capaz de simular automaticamente seu funcionamento.

3.3.3 Orientação a Objetos

A utilização de técnicas da Orientação a Objetos tais como herança, polimorfismo, sobrecarga e encapsulamento, trouxe novos desafios aos testes estruturais tradicionais bem como novos termos no vocabulário de testes, definidos por Scott Ambler [AMBLER] como:

- Teste de Métodos – o ato de assegurar que um método opera conforme definido. O teste de método é comparável ao teste de unidade no mundo estruturado;

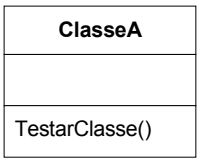
- Teste de Classes – pode ser visto como a combinação de teste de unidade com teste de integração. Unidade pois as classes e suas instâncias são testadas isoladamente e integração pois são verificados como os métodos e atributos das classes trabalham em conjunto;

- Teste de Regressão de Herança – ato de aplicar os casos de teste em todas as superclasses de uma subclasse dada.

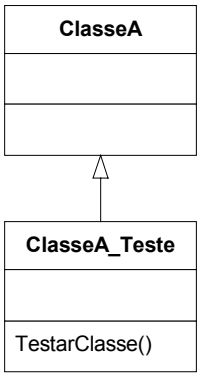
Basicamente temos duas formas de criar casos de teste para programas orientados a objetos.
A primeira delas é que cada classe contenha um método, chamados por exemplo `TestarClasse`, que é responsáveis pelo teste da classe. Este método de teste pode exercitar os outros métodos da classe (públicos e privados) bem como ser utilizado nas classes derivadas. Uma outra vantagem é que a manutenção deste caso de teste é simples já que ele está dentro do próprio código da classe.

Vale observar que o métodos de teste não deve constar na versão final do aplicativo. A segunda forma é criar uma subclasse derivada da classe que desejamos testar que implemente o método de teste. A diferença desta opção, em relação à primeira, é que o código de teste fica isolado.

Método de Teste na Classe

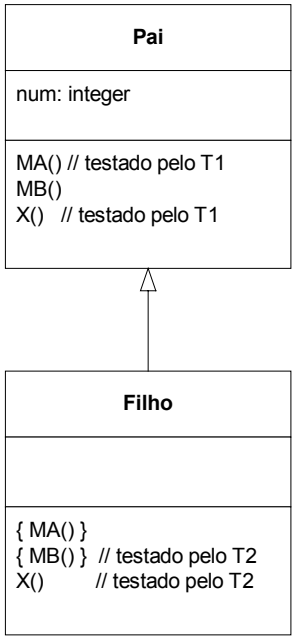


Método de Teste em Subclasse




```
class Pai
{
public:
    int num;

    void MA()
    {
        num = 1;
        X();
    }
    void MB()
    {
        num = 0;
        X();
    }
    virtual void X()
    {
        int a = 2 / num;
    }
};

class Filho : public Pai
{
public:
    void X()
    {
        int a = 2 * num;
    }
};

void Teste(void)
{
    Pai p;
    Filho f;

    p.MA(); // Caso de Teste T1
    f.MB(); // Caso de Teste T2
}
```

No exemplo ao lado, aparentemente o código apresenta 100% de cobertura das declarações, ou seja, os métodos `MA`, `MB` e `X` foram executados. No entanto se analisarmos com cuidado, perceberemos que `X` somente foi executado no **contexto** da classe Filho.
Os métodos `MA` e `MB` estão representados entre chaves { } na classe Filho pois estão implementados na classe Pai.



Se observamos bem, `MB` no **contexto** da classe Pai apresenta um erro, uma divisão por zero.

Este exemplo, com apenas algumas linhas, reflete as sutilezas dos testes em classes que apresentam polimorfismo.

3.3.4 Smoke

Normalmente um projeto de software envolve dezenas ou centenas de arquivos, compartilhados por um time de desenvolvimento, que são compilados, 'linkados' e combinados para a geração de um produto executável. Este processo é chamado montagem (*built*) do produto.

