Revista Engenho, vol. 3 / 2011

TESTABILIDADE DE SOFTWARE: VISÃO GERAL

César Tegani Tofanini[1] e Fábio Vieira Teixeira[2]

[1]Centro Universitário Padre Anchieta, Jundiaí, SP, Brasil

[2]Universidade Estadual de Campinas, Campinas, SP, Brasil

Resumo

Neste trabalho, são apresentados os conceitos relacionados à testabilidade de *software*. As características desejáveis para se obter testabilidade são: controlabilidade, observabilidade, disponibilidade, simplicidade, estabilidade e informação, sendo que as mais importantes são controlabilidade e observabilidade. É apresentado o conceito de testabilidade de domínio, através das extensões de observabilidade e controlabilidade, assim como, o conceito de testabilidade como probabilidade de se revelar defeitos, que tem como base a ideia da perda de informação. Medidas para avaliar e aumentar o grau de testabilidade são discutidas por meio de exemplos práticos que podem ser aplicados no processo de desenvolvimento de um *software*.

Palavras-chave: testabilidade, controlabilidade, observabilidade

Abstract

This paper shows the concepts related to testability of software. Desirable characteristics for achieving testability are: controllability, observability, availability, simplicity, stability and information, and the most important are controllability and observability. It shows the concept of domain testability, through the extensions of observability and controllability, it also shows the concept of testability as likely to reveal defects, which is based on the idea of information loss. Measures to assess and increase the degree of testability are discussed through practical examples that can be applied in development of software.

Keywords: testability, controllability, observability

1- Introdução

O processo de teste pode ser considerado o mais custoso do ciclo de vida de um *software*. Geralmente, no desenvolvimento de um projeto de *software* é muito comum a necessidade de redução de custos devido às limitações estipuladas no projeto. O ponto fundamental é tornar a tarefa de teste mais simples buscando maior eficiência na tarefa de revelar defeitos. Essa característica pode ser alcançada nas diferentes fases do projeto, proporcionando a redução dos custos, a simplificação das operações de teste e aumento da qualidade do *software*. Segundo as definições clássicas (PRESSMAN, 2001; CHOWDHARY, 2009; GUPTA; SINHA, 2009), testabilidade é a aptidão de um *software* para atender um critério de testes através da simulação de todas as situações de forma eficiente.

Grandes companhias de desenvolvimento de *software*, onde o suporte a um produto pode durar muitos anos (CHOWDHARY, 2009), têm demonstrado interesse em desenvolver a testabilidade para melhorar seus processos de teste.

Na literatura, não existe um consenso sobre todas as características de um *software* testável. Diversos autores dão sugestões de como um *software* deveria ser projetado para se atingir um alto grau de testabilidade (CHOWDHARY, 2009; GUPTA; SINHA, 2009; BACH, 2003; GAO, 2000; MICROSOFT, 2009; VOAS; MILLER, 1995; FREEDMAN, 1991; PETTICHORD, 2002). Entretanto, é importante ressaltar que as características relacionadas têm muita coisa em comum. Os principais aspectos relacionados à testabilidade, que são afetados no projeto de *software* e que interferem diretamente na viabilidade e automação de testes, são a controlabilidade e a observabilidade.

Além de técnicas utilizadas para aumentar a testabilidade de um *software*, é importante termos ferramentas para avaliar esta testabilidade. As métricas existentes incluem as baseadas nas definições de observabilidade e controlabilidade de Freedman (1991), baseadas no conceito de perda de informação, definida por Voas e Miller (1995) e baseadas na técnica de análise de sensibilidade. Por fim, os impactos enfrentados no projeto e na implementação de um *software* quando utilizamos os conceitos de testabilidade são discussões importantes.

2- Conceitos e Definições

> (1) O grau em que um sistema ou componente facilita o estabelecimento de critérios de teste e a execução dos de testes para determinar se esses critérios foram cumpridos. (2) O grau em que um requisito é expresso em termos que permitem o estabelecimento de critérios de teste e execução de testes para determinar se esses critérios foram cumpridos. (THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, 1990, p. 76).

Nesta mesma linha, Pressman (2001) e Chowdhary (2009) definem o conceito de testabilidade como o quão fácil um programa de computador pode ser testado. Gupta e Sinha (2009) definem a testabilidade como a preparação que pode ser realizada durante as fases de projeto e codificação de um *software*, de forma a permitir que o processo de teste seja realizado de forma fácil e sistemática.

