Capítulo 2

Técnicas e Ferramentas de Teste de Software

Marcio Eduardo Delamaro, Marcos Lordelo Chaim, Auri Marcelo Rizzo Vincenzi

Abstract

The activity of software testing has received much attention from industry recently. This is an essential activity for ensuring quality of the software product but, on the other hand, has a high cost associated to it. Thus, techniques and tools that increase productivity and thereby reduce its cost are essential. In this chapter we show some of these techniques and tools and also address some issues related to testing and its automation and that usually are not addressed in text books and courses of the area.

Resumo

A atividade de teste de software tem recebido muita atenção por parte da indústria recentemente. Trata-se de uma atividade indispensável para garantir a qualidade do produto de software mas que, por outro lado, tem associado a si um alto custo. Assim, técnicas e ferramentas que m aumentar a produtividade e com isso reduzir o custo são essenciais. Nesse capítulo procuramos mostrar algumas dessas técnicas e ferramentas e também abordar alguns temas relacionados ao teste e à sua automatização e que não são tratados em textos didáticos e em disciplinas de cursos da área.

2.1. Introdução

Neste capítulo vamos abordar o tema de teste de software. Consideramos um assunto de real importância no cenário de desenvolvimento de software. Prova disso é o interesse crescente que temos sentido tanto na indústria quanto na academia.

E esse interesse vem também dos nossos estudantes que cada vez mais se se engajam na condução de trabalhos relacionados ao teste de software. Ainda assim, muitos cursos de graduação tratam do assunto de forma superficial, frequentemente diluído em de uma disciplina de Engenharia de Software. Assim, o objetivo deste minicurso é apresentar uma introdução ao tema, discutindo algumas técnicas de teste.

Procuramos apresentar, dentro das limitações de espaço deste texto, técnicas de teste que são amplamente utilizadas. Por outro lado, procuramos

destacar alguns aspectos que não são facilmente percebidos pelo iniciante, e são frutos de nossa experiência. Além de discutir técnicas e ferramentas de apoio, tratamos também de alguns assuntos que estão intimamente a eles ligados e que raramente se encontram em textos didáticos sobre teste de software. Assim, falamos sobre automatização de teste, teste em sistemas Web, geração automática de dados de teste, oráculos e depuração.

2.2. Conceitos e definições

Inicialmente precisamos definir o que é teste de software. Consideramos, pelo menos no escopo deste texto, que testar é a atividade de escolher dados para executar um determinado software, executá-lo e verificar se o resultado produzido corresponde àquilo que dele se esperava. Dessa forma, a Figura 2.1 resume o que entendemos por teste de software. É importante ressaltar o objetivo que se tem ao testar um programa: descobrir defeitos. Já que é impossível provar a correção de um programa por meio de teste, o que se espera é que o desenvolvedor gaste os recursos disponíveis buscando mostrar que em alguma situação o software não funciona como esperado e, caso não consiga fazê-lo, tem-se uma indicação de que ele vai, sempre ou pelo menos na maioria dos casos, funcionar sem problemas.

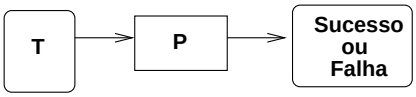


Figura 2.1. Visão sobre a atividade de teste

É interessante que tenhamos, durante a realização dessa atividade, algum tipo de controle que nos permita aferir como estamos indo e responder a uma importante pergunta: até quando devemos testar? Para isso, existem algumas técnicas que serão tratadas neste capítulo. Para começar a entendê-las vamos adicionar alguns elementos à Figura 2.1, criando então um modelo um pouco mais elaborado, mostrado na Figura 2.2, adaptada do livro de Delamaro et al. [Delamaro et al. 2007]. O elemento central desta figura é o programa que se deseja testar, representado pelo retângulo rotulado como **P**.

O primeiro elemento que aparece na Figura 2.2 é **I**, o **domínio de entrada** ou simplesmente o domínio de **P**. Esse é o conjunto de todos os dados que podem ser utilizados para executar **P**. Por exemplo, se queremos testar um programa que computa o fatorial de um número então o domínio de entrada corresponde ao conjunto dos números naturais. Ou, mais precisamente, o conjunto dos números naturais que podem ser representados em um determinado ambiente ou linguagem de programação.

Nem sempre é fácil definir qual é o domínio do programa em teste (PUT, do inglês, *program under teste*) pois qualquer fator que influencie o seu comportamento do programa caracteriza o domínio de entrada. Por exemplo, um sistema

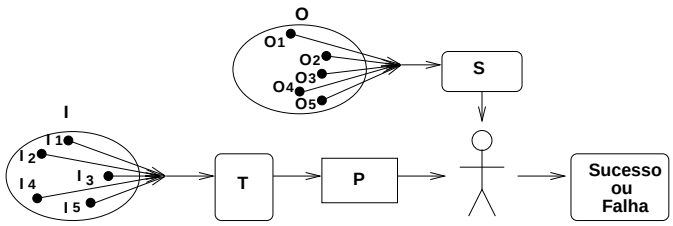


Figura 2.2. Elementos da atividade de teste

de informação que utilize um banco de dados para armazenar informações executa de acordo com os registros armazenados nesse banco de dados que, portanto, compõe o seu domínio de entrada. Assim como no caso do fatorial, o domínio de entrada é, em teoria, infinito, ficando, na prática, restrito ao volume de informação que se possa armazenar no banco de dados.

Da mesma forma, definimos o conjunto **O**, que chamamos de **domínio de saída**. Ele é composto por todos os possíveis resultados que podem ser produzidos por **P**. Por exemplo, no caso do programa que calcula o fatorial, teríamos o conjunto: {1, 2, 6, 24, 120, ...}. Se pensarmos em **P** como uma função, **O** corresponderia à imagem dessa função.

A Figura 2.2 ilustra que do domínio de entrada são selecionados alguns elementos, $\{I_1, I_2, ..., I_5\}$ que serão usados na execução de **P**. Cada elemento I_i é chamado de um **dado de teste** e **T** é o **conjunto de dados de teste**. Da mesma forma, o conjunto **S** contém os elementos $\{O_1, O_2, ..., O_5\}$ que são os **resultados esperados** da execução de **P** com os elemntos de **T**. Um par formado por um dado de teste e seu correspondente resultado esperado, $\langle I_i, O_i \rangle$ é chamado de um **caso de teste**. O conjunto de todos os casos de teste usados para executar o programa **P** é chamado de **conjunto de teste** ou **conjunto de casos de teste**.

Um ponto importante, e que às vezes gera dúvida e controvérsia, é o seguinte: não faz sentido selecionar dados de teste fora do domínio de entrada de **P**. E o motivo é bastante simples: se um determinado dado não está no domínio então não existe no domínio de saída um elemento correspondente, ou seja, não existe um comportamento definido que possa ser usado para definir se a execução está correta ou não.

A possível polêmica vem do fato de que, em geral, comportamentos "anormais" devem, também, ser exercitados durante o teste. Por exemplo, no caso do fatorial, todo testador com alguma experiência sabe que seria interessante verificar qual o comportamento do programa usando como entrada um valor negativo, esperando supostamente como resultado uma mensagem de erro. O problema aqui é a forma como definimos os domínios desse programa. Devemos incluir os valores negativos como parte do conjunto **I**, bem como a mensagem "Valor negativo não válido" como parte do conjunto **O** e não con-

fundir o domínio do programa **P** com o domínio da função matemática fatorial. Alguns casos de teste que poderíamos então estabelecer, complementando os domínios do programa fatorial seriam:

- ⟨-1, erro: Valor negativo não válido⟩
- ⟨1.4, erro: Valor inteiro requerido⟩
- ⟨“abc”, erro: Valor inteiro requerido⟩

Voltando à Figura 2.2, aquele “bonequinho” que compara os resultados produzidos pela execução de **P** com os resultados esperados é o que costumamos chamar de **oráculo**. Optamos por desenhá-lo com o aspecto humano pois muitas vezes esse papel é desempenhado pelo próprio testador, que tem a incumbência de analisar os resultados e dar o veredito sobre a correção. Isso está relacionado com o fato de que muitas vezes **S** não é dado explicitamente, mas sim é definido por meio de uma especificação descritiva informal.

Por outro lado, se **S** for definido explicitamente – por meio de uma tabela, de uma especificação formal ou de imagens, por exemplo – é possível automatizar a função de oráculo, substituindo o testador por um programa-oráculo. Ainda assim, mesmo que se tenha explicitamente uma definição dos valores esperados para cada um dos dados de teste, nem sempre é simples obter-se um oráculo automatizado, uma vez que as saídas de um programa podem assumir formatos variados como: imagens, sons, sinais, etc.

Existem alguns outros termos que causam confusão, principalmente por causa de suas traduções dos originais em inglês. Definimos **defeito** (do inglês, *fault*) como sendo um passo, processo ou definição de dados incorreto. Por exemplo, no programa de ordenação da Figura 2.3 existe um defeito na inicialização da variável `last`. Essa variável indica qual é o último elemento, contado a partir do fim do array, que ainda precisa ser ordenado. O correto seria inicializá-la com o valor 1 e não com o valor 0.

```
public static void bubbleSort(int[] x) {
    int n = x.length;
    for (int last = 0; last < n; last++) {
        for (int i = 0; i < n - last; i++) {
            if (x[i] > x[i+1]) {
                int aux = x[i];
                x[i] = x[i+1];
                x[i+1] = aux;
            }
        }
    }
}
```

Figura 2.3. Programa exemplo: defeito, erro e falha

O estado de uma execução de um programa em determinado instante é dado pelo valor da memória (ou das variáveis do programa) e do apontador

de instruções. Um **erro** (*error*) durante uma execução do programa, que se caracteriza por um estado inconsistente ou inesperado, é fruto da execução de um defeito. No exemplo da Figura 2.3 logo após a execução da atribuição `last = 0` temos um erro. Nesse ponto da execução essa variável deveria conter o valor 1, porém contém o valor 0.

Esse estado inconsistente pode ou não levar a uma **falha**, ou seja, a um comportamente que difere do comportamento esperado. Por exemplo, para um array de tamanho maior que 0 teríamos uma falha ao executar a comparação `x[i] > x[i+1]` com a indicação de uma tentativa de acesso a uma posição ilegal do array. Já para um dado de teste em que o tamanho do array é 0, mesmo com a execução do defeito e com a ocorrência do erro, não ocorre nenhuma falha.

Note-se que essas definições não são seguidas o tempo todo nem são unanimidade entre os pesquisadores e engenheiros de software, principalmente em situações informais do dia-a-dia. Em particular, utiliza-se "erro" de uma maneira bastante flexível, muitas vezes significando defeito, erro ou até falha.

E então chegamos às técnicas de teste. Como dissemos, são elas que nos auxiliam na escolha de conjuntos de teste para nosso software. As técnicas de teste abordadas neste texto são conhecidas como **baseadas em subdomínios** pois a seleção de dados de teste é feita com base na divisão do domínio de entrada em subdomínios para que, então, dados de teste sejam selecionados dos subdomínios. Na verdade, uma vez definidos os subdomínios, basta que se selecione apenas um caso de teste em cada subdomínio pois, pelo menos em teoria, qualquer dado dentro de um subdomínio faz com que o PUT se comporte da mesma maneira.

Nesta altura, fica a questão de como fazer a divisão do domínio de entrada. Existem para isso os **critérios de teste**. Eles definem **requisitos de teste** que estabelecem quais são os subdomínios. Por exemplo, vamos definir o critério de teste C_1 que tem como requisitos de teste o conjunto $\{S_1, S_2, ..., S_n\}$ em que S_i é um dos comandos do PUT. Cada S_i define um subdomínio I_i do domínio de entrada **I**, definido como $I_i = \{d \in I \mid$ a execução de **P** com d faz com que S_i seja executado $\}$. Um conjunto de teste que possui pelo menos um elemento de cada subdomínio definido por C_1 é dito satisfazer o critério C_1, ou ser C_1-adequado. Nesse caso específico, um conjunto C_1-adequado possui a característica de executar pelo menos uma vez cada um dos comandos do PUT.

Na próxima seção vamos mostrar como esses conceitos se materializam em critérios de teste que são amplamente conhecidos e utilizados na prática.

Destacamos ainda que a atividade de teste é dividida em algumas etapas que servem para diminuir a sua complexidade. Em cada uma das etapas, objetivos diversos são estabelecidos e tipos diferentes de defeitos são tratados. No **teste de unidade**, o foco é assegurar que cada uma das unidades do software (método, função, rotina, módulo, dependendo da linguagem e do paradigma de

programação) seja exercitada. O objetivo é revelar defeitos relacionados com a lógica, algoritmos e implementação de cada unidade.

No **teste de integração** tem-se como objetivo assegurar que a estrutura do software, quando suas unidades são integradas, está adequada e as interações entre essas unidades funcionam corretamemte. O tipo de defeito que se procura revelar é relacionado com a forma como interagem as unidade, por exemplo, por meio de chamadas de funções, invocação de métodos, retorno de valores e controle de exceções.

A etapa de mais alto nível é o **teste de sistema**. Nele, procura-se garantir que todos os requisitos especificados para o software estão corretamente implementados. Isso inclui requisitos funcionais e não funcionais. Assim, além da correção, características como desempenho, proteção e segurança, podem também ser alvos desta etapa de teste.

Para cada uma dessas etapas podemos ter características distintas como: momento em que são realizadas, pessoas que as realizam e técnicas utilizadas em cada uma delas. A definição dessas características de forma sincronizada é chamada de **estratégia de teste**.

2.3. Técnicas de teste

Três técnicas de teste são as mais conhecidas e mais utilizadas. O que as diferencia é o tipo de informação que se utiliza para derivar os requisitos de teste e, portanto, definir os subdomínios de teste. A técnica de teste funcional, também conhecida como teste caixa preta, pois o "interior" do programa em teste não é importante para esse propósito, utiliza apenas a especificação do software. O teste estrutural ou caixa branca, em contraposição ao nome caixa preta, utiliza a estrutura interna do programa para definir como ele deve ser testado. A ténica baseada em defeitos utiliza, além do código do programa, informação sobre defeitos típicos que podem afetar a implementação.

Um ponto importante a se destacar é que ao utilizar essas técnicas não se discute qual é melhor ou qual delas se deve escolher. O que queremos dizer é que elas devem ser utilizadas de maneira conjunta, definindo uma estratégia de teste, de modo que as melhores características de cada técnica sejam aproveitadas.