Smoke Teste é o processo no qual um produto de software é diariamente montado e submetido a uma série de testes para verificar sua funcionalidade básica [MCCONNELL].

O Smoke Teste, por ser simples, deve ser colocado em prática desde do início da implementação do produto e quando realizado diariamente:

- minimiza os riscos de integração pois incompatibilidades entre módulos são descobertas assim que surgem;
- fornece um mecanismo para monitorar o progresso do sistema;
- mantém o moral da equipe de desenvolvimento elevado através do acompanhamento diário da evolução do sistema;
- facilita o diagnóstico de defeitos. Por exemplo, se o produto funcionava no dia 10 e não funciona no dia 11, alguma coisa aconteceu neste período de 1 dia. Se o teste fosse feito semanalmente, teríamos 5 dias para analisar ao invés de 1.

O termo *smoke test* vem da indústria eletrônica. Depois de um reparo, o técnico ligaria o dispositivo e procuraria por fumaça. A presença de fumaça saindo de algum circuito diria ao técnico que alguma peça estaria recebendo muita corrente. Se não aparecesse fumaça imediatamente, o técnico, confiante, faria outros testes.

3.3.5 Stress e Carga

Boris Beizer [BEIZER] diz:

"Um dos mais comuns, porém inoportuno abuso da terminologia é tratar 'teste de carga' e 'teste de stress' como sinônimos. A conseqüência deste abuso, sem conhecimento, da semântica é que, geralmente, os sistemas não são devidamente testados à carga e nem sujeitados a significativos testes de stress.

1. **Teste de Stress** é sujeitar o sistema a uma carga desproporcional enquanto impede-se o uso dos recursos (e.g. RAM, discos) necessários para o processamento da carga. A idéia é 'stressar' o sistema até seu ponto de falha visando encontrar bugs que tornarão esta falha potencialmente prejudicial. Não espera-se que o sistema processe a sobrecarga sem os recursos adequados mas que se comporte (falhe) de uma maneira decente (e.g., não corrompendo nem perdendo dados). Os bugs e os modos de falhas descobertos durante o Teste de Stress podem ou não serem corrigidos dependendo da aplicação, modo de falha, conseqüências, etc.
A carga (fluxo imposto de transações) no Teste de Stress é freqüentemente distorcida de forma deliberada para forçar o sistema à condição de consumo rápido de recursos.

2. **Teste de Carga** é sujeitar o sistema à carga usual ou estatisticamente representativa. As duas principais razões para usar tal carga é testar a confiabilidade e performance do software. O termo 'Teste de Carga' é por si só muito vago e impreciso para garantir seu uso. Queremos dizer carga representativa, sobrecarga, carga alta ? Nos Testes de Performance, a carga varia de um mínimo (zero) até um valor máximo onde o sistema ainda opera sem falta de recursos ou suas transações sofrendo demoras excessivas."

3.3.6 Semeando Erros e Mutação de Programa

Semeando Erros (*Error Seeding*) e Mutação de Programa são técnicas para medir a qualidade dos dados de teste de acordo com sua eficiência e habilidade de detectarem defeitos [ZHU].

Semeando Erros é utilizado para estimar o número de defeitos ainda remanescentes em um software quando defeitos artificiais são introduzidos no programa em teste de uma forma aleatória e desconhecida pela pessoa que realiza os testes. Assume-se que estes defeitos artificiais apresentam a mesma dificuldade de detectar que os defeitos reais (inerentes).
O software é testado e os defeitos artificiais e inerentes são descobertos e contados separadamente.

Sejam:

$$r = \frac{dA}{tA}$$

dA defeitos artificiais descobertos
tA total de defeitos artificiais

dI defeitos inerentes descobertos
tI total de defeitos inerentes

então o número total de defeitos (*tI*) no programa pode ser estatisticamente previsto como:

$$tI = \frac{dI}{r}$$

O método Semeando Erros também pode ser utilizado para medir a qualidade do teste sendo a razão *r* (número de defeitos artificiais descobertos sobre o total de erros artificiais) a própria medida. Assim, se somente uma pequena quantidade de erros artificiais forem encontrados, concluímos que a qualidade do teste é baixa. Entretanto, a precisão desta medida depende de como os defeitos artificiais foram introduzidos. Normalmente, estes defeitos são plantados manualmente porém, na prática, é difícil introduzir defeitos da mesma natureza dos inerentes. Desta forma, os defeitos artificiais são bem mais fáceis de se localizar que os inerentes. Uma tentativa de superar este problema é através da Mutação de Programa.