As definições de testabilidade acima nos forçam a possuir um critério para teste se desejarmos obter o grau de testabilidade de um *software*. Assim, o conceito testabilidade tem sido usado para discutir a facilidade com que um determinado critério de seleção de dados de teste pode ser satisfeito durante um teste. Adicionalmente, pode-se dizer que a atividade de

teste pode revelar defeitos, a testabilidade não, mas testabilidade pode sugerir os lugares onde estão os defeitos, algo que o teste não pode.

2.1- Características que determinam a testabilidade

De forma geral, algumas características são desejáveis para que um *software* seja testável. A seguir, seis características elencadas por Bach (2003) e referenciada por outros autores (PRESSMAN, 2001; CHOWDHARY, 2009; GAO, 2000): 1) Controlabilidade: existe uma interface para conduzir os testes, os estados de variáveis e condições de *hardware* podem ser controlados pelo testador e os componentes de *software* podem ser testados de forma independente; 2) Observalidade: registros de estados atuais e anteriores do sistema estão disponíveis para consulta, resultados diferentes são gerados para entradas diferentes, todos os elementos que afetam a saída são visíveis e a saída incorreta pode ser facilmente percebida; 3) Disponibilidade: o código fonte do programa deve ser acessível, defeitos não impedem que os testes continuem e o produto evolui em fases; 4) Simplicidade: a implementação deve ser o mínimo necessário para atender os requisitos, os módulos devem ser coesos e com baixo acoplamento; 5) Estabilidade: as alterações de *software* são controladas e não frequentes. Alterações realizadas não devem invalidar os testes já existentes; 6) Conhecimento: metodologia e tecnologia utilizada é similar a outros projetos já conhecidos pelos testadores. A documentação é acessível, organizada, detalhada e precisa.

2.1.1- Observabilidade e Controlabilidade

Um componente de *software* é observável se seu comportamento externo bem como seu comportamento interno pode ser observado em tempo real ou gravado para posterior análise (GUPTA, 2009). As entradas fornecidas, como parte do teste, devem produzir resultados distintos. Além disso, um resultado incorreto e erros internos podem ser facilmente identificados (PRESSMAN, 2001). O grau de observabilidade de um programa pode ser obtido pela capacidade de determinar de forma econômica se um resultado obtido por um teste é o resultado esperado (MICROSOFT, 2009).

Um componente de *software* é controlável se através de sua interface de comandos pode ser inicializado com diferentes estados conforme requerido pelos testes (GUPTA, 2009). Testes podem ser especificados, automatizados e reproduzidos convenientemente (PRESSMAN, 2001). Obviamente, quanto mais controlável é um componente, mais previsível é o seu comportamento (MICROSOFT, 2009).

2.2- Testabilidade de domínio

Se um componente de *software* for testado duas vezes com o mesmo valor de entrada, os dois resultados de execução deveriam ser iguais. Se isso não acontecer, os dados de entrada estão incompletos, ou seja, as saídas não estão dependendo somente das entradas. Neste caso, estados internos do componente, não conhecidos pelo testador, interferem no resultado de saída. Nesta situação, diz-se que o componente está com inconsistências de entrada.

Por outro lado, se a saída de um componente de *software* é especificada dentro de um intervalo de valores, deveria ser possível construir uma entrada de teste cuja execução poderia cobrir qualquer um dos valores do intervalo de saída desejado. Se não for possível especificar um valor de entrada para algum dos valores do intervalo de saída, afirma-se que o componente está com inconsistências de saída.

Componentes de *software* com inconsistências de entrada e saída não são facilmente testáveis. Estas inconsistências podem ser evidências de um defeito de *software*, mas não implica, de fato, em sua existência.

Uma vez que o processo de teste verifica a tendência de um *software* realizar as entradas e saídas segundo uma determinada especificação, é necessário identificar as características das entradas e saídas explícitas, dada pela interface do componente de *software* e das entradas e saídas implícitas dada pelo estado interno do componente de *software*.

Parte das dificuldades de uma atividade de teste está relacionada à identificação e à especificação das entradas e saídas implícitas de um componente de *software* para utilização como itens auxiliares que podem ser controlados e observados durante os testes.