A seguir, iremos discutir duas dessas técnicas. A técnica estrutural e a baseada em defeitos. Optamos por deixar de fora a técnica funcional para que pudessemos, no espaço que temos disponível, aprofundar a discussão nas outras duas. Além disso, a técnica funcional é aquela para a qual encontramos mais material didático disponível. Por exemplo, o leitor pode consultar o capítulo de Fabbri et al [Fabbri et al. 2007] que traz uma boa compilação sobre critérios de teste funcional.

2.3.1. Teste estrutural

Ao contrário dos critérios funcionais, a utilização da técnica estrutural requer a existência e análise de uma implementação. A partir dessa implementação são identificadas "estruturas" de interesse e que devem ser exercitadas

pelos casos de teste. Essas estruturas como comandos, desvios, exceções ou caminhos, por exemplo, são os requisitos de teste. São esses requisitos que determinam os subdomínios do domínio de entrada. Por exemplo, um comando S_i do programa define um conjunto de dados de teste I_i – ou um subdomínio – que faz com que aquele comando seja executado.

Antes de apresentarmos os critérios de teste, iniciamos esta seção apresentando um instrumento essencial na definição desses critérios.

Grafo de fluxo de controle

O Grafo de Fluxo de Controle (GFC) ou Grafo de Programa é uma abstração de uma unidade – função, método ou procedimento – que procura representar as possíveis sequências de execução dos comandos dessa unidade. Um GFC $G = \langle$N, E, s$\rangle$ é um grafo direcionado formado por um conjunto de vértices, um conjunto de arestas e um nó inicial, respectivamente.

Os vértices (ou nós) do grafo correspondem a blocos indivisíveis de comandos. Isso significa que em um vértice são colocados comandos que são sempre executados juntos, ou seja: 1) toda vez que o primeiro comando de um vértice é executado, todos os demais comandos são executados em sequência; 2) não existem desvios para o meio do bloco, ou seja, para executar um comando no meio do bloco, seu antecessor deve ter sido executado imediatamente antes dele.

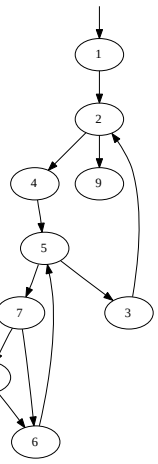
A Figura 2.4(a) mostra o exemplo do *bubbleSort* dividido em blocos, que coresponderiam aos vértices do GFC. É possível perceber que a identificação desses blocos pode não ser tão simples quanto se possa imaginar a princípio. Na verdade, cada bloco contém sequências de trechos de código, o que não corresponde a comandos completos, necessariamente. Por exemplo, o bloco número 1 contém o comando de atribuição a `n` e a inicialização da variável `last`, que sempre são executados nessa sequência. Já o trecho seguinte que é a expressão de controle do `for` é executado na sequência mas não sempre junto com os comandos anteriores. A cada iteração do `for` essa expressão é recalculada sem que os comandos do bloco 1 sejam executados, assim, se colocassemos essa expressão no bloco 1 teríamos um possível desvio para o meio do bloco.

Continuando, o comando de incremento do `for` não é executado imediatamente após a expressão de controle mas sim no final da interação. Assim, embora esteja contiguo à expressão, não deve ser colocado no mesmo bloco que ela, mas sim, em um bloco separado.

Se é possível executar-se o primeiro comando de um vértice i imediatamente após o último comando de um vértice j, então existe uma aresta (i, j) no GFC. No programa da Figura 2.4(a), podemos identificar, por exemplo, que depois do bloco 1, executa-se necessariamente o bloco 2, caracterizando a aresta (1, 2); depois do bloco 2 pode-se executar o bloco 4 ou o bloco 9, caracterizando as arestas (2, 4) e (2, 9).

```
public static void bubbleSort(int[] x) {
   int n = x.length;              /* 1 */
   for (int last = 1;             /* 1 */
        last < n;                 /* 2 */
        last++)                   /* 3 */
   {
       for (int i = 0;            /* 4 */
            i < n - last          /* 5 */
            i++)                  /* 6 */
       {
           if (x[i] > x[i+1]) {   /* 7 */
               int aux = x[i];    /* 8 */
               x[i] = x[i+1];     /* 8 */
               x[i+1] = aux;      /* 8 */
           }
       }
   }
}                                 /* 9 */
```

(a) Blocos



(b) GFC

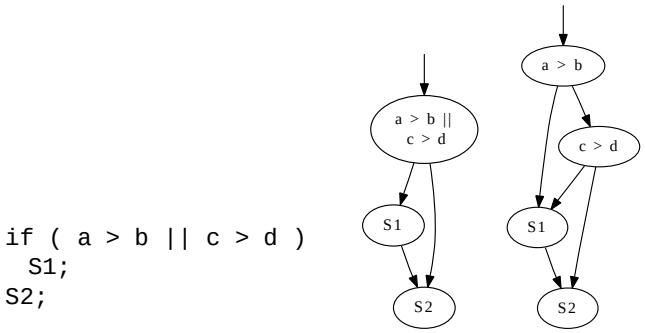
Figura 2.4. Blocos de comandos e GFC do *bubbleSort*

Todo GFC possui um único vértice *s* que é o vértice inicial. Ele corresponde ao bloco que contém o comando de entrada da unidade. Pode possuir também um ou mais nós que não têm sucessores. São os nós de saída e correspondem a pontos finais de execução da unidade como comandos de retorno. No nosso exemplo, esse comando de retorno está implícito e por isso tivemos que associar um bloco ao fecha chave do método. O vértice 9 é portanto o único final da unidade. A Figura 2.4(b) mostra o GFC para o método *bubbleSort*, com o "1" como nó de entrada.

Por esse exemplo simples podemos notar que a construção do GFC depende de uma interpretação do código. Essa interpretação depende obviamente da linguagem do programa sendo analisado mas, mais do que isso, depende também da decisão de como cada estrutura da linguagem deve ser considerada na construção do grafo. É o que chamamos de **modelo de fluxo de controle**. Podemos dizer que esse modelo descreve como uma unidade deve ser abstraída na forma de um grafo. Quando temos uma ferramenta que analisa o programa fonte e cria automaticamente seu GFC, temos uma implementação do modelo de fluxo de controle que escolhemos.

Algumas características das linguagens de programação não são tão facilmente abstraídas ou podem ser abstraídas de maneiras diferentes e, então, requerem que o testador saiba qual é o modelo de fluxo de controle que deve ser usado. Um exemplo disso é mostrado na Figura 2.5. Podemos ter duas formas diferentes de abstrair o comando `if`. Uma que considera a expressão

de controle como única e outra que considera a ordem de execução das subexpressões. Nesse segundo caso, o GFC mostra que executando apenas uma das subexpressões, em alguns casos é possível fazer o desvio diretamente para o comando dentro do `if`. No modelo de fluxo de controle usado para criar o primeiro grafo isso não é considerado explicitamente. Não podemos dizer que um dos dois modelos é melhor ou pior que o outro mas eles certamente terão consequências quando formos definir os critérios de teste baseados nesses GFCs.



(a) Código comando `if` (b) Um GFC (c) Outro GFC

Figura 2.5. Diferentes modelos de fluxo de controle

Uma vez apresentado o GFC vamos definir alguns conceitos que deverão ser úteis no resto desta seção.

- Um caminho é uma sequência de vértices que são ligados por arestas. Por exemplo, no GFC da Figura 2.4(b), (4, 5, 7, 6, 5, 3, 2) é um caminho.
- Um caminho completo é aquele que inicia no vértice de entrada e termina em um vértice de saída. Por exemplo: (1, 2, 9) ou (1, 2, 4, 5, 3, 2, 9) na Figura 2.4(b). Cada caminho completo no GFC corresponde a uma execução da unidade sendo analisada. Mais precisamente, corresponde a um conjunto de possíveis execuções da unidade, uma vez que diversos dados de entrada podem levar à execução do mesmo caminho completo. Em alguns casos esse conjunto pode ser vazio pois pode não existir dado de entrada que leve à execução do caminho.
- Um caminho (completo ou não) é dito não executável quando não existe nenhum dado no domínio de entrada do programa que leve à sua execução.

Nas próximas seções utilizaremos o GFC e estes conceitos para definir alguns critérios de teste estrutural.

Critérios baseados no fluxo de controle

Como o nome já diz, os critérios baseados no fluxo de controle, ou simplesmente critérios de fluxo de controle, utilizam apenas as informações contidas no GFC para derivar seus requisitos de teste. Vamos, então, definir dois desses critérios.

O critério **todos-nós** estabelece que um conjunto de teste adequado deve fazer com que cada um dos vértices seja executado pelo menos uma vez. Dito de outra forma, se tomarmos um conjunto de teste $T = \{t_1, t_2, ..., t_n\}$ e os respectivos caminhos completos por ele definido $\Pi = \{\pi_1, \pi_2, ..., \pi_n\}$, esse critério exige que cada um dos vértices apareça pelo menos uma vez em algum caminho de Π.

O critério **todas-arestas** (ou todos-arcos) é similar mas os requisitos de teste são as arestas do GFC. O critério requer que cada uma das arestas seja executada pelo menos uma vez, ou seja, que apareça pelo menos uma vez no conjunto de caminhos executados pelos casos de teste.

Olhando o GFC do *bubbleSort*, vemos que para satisfazer o critério todos-nós podemos utilizar o caso de teste [3, 2, 1]. Com esse único caso de teste garantimos que todos os vértices da unidade – e portanto todos os seus comandos – são executados pelo menos uma vez. Não conseguimos entretando cobrir o critério todas-arestas. Se observamos, veremos que em nenhuma das vezes em que o comando `if` é executado, obtem-se o resultado falso para a condição de controle e portanto, nenhuma vez a aresta (7, 6) é exercitada. Para tanto podemos adicionar o caso de teste [1, 2, 3]. É fácil perceber que o critério todas-arestas inclui (do inglês *subsumes*) o critério todos-nós, ou seja, sempre que o conjunto de teste **T** for todas-arestas-adequado, será também todos-nós-adequado.

Um problema relacionado ao teste estrutural é a existência de requisitos não executáveis, ou seja, elementos que são requeridos por algum critério mas que são impossíveis de serem satisfeitos. Sabe-se que, no caso geral, é um problema indecidível verificar se um requisito de teste é executável ou não, ficando por conta da análise do testador o julgamento sobre a possibilidade ou não de se satisfazer tais requisitos.

No caso de nós e arestas, em teoria, não deveriam existir requisitos não executáveis pois, se tais requisitos não podem ser executados, poderiam ser removidos do programa em teste. Existem, porém, casos em que isso não é verdade. Por exemplo, na Figura 2.6 vemos o caso em que a chamada a um método faz com que o programa termine, não permitindo que a aresta (2, 3) seja executada.

Critérios baseados no fluxo de dados

Os critérios baseados no fluxo de dados acrescentam à abstração do GFC informação sobre como as variáveis do programa são utilizadas. A idéia é que

```
int div(int a, int b) {
    if ( b == 0 )
        System.exit(-1);
    return a / b;
}
```

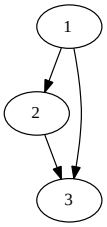


Figura 2.6. exemplo de aresta não executável

cada vez que uma variável recebe um valor ele deve ser verificado em algum ponto do programa. Assim, classificam-se as possíveis formas de utilizar um variável em:

- definição: é toda referência feita a uma variável que faz com que o valor dessa variável possa ser alterado. Por exemplo: `a = 10` ou `read(a)` ambos fazem a definição da variável `a`;
- uso: são todas as demais referências a variáveis, ou seja, quando o valor armazenado na variável é utilizado mas não modificado. Um uso pode ainda ser subclassificado em:
  - uso predicativo ou p-uso: quando o valor da variável é usado para definir o fluxo de controle do programa. Por exemplo `if (a > 10)` faz um uso predicativo de `a`;
  - uso computacional ou c-uso: todos os demais usos que não são p-usos. Por exemplo, `a = b * c` possui um uso computacional de `b` e um de `c`.

Assim como na construção do GFC, a definição de como se dão as definições e os usos de um programa requer a adoção de um **modelo de fluxo de dados**. Embora possa não parecer, existem diversos problemas que precisam ser abordados ao se definir tal modelo. Alguns exemplos são:

- definição e uso de arrays. Quando se atribui um valor a um elemento de um array, por exemplo `v[i]` é, em geral, impossível saber qual é o elemento definido. Assim, quando um uso de `v` ocorre é impossível saber se ele corresponde ou não à definição feita anteriormente;
- chamada de subrotinas com passagem de parâmetro por referência. Tais chamadas podem fazer com que o valor da variável usada como argumento seja modificado. Então é preciso identificar se e quando chamadas de subrotinas correspondem a definições de variáveis;
- utilização de ponteiros ou referências a objetos. Neste caso, a atribuição de um valor à variável `x->idade`, por exemplo, e seu subsequente uso,

requer uma análise complexa. No momento do uso, a variável x pode referenciar uma posição de memória diferente daquela que referenciava na atribuição, não caracterizando portanto um par definição/uso da mesma variável.

Uma vez estabelecido um modelo de fluxo de dados, precisamos ainda estender o GFC adicionando a ele informação sobre definições e usos de variáveis. A Figura 2.7 mostra o **Grafo Def-Uso** do método *bubbleSort*. Ele possui informação sobre todas as definições e usos de cada uma das variáveis. Ele auxilia a definição dos critérios por meio da identificação de quais são as associações definição/uso existentes na unidade em teste.