Mutação de Programa consiste em construir uma coleção de programas alternativos que diferem do programa original de alguma forma. Estes programas alternativos são chamados mutantes. Dizemos que um mutante morre quando, submetido a um teste, ele produz um resultado diferente do programa original e vive quando ele completa o teste igual ao original. Cada mutante é executado sobre o conjunto de testes até que ele produza um resultado diferente do programa original ou complete o teste.
Mutantes permanecem vivos por dois motivos:

- o conjunto de dados de teste é inadequado;
- o mutante é equivalente ao programa original

Quando um grande número de mutantes sobrevivem a um teste, não temos nenhuma razão a mais para acreditar que o programa original é o correto. Provavelmente o comportamento do programa

não é governado pelos dados de teste escolhidos ou estes dados não exercitaram o trecho do programa que sofreu a mutação.
Mutação de Programa pode ser um teste estrutural quando os mutantes são produzidos manualmente e funcional quando são criados por uma ferramenta automatizada.

Vamos considerar, como exemplo, a função `NdxMax` que varre um array e retorna o índice do maior elemento deste array e seu mutante `NdxMaxMut` que teve a comparação maior (>)trocada por maior igual (>=).

Código Original

```
int NdxMax(int *array, int qtd)
{
    int ndx = 0;

    for (int i=1; i<qtd; i++)
        if (array[i] > array[ndx])
            ndx = i;

    return ndx;
}
```

Código Mutante

```
int NdxMaxMut(int *array, int qtd)
{
    int ndx = 0;

    for (int i=1; i<qtd; i++)
        if (array[i] >= array[ndx])
            ndx = i;

    return ndx;
}
```

Dados de Teste	Resultado Esperado	Resultado `NdxMax`	Resultado `NdxMaxMut`
`int a[5] = {0,3,9,6,7};`	2	2	2
`int b[5] = {0,3,9,6,9};`	2	2	4

Para os dados de teste contidos no array `a`, as duas funções se comportam da mesma maneira, porém, com o array `b` conseguimos matar o mutante.

4. Onde procurar Bugs

Steve Maguire em "Writing Solid Code" [MAGUIRE], ressalta que rotinas de tratamento de erro geralmente apresentam bugs pois elas são raramente testadas. Além destas rotinas, podemos citar outras situações candidatas a apresentarem bugs [SHIMEAL]:

4.1 Tratamentos de Erro

Algumas rotinas de tratamento de erro não são efetivamente testadas pois os erros que elas tratam não ocorrem frequentemente ou são improváveis de acontecer, deixando, desta forma, bugs escondidos. Cobertura dos Caminhos é eficaz neste caso.

```
char *p = (char *)malloc (32);
if (p == NULL)
{
    printf ("Erro na função %s\n");
    // onde está a variável com o nome da função ?
}
```

4.2 Extrema

Extrema são trechos de código que são executados em raras ocasiões ou quando problemas acontecem, por exemplo, um disco atinge sua capacidade ou um erro de comunicação na rede. Tratamentos de Erro são um caso especial de Extrema.

4.3 Algoritmos Complexos

Códigos que implementam algoritmos têm maior possibilidade de apresentarem defeitos do que códigos não-algorítmicos. Somente executar o código não é suficiente para localizar bugs. O correto seria submeter ao código dados que já foram processados por uma fonte independente e comparar os resultados.

4.4 Código "Esperto" (*cool code*)

Código "Esperto" é aquele que apresenta um artifício de programação ou foi, por exemplo, otimizado ou escrito de forma compacta (*terse code*). Mesmo que o programador que escreveu o código originalmente sabia o que ele estava fazendo, este tipo de código é um forte candidato a bugs quando sofre uma manutenção por outra pessoa.