Segundo Freedman (1991), a princípio, a maioria dos componentes de *software* não é observável ou controlável. As alterações necessárias para se obter um componente de *software* observável são chamadas de *extensões observáveis*. As alterações necessárias para se obter um componente de *software* controlável são chamadas de *extensões controláveis*.

2.2.1- Definição Formal de Observabilidade

De forma geral, dado um componente de *software* $\mathbb{P}$ que recebe como entrada um conjunto de dados $\mathbb{E}$ e tem como saída $\mathbb{S}$, $\mathbb{P}$ é observável se, para um subconjunto $\mathbb{E}_1$ da entrada, resulta em um subconjunto $\mathbb{S}_1$ da saída e, para um subconjunto $\mathbb{E}_2$ da entrada, resulta em um $\mathbb{S}_2$ da saída de forma que $\mathbb{E}_1$ é diferente de $\mathbb{E}_2$ e $\mathbb{S}_1$ é diferente de $\mathbb{S}_2$. Desta maneira, um componente de *software* é observável se diferentes resultados são gerados a partir de diferentes dados de entrada.

Por exemplo, considere a seguinte função:

```
function F(X: in INTEGER) return INTEGER is
begin
        return x * VARIAVEL_GLOBAL;
end F;
```

Esta função não é observável, pois ela depende de uma variável global. Execuções com os mesmos dados de teste podem resultar em saídas diferentes.

2.2.2- Definição Formal de Controlabilidade

De forma geral, dado um componente de *software* $\mathbb{P}$ que recebe como entrada um conjunto de dados $\mathbb{E}$ e tem como saída $\mathbb{S}$, $\mathbb{P}$ é *controlável* se, para todo conjunto $\mathbb{E}_i$: { todas as execuções de $\mathbb{P}(\mathbb{E}_i)$ } = $\mathbb{TF}$, onde $\mathbb{TF}$ é o conjunto de valores possíveis para o tipo de dado de $\mathbb{S}$. Desta maneira, um componente de *software* é controlável se o conjunto de resultados que ele retorna é igual ao conjunto dos valores dados pelo seu tipo de retorno.

Por exemplo, considere a seguinte função:

```
POSITIVE = {0,1,2,...,MAX_INTEGER}
function G(X: in POSITIVE) return POSITIVE is
begin
        return x mod 3;
end G;
```

Esta função não é controlável, pois $\mathbb{G}$ retorna um subconjunto do tipo POSITIVE (conjunto {0,1,2}) para toda entrada. Portanto, não há dados de entrada que retornem valores entre 3 e MAX_INTEGER.

A observabilidade não é requerida para a controlabilidade: componentes de *software* sem dados de entrada podem ser controláveis, mas não observáveis. Se um componente de *software* é observável e controlável, então a função dada pelo componente é do tipo *onto*[1] (FREEDMAN , 1991).

2.3- Testabilidade como probabilidade de se revelar defeitos

Voas e Miller (1995) definem testabilidade de maneira diferente da tradicional: "Testabilidade é a probabilidade que um *software* falhará em sua próxima execução durante o teste se o *software* possuir um defeito". Ou seja, um *software* suscetível a revelar seus defeitos tem alta testabilidade.

Com esta concepção, a testabilidade dá o grau de dificuldade para a detecção de um defeito em um lugar específico durante a atividade de testes. Se após a realização de testes, de acordo com o grau especificado pela testabilidade, nenhuma falha for observada, então se pode ter uma razoável certeza que o programa está correto.

---

[1] Uma função $\mathbb{F}$ de $\mathbb{A}$ para $\mathbb{B}$ é chamada *onto* se para todo $\mathbb{b}$ em $\mathbb{B}$ existe um $\mathbb{a}$ em $\mathbb{A}$ de forma que $\mathbb{f}(\mathbb{a}) = \mathbb{b}$. Ou seja, todos os elementos em $\mathbb{B}$ são referenciados.

2.3.1- Perda de Informação

A observação empírica do que ocorre na execução de um programa sugere que um erro[2] pode ser cancelado. O grau em que este fenômeno ocorre pode ser quantificado por meio de uma análise estática do código ou da inspeção da especificação de um componente de *software*. Assim, são adquiridas informações que dão uma ideia sobre a probabilidade de ocorrer o cancelamento de um erro. Isto nos dá indícios para mensurar a quantidade de defeitos que permanecerão escondidos durante os testes. Este acontecimento é denominado perda de informação (VOAS; MILLER, 1995). A perda de informação aumenta a chance do cancelamento de um erro e este fato diminui a testabilidade do *software*.