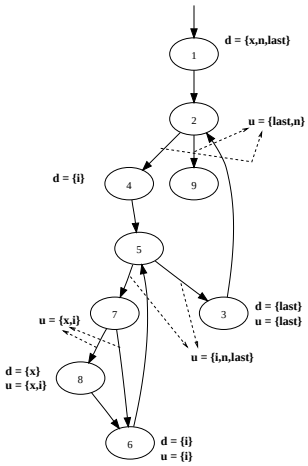


Figura 2.7. Grafo def-uso do método *bubbleSort*

Algumas observções importantes sobre o grafo da Figura 2.7:

- os usos predicativos são sempre associados às arestas que as respectivas variáveis influenciam. Na figura usamos as setas tracejadas apenas para identificar esse fato de maneira mais clara;
- para as variáveis que não são definidas no método, como variáveis de classe ou parâmetros formais, supõe-se uma definição no nó de entrada do grafo;
- usos que são locais, ou seja, que referem-se a definições feitas no mesmo nó não são importantes e não precisam aparecer no grafo def-uso. É o caso do uso da variável `aux` no vértice 8; Os usos que aparecem no grafo são chamados usos globais;

- definições locais, ou seja, aquelas que não possuem pelo menos um uso global também são irrelevantes. É o caso da definição de `aux` no vértice 8;
- para que uma associação definição/uso de uma variável `x` exista entre dois pontos do programa, é preciso que entre a definição e o uso exista pelo menos um caminho livre de definição em relação a `x`. Um **caminho livre de definição** em relação a uma variável `x` do vértice `i` ao vértice `j` ou à aresta (j, k) é uma caminho que não possui nenhuma definição de `x`, exceto, possivelvente, no vértice inicial, `i`.

Assim, as associações definição/uso que se caracterizam no *bubbleSort* são as mostradas na Tabela 2.3.1.

Tabela 2.1. Associações definição/uso do *bubbleSort*

Variável	Definição	Uso
i	4	6
i	4	8
i*	4	(5,3)
i	4	(5,7)
i	4	(7,6)
i	4	(7,8)
i	6	8
i	6	(5,3)
i	6	(5,7)
i	6	(7,6)
i	6	(7,8)
last	3	(5,3)
last	3	(5,7)
last	3	(2,4)
last	3	(2,9)

Variável	Definição	Uso
last	1	3
last	1	(5,3)
last	1	(5,7)
last	1	(2,4)
last*	1	(2,9)
n	1	(5,7)
n	1	(5,3)
n	1	(2,4)
n	1	(2,9)
x	0	8
x	0	(7,6)
x	0	(7,8)
x	8	8
x*	8	(7,6)
x	8	(7,8)

Os critérios de fluxo de dados mais conhecidos foram definidos por Rapps e Weyuker [Rapps e Weyuker 1982]. Eles formam uma família de critérios, dos quais mencionaremos dois. O critério **todas-definições** requer que para cada definição de variável, um uso seja exercitado. O critério **todos-usos** requer que para cada definição de variável, todos os usos existentes sejam exercitados. Olhando na tabela de requisitos (Tabela 2.3.1), o critério todas-definições requer que uma das associações de cada divisão da tabela seja coberta pelo conjunto de teste. O critério todos-usos requer que todas as associações relacionadas na tabela sejam cobertas. Claramente este é um outro exemplo de um critério de teste que inclui o outro.

Os requisitos marcados com * na Tabela 2.3.1 são aqueles que não foram cobertos pelo conjunto de casos de teste que haviamos criado com os critérios de fluxo de controle, mais especificamente com o critério todas-arestas. O requisito ⟨last, 1, (2,9)⟩ requer que a definição da variável `last` no vértice 1 seja exercitada na aresta (2,9) sem ser redefinida. Para tanto necessitamos de um caso de teste cuja execução não entre no laço do `for`, caso contrário a

variável `last` seria redefinida. Portanto um caso de teste cuja entrada tenha tamanho menor ou igual a 1 deve cobrir tal requisito.

O requisito ⟨i, 4, (5,3)⟩ é um caso semelhante. Precisaríamos fazer com que a execução, em alguma iteração, não entrasse no `for` mais interno. Porém, como a variável de controle `i` é inicializada com 0 e o valor limite `n - last` é sempre maior que 0, então é impossível exercitar esse requisito, ou seja, trata-se de um requisito não executável.

Para cobrir o último requisito que falta devemos escolher um array no qual façamos uma troca dos elementos e não a façamos na comparação seguinte para que o valor de x seja alterado na primeira vez (definição no nó 8) mas que a execução na próxima comparação passe pela aresta (7, 6) (p-uso na aresta). Então, o array [2, 1, 3] iria cobrir esse requisito.

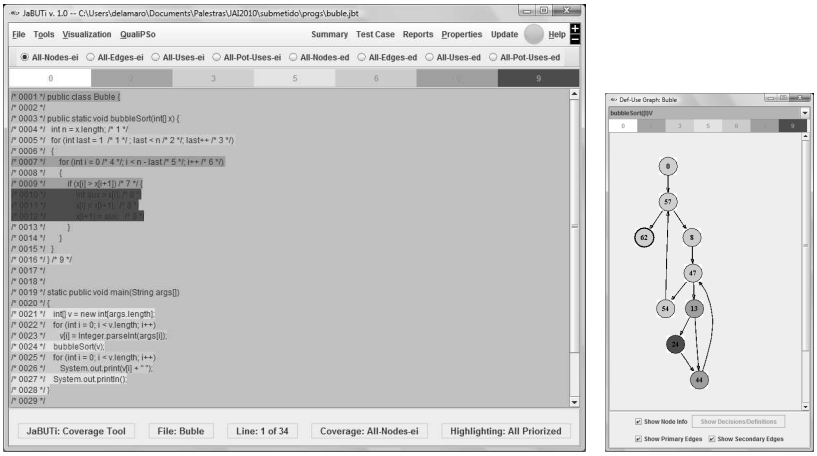
A ferramenta JaBUTi

A JaBUTi (Java Bytecode Understanding and Testing) é uma ferramenta de apoio à aplicação de critérios estruturais baseados no fluxo de controle (todos-nós e todas-arestas) e no fluxo de dados (todos-usos) para programas Java. Mais precisamente, ela apoia o teste de código objeto Java – *bytecode* da Máquina Virtual Java – pois toda a análise estática (geração do grafo def-uso e derivação dos requisitos de teste) e dinâmica (análise de cobertura dos casos de teste) é feita sobre o *bytecode* e não sobre o código fonte. Na verdade, o código fonte não precisa sequer estar disponível para o testador, embora a ferramenta também faça o mapeamento das informações coletadas durante o teste para o nível de código fonte, quando ele estiver disponível.

Assim, a JaBUTi não é uma ferramenta de teste apenas para Java. Se o desenvolvedor possuir um compilador que gere *bytecode* Java para outras linguagens como C, Pascal, Phython etc, pode usar a JaBUTi para testar seus programas também.

Ao criar uma sessão de teste, o testador informa qual a classe “principal” de seu programa e a ferramenta trata de procurar e analisar as demais classes necessárias. O testador pode selecionar quais são as classes que deseja testar, ou seja, aquelas cujas coberturas serão medidas. O testador tem à sua disposição diversas formas de visualizar o programa e as informações estáticas dele colhidas. Por exemplo, pode ver: código fonte, *bytecode*, grafo def-uso e lista de requisitos. As Figuras 2.8(a) e 2.8(b) mostram o código fonte e o grafo def-uso do *bubbleSort*.

Os elementos, tanto no programa fonte quanto no grafo, estão coloridos indicando o “peso” de cada elemento, de acordo com a legenda mostrada na parte superior da janela. Por exemplo, o nó 24 está colorido de vermelho, que na legenda corresponde ao número 9. Isso significa que ao cobrir o requisito de teste – no caso o próprio nó – outros 9 requisitos seriam coberto também. O número 24 representa o número no *bytecode* da instrução que inicia o bloco.



(a) Visualização do programa fonte

(b) Visualização do grafo def-uso

Figura 2.8. Modos de visualização da JaBUTi

Também na parte superior, o testador pode selecionar os diversos critérios implementados na ferramenta. Pode-se notar que cada um dos critérios que comentamos anteriormente está dividido em dois na JaBUTi. Por exemplo, temos todos-nós “ei” e “ed”. O primeiro é relativo ao código normal do método e o segundo relativo ao código que faz tratamento de exceções. Por isso os nomes significam independente de exceção e dependente de exceção, respectivamente.

Para executar seus casos de teste o usuário pode instrumentar as suas classes usando a operação “*File/Save instrumented classes*”. Então, pode executar normalmente sua aplicação que, enquanto executa, gera informações de rastro, indicando quais pontos do código foram executados. O arquivo de rastro é analisado pela ferramenta que mostra as informações de cobertura. Casos de teste podem ser definidos também utilizando conjuntos de teste no formato da JUnit, ferramenta que automatiza o teste de unidades Java.

Diversos tipos de relatórios podem ser vizualizados ou gerados em arquivos HTML. Além disso, a JaBUTi possui a funcionalidade de calcular métricas estáticas de orientação a objetos, populares na literatura. Possui também uma ferramenta de *slicing* que auxilia a localização de defeito. O testador, ao perceber que o software falha com um caso de teste, pode marcá-lo e usar mais dois outros casos de teste que não falharam. A ferramenta, fazendo a diferença entre os caminhos executados por esses casos de teste indica no código, também com cores, os possíveis pontos onde o defeito se localiza.

A JaBUTi está disponível como software livre na incubadora virtual da FAPESP (`http://incubadora.fapesp.br/projects/jabuti/`).

Teste estrutural no GCC

Existem diversas ferramentas de apoio ao teste estrutural. Uma que vem ganhando popularidade é o GCC (Gnu Compiler Collection), que na verdade é um compilador que incorporou facilidades para instrumentação de código e coleta de rastros de execução. O GCC possui diversos *front-ends* para linguagens como C, C++, Objective-C, FORTRAN, Java, e Ada e é um padrão *de facto* na indústria de software.

Usando alguns argumentos como `--coverage` ao compilar os programas, o GCC gera código instrumentado que, ao executar, gera arquivos com informações de cobertura de fluxo de controle (comandos e desvios). Para processar os arquivos gerados pelo código instrumentado, o GCC é acompanhado pelo programa *gcov*, que gera um arquivo texto legível pelo testador. A Figura 2.9 mostra um trecho do arquivo gerado pela *gcov* para o *bubbleSort*, depois de executado com o caso de teste [1, 2, 3]. Cada linha mostra o número de vezes que o comando foi executado e o número do comando. Quando não há número de execuções isso indica que aquela linha não gera código executável. A cadeia ##### indica que a linha não foi executada.

```
        -:    0:Source:buble.c
        -:    0:Graph:buble.gcno
        -:    0:Data:buble.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        1:    3:void bubbleSort(int x[], int n) {
        3:    4:    for (int last = 1; last < n; last++) {
        5:    5:        for (int i = 0; i < n - last; i++) {
        3:    6:            if (x[i] > x[i+1]) {
    #####:    7:                int aux = x[i];
    #####:    8:                x[i] = x[i + 1];
    #####:    9:                x[i+1] = aux;
        -:   10:            }
        -:   11:        }
        -:   12:    }
        1:   13:}
```

Figura 2.9. Informação de cobertura no *gcov*

Como o arquivo gerado pela *gcov* não é muito ilustrativo, outros programa têm surgido para processar a saída do programa instrumentado. Por exemplo o *lcov* (`http://ltp.sourceforge.net/coverage/lcov.php`), que gera arquivos HTML com informações bem mais completas sobre a cobertura obtida.

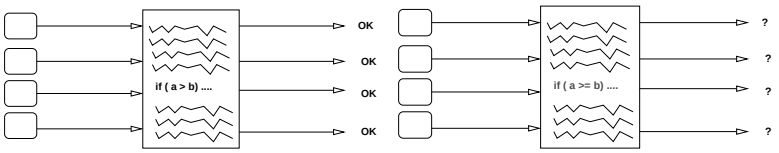
2.3.2. Teste baseado em defeito

Nesta seção falaremos especificamente sobre um critério conhecido como teste de mutação. Ele foi proposto no final da década de 1970 por DeMillo et al [DeMillo et al. 1978]. Nestas três décadas, muita pesquisa foi e continua sendo dedicada a esse critério, pricipalmente por mostrar-se de grande eficácia na criação de conjuntos de teste com grande probabilidade de revelar defeitos. Além de encontrarmos trabalhos sobre teste de mutação nas principais publicações e eventos da área, existe um congresso dedicado especificamente a esse tema, que se realiza a cada dois anos. O primeiro aconteceu no ano de 2000 e em 2010 teremos um nova edição (`http://www.st.cs.uni-saarland.de/mutation2010/`).

Conceitos básicos

Para explicar, de forma intuitiva, como funciona o teste de mutação, vamos pensar na seguinte situação: suponha que você, leitor, seja um programador e que seu chefe peça para você implementar uma determina função. Além disso, ele pede para você testar essa função e apresentar para ele, junto com o código desenvolvido, o conjunto de teste que foi utilizado. Você entrega os artefatos solicitados e seu chefe executa seu programa com os casos de teste fornecidos. Obviamente tudo funciona como deveria – afinal você já havia realizado essa tarefa – e tanto você quanto seu chefe ficam orgulhosos e felizes.

Imagine, ainda, que seu chefe tem uma idéia, meio estranha, mas aceitável (afinal ele é o chefe). Ele escolhe um dos comandos do seu programa e modifica-o. Por exemplo, um comando `if (a>b)` é re-escrito como `if (a>=b)`. Depois, executa essa versão modificada com os mesmos casos de teste que funcionaram bem com o seu programa e observa o resultado dessas novas execuções, como ilustrado na Figura 2.10. O que se espera que aconteça?



(a) Programa original (b) Programa modificado

Figura 2.10. Casos de teste no programa modificado

Intuitivamente, esperamos que o programa falhe com um ou alguns dos casos de teste. Afinal, se o programa original estava correto, o programa modificado não deveria estar. Se o conjunto de teste utilizado não consegue mostrar isso, ou seja, se o programa modificado continua produzindo resultados corretos é provavel que nenhum dos casos de teste seja capaz de exercitar de

maneira adequada aquele comando que foi modificado. Portanto, deveríamos melhorar esse conjunto de teste para mostrar que esse segundo programa "está errado", como ilustrado na Figura 2.11

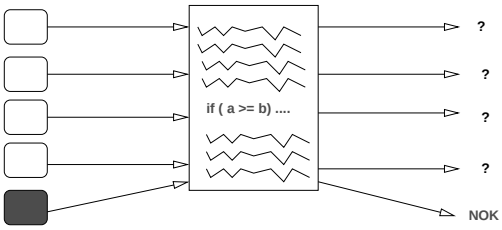


Figura 2.11. Novo caso de teste para melhorar o conjunto inicial

É exatamente essa a idéia por trás do teste de mutação. Não com uma versão modificada, mas com muitas delas. Cada uma das versões alternativas é um mutante. O mutante que apresenta resultado incorreto (diferente do resultado do programa original) é dito "morto". E o objetivo deste critério é encontrar um conjunto de teste que mate todos os mutantes. Quando isso acontece, temos um bom conjunto de teste, ou melhor, um conjunto adequado, de acordo com esse critério.