Código Compacto

```
for (i=0; i<strlen (szStr); i++)
{
    ...
}
```

código compacto porém ineficiente pois o tamanho da string é calculado a cada iteração do loop

Código Não Otimizado

```
while (expressão)
{
    A;
    if (f)
        B;
    else
         C;
    D;
}
```

Código Otimizado

```
if (f)
{
    while (expressão)
    {
        A;
        B;
        D;
    }
}
else
{
    while (expressão)
    {
        A;
        C;
        D;
    }
}
```

No exemplo acima, o código otimizado é mais difícil de ser mantido. Por exemplo, se a `expressão` for alterada ou acrescentarmos uma nova função `E` após a chamada da função `D`, dois pontos do código terão que ser verificados.

Rick Booth [BOOTH] sustenta que código rápido (otimizado) geralmente viola os princípios da estruturação de um programa tornando-o perigoso e de difícil manutenção.

4.5 Interfaces Complexas

Interfaces complexas geralmente envolvem códigos compartilhados por mais de um programador. Nestas condições podem não ficar claro para os programadores de quem é a responsabilidade por partes chaves da interface, levando a erros como, por exemplo, dupla deleção e *memory leaks*.

Dupla Deleção

```
// autor: programador A
void F1()
{
    ...
    OBJ *p = new OBJ;
    Funcao(p);
    // p já foi deletado !
    delete p;
    ...
}

// autor: programador B
void Funcao(OBJ *p)
{
    ...
    delete p;
}
```

Memory Leak

```
// autor: programador A
void F1()
{
    ...
    OBJ *p = new OBJ;
    Funcao(p);
    ...
    // ninguém deletou p !!
}

// autor: programador B
void Funcao(OBJ *p)
{
    ...
}
```

4.6 Códigos e Manutenções de Emergência

Quando as pessoas envolvidas em um projeto estão trabalhando sob grande pressão e correndo contra o relógio, as regras básicas de programação, comentários nos programas, documentação e testes são deixados de lado. Desta forma o sistema se torna cada vez mais instável, o número de defeitos aumenta, mais códigos e manutenções de emergência são necessários. O projeto corre grande risco.

4.7 Novos Programadores

Novos programadores não são necessariamente programadores inexperientes, são aqueles recém integrados à equipe de desenvolvimento, ou seja, aqueles que ainda não têm familiaridade com o código. Eventualmente defeitos são introduzidos no sistema por novos programadores.

5. Conclusão

Softwares podem ser testados durante todo o seu ciclo de desenvolvimento com diversos níveis de rigor e detalhe. Podemos testar desde a sua estrutura interna até sua funcionalidade básica passando pela interface com o usuário. Podemos também realizar testes quando os softwares sofrem alterações de qualquer natureza.

Porém, das diversas técnicas apresentadas neste artigo, devemos escolher aquelas que melhores se adaptam ao nosso ciclo de desenvolvimento e categoria de sistema. Os testes devem ser rigorosos o suficiente para serem efetivos, ou seja, descobrirem defeitos e, ao mesmo tempo, flexíveis para se adaptarem às constantes evoluções e alterações dos sistemas.

No mercado atual, exigente e competitivo, que demanda sistemas cada vez mais complexos com prazos e custos cada vez mais curtos, a atividade de teste é vital dentro do ciclo de desenvolvimento.

6. Glossário

- bug

O termo bug teve sua origem nos Estados Unidos, na época que os computadores eram compostos por válvulas, quando uma série de erros inexplicáveis foram causados por mariposas que voavam dentro dos computadores [IPL]. Bug é sinônimo de defeito.

- *debug*

Processo de localização e correção de um defeito **conhecido** num programa. Depurar.

- defeito, falta

No contexto, problema ou fraqueza no código fonte de um programa. Defeitos são geralmente introduzidos através de um erro que podem ser de digitação, lógica ou matemática.

- encapsulamento

Habilidade de conter ou esconder informações sobre um objeto tais como suas estruturas internas de dados e código. Encapsulamento isola a complexidade interna da operação de um objeto do restante da aplicação [MSDN].