2.3.1.1- Perda de Informação Explícita

A perda de informação explícita ocorre quando variáveis não são validadas durante a execução do *software* ou ao final, no resultado da execução. Este tipo de perda de informação, normalmente ocorre como resultado do encapsulamento de dados. Apesar do encapsulamento de dados ser aceito como boa prática de programação, uma vez que as variáveis locais não estão disponíveis para identificar um estado incorreto e ajudar na revelação de defeitos, esta técnica não é boa do ponto de vista da testabilidade, pois causa a diminuição da observabilidade do componente de *software*.

2.3.1.2- Perda de Informação Implícita

A perda de informação implícita ocorre quando dois ou mais parâmetros de entrada diferentes, passados para um componente de *software*, produzem o mesmo resultado na saída. Como exemplo, considere uma função definida pelo programador que recebe dois parâmetros inteiros e produz uma saída lógica, nesta função, muitos valores diferentes de entrada são possíveis, enquanto a saída somente pode assumir o valor ‘0’ ou o valor ‘1’.

3- Critérios e métricas de testabilidade

3.1- Métricas para testabilidade de domínio

[2] Voas e Miller (1995) usam o termo *data state error*. Neste texto usamos apenas o termo *erro.*

Freedman (1991) apresenta o conceito de Testabilidade de Domínio. Formalmente, as métricas de testabilidade de domínio são definidas em termos das extensões observáveis e controláveis.

Dada a seguinte função:

function **F** (E: in *T*) return *TF*;

e a extensão observável com índice de observabilidade $\mathbb{m}$:

function **FO** (E: in *T*; E1: in *T1*; E2: in *T2*; ...; En: in *Tn*)
return *TF*;

Para um teste exaustivo, o número extra de casos de teste requeridos seria um múltiplo de

$$|T1| * \ldots * |Tn| \leq |T_{max}|^n$$

|Tn| denota a cardinalidade do domínio e $|T_{max}|$ é a cardinalidade máxima do domínio considerando todos os tipos de entrada.

O esforço extra, associado com as $\mathbb{m}$ entradas extras, depende dos seus tipos de dados.

Este fator é normalizado considerando o número efetivo de entradas binárias extras:

$$Ob = \log_2(|T1|*\ldots*|Tn|)$$

onde *Ob* é a medida de observabilidade e corresponde ao número de entradas binárias extras requeridas para transformar em observável o componente de *software*.

Dada a seguinte sub-rotina:

procedure **P** (E: in *T*; O1: out *T1*; ...; Om: out ***Sm***);

e a extensão controlável com índice de controlabilidade $\mathbb{m}$:

procedure **PC** (E: in *T*; OC1: out *T1*; ...; Om: out ***Tm***);

define-se:

$$Ct = \log_2(|T1|*\ldots*|T_m|)$$

onde *Ct* é a medida de controlabilidade e corresponde ao número de saídas binárias que precisam ser modificadas para transformar em controlável o componente de *software*.

Nota-se que:

$$0 \leq Ob \leq \mathbb{m} * \log_2(|T_{max}|)$$

$$0 \leq Ob \leq \mathbb{mm} * \log_2(|T_{max}|)$$

Todo componente de *software* tem um grau de testabilidade que é intrínseco (GUPTA, 2009). Estas duas medidas (*Ob* e *Ct*) dão a ideia do esforço necessário para os testes, depois de aumentarmos este grau de testabilidade.

Com o uso das extensões, observamos que testes adicionais precisarão ser realizados para validarmos estas extensões. Entretanto, é importante observar que os testes exigidos anteriormente eram fracos, pois não exercitaria o *software* da maneira adequada.

3.2- Razão domínio/variação

Voas e Miller (1995) apresentam a razão domínio/variação. Para isso, dividem a perda de informação entre implícita e explícita. A análise estática do código é usada para quantificar

o grau de perda de informações explícitas e a inspeção da especificação é usada para quantificar o grau de perda de informação implícita.