Para medir a adequação de um conjunto de teste em relação a esse critério define-se o **escore de mutação**. Ele é calculado como:

$$MS = \frac{\text{número de mutantes mortos}}{\text{número de mutantes}}$$

Ou seja, é a razão entre o número de requisitos satisfeitos pelo número total de requisitos de teste (os mutantes).

Na verdade, precisamos adicionar a essa expressão uma dificuldade inerente ao teste de mutação. São os **mutantes equivalentes**, ou seja, aqueles que sempre se comportam como o programa original para todos os elementos do domínio de entrada e, portanto, não podem ser mortos. No caso geral, não é possível identificar automaticamente os mutantes equivalentes, ficando por conta do testador fazê-lo. E o escore de mutação deve ser calculado como:

$$MS = \frac{\text{número de mutantes mortos}}{\text{número de mutantes não equivalentes}}$$

Esse é um critério baseado em subdomínios pois cada mutante estabelece um conjunto de casos de teste que o mata e o testador deve selecionar um deles como parte do seu conjunto de teste.

Operadores de mutação

A criação dos mutantes não é feita de qualquer forma. Ela é baseada num conjunto de **operadores de mutação**, que depende muito da linguagem de programação do PUT. Por exemplo, para a linguagem C foram definidos 75 operadores de mutação, divididos em 4 classes: comandos, operadores, variáveis e constantes. Usando uma versão C do *bubbleSort*, da Figura 2.3, mostramos em seguida um exemplo de operador para cada uma dessas classes.

Mutação de comando. Os operadores desta classe atuam sobre a estrutura dos comandos do programa. O operador "SSDL – remoção de um comando" remove um dos comandos do programa original para criar um mutante. Assim, o número de mutantes criados é igual ao número de comandos do programa. Um dos mutantes gerados para o *bubbleSort* seria o da Figura 2.12

```
void bubbleSort(int x[], int n) {
    for (int last = 1; last < n; last++) {
        for (int i = 0; i < n - last; i++) {
            if (x[i] > x[i+1]) {
                int aux = x[i];
                ; // comando removido
                x[i+1] = aux;
            }
        }
    }
}
```

Figura 2.12. Mutante gerado pelo operador SSDL

Para matar esse mutante o testador deveria criar um dado de teste em que pelo menos uma troca de valores fosse realizada, por exemplo: [3, 2, 1]. Já um dado de teste como [1, 2, 3] não seria capaz de matar o mutante;

Mutação de operadores. Nesta classe, os operadores da linguagem C são o alvo das modificações. Por exemplo, o operador "ORRN – troca de operadores relacionais" faz com que a ocorrência de um operador relacional seja trocada pelos outros operadores relacionais. Note que a aplicação desse operador em um único ponto de mutação não gera apenas um, mas cinco mutantes (considerando que temos seis operadores relacionais em C). Para o *bubbleSort* teriamos como um dos mutantes aquele mostrado na Figura 2.13. Nele, a troca é feita toda vez que os elementos comparados forem diferentes. Portanto, um dado de teste que mataria o mutante é [1, 2, 3]. Já [3, 2, 1] não conseguiria fazê-lo.

Mutação de variáveis. Nesta classe as modificações referen-se ao uso das variáveis do programa. Em geral, cada ocorrência de uma variável é substituída por outra, de tipo compatível. O operador "VSSR – troca de variável escalar" troca cada ocorrência de uma variável escalar (não estruturada)

```
void bubbleSort(int x[], int n) {
    for (int last = 1; last < n; last++) {
        for (int i = 0; i < n - last; i++) {
            if (x[i] != x[i+1]) { /* troca de operador */
                int aux = x[i];
                x[i] = x[i+1];
                x[i+1] = aux;
            }
        }
    }
}
```

Figura 2.13. Mutante gerado pelo operador ORRN

por todas as outras variáveis de tipos compatíveis que aparecem na mesma função. Por exemplo, temos o mutante da Figura 2.14.

```
void bubbleSort(int x[], int n) {
    for (int last = 1; last < n; last++) {
        for (int i = 0; i < n - last; i++) {
            if (x[i] > x[i+1]) {
                int aux = x[last]; /* troca de variável */
                x[i] = x[i+1];
                x[i+1] = aux;
            }
        }
    }
}
```

Figura 2.14. Mutante gerado pelo operador VSSR

Para matar esse mutante podemos selecionar a entrada [3, 2, 1]. Uma entrada que não mata esse mutante seria [1, 2, 3].

Mutação de constantes. De maneira semelhante à anterior, esta classe de operadores manipula as constantes que aparecem no programa. O operador "CCCR – troca de constantes" troca a ocorrência de uma constante por todas as constantes de tipos compatíveis que aparecem no programa. Por exemplo, temos o mutante da Figura 2.15.

Da mesma forma, os casos de teste mencionados para o mutante anterior fazem com que o mutante falhe e funcione como o programa original.

Existem, ainda, alguns operadores que geram os chamados **mutantes instrumentados** que, ao contrário dos demais, não representam defeitos típicos da linguagem alvo. Esses operadores têm como objetivo garantir que o conjunto de casos de teste adequados ao critério tenha alguma característica de interesse. Por exemplo, o operador "STRP – armadilha na execução de comandos" substitui cada um dos comandos do programa original por uma chamada de função "TRAP( )". A execução dessa função força a morte do mutante. Com isso, para matar todos os mutantes gerados por esse operador, o testador deve fornecer um conjunto de teste que execute, pelo menos uma

```
void bubbleSort(int x[], int n) {
    for (int last = 1; last < n; last++) {
        for (int i = 0; i < n - last; i++) {
            if (x[i] > x[i+1]) {
                int aux = x[i];
                x[i] = x[i+0]; /* troca de constante */
                x[i+1] = aux;
            }
        }
    }
}
```

Figura 2.15. Mutante gerado pelo operador CCCR

vez, cada um dos comandos do programa, garantindo a cobertura do critério estrutural todos-nós.

Comentários sobre equivalência

A tarefa que mais demanda esforços no teste de mutação é a identificação de mutantes equivalentes. É sabido que decidir se dois programas computam a mesma função é um problema indecidível. Mesmo quando a diferença entre os dois programas é pequena, como no caso do teste de mutação, isso não pode ser feito, se considerarmos o caso geral.

É claro que existem casos em que seria possível identificar automaticamente os mutantes equivalentes. Existem alguns trabalhos que tratam desse assunto como, por exemplo, o trabalho de Offutt e Craft [Offutt e Craft 1994]. A questão é se vale a pena fazê-lo. Em geral, os mutantes que podem ser identificados automaticamente como equivalentes são aqueles que podem, também, ser facilmente identificados pelo testador. Aqueles que são difíceis de analisar continuariam sem serem identificados e continuariam a demandar esforços do testador.

Algumas situações fazem com que a identificação de equivalência seja trivial. Por exemplo, ao aplicar o operador que troca um operador de atribuição simples por um operador de atribuição com operação aritmética, frequentemente temos a seguinte situação: `a = 0` substituído por a `*= 0`, o que claramente cria um mutante equivalente, qualquer que seja o contexto em que ocorra.

Existem diversos outros padrões que, embora um pouco mais elaborados, ocorrem muito frequentemente e que são facilmente identificados. Por exemplo:

Comando original	Mutante equivalente
`for (i = 0; i < n; i++)`	`for (i = 0; i != n; i++)`
`if ( (a > b) || (c < d) )`	`if ( (a > b) + (c < d) )`
`if ( (a > b) && (c < d) )`	`if ( (a > b) * (c < d) )`
`if ( (a < 0) || (a > 10) )`	`if ( (a < 0) - (a > 10) )`

Existem porém os mutantes equivalentes que são bastantes difíceis de serem identificados. Em geral, sua análise depende do contexto em que a função alterada é executada, como, por exemplo, o domínio dos seus parâmetros de entrada, o estado do programa no momento da sua chamada, entre outros. Existe ainda alguns mutantes que estão “no limbo”, e o testador não sabe exatamente o que fazer com eles. Por exemplo, para o *bubbleSort*, vamos considerar o mutante da Figura 2.16.

```
void bubbleSort(int x[], int n) {
    for (int last = 0; last < n; last++) {
        for (int i = 0; i < n - last; i++) {
            if (x[i] > x[i+1]) {
                int aux = x[i];
                x[i] = x[i+1];
                x[i+1] = aux;
            }
        }
    }
}
```

Figura 2.16. Mutante “quase-equivalente”.

A variável `last` foi inicializada com o valor `0` em vez do valor `1`. Como vimos, quando comentamos a Figura 2.3, esse programa em Java apresentaria uma falha ao realizar-se a comparação do comando `if`. Em C, o que acontece é que é feita, para cada iteração do `for` mais externo, uma comparação a mais. Na primeira iteração isso pode ou não levar a uma falha de acesso à memória. Nas demais iterações essa comparação a mais não faz diferença pois na iteração anterior colocou-se na posição `i+1` um valor que é maior do que todos os valores ainda não ordenados, que estão nas posições inferiores do array. Assim, a única forma de matar esse mutante seria torcer para que ocorra um erro de acesso e o mutante seja morto. Uma vez que não existe uma forma do testador escolher um caso de teste que faça com que isso aconteça, aconselha-se o testador a marcar esse mutante como equivalente, embora ele não o seja de verdade.

Para finalizar esta subseção, vale a pena, ainda, mencionar o caso em que a mutação faz com que o mutante entra em um laço infinito, por exemplo ao se remover um comando que incrementa um contador dentro de um comando de repetição. Esse mutante, claramente, não pode ser considerado equivalente. Ao contrário, todo mutante com esse comportamento deveria ser considerado morto. O problema aqui é como comparar o resultado do programa original e de um programa cuja execução não termina. Uma das soluções, adotada na ferramenta PROTEUM, que será apresentada a seguir é a de monitorar o tempo de execução do programa original e dos mutantes, com cada caso de teste. Se o mutante extrapolar demasiadamente o tempo de execução do programa original, é considerado morto.

Ferramenta PROTEUM

A ferramenta PROTEUM foi desenvolvida no ICMC/USP a partir de 1991 e apresentada pela primeira vez no Simpósio Brasileiro de Engenharia de Software de 1993 [Delamaro et al. 1993]. É uma das poucas, senão a única ferramenta para o teste de mutação para a linguagem C. Por isso, e por oferecer algumas características que permitem a realização de experimentos de maneira simples, tem sido muito utilizada, em todo o mundo, principalmente no meio acadêmico.

É uma ferramenta fácil de se usar, com uma interface bastante simples. Na verdade, a interface que vamos rapidamente descrever aqui é apenas um "*shell*" de acesso à ferramenta real. Esta é composta por uma série de programas que foram feitos para serem invocados por meio de um terminal de comandos. Isso permite que sejam criados *scripts* de teste que reproduzem as mesmas operações que podem ser realizadas pela interface gráfica. A Figura 2.17 mostra a janela principal dessa interface.

Figura 2.17. Menu principal da PROTEUM

As operações incluídas nessa interface são:

- *Program test*: operações para criar uma nova sessão de teste e para recarregar uma sessão que tenha sido criada anteriormente e interrompida;
- *Test Case*: operações para manipular o conjunto de teste a ser avaliado. Permite, entre outros, adicionar um novo caso de teste, excluir ou desabilitar casos de teste existentes, habilitar casos de teste desabilitados, visualizar cada um dos casos de teste e importar conjuntos de teste prontos, de arquivos texto ou de outras sessões da própria PROTEUM;
- *Mutants*: operações para manipular o conjunto de mutantes que serão usados no teste. Permite criar, habilitar/desabilitar, executar e visualizar mutantes;
- *Reports*: operação para criação de relatório sobre a sessão de teste;
- *Properties*: permite que o testador defina alguns parâmetros como, por exemplo, o diretório corrente das suas sessões de teste;
- *Status*: exibe uma janela com o sumário da sessão de teste naquele momento. Mostra informações como o número de mutantes, de casos de teste e o escore de mutação.

Vamos mostrar algumas dessas operações numa sequência típica de uma sessão de teste. Usaremos para isso o mesmo programa *bubbleSort*, ao qual adicionamos uma função "`main`", que pega os valores passados na linha de comando, coloca-os em um array e chama a função a ser testada. Ou seja, a função `main` funciona como um *driver* para o teste do *bubbleSort*.

Inicialmente, cria-se a sessão de teste selecionando-se a opção "*Program Test/New*", o que nos leva à janela apresentada na Figura 2.18. Nela são fornecidos os argumentos para a criação da sessão de teste, ou seja, o diretório onde devem ser criados os arquivos correspondentes, o nome que se deseja atribuir à sessão, o nome do arquivo fonte do programa, do arquivo executável e o comando para criação do arquivo executável a partir do fonte. Esse último é necessário para que a ferramenta possa criar os mutantes executáveis a partir dos seus fontes, de maneira flexível e que pode ser definida pelo testador. Por fim, o tipo da sessão define como os mutantes serão executados: *test* significa que um mutante é executado até que seja morto por um caso de teste; *research* significa que os mutantes são executados com todos os casos de teste.

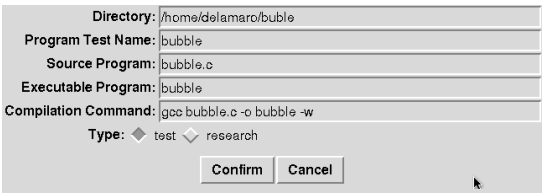


Figura 2.18. Criação de uma sessão de teste

Em seguida, podemos adicionar alguns casos de teste que constituirão o conjunto inicial de casos de teste. Para isso, basta selecionar a opção "*Test Case/Add*" e a ferramenta solicita os argumentos para execução do programa em teste e, em seguida, coloca-o em execução. Caso o programa seja interativo, ou seja, necessite de dados fornecidos em tempo de execução, o testador interage normalmente com o programa e essas interações são registradas. Tanto as entradas fornecidas quanto os resultados produzidos ficam armazenados para que os mutantes possam ser executados com as mesmas estradas e para que os seus resultados possam ser comparados com os resultados originais. Note-se, porém, que apenas as entradas e saídas padrões são tratadas (teclado e tela do terminal). Repetindo algumas vezes a operação de inserção de casos de teste Para o nosso exemplo vamos utilizar como entradas: [1, 2, 3], [3, 2, 1], [2, 1, 3] e [2, 3, 1].