- erro

Ato cometido por uma pessoa que resulta num defeito [CORNETT].
A definição de erro apresentada pelo Cambridge Dictionary é: ação, decisão ou julgamento que produz um resultado não desejado ou não intencional.

- falha

No contexto de Engenharia de Software falha é a não conformidade com os requisitos de software.
É a divergência no comportamento esperado de um sistema.
É a manifestação de um defeito [CORNETT].

- polimorfismo

Habilidade de ter métodos com mesmo nome mas com diferentes conteúdos para diferentes classes. Polimorfismo significa que podemos ter duas classes com diferentes implementações ou códigos mas com a mesma interface [MSDN].

- *profiler*

Ferramenta de diagnóstico para analisar o comportamento, em tempo de execução, de programas [MSDN].

- *short circuit*

Na linguagem C expressões contendo os operadores lógicos E e OU são avaliadas da esquerda para a direita. Se o valor da primeira operado for suficiente para determinar o resultado da operação, o segundo operador não é avaliado [MSDN].

- sobrecarga

Prática de fornecer mais de uma definição para um dado nome de função no mesmo escopo [MSDN].

- validação

Conjunto de atividades para assegurar que o programa está sendo construído de acordo com os requisitos [PRESSMAN].
Durante a validação descobrimos se estamos fazendo o produto correto, descobrimos se o que foi especificado era realmente o que o usuário queria.

- verificação

Conjunto de atividades para assegurar que um programa implementa corretamente uma função especificada [PRESSMAN].
Durante a verificação descobrimos se estamos fazendo corretamente o produto.

- *widgets = windows+ gadgets*

Componentes básicos da interface gráfica tais como botões, *list boxes*, *check boxes*, *scroll bars*, *radio buttons*, etc.

7. Bibliografia

[ADRION] Adrion, W. Richards; Branstad, Martha A.; Cherniavsky, John C., "Validation, Verification, and Testing of Computer Software", *ACM Computer Surveys*, Vol. 14, No. 2, Junho 1982

[AMBLER] Ambler, Scott, "Testing Objects", *Software Development*, Agosto 1996

[BECK] Beck, Kent, *Extreme Programming Explained*, Addison Wesley, 2000, p. 100

[BEIZER] Beizer, Boris; *Frequently Asked Questions (FAQ)*; comp.software.testing newgroups

[BOOTH] Booth, Rick, *Inner Loops*, Addison Wesley, 1997

[BRADLEY] Bradley, N. Scott, "Software Testing Cycles", *Dr. Dobb's Journal*, Fevereiro 1994

[CORNETT] Cornett, Steve, "Code Coverage Analysis", Bullseye Testing Technology

[GIANTURCO] Gianturco, Mark D., "Testing Techniques for Quality Software", *Software Development*, Agosto 1994

[IPL] Information Processing Ltd., "An Introduction to Software Testing", 1996

[IPLa] Information Processing Ltd., "Designing Unit Test Cases", 1996

[MAGUIRE] Maguire, Steve, *Writing Solid Code*, Microsoft Press, 1993, p. 78

[MCCONNELL] McConnell, Steve, *Rapid Development*, Microsoft Press, 1996, p. 73, 74

[MSDN] Microsoft Developer Network

[PRESSMAN] Pressman, Roger S., *Software Engineering – A Practitioner's Approach*, McGraw-Hill, p.479

[PRESSMANa] Pressman, Roger S., *Software Engineering – A Practitioner's Approach*, McGraw-Hill, p.465

[RABIN] Rabin, Steve, "Software Testing: Concepts, Tools, and Techniques", *Software Development*, Novembro 1993

[SHIMEAL] Shimeall, Stephen, "Writing Robust Regression Tests", *Software Development*, Agosto 1996

[ZHU] Zhu, Hong; Hall, Patrick A. V.; May, John H. R., "Software Unit Test Coverage and Adequacy", *ACM Computer Surveys*, Vol. 29, No. 4, Dezembro 1997