Investigações sugerem que o grau de perda de informação implícita que pode ocorrer durante a execução pode ser visível na especificação do programa. A métrica que pode dar esta informação é denominada de razão domínio/variação (DRR[3]). A DDR de uma especificação é a razão entre a cardinalidade do domínio de entrada e a cardinalidade do domínio de saída. É denotada por $\alpha : \beta$[4], onde α é a cardinalidade do domínio de entrada e β é a cardinalidade do domínio de saída. Em geral, com o aumento da DRR para uma especificação, o potencial para perda de informação implícita em uma implementação para esta especificação também aumenta.

3.2.1 Relação Entre Perda de Informação e DRR

A perda de informação implícita é comum em muitos operadores internos das linguagens de programação modernas. Considere as funções listadas na tabela 2. Esta tabela mostra funções com diversos graus de perda de informação implícita e DRR. Uma função classificada como 'sim' para perda de informação implícita[5] tem maior probabilidade de produzir uma saída idêntica para diferentes parâmetros de entrada. Uma função classificada como 'não'[5] sugere que diferentes parâmetros de entrada produziriam diferentes saídas. Considere o baixo grau de DRR da função 'not'[6]. A função tem este grau devido ao fato das cardinalidades de entrada e saída serem iguais, ou seja, $\alpha = \beta$.

[3] Do termo original *domain/range ratio*, conforme definido por Voas (1992).
[4] Conforme notação utilizada por Voas e Miller (1995).
[5] Classificação dada pela terceira coluna na tabela 2.
[6] Função definida na tabela 2: Linha 14, Coluna 2.

	Função	Perda de Informação Implícita	DRR
1	$f(a) = \begin{cases} 0 & \text{if } a < 0 \\ a & \text{otherwise} \end{cases}$	sim	$\infty_I : \infty_I/2$
2	$f(a) = a + 1$	não	$\infty_I : \infty_I$
3	$f(a) = a \bmod b$	sim	$\infty_I : b$
4	$f(a) = a \text{ div } b$	sim	$\infty_I : \infty_I/b$
5	$f(a) = \text{trunc}(a)$	sim	$\infty_R : \infty_I$
6	$f(a) = \text{round}(a)$	sim	$\infty_R : \infty_I$
7	$f(a) = \text{sqr}(a)$	não	$2 \cdot \infty_R : \infty_R$
8	$f(a) = \text{sqrt}(a)$	não	$\infty_R : \infty_R$
9	$f(a) = a/b$	não	$\infty_R : \infty_R$
10	$f(a) = a - 1$	não	$\infty_I : \infty_I$
11	$f(a) = \text{even(a)}$	sim	$\infty_I : 2$
12	$f(a) = \text{sin(a)}$	sim	$\infty_I : 360$
13	$f(a) = \text{odd(a)}$	sim	$\infty_I : 2$
14	$f(a) = \text{not(a)}$	não	$1 : 1$
15	$f(a, b) = \text{(a)or(b)}$	sim	$4 : 2$

Tabela 2 - DRR e perda de informação implícita de várias funções; ∞_I é a cardinalidade dos números inteiros e ∞_R é a cardinalidade dos números reais. (VOAS; MILLER, 1995).

3.3- Análise de sensibilidade

Voas (1992) apresenta uma interessante técnica dinâmica para estimar estatisticamente três características de um programa: 1) A probabilidade de um local específico de um programa ser executado, 2) A probabilidade deste local afetar o estado do programa e 3) A probabilidade do estado produzido por este local afetar o resultado de saída do programa.

A técnica descrita é chamada de propagação, infecção e execução (PIE). Esta técnica é baseada nas técnicas de teste baseados em defeito e teste de mutação. Entretanto, PIE é distinta destas técnicas, pois não revela a existência de defeitos. Ao invés disso, identifica os locais de um programa onde defeitos, se existirem, são mais propensos a permanecerem escondidos durante a atividade de testes.

A análise de sensibilidade, com o uso das informações obtidas pela análise PIE, procura responder as seguintes questões: 1) Onde os recursos limitados de teste podem ser usados para obtermos mais benefícios? 2) Quando deveríamos usar uma outra técnica, diferente de teste, para validar nosso *software*? 3) Qual o grau de testes necessários para garantir que um determinado local, provavelmente, não está escondendo um defeito? 4) Quando precisamos reescrever o *software* para torná-lo menos propenso a esconder defeitos?

Um local no modelo PIE é uma atribuição, entrada, saída ou a condição associada a uma estrutura de programação. Para cada parte do modelo existe um algoritmo distinto.