Em seguida, podemos gerar os mutantes com os quais queremos trabalhar. Com a operação "*Mutants/Generate Unit*" podemos selecionar quais funções devem ser testadas e, em seguida, quais operadores de mutação iremos utilizar. A Figura 2.19 mostra a janela na qual podemos escolher, para a classe de mutação de operadores, quais operadores de mutação desejamos utilizar. Para

cada operador podem ser definidos dois parâmetros de geração. O primeiro é a porcentagem do número total de mutantes que devem ser realmente gerados, utilizando uma seleção aleatória. O segundo é o número máximo de mutantes a serem gerados para cada ponto de mutação, também com seleção aleatória. No exemplo da figura, estamos gerando mutantes apenas para o operador ORRN, sem nenhuma restrição quanto ao número de mutantes a serem gerados, ou seja, parâmetros definidos como 100% e 0 (que indica que não há limite por ponto de mutação). A terceira coluna na Figura 2.19 mostra quantos mutantes foram anteriormente gerados para cada operador de mutação.

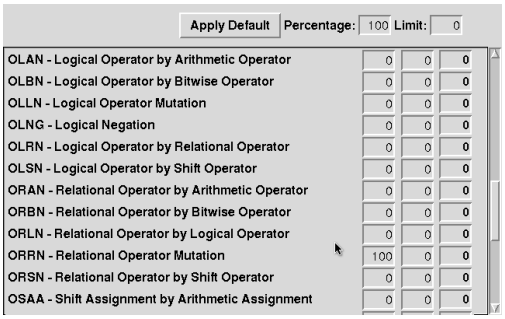


Figura 2.19. Escolha dos operadores de mutação

Uma vez gerados os mutantes, devemos executá-los para obter o nível de adequação dos casos de teste que introduzimos anteriormente. Isso é feito pela operação "*Mutants/Execute*". A janela de status, mostrada na Figura 2.20 resume o resultado alcançado. Dos 15 mutantes gerados, 4 não foram mortos pelo conjunto de teste.

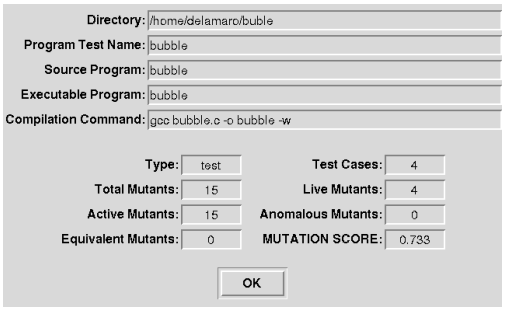


Figura 2.20. Resultado da execução inicial dos mutantes

Para analisar esse resultado é preciso verificar cada um dos mutantes vivos e identificar a razão pela qual eles não foram distinguidos. Isso é feito com a opção "*Mutants/View*" que leva à janela mostrada na Figura 2.21. Nela são mostrados o código original e o código dos mutantes. O testador pode selecionar o tipo de mutantes que deseja visualizar. Por exemplo, pode deixar marcada apenas o tipo "*Alive*", para ver apenas os mutantes vivos. E pode ainda marcar um mutantes como equivalente.

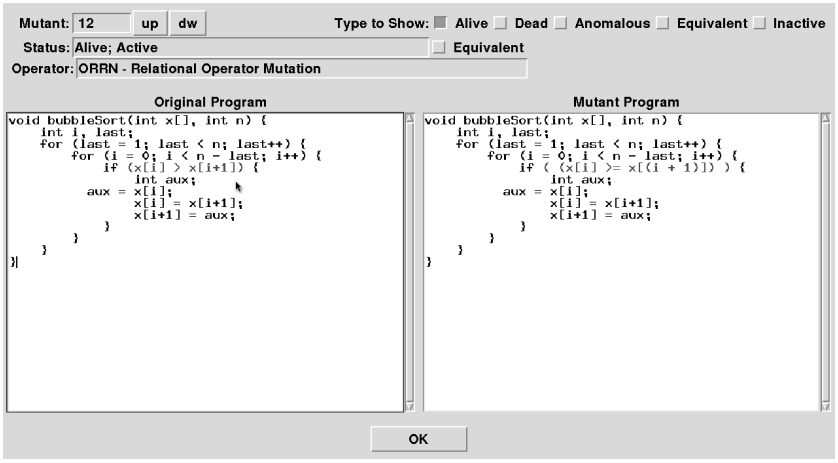
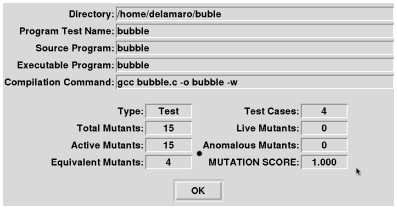


Figura 2.21. Análise dos mutantes vivos

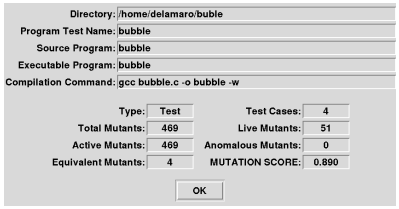
No nosso exemplo, iremos concluir que todos os 4 mutantes vivos são equivalentes. Marcando-os como tal e reexecutando os mutantes obtemos então um novo escore de mutação, conforme mostrado na Figura 2.22(a). Alcançando escore 1,0 terminamos o teste desse programa, para esse conjunto de mutantes, tendo obtido um conjunto adequado. Obviamente esse conjunto não é muito completo pois utilizamos um conjunto muito restrito de operadores de mutação. Podemos, porém, adotar uma estratégia incremental, gerando mutantes em diversas etapas, enquanto tivermos tempo e recursos para realizar a atividade de teste. Só como curiosidade, geramos todos os possíveis mutantes para o *bubbleSort* e os executamos com o conjunto de 4 casos de teste que fornecemos. O resultado é mostrado na Figura 2.22(b).

2.4. Automatização

Uma parcela considerável do custo das atividades de teste está associada à execução dos casos de teste. Tradicionalmente, os conjuntos de teste que foram gerados durante o desenvolvimento são reexecutados toda vez que uma manutenção ocorre. Este tipo de teste é chamado de *teste de regressão* e tem como objetivo "regredir" um software ao estado anterior antes da manutenção



(a) Resultado com os equivalentes identificados



(b) Resultado com todos os mutantes gerados

Figura 2.22. Progresso da sessão de teste

realizada, isto é, ao estado em que todos os casos de teste são executados com sucesso.

A reexecução dos casos de teste é uma atividade cara porque os mesmos dados de entrada fornecidos durante o teste realizado no desenvolvimento deverão ser fornecidos novamente e a validação das saídas obtidas deverá também ocorrer de novo. Sem mecanismos para automatizar a execução do teste de regressão, duas situações podem ocorrer: (1) o teste tem seu custo aumentado significativamente, pois os testadores deverão realizá-lo manualmente; ou (2) o teste de regressão será negligenciado, afetando-se assim a qualidade do software.

No contexto dos métodos de desenvolvimento ágil, a automatização do teste é uma prática fundamental. Segundo Beck [Beck 2002], um requisito de teste somente existe, isto é, está especificado, se houver um teste automatizado associado. A evolução desse conceito deu origem ao Desenvolvimento Dirigido a Teste (*TDD – Test-Driven Development*). TDD corresponde a uma estratégia de desenvolvimento de software na qual, para cada unidade de código-fonte que compõe uma funcionalidade do software, testes de unidade são desenvolvidos anteriormente à sua implementação [Crispin 2006].

A abordagem *Red-Green-Refactor* corresponde a um conjunto de atividades utilizadas na maioria dos métodos ágeis, sendo uma característica implícita à prática de TDD. Esta abordagem é composta pelos seguintes passos [Beck 2002, Andrea 2008].

1. (***Red***): Casos de teste automatizados são escritos. Ao serem executados, eles possivelmente nem mesmo compilarão, pois a funcionalidade ainda não foi implementada.
2. (***Green***): A funcionalidade é desenvolvida o suficiente para fazer com que os casos de teste executem com sucesso.
3. (***Refactor***): Uma vez que todos os casos de teste passarem, o código é melhorado de modo que as melhorias continuem assegurando o sucesso de execução do conjunto de teste.

De acordo com Beck [Beck 2002], apenas o uso de TDD pode não atender aos requisitos de usuário adequadamente pois, neste tipo de desenvolvimento, testes de unidade são escritos pelos desenvolvedores, e estes podem não implementar necessariamente o que o cliente deseja [Beck 2002]. Nesse contexto, surgiu a estratégia de Desenvolvimento Dirigido a Teste de Aceitação (*ATDD – Acceptance Test-Driven Development*) na qual as atividades decorrentes da abordagem *red-green-refactor* passam a ser realizadas para o teste de validação [Hendricksons 2008]. Deste modo, o cliente, com auxílio da equipe de desenvolvimento, escreve casos de teste de aceitação para uma determinada funcionalidade do sistema que, a seguir, é codificada pelos desenvolvedores. Por fim, o cliente executa os casos de teste de aceitação criados para validar a funcionalidade implementada. Para apoiar as características inerentes à prática de métodos ágeis é fundamental que o conjunto de teste de aceitação seja automatizado e facilmente modificável [Crispin 2002].

A Figura 2.23 ilustra a interação existente entre estas duas abordagens do desenvolvimento ágil de software. Conforme observado na figura, em ATDD casos de teste de aceitação são modelados pelo cliente. Em seguida, os desenvolvedores implementam, utilizando TDD, todo código suficiente para que aquela funcionalidade modelada seja atendida. Por fim, o cliente, executa o conjunto de teste de aceitação para validar as funcionalidades implementadas.

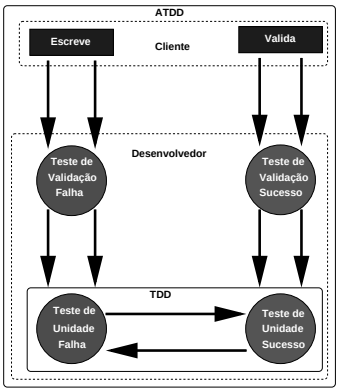


Figura 2.23. Interação entre ATDD e TDD.

Pode-se, portanto, afirmar que a automatização é uma atividade fundamental em qualquer abordagem de desenvolvimento, mas especialmente nos processos de desenvolvimento ágil. A seguir discutiremos os principais tipos de ferramentas que auxiliam o desenvolvedor a criar conjuntos de testes automatizados.

Ferramentas de automatização de testes

O termo **automatização de teste** é normalmente associado à execução automática de um conjunto de teste e à avaliação automática dos resultados gerados. Tradicionalmente, a execução automática de teste pode ocorrer em dois níveis. No nível de unidade ocorre por meio de ferramentas que executam casos de teste associados a uma unidade. As ferramentas utilizadas na automação do teste de unidade, como JUnit[1] e PHPUnit[2], também podem ser utilizadas no teste de integração, desde que os procedimentos ou objetos invocados utilizem a implementação final, e não versões simplificadas. As ferramentas que apoiam o teste de unidade são essenciais para a utilização do paradigma TDD.

A automatização dos testes de integração e aceitação, por sua vez, é tipicamente realizada por meio de ferramentas de captura e reexecução. Essas ferramentas automatizam casos de teste que fazem acesso direto à interface (gráfica, em geral) do sistema. Elas podem acessar a interface de uma aplicação *desktop* ou de uma aplicação Web, por isso, pressupõem que um conjunto de funcionalidades esteja implementado com a sua interface disponível. Semelhante às ferramentas de teste de unidade, as ferramentas de captura e reexecução fornecem mecanismos para comparação de saídas para permitir a validação automática dos resultados obtidos. A utilização do paradigma ATDD de desenvolvimento é baseada na disponibilidade desse tipo de ferramenta. A seguir descrevemos as ferramentas de teste de unidade e de captura e reexecução.

Teste de unidade automático

As ferramentas para teste de unidade são genericamente referidas como bibliotecas *xUnit*. A letra *x* é uma referência à linguagem alvo, que pode ser Java, C, PHP etc. A mais famosa biblioteca *xUnit* é a que apoia o teste de programas escritos em Java, conhecida como JUnit. A seguir descrevemos as principais funcionalidades da biblioteca JUnit; as demais bibliotecas funcionam de maneira semelhante.

A biblioteca JUnit permite que sejam criados casos de teste em uma classe Java por meio de anotações. Por exemplo, os casos de teste de unidade de uma classe chamada `Money` são definidos em uma classe de teste `MoneyTest`. Essa classe possui dois métodos `setUp()` e `tearDown()`. `setUp()` é um método chamado sempre antes de um caso de teste ser executado. Para isso adicionamos à sua declaraçção a anotação `@Before`. Sua função é inicializar variáveis, estruturas de dados ou conexões que serão utilizadas nos casos de teste. `tearDown()`, por sua vez, é chamado sempre depois de

[1] `http://www.junit.org`

[2] `http://www.phpunit.de`

executados os casos de teste e é responsável por “desfazer” estruturas ou conexões criadas em `setUp()`. Para tanto, adcionamos a ele a anotação `@After`. Na Figura 2.24 é exibida a classe de teste `MoneyTest`, bem como a definição do construtor, das variáveis de instância e dos métodos `setUp()` e `tearDown`.

```
package money;

import org.junit.* ;
import static org.junit.Assert.*;

public class MoneyTest {

// Variáveis de instância
private Money f12CHF;
private Money f14CHF;
private Money f28USD;

   // construtor
   public MoneyTest(String arg0)
   {
       super(arg0);
   }

   @Before
   public void setUp()
   {
       f12CHF = new Money(12, "CHF");
       f14CHF = new Money(14, "CHF");
       f28USD = new Money(28, "USD");
   }

   @After
   public tearDown()
   {
      f12CHF = null;
      f14CHF = null;
      f28USD = null;
   }
}
```

Figura 2.24. Criação da estrutura de teste de unidade com JUnit

Os demais métodos da classe `MoneyTest` constituem os casos de teste automatizados por JUnit. Na primeiras versões da biblioteca, os métodos que tinham o prefixo `test` nos seus nomes eram entendidos com um caso de teste a ser executado automaticamente. As novas versões permitem que métodos sem este prefixo sejam considerados como tal desde que anotados com `@Test` antes do método. Os métodos de teste são executados por reflexão quando é incluída uma chamada ao *test runner* no método `main()`.