3.3.1- Análise de Execução

A análise de execução é um método baseado na estrutura de um programa. Ela estima a probabilidade da execução de um local[7] quando dados de entrada são selecionados a partir de uma distribuição de entrada indicando a probabilidade de execução de um determinado comando. Esta probabilidade é determinada pelo número de vezes que um local é executado em relação ao total de casos de teste executados. Por exemplo, se 100 casos de teste são executados e o local é exercitado para 40 destes testes, a probabilidade de execução é 0,4.

3.3.2- Análise de Infecção

Para estimar a probabilidade de infecção é realizada uma série de mutações sintáticas de um determinado local do programa. Após cada mudança, o programa é re-executado com dados de entrada aleatórios. Os estados dos programas mutantes são então comparados com o estado do programa original, considerando o mesmo local de execução. Se o estado de um programa mutante for diferente do estado do programa original, é dito que aconteceu uma infecção. A probabilidade de infecção é determinada pelo número de vezes que um local é infectado em relação ao total de casos de teste executados. Por exemplo, se 100 casos de teste são executados e o local é infectado para 20 destes testes, a probabilidade de infecção é 0,2.

3.3.3 Análise de Propagação

A análise de propagação estima a probabilidade de propagação de locais do programa com infecções simuladas. Um determinado local do programa é monitorado durante a execução de testes aleatórios. Após a execução do local, o estado resultante do programa é infectado, atribuindo um valor aleatório para algum dado deste estado. Em seguida, o programa continua a execução dos comandos subseqüentes até um resultado de saída. O resultado obtido pela saída do programa executado com a infecção é então comparado com o resultado que seria obtido pela saída do programa original, com os mesmos dados de teste, mas sem a infecção simulada. Se os resultados de saída forem diferentes, é dito que ocorreu a propagação. A probabilidade de propagação é determinada pelo número de vezes que com o

[7] O termo local refere-se a um determinado comando de um programa.

uso de um local com infecção simulada se obtém um resultado de saída diferente do estado de saída do programa original com o mesmo dado de teste em relação ao total de casos de teste executados. Por exemplo, se 100 casos de teste são executados e há a propagação do estado incorreto do programa para 10 destes testes, a probabilidade de propagação é 0,1.

3.3.4 Estimativas Resultantes

Quando análise de sensibilidade é finalizada, três conjuntos de probabilidades estimadas para cada local no programa são obtidos.

Para um teste aleatório identificar a presença de um defeito, deve ocorrer a execução, a infecção e a propagação. Neste caso, a execução resulta em uma falha. A análise de sensibilidade usa os dados da análise PIE para estimar a probabilidade mínima de um local com defeito, se o defeito existir, gerar uma falha. Assim, esta análise ordena os locais do programa baseado em sua habilidade em esconder defeitos.

Desta forma, o produto da média das estimativas resulta na probabilidade estimada de falha se este local tivesse um defeito. A sensibilidade de um local é a predição da probabilidade mínima que um defeito no local irá resultar em uma falha na execução do programa. Se um local tem sensibilidade 1.0 para um conjunto de dados de testes, então é previsível que cada entrada do conjunto de dados de teste resultará em uma falha na execução do programa. Se o local tem sensibilidade 0.0 para um conjunto de dados de testes, então é previsível que não importa se um defeito está presente no local, nenhum dado de entrada do conjunto de dados de teste poderá revelar o defeito durante o teste.

4- Obtendo testabilidade

A busca por testabilidade foca no desenvolvimento de testes de forma mais simples, utilizando a especificação e os recursos disponíveis em um projeto de *software*. Vários fatores contribuem para a baixa testabilidade de um sistema, portanto diversas técnicas são propostas para minimizar esse problema.

Pettichord (2002) sugere aumentar a visibilidade para prover maior testabilidade a um programa. Esse atributo permite observar os resultados, efeitos colaterais e os estados internos de um *software*. É basicamente a disponibilidade proporcionada para acesso ao código fonte, documentos de projetos e os registros de mudança do programa.