Na Figura 2.25 são apresentados o método `main()` e o método de teste `simpleAddTest()`. Este último método testa se a soma de 12 francos suíços com 14 francos suíços é realizada corretamente por objetos da classe `Money`.

Para realizar a comparação dos resultados obtidos, JUnit possui um classe chamada `Assert`. Esta classe contém métodos que comparam um resultado obtido com o esperado. No exemplo da Figura 2.25 o método `assertEquals()` verifica se os dois objetos `Money` passados como parâmetros são iguais. Se forem iguais, o fluxo de execução continua normalmente. Caso contrário, `assertEquals()` lança uma exceção informando que o teste falhou.

```
@Test
public void simpleAddTest()
{
   Money expected = new Money(26, "CHF");
   Money result = f12CHF.add(f14CHF);
   Assert.assertEquals(result, expected);
}

public static void main ( String args[ ] ) {
    org.junit.runner.JUnitCore.main( "money.SimpleTest" ) ;
}
```

Figura 2.25. Caso de teste implementado com JUnit

Captura e reexecução

Ferramentas para captura e reexecução (*capture & playback*) já estavam disponíveis nos anos 70. Estas ferramentas têm por objetivo capturar os comandos realizados pelos testadores durante o teste e registrá-los na forma de *scripts*. Os *scripts* são programas gerados normalmente em uma linguagem de alto nível. Os comandos dessa linguagem são capazes de posicionar o cursor em determinado campo, de inserir valores númericos ou alfanuméricos e de emitir comandos (e.g., clicar em um botão ou enviar o caractere ENTER) para a execução do software. Quando executados os *scripts*, os comandos originalmente realizados pelos testadores são submetidos automaticamente ao software.

Selenium IDE é uma ferramenta *open-source* que realiza as funções de captura e reexecução para aplicações Web. Ela pode ser instalada em um navegador Mozilla Firefox para capturar os comandos emitidos pelo testador para testar uma aplicação Web. A ferramenta ainda fornece a possibilidade de se comparar a saída obtida com os valores esperados. Por exemplo, é possível verificar se um determinado texto da página retornada está ou não presente. A Figura 2.26 apresenta o sítio *Google Gmail* para o qual foi fornecido a cadeia de caracteres “vocemesmo” no campo *Username* e “123” no campo *Password*. No canto esquerdo da figura, observa-se a ferramenta Selenium IDE que registra todos os comandos emitidos pelo testador. Ao final do registro dos comandos, é solicitado que seja verificada a página de retorno quanto à presença do texto “Username”. Se a página de retorno contém a cadeia “Username” o teste é bem-sucedido, caso contrário ele falha.

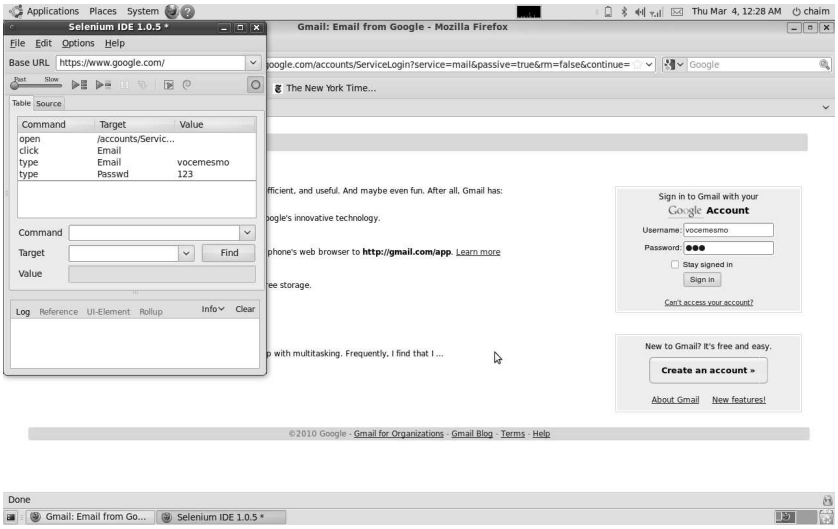


Figura 2.26. Exemplo de ferramenta de captura e reexecução – Selenium IDE.

O programa Java da Figura 2.27 é o *script* capturado automaticamente pela Selenium IDE. Note-se que este *script* pode ser executado automaticamente sem que haja intervenção humana. Nada impede, porém, que o programa Java seja criado por um programador sem a utilização do Selenium IDE, pois a biblioteca utilizada neste programa Java é parte do pacote *Selenium RC*. Na Seção 2.5 os componentes da ferramenta *Selenium* são descritos.

Existem várias ferramentas deste tipo disponíveis, comerciais ou não. *Vermont High Test* [3] é um exemplo de ferramenta comercial para o teste de aplicações *desktop*.

2.5. Teste de sistemas WEB

A principal característica das aplicações Web é utilizar os recursos da rede mundial de computadores – a *World Wide Web* (Web) – como meio para a implementação da arquitetura da aplicação.

Tipicamente, essas aplicações são caracterizadas por serem estruturadas em três camadas: de apresentação, de negócios e de acesso aos dados [Nguyen 2001]. A camada de apresentação utiliza navegadores (e.g., Internet Explorer, Mozilla Firefox) como *front end* para interagir com os usuários finais; a camada de negócios implementa as regras que constituem o núcleo da

[3] `http://www.vtsoft.com`

```
package com.example.tests;

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

public class testescript extends SeleneseTestCase {

   public void setUp() throws Exception {
     setUp("http://www.gmail.com/", "*chrome");
   }

   public void testTestescript() throws Exception {
selenium.open("/accounts/ServiceLoginAuth?service=mail");
selenium.click("Email");
selenium.type("Email", "vocemesmo");
selenium.type("Passwd", "123");
selenium.click("signIn");
selenium.waitForPageToLoad("30000");
assertTrue(selenium.isTextPresent("Username"));
   }
}
```

Figura 2.27. *Script* de teste gerado pela ferramenta Selenium

aplicação; e a camada de acesso aos dados contém as informações que serão acessadas e armazenadas durante a interação com a aplicação.

A Web permite que as camadas estejam distribuídas em diferentes localidades e máquinas. Tipicamente, a apresentação é realizada em uma máquina cliente e as camadas de negócio e de acesso aos dados em máquinas servidoras. Obviamente, nada impede que todas as camadas, ou duas delas, fiquem localizadas em uma única máquina. Porém, os recursos utilizados permitem que haja a distribuição dos diferentes componentes da aplicação.

Outra característica das aplicações Web são os seus requisitos não funcionais [Laprévote et al. 2009]. Essas aplicações tendem a ser utilizadas por vários usuários ao mesmo tempo em razão da arquitetura distribuída possiblitada pela Web. Além disso, a arquitetura utilizada permite que novas máquinas clientes e servidoras sejam incluídas, ou seja, as aplicações Web são *escaláveis*.

Assim, é esperado que as aplicações Web tenham requisitos estritos de tempo de resposta, de número de usuários atendidos e de disponibilidade da aplicação (e.g., 24 horas, 7 dias por semana), entre outros. Requisitos não funcionais como usabilidade e segurança são também de fundamental importância nessas aplicações haja vista o carácter interativo da maioria delas e o fato de a informação trocada entre os componentes da aplicação poder circular pela rede mundial de computadores.

Nesta seção pretende-se dar uma visão das atividades de teste das aplicações Web. Essas atividades levam em consideração os tipos de teste e as técnicas que podem ser usados, as estratégias mais adequadas às aplicações Web e os requisitos não funcionais típicos. Por fim, as ferramentas que podem

ser utilizadas para o teste de aplicações Web são apresentadas. O texto a seguir é baseado no contéudo do relatorório técnico "*Test suites and benchmarks for the chosen set of Open Source projects and artifacts. Methodology for creating test suites and benchmarks for arbitrary systems*" [Laprévote et al. 2009] produzido dentro do escopo do projeto QualiPSo[4].

2.5.1. Tipos e técnicas de teste

Como a maioria dos sistemas de software, aplicações Web necessitam que sejam testados os seus requisitos funcionais e não funcionais. Para testá-los, há diferentes tipos e técnicas de teste. A seguir, discutimos os testes essenciais das aplicações Web e as técnicas de teste que podem auxiliar no seu desenvolvimento.

Teste de requisitos funcionais

As técnicas de teste mais utilizadas para testar os requisitos funcionais das aplicações Web são as técnicas caixa branca e caixa preta.

O teste caixa branca pode ser utilizado em qualquer nível (teste de unidade ou integração; teste de sistema ou aceitação); porém, é mais utilizado para o teste das unidades (e.g., métodos, classes) que compõem as aplicações. Normalmente, o teste é automatizado utilizando mecanismos como JUnit e PHPUnit.

A técnica caixa preta, por sua vez, é também utilizada no teste de unidade. Porém, seu uso mais frequente é nos testes de sistema e de aceitação por ser apropriada para aplicações grandes, complexas e com muitos caminhos de execução possíveis [Laprévote et al. 2009].

Teste de requisitos não funcionais

Testes que verificam requisitos não funcionais como o teste de desempenho, teste de carga/volume e o teste de estresse são fundamentais em aplicações Web [Laprévote et al. 2009]. O teste de desempenho tem por objetivo produzir dados que permitam prever o desempenho da aplicação quando diferentes níveis de carga (e.g., número de requisições) são submetidos. Em especial, espera-se identificar os níveis de carga que provocam a exaustão do recursos do sistema.

O teste de carga/volume é semelhante ao teste de desempenho, porém visa a verificar como a aplicação comporta-se quando submetida a um grande volume de dados, cálculos e processamento intensivo. Já o teste de estresse força a aplicação a operar em condições de recursos reduzidos. O objetivo é verificar o comportamento em situações acima do limites estabelecidos para

[4] `http://www.qualipso.org`

garantir que a aplicação funciona adequadamente ou que as falhas sejam tratadas adequadamente sem perda ou corrupção dos dados.

A técnica de teste indicada para o desenvolvimento dos testes de desempenho, carga/volume e estresse é a técnica teste caixa cinza [Nguyen 2001]. Semelhantemente ao teste caixa preta, o foco deste teste é verificar se a saída obtida corresponde à saída esperada. A diferença é que o testador utiliza conhecimento das estruturas de dados, dos algoritmos utilizados, do ambiente de execução, dos limites estabelecidos e da arquitetura da aplicação para desenvolver os testes. Note-se, no entanto, que não é necessário conhecer o código fonte. Com essas informações e ferramentas adequadas (ver Seção 2.5.3), é possível desenvolver testes que avaliem requisitos não funcionais das aplicações Web.

Outros requisitos não funcionais como segurança, usabilidade e compatibilidade não podem ser negligenciados em aplicações Web. Para o teste desses requisitos não existem técnicas de teste em particular. Há estratégias como inspeção do código para detecção de falhas de segurança, avaliação do uso da aplicação para identificação de falhas de usabilidade e ergonomia e avaliação da compatiblidade da aplicação com diferentes navegadores.

2.5.2. Estratégias de teste

Nesta seção discutimos as estratégias para o teste das camadas componentes das aplicações Web, bem como as estratégias para o teste dos aspectos de segurança, usabilidade e compatiblidade.

Teste das camadas

A arquitetura típica de uma aplicação Web é (ou deveria ser) dividida em camadas. A divisão clássica estabelece três camadas, a saber, de apresentação, de negócios e de acesso aos dados. Essa arquitetura direciona a estratégia de teste a ser utilizada. Por ser dividida em camadas, o testador pode concentrar o esforço de teste em cada uma das camadas.

Durante o teste de uma camada, os objetos da camada inferior invocados são substituídos por “dublês de teste” (*test doubles*) [Meszaros 2007]. Essa abordagem evita que falhas devidas ao mau funcionamento de outras camandas sejam relatadas e permite que o desenvolvimento das camadas ocorra em paralelo. Os dublês de teste podem ser de vários tipos, a saber:

- objetos burros (*dummy*): utilizados somente para preencher uma lista de parâmetros e nunca são usados;
- objetos falsos (*fake*): possuem uma implementação simplificada, por exemplo, com valores fixos (*hardcoded values*);
- objetos substitutos (*stub*): são também objetos falsos com implementação simplificada, mas que possuem funcionalidade adicional como registrar as chamadas de seus métodos;

- objetos imitadores (*mock*): são os dublês mais complexos, pois podem ser programados para esperar chamadas específicas, tratar chamadas inesperadas e também verificar se todas as chamadas esperadas foram recebidas.

É importante observar que a estratégia delineada acima não é exclusiva das aplicações Web, podendo ser aplicada em outros tipos de sistemas que utilizam arquiteturas em camadas. A seguir, discutimos estratégias específicas para cada uma das camadas das aplicações Web.

Teste do acesso aos dados

A camada de acesso aos dados é, em geral, construída utilizando Objetos de Acesso a Dados (*DAOs – Data Access Objects*). Existem duas estratégias para testar DAOs. A primeira utiliza a base de dados real como se fosse a própria aplicação. Dependendo do tamanho e do tipo da base de dados, pode ser a base de dados local do desenvolvedor, uma base de dados dedicada para testes ou mesmo a base de dados de desenvolvimento (não recomendado) [Laprévote et al. 2009].

Outra abordagem é baseada em objetos imitadores. A utilização desse tipo de objetos tem a vantagem de não utilizar a base de dados real, de aumentar a rapidez do teste, pois não há dependência de conexão com a base de dados, e de facilitar tratamento de erros. Por outro lado, objetos imitadores precisam ser implementados, o que significa um custo adicional de programação que não ocorre quando a base de dados real é utilizada.

Teste da lógica de negócios

O teste da camada de negócios é similar ao teste de outros tipos de software [Laprévote et al. 2009]. Na maioria das vezes, são utilizados objetos imitadores para fazer o papel da camada de acesso aos dados, ou seja, são criados imitadores DAO. Ao utilizar dublês de teste, não é necessário considerar aspectos como *drivers* para acesso à base de dados. Além disso, os dublês facilitam o tratamento de erros e a automatização do teste. Porém, é também possível utilizar os DAOs reais que acessam a camada de dados. Neste caso, estaremos realizando o teste de integração, além do teste das regras de negócio.

Teste da camada da apresentação

A camada de apresentação é realizada utilizando o formato HTML. Páginas neste formato são enviadas ao cliente por meio do protocolo HTTP. Navegadores apresentam as páginas enviadas ao usuário final que interage por meio delas.

Ferrramentas como HTTPUnit[5], Selenium[6] e HTMLUnit[7] aproveitam o conhecimento do formato HTML para simular as ações do usuário como, por exemplo, preencher um campo ou apertar um botão. Dessa maneira, é possível desenvolver programas ou *scripts* que executem e validem os casos testes da camada de apresentação. A validação dá-se por meio da comparação das páginas recebidas como resposta com o resultado esperado. Os testes realizados na camada de apresentação em geral são o de aceitação e o de sistema. O seu desenvolvimento na maioria das vezes utiliza as técnicas caixa preta e caixa cinza.

Uma tarefa padrão do teste da camada de apresentação é a validação de *links*. Ela consiste em verificar se todos *links* levam a sítios válidos e disponíveis. *Links* inválidos frequentemente indicam a existência de defeitos. Eles dificultam a navegação e fazem com que parte das funcionalidades da aplicação fiquem indisponíveis. É uma tarefa importante e fácil de automatizar utilizando as ferramentas adequadas.

Teste de segurança

Como muitas outras, as aplicações Web armazenam dados dos usuários. Diferentemente das maioria das outras, essas aplicações estão disponíveis na Web para acesso por qualquer pessoa. Isto é um risco se pessoas mal intecionadas obtêm acesso aos dados dos usuários. Portanto, segurança é provavelmente o requisito não funcional mais importante, embora seja extremamente difícil de testar e verificar [Laprévote et al. 2009].

Existem diretrizes e padrões a serem seguidos para garantir a segurança como, por exemplo, verificar ameaças comuns como injeção de SQL e de JavaScript. Ferramentas que realizam análise estática de código podem ser configuradas para fazer verificação de potenciais padrões de código que possam levar a problemas de segurança.

Esses procedimentos, apesar de serem fortemente recomendados, não são suficientes para proteger uma aplicação Web de todas as possíveis ameaças. Sistemas empresariais ou críticos necessitam de revisões de código, conduzidas por especialistas que identifiquem ameaças específicas para uma determinada aplicação. Infelizmente, essas auditorias são realizadas por terceiros e, normalmente, são caras.

Teste de compatibilidade

O principal problema de compatiblidade das aplicações Web são os navegadores utilizados para acessá-las. Existem vários navegadores disponíveis

[5] `http://httpunit.sourceforge.net`

[6] `http://seleniumhq.org`

[7] `http://htmlunit.sourceforge.net`

(e.g., Internet Explorer, Mozilla Firefox, Opera, Safari), o que torna normalmente difícil, e frequentemente caro, desenvolver uma aplicação Web compatível com todos. A solução é identificar os navegadores mais populares entre os usuários e garantir a compatibilidade em relação a eles.

Uma diretriz para garantir a compatibilidade é seguir padrões durante o desenvolvimento da camada de apresentação. Um teste simples de compatibilidade com vários navegadores é verificar se as páginas se comportam e apresentam visual semelhante. Mesmo para aquelas páginas compatíveis, é preciso verificar se a aparência e o funcionamento possuem pequenas diferenças por causa do mecanismo de renderização e a diferentes implementações de *scripts*.

Os casos testes projetados para a camada de apresentação podem ser executados para verificar o comportamento em diferentes navegadores. A verificação da aparência, no entanto, deve ser realizada por especialistas em interação humano-computador via Web.

Teste de usabilidade

Usabilidade é o requisito não funcional mais subjetivo de qualquer sistema de software. Existem diretrizes a serem seguidas no desenvolvimento do projeto de interação humano-computador para aplicações em geral [Tidwell 2005] e para aplicações Web [Scott e Neil 2009], em particular. Essas diretrizes devem ser seguidas no desenvolvimento, porém, é indispensável testar a usabilidade do sistema com usuários típicos do sistema.

A partir dos resultados desse teste, é possível avaliar a facilidade com que os usuários se acostumam com a interface e realizam suas tarefas e como eles conseguem recuperar-se de ações incorretas, por exemplo. Medidas objetivas como o tempo para realizar uma tarefa e avaliações subjetivas como comentários dos usuáros auxiliam o desenvolvimento da interação-humano computador da aplicação.

2.5.3. Ferramentas

A seguir, descrevemos dois tipos de ferramentas para auxiliar o teste de aplicações Web: ferramentas para automatização e para avaliação de desempenho e carga. O objetivo não é apresentar o estado da arte em termos de ferramentas para testes Web, mas descrever exemplos de ferramentas que são úteis ao teste Web.

Ferramentas para automatização

HttpUnit é uma biblioteca Java para auxiliar o teste de aplicações Web. É utilizada juntamente com a biblioteca JUnit para executar casos testes que verifiquem a funcionalidade de uma página Web. Operações como preencher

um formulário e “clicar” em um botão são fornecidas. Além disso, HttpUnit permite que as páginas retornadas sejam comparadas com as saídas esperadas, possibilitando a validação automática dos testes. Os testes gerados utilizando HttpUnit implicam que o testador deve escrever programas Java que realizem as operações fornecidas pela biblioteca. HtmlUnit é uma biblioteca Java que possui recursos semelhantes aos da biblioteca HttpUnit.

Selenium corresponde a um conjunto de ferramentas voltadas à automatização do teste de aplicações Web. Seu principal objetivo é automatizar a interação entre usuário e aplicação Web. Seu conjunto de ferramentas é mais completo que HttpUnit e HmtlUnit, visto que, além de fornecer uma biblioteca para programação dos casos de testes, permite também que eles sejam registrados na forma de *scripts* como uma ferramenta de captura e reexecução. Os *scripts* podem ser gerados em várias linguagens como Java, Groovy, entre outras. As principais ferramentas de Selenium são as descritas abaixo:

- **Selenium Remote Control (SRC):** ferramenta responsável pela criação e execução de *scripts* de testes para aplicações Web. Os *scripts* descrevem ações que devem ser realizadas na aplicação sob teste.
- **Selenium IDE:** extensão do navegador Mozilla Firefox para gravação e reprodução de teste. Esta ferramenta corresponde a um ambiente gráfico no qual é gravada a interação usuário-navegador, gerando assim um *script* executável em SRC. A ferramenta também possui a funcionalidade de executar o *script* gerado.

Ferramentas para avaliação de desempenho e carga

Httperf[8] é uma ferramenta *open-source* para análise quantitativa de servidores Web [Laprévote et al. 2009]. Essa ferramenta tem por objetivo auxiliar o teste de desempenho por meio da geração de diferentes cargas de acordo com o protocolo HTPP. Os protocolos HTTP/1.0, HTTP/1.1 e SSL são apoiados.

O objetivo principal é fornecer estatísticas relativas ao desempenho do sevidor Web, medindo a sua capacidade de atendimento de requisições. Para realizar essa tarefa, a ferramenta mantém a taxa de requisições ao servidor acima da sua capacidade. A sobrecarga dos servidores é realizada de duas formas: por meio da geração de requisições e de URLs.

A ferramenta coleta as seguintes estatísticas relacionadas com as requisições: número de conexões; tempo de conexão; taxa e tamanho das requisições; taxa, tempo e duração das respostas e status; erros como tempo esgotado nos clientes, tempo esgotado nos *sockets*, conexões recusadas etc. Httperf também fornece medidas relacionadas com as sessões, a saber, tempo médio para completar uma sessão; tempo médio antes que uma sessão falhe e número de sessões por conexão. Httperf é uma ferramenta extensível, podendo ser adicionados novos geradores de carga e novas medidas de desempenho.

[8] `http://www.hpl.hp.com/research/linux/httperf`

Apache JMeter[9] é provavelmente a ferramenta mais conhecida e utilizada para a avaliação de desempenho de sítios Web. Trata-se de uma ferramenta *open-source* desenvolvida pelo projeto Apache Jakarta. JMeter permite aos usuários criar cenários reais e executá-los concorrentemente em múltiplas linhas de execução (*threads*). A ferramenta é capaz de testar aplicações que utilizam as seguintes tecnologias: Web – HTTP (HyperText Transfer Protocol), HTTPS (HyperText Transfer Protocol Secure), SOAP (Simple Object Access Protocol); base de dados por meio de JODBC (Java Database Connectivity) LDAP (Lightweight Directory Access Protocol), JMS (Java Message Service); e correio eletrônico – POP3 (Post Office Protocol).

Os cenários de teste são criados utilizando a interface gráfica de JMeter. Todo plano de teste é composto de elementos simples como: *ThreadGroup* – elemento de configuração da linha de execução; *Samplers* – conectam com o componente em teste e obtém resposta; *Logic Controllers* – permite a definição de laços e condições; *Listeners* – elementos que coletam e visualizam informação do caso de teste como tempo de resposta e resultado; *Assertions* – elementos que permitem verificar valores específicos de resposta esperados; e *Processors* – pré e pós-processadores que podem mudar os valores da requisição ou processar o valor recebido como resposta.

Os planos de teste podem ser executados simultaneamente em um número de *threads* concorrentes definidos pelo testador. Os resultados e medidas obtidos são então registrados e visualizados pelos *listeners*. Outra característica interessante de JMeter é distribuir o teste. Um grupo de máquinas pode cooperar para simular tráfego impossível de se gerar com uma única máquina. Nesse grupo há um nó que funciona como mestre/coordenador, coletando os dados das outras máquinas [Laprévote et al. 2009].

2.6. Geração automática de dados de teste

Para garantir um sistema livre de defeitos, todo software deveria ser executado com todas as entradas possíveis de forma exaustiva, de modo que todas as possibilidades de execução do sistema sejam exploradas, antes de sua liberação. Infelizmente, diversos problemas inerentes da própria computação ou do grande número de entradas que a maioria dos produtos de software aceitam, além do alto custo envolvido em sua execução, tornam o teste exaustivo impraticável.

Para auxiliar a sistematizar a atividade de teste e orientar o testador quando um software foi suficientemente testado é que foram criados os critérios de teste definidos nas seções anteriores. O principal objetivo desses critérios é subdividir o domínio de entrada em subdomínios, não necessariamente disjuntos, e exigir do testador que pontos (dados de teste) de cada subdomínio sejam selecionados. Mas quais pontos devem ser escolhidos? A princípio, deveriam ser escolhidos aqueles com maior probabilidade de detectar a presença de de-

[9] `http://jakarta.apache.org/jmeter/index.html`

feitos no programa em teste. A estratégia empregada para a geração de dados de teste (GDT) é de fundamental importância pois dela depende a eficácia dos dados de teste obtidos.

A completa automatização para a geração automática de dados de teste é prejudicada por problemas, tais como: caminhos ausentes, caminhos não executáveis e equivalência de programas. Entretanto, mesmo na presença dessas limitações, várias pesquisas são realizadas nessa área e esta seção visa a apresentar uma síntese do estado atual dessa área, descrevendo os principais trabalhos desenvolvidos.

2.6.1. Conceitos básicos

O teste exaustivo significa executar o programa em teste com todos os possíveis valores de seu domínio de entrada. Uma vez que o domínio de entrada pode ser (pelo menos em teoria) infinito, executar o programa para todo o domínio é, em geral, impossível de ser conduzido. Mesmo para domínios finitos mas com muitos elementos, o teste exaustivo pode ser proibitivo por causa das restrições de tempo e custo impostas pelos projetos de software. Desse modo, a única opção é usar apenas parte do domínio durante a atividade de teste de modo que a questão a ser respondida seja quais valores de entrada seriam selecionados para maximizar a chance de detecção de defeitos e permitir a execução dos casos de teste respeitando as restrições de tempo e custo impostas.

O exemplo a seguir é uma adaptado do trabalho de Dijkstra [Dijkstra 1970] “On the reliability of mechanisms” no qual ele demonstra a impossibilidade do teste exaustivo e define o famoso corolário: “o teste de programa somente pode ser utilizado para detectar a presença de defeitos, mas nunca para detectar a sua ausência”.

Considere um simples método em Java que recebe como parâmetro dois argumento de um tipo primitivo `double`, cada um com uma representação de 64 bits. Esse método tem claramente um domínio de entrada finito com 2^{128} elementos ($2^{64} * 2^{64}$) considerando todas as combinações possíveis. Supondo que esse método seja executado em uma máquina capaz de realizar 2^{40} instruções por segundo, que é compatível com a velocidade dos processadores atuais, o teste exaustivo desse método levaria ($\frac{2^{128}}{2^{40}} = 2^{88} \approx 10^{26}$) segundos para ser completado. Como um ano tem aproximadamente 10^7 segundos, a execução dos casos de teste terminariam em torno de $\frac{10^{26}}{10^7} = 10^{19}$ anos, claramente um prazo inviável de ser cumprido.

A partir do exemplo acima podemos observar que a grande dificuldade do teste consiste em identificar quais os melhores elementos a serem selecionados para executar o programa em teste de modo a detectar a maior quantidade de defeitos no menor tempo e no menor custo, ou seja, como criar os melhores casos de teste. Conforme descrito na seção introdutória, um caso de teste é composto de duas partes, o dado de teste – correspondente ao valor de entrada a ser utilizado para executar o programa – e a saída esperada,

determinada pelo oráculo de teste. A atividade de geração de dados de teste é bastante complexa e envolve diversos passos e a cada passo, uma série de questões relacionadas.

Em termos de pesquisa, diferentes representações para distinguir as técnicas de geração automática de dados de teste (GADT) são possíveis, tais como, baseadas em especificação, baseada em código, aleatória, estática ou dinâmica. A Figura 2.28, adaptada de Mahmood [Mahmood 2007], destaca as áreas às quais as técnicas de GADT podem pertencer. Aqui, também, podemos observar que existe um complemento entre as técnicas funcional e estrutural, principalmente pelo fato que que, em geral, elas são utilizadas em momentos diferentes no teste dos produtos de software.