Outra técnica sugerida é a utilização de saídas explícitas, chamadas de modo *verbose*. Essa técnica permite a visualização da operação interna de um *software*. Se considerarmos,

por exemplo, o envio de um email em sistemas UNIX, o que o usuário terá em sua interface será basicamente os parâmetros passados para realizar o envio, entretanto, utilizando o modo *verbose*, é possível que ele acompanhe a comunicação entre o cliente e o servidor e verifique todas as operações. A técnica de *verbose* nada mais é do que uma verificação de eventos (*logging*). Um *log* torna mais fácil o entendimento da operação de um sistema, pois facilita a identificação de falhas, permitindo então a depuração. Um bom arquivo de *log*, do ponto de vista da testabilidade, deve incluir as seguintes informações: eventos de usuários, processamentos significativos do sistema, marcos principais do sistema como inicialização ou desligamento, assim como estabelecimento de conexões, estados inconsistentes ou inesperados do programa, eventos não usuais e conclusão das operações.

Outra técnica muito útil para facilitar os testes é a injeção de defeitos. Através dela é possível que o testador realize uma simulação do comportamento do sistema diante de uma situação específica. Uma possível aplicação seria o teste referente ao comportamento do programa durante o armazenamento de um determinado tipo de arquivo em um local com espaço de armazenamento insuficiente. Ao invés do testador, por exemplo, ter que preencher a mídia de armazenamento para então realizar os testes, basta ele injetar um defeito que possibilite a simulação desta situação.

Segundo Pettichord (2002), as interfaces de programação[8] proporcionam maior testabilidade a um *software*. Este tipo de interface possibilita a automação do processo de teste, ao contrario das interfaces gráficas de usuário.

4.1- Minimização da perda de informação

Voas e Miller (1995) mencionam as características desejáveis para que um projeto de *software* tenha robustez em relação à perda de informação. Se o teste para avaliação do *software* for do tipo caixa-preta[9] então uma grande quantidade de teste é necessária para se estabelecer uma pequena probabilidade de falhas. Para reduzir esse número de testes uma de duas técnicas podem ser utilizadas: 1) selecionar teste com grande potencial para revelar defeitos; 2) projetar *software*s com grande habilidade de falhar quando defeitos existem.

Fica claro que programas com maior probabilidade de ocultar defeitos tornam a tarefa de teste mais complexa. Assim, Voas e Miller (1995) propõem métodos para identificar partes

[8] Dados, funções ou sub-rotinas que podem ser acessadas programaticamente.
[9] Um teste caixa-preta examina aspectos funcionais de um *software*, não se preocupando com a estrutura lógica interna deste *software* (PRESSMAN, 2001).

do código com maior probabilidade disto acontecer e, desta forma, concentrar os esforços de teste nestas partes.

O primeiro método proposto pelos autores é o isolamento de perdas de informação implícita através da decomposição de especificação. Essa decomposição é feita de tal forma que os componentes do programa são divididos e então é possível verificar qual deles tem maior DRR, o que permite ao testador saber qual o nível do teste que deve ser aplicado. Outro método sugerido de Voas e Miller (1995) para minimizar a perda de informação é a redução da perda de informação explicita através do aumento dos parâmetros de saída.

4.2- Extensões testáveis para testabilidade de domínio

Dentro da testabilidade de domínio, podem-se criar extensões para aumentar o grau de testabilidade de determinados componentes de um *software*. Freddman (1991) propõe duas extensões: a extensão de observabilidade e a extensão de controlabilidade.

De maneira formal, uma extensão de observabilidade é dada da seguinte forma:

Consideremos que $\mathbb{F}$ denota a expressão

```
function F(E: in T) return TF;
```

$\mathbb{F}$ tem uma extensão observável com índice de observabilidade $\mathbb{m}$ se existir um procedimento observável FO definido como

```
function FO (E: in T; E1: in T1; …; E: in Tn)
    return TF;
```

Tal que para toda entrada de $\mathbb{F}$ exista entradas para FO, da forma

F(E) = FO (E1, E2, ..., En)

Por exemplo, a função

```
function F(X: in INTEGER) return INTEGER is
begin
        return x * VARIAVEL_GLOBAL;
end F;
```

Tem uma extensão observável FO

```
function FO(X: in INTEGER, G: in INTEGER) return INTEGER is
begin
        return x * G;
end F;
```

Além da extensão de observabilidade, Freddman (1991) também formaliza a extensão de controlabilidade, que é dada da seguinte forma:

Consideremos a função 𝔽, denotada por

```
function F (E1: in T1; …; En: in Tn) return TF;
```

𝔽 tem uma extensão controlável se existe um tipo TC que é um subconjunto de TF e

uma função de expressão controlável FC definida por

```
function FC (E: in T; E1: in T1; E2: in T2; …; En: in Tn)
  return TC;
```