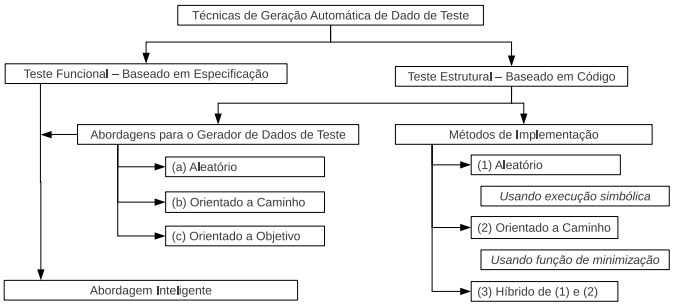


Figura 2.28. Revisão das técnicas de GADT (adaptada de Mahmood [Mahmood 2007])

Conforme ilustrado na Figura 2.28, as técnicas da GADT estão divididas em duas classes, ou seja, teste funcional ou teste estrutural, detalhadas a seguir em conjunto com as subdivisões apresentadas [Mahmood 2007].

2.6.2. Técnicas de geração

Abordagem de GDT Aleatória.

Como definida por Korel [Korel 1996], geração aleatória de dado de teste consiste simplesmente na geração de entradas ao acaso, até que uma entrada útil seja encontrada. Essa abordagem é rápida e simples mas pode não ser uma boa escolha para sistemas complexos ou se utilizada em conjunto com critérios de adequação complexos. A probabilidade de selecionar uma entrada adequada ao acaso pode ser muito baixa.

Abordagem de GDT orientada a caminho.

A abordagem típica utilizada aqui é a geração de caminhos a partir do grafo de fluxo de controle do programa. Nessa abordagem, inicialmente o grafo é

criado e, posteriormente, um caminho particular é selecionado. Auxiliado por uma técnica de execução simbólica, considerando análise estática, ou função de minimização, considerando uma análise dinâmica, o dado de teste que executa o caminho pode ser gerado. No caso de execução simbólica, variáveis simbólicas são usadas no lugar de valores reais durante a travessia no caminho [Pargas et al. 1999, Hajnal e Forgács 1998]. A existência de caminhos não executáveis é um problema que impossibilita a geração de dados de teste para todo caminho selecionado. A complexidade dos tipos de dados empregados no programa em teste também dificulta sua aplicação.

Abordagem de GDT orientada a objetivo.

Nessa abordagem, um dado de teste é selecionado a partir de um conjunto de entrada já disponível visando à execução do objetivo desejado, tal como um comando, independentemente do caminho escolhido para atingir o objetivo [Pargas et al. 1999]. Dois passos básicos são realizados: 1) identificar o conjunto de comandos (e os arcos respectivos) que se cobertos implicam em cobrir o critério; e 2) gerar o dado de entrada que executa cada comando selecionado (e os arcos respectivos) [Gotlieb et al. 1998]. As abordagens “baseada em assertiva” e “encadeamento” são caracterizadas como orientadas a objetivo. Na primeira, assertivas são inseridas e então resolvidas enquanto que a segunda exige que seja realizada uma análise de dependência de dados.

Abordagem de GDT inteligente.

Consiste na utilização de técnicas sofisticadas de análises do código ou de sua especificação, empregando métodos de Inteligência Artificial para guiar a busca por novos dados de teste, tanto para o teste funcional quanto estrutural [Michael e McGraw 1998, Pargas et al. 1999, Tracey et al. 1998].

Métodos estáticos.

Consistem naqueles utilizados para a análise e verificação de representações do sistema, tais como o documento de requisitos, diagramas de projeto e o código fonte do software, de forma manual ou automática, sem exigir sua execução [Chu et al. 1997, Sommerville 2007]. Em geral, a análise estática é realizada por meio de execução simbólica que visa a identificar as restrições das variáveis de entrada para um critério de teste particular. Soluções a essas restrições representam dados de teste [Tracey et al. 1998].

Métodos dinâmicos

Em vez de empregarem substituição de variáveis como na execução simbólica, executam o programa em teste com algum valor de entrada, gerado,

possivelmente, de forma aleatória [Chu et al. 1997]. Monitorando o fluxo de execução do programa é possível identificar se o caminho desejado foi ou não percorrido. Em caso negativo, retorna-se até o ponto onde o caminho foi desviado e, utilizando-se métodos de busca diferentes, as entradas podem ser manipuladas até que a direção correta seja tomada.

Métodos híbridos

Combinam métodos estáticos e dinâmicos de modo que os benefícios de cada método possam ser utilizadom de forma sincronizada visando a facilitar a geração do dado de teste desejado. Por exemplo, no trabalho de Meudec [Meudec 2001] a execução simbólica é utilizada pela abordagem dinâmica.

2.6.3. Exemplo de aplicação

Para ilustrar o processo de criação de um dado de teste, foi escolhida a abordagem proposta por Korel [Korel 1990] por ser um dos representantes das abordagens com mais demandas de pesquisa. Ela apoia a geração de dados de teste para critérios caixa-branca, utilizando uma abordagem orientada a caminho e que emprega métodos dinâmicos.

Utilizando o exemplo clássico da classificação de triângulos [McMinn 2004], a Figura 2.29 apresenta uma possível implementação desse programa em linguagem C e ao lado está o grafo de fluxo de controle da função `tri_type`. Nesse grafo, o nó s (*start node*) representa o nó de entrada e o nó e (*exit node*) o nó de saída. O rótulo dos demais nós faz referência aos comandos que eles contém (comentários associados ao início das linhas da Figura 2.29).

Na abordagem de Korel [Korel 1990], o procedimento de geração de dado de teste é executado em uma versão instrumentada do programa original. Buscando executar um determinado caminho do programa, uma entrada arbitrária é escolhida. Se, durante a execução, um ramo diferente do previsto é seguido – um que desvie do caminho desejado – uma busca local por entradas do programa é realizada utilizando uma função objetivo derivada a partir do predicado de interesse, referente ao ramo alternativo. A função objetivo descreve quão próximo se está do predicado desejado ser verdadeiro. O valor computado é dito a distância do ramo.

Considere que desejamos executar o caminho `s,1,5,9,10,11,12,13,14,`e. Se a função `tri_type` for executada com a entrada $(a = 10, b = 20, c = 30)$, o fluxo de controle segue com sucesso os ramos falsos entre os nós de 1 a 5. Entretanto, o fluxo de controle diverge do desejado no nó 9. Nesse ponto, a busca local é iniciada visando a alterar os dados de entrada de modo que o ramo desejado seja executado. Assumindo que os predicados dos ramos são da forma a *op* b, sendo a e b expressões aritméticas e *op* um operador relacional, uma função objetivo na forma f *rel* 0 é derivada, sendo f e *rel* definidos conforme a Tabela 2.2.

```
/*  s  */ int tri_type(int a, int b, int c)
          {
            int type;
/*  1  */   if (a > b)
/* 2-4 */   {   int t = a; a = b; b = t;   }
/*  5  */   if (a > c)
/* 6-8 */   {   int t = a; a = c; c = t;   }
/*  9  */   if (b > c)
/*10-12*/   {   int t = b; b = c; c = t;   }
/* 13  */   if (a + b <= c)
            {
/* 14  */     type = NOT_A_TRIANGLE;
            }
            else
            {
/* 15  */     type = SCALENE;
/* 16  */     if (a == b && b == c)
              {
/* 17  */       type = EQUILATERAL;
              }
/* 18  */     else if (a == b || b == c)
              {
/* 19  */       type = ISOSCELES;
              }
            }
/*  e  */   return type;
          }
```

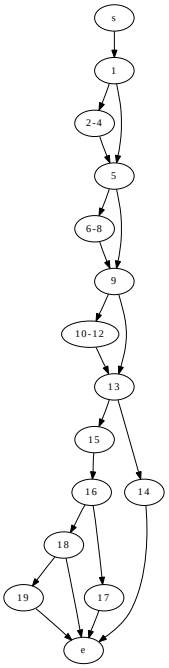


Figura 2.29. Programa exemplo de classificação de triângulos (extraído de McMinn [McMinn 2004])

Tabela 2.2. Funções objetivo para predicados relacionais [Korel 1990]

Predicado relacional	f	*rel*
$a > b$	$b - a$	$<$
$a \geq b$	$b - a$	$\leq$
$a < b$	$a - b$	$<$
$a \leq b$	$a - b$	$\leq$
$a = b$	*abs* $(a - b)$	$=$
$a \neq b$	*-abs* $(a - b)$	$<$

A função f deve ser minimizada para um valor positivo (ou zero, se *rel* é $<$) quando o predicado do ramo atual para o ramo desejado for falso, ou para um valor negativo (ou zero, se *rel* é $=$ ou $\leq$) quando for verdadeiro. Considerando o predicado para o ramo verdadeiro a partir do nó 9, a função objetivo é $c - b > 0$. O valor dessa função para a entrada $(a = 10, b = 20, c = 30)$ é $30 - 20 = 10$. Desse modo, o programa deve ser instrumentado de modo que tais valores

possam ser computados. Isso pode ser feito por meio de uma expressão de ramo, como no exemplo a seguir:

```
if (eval_obj(9, b, c))
{
    ...
```

A função *eval_obj* reporta a distância para que a execução siga pelo ramo verdadeiro a partir do nó 9, usando as variáveis b e c. A função retorna um valor verdadeiro correspondente à avaliação da expressão original de modo a não alterar a semântica do programa. Esse mecanismo de busca local para derivar valores de entrada utilizando uma função objetivo recebe o nome de **método de alternância de variáveis** (*alternating variable method*) [McMinn 2004]. O valor de cada variável de entrada é ajustado individualmente, mantendo o valor das demais variáveis constantes. O primeiro estágio de manipulação das variáveis é chamado de fase exploratória. Ele investiga a vizinhança da variável, incrementando ou decrementando seu valor original e reavaliando a função objetivo para identificar qual padrão causa uma melhoria na busca pelo objetivo desejado, levando à próxima fase, denominada fase padrão. Nessa fase, uma série de alterações maiores no valor da variável é realizada visando a encontrar o mínimo para a função objetivo. Atingido esse objetivo, a próxima variável é então selecionada e a fase exploratória é reiniciada.

Retomando o exemplo apresentado em que a execução tomou um rumo diferente daquele desejado a partir do nó 9, incrementar ou decrementar o valor da variável a nesse caso não altera o valor da função objetivo, de modo que essa variável tem o seu valor original mantido. Em seguida, a variável b é analisada e um decremento na variável b piora o resultado da função objetivo, mas um incremento leva a uma melhoria. Assim, a fase padrão tem início e a variável b é incrementada até $b > c$. Supondo que o valor 31 é atingido, o novo vetor de entrada ficaria $(a = 10, b = 31, c = 30)$. A partir dessa nova entrada, a execução segue corretamente a partir do nó 9 como era desejado mas ela diverge novamente no nó 13 uma vez que o valor de $a+b$ nesse nó é maior que o valor de c. Uma busca local é iniciada novamente para ajustar o valor das variáveis de modo que o ramo verdadeiro seja seguido, mantendo a execução correta dos ramos anteriores. A função objetivo derivada do predicado do ramo verdadeiro é $(a + b) - c \leq 0$. Um decremento no valor da variável b causa uma violação do sub-caminho a partir do nó 9, já um incremento causa uma melhoria na função objetivo uma vez que os valores de b e c são trocados nos nós de 10 a 12. Eventualmente, o vetor de entrada $(a = 10, b = 40, c = 30)$ é encontrado e o caminho completo desejado é executado.

É importante observar que, como todo método de busca local, o resultado final é dependente da solução inicial fornecida e, dependendo desse vetor de entrada inicial a solução pode não ser encontrada por causa das violações que podem ocorrer. Para mais informações sobre métodos de busca e outros

exemplos de geração de dados de teste por meio dessa abordagem o leitor pode consultar o trabalho de McMinn [McMinn 2004].

2.7. Oráculos de teste

Nesta seção discutimos um pouco mais sobre oráculos de teste. Em particular, vamos abordar o problema da sua automatização e daqueles programas cujas saídas não são facilmente interpretáveis, de modo que decidir se uma execução está correta não é tarefa trivial

De acordo com a definição de Baresi e Young [Baresi e Young 2001], um oráculo de teste é um método utilizado para verificar se o sistema em teste se comporta corretamente em uma particular execução. O papel de oráculo pode ser desempenhado por um outro programa – no caso de um oráculo automatizado – ou por um ser humano que, baseado em uma especificação, decide se o comportamento obtido é aquele esperado.

Essa definição não dá exatamente a dimensão dos problemas existentes na aplicação de um oráculo, seja ele humano ou automatizado. Weyuker [Weyuker 1982] faz uma análise bastante interessante sobre o teste de programas chamados “não testáveis”, ou seja, aqueles para os quais não existe um oráculo ou não existe um oráculo que seja aplicável na prática, por exemplo, por causa do seu alto custo. A pesquisadora mostra que, em muitos casos, embora não se possa saber exatamente o resultado esperado para uma determinada execução, é possível determinar características desejadas ou indesejadas que dão uma noção sobre a correção ou incorreção de uma execução do programa em teste. Aponta, também, algumas alternativas para que se possa testar tais programas para os quais não existem oráculos.

Um dos exemplos fornecidos é o problema de se calcular a milésima casa decimal do número π. Nesse caso, não apenas não existe um oráculo mas também não é possível dizer o quão plausível é a resposta fornecida pelo programa. Uma solução, embora limitada, seria alterar ligeiramente o programa para que ele calculasse a décima casa decimal, de forma que se o resultado apresentado fosse correto, o testador poderia ter alguma confiança de que o programa funcionaria também para o problema original. Existem também os casos em que é fácil excluir alguns resultados que claramente divergem daquele esperado mas para os quais dificilmente se consegue determinar exatamente o valor esperado. Por isso, é necessário admitir uma margem de erro dentro da qual a resposta pode ser considerada correta.

A automatização de oráculos é um item fundamental para promover a produtividade na atividade de teste. A existência de um programa que possa decidir – ainda que não de forma exata – sobre a correção de uma execução do programa em teste pode representar economia de tempo e maior precisão, uma vez que a intervenção humana pode ser, além de lenta, imprecisa e sujeita a variações.

Um caso bastante comum de programas que são difíceis de se decidir sobre a correção de suas saídas ocorre quando resultados se apresentam no

formato gráfico. Programas com interfaces gráficas (GUI