Tal que para todas as entradas e qualquer estado s para F, existe um estado s* para FC dado que:

F(E1,..., En) = FC (E1,..., En)

Por exemplo, a função 𝔾

```
POSITIVE = {0,1,2,...,MAX_INTEGER}
function G(X: in POSITIVE) return POSITIVE is
begin
        return x mod 3;
end G;
```

Possui uma extensão controlável GC

```
Type SMALL is POSITIVE range 0..2;
function GC(X: in POSITIVE) return SMALL is
begin
        return SMALL(x mod 3);
end GC;
```

GC é controlável pois o conjunto de valores que é retornado é o mesmo denotado pelo tipo de retorno SMALL:

$$\{GC(E) \text{ para todo } E\} = \{ 0,1,2 \} = SMALL$$

Além disso, para todos os estados 𝕤 de 𝔾, existe um 𝕤* para GC tal que:

$$G(E) = GC(E)$$

5- Conclusão

A testabilidade está ligada à dificuldade de se testar um sistema de forma adequada e à ideia de onde os defeitos podem existir sem se manifestar. A capacidade de entender estes conceitos e trabalhar com as limitações impostas determinam a qualidade da atividade de testes e conseqüentemente a confiabilidade e a qualidade de um *software*.

Freedman (1991) menciona que a métrica para testabilidade de domínio pode ser usada antes da implementação. Isso realmente é verdade, entretanto há a necessidade de já haver uma especificação muito próxima ao que será implementado. Outra desvantagem desta

medida é que ela só pode ser obtida após a criação de uma extensão do componente original. A DRR pode nos dar uma visão da testabilidade em um estágio bem inicial da especificação, pois pode ser obtida apenas através da análise da interface dos componentes de *software*. Desta forma, pode-se usar a métrica para pensar em implementações que minimizem este problema. A análise de sensibilidade consegue apresentar uma visão geral da testabilidade de um *software*. Entretanto, é uma medida que só pode ser utilizada após a fase de implementação.

A utilização em conjunto dos conceitos vistos, pode levar a melhora de todo o processo de desenvolvimento. A DRR poderia ser um excelente termômetro para avaliar os módulos onde haveria mais perda de informação e consequentemente, um menor grau de testabilidade. Neste ponto, a decomposição da especificação poderia ajudar a reduzir ou isolar os componentes problemáticos. Em um segundo momento, poderíamos fazer uso dos conceitos de testabilidade de domínio para construir extensões testáveis dos componentes com maior DRR. Finalmente, a análise de sensibilidade poderia ser usada para avaliar o *software* como um todo, dando pistas dos locais onde os testes deveriam ser realizados com um maior grau de atenção.

REFERÊNCIAS BIBLIOGRÁFICAS

PRESSMAN, Roger S.. **Software Enginerring: A Practitioner Approach**. 5. ed. New York: Mcgraw-hill, 2001.

CHOWDHARY, Vishal. **Practicing Testability in the Real World**. IEEE Computer Society, n. , p.260-268, 2009.

GUPTA, S. C.; SINHA, M. K.. **Impact of Software Testability Considerations on Software Development Life Cycle**. IEEE Computer Society, n., p.105-110, 2009.

BACH, James. **Heuristics of Software Testability**, 2003.

GAO, Jerry. **Component Testability and Component Testing Challenges.** Washington DC, 2000.

MICROSOFT. **Improving the Testability of Software.** Disponível em: <http://msdn.microsoft.com/en-us/library/cc500353.aspx>. Acesso em: 23 nov. 2009.

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, 1990, New York. **IEEE Standard Glossary of Software Engineering Terminology.** New York: The Institute Of Electrical And Electronics Engineers, 1990.

VOAS, Jeffrey; MILLER, Keith W.. **Software Testability: The New Verification**. IEEE Software**,** p. 17-28. 1995.

FREEDMAN, Roy S.. **Testability of Software Components**. IEEE Transactions On Software Engineering, n. , p.553-564, 1991

VOAS, Jeffrey M.. **PIE: A Dynamic Failure-Based Technique**. IEEE Transactions On Software Engineering, n., p.717-727, 1992.

PETTICHORD, Bret. **Design for Testability**. In: STAR WEST, Anaheim, California, 2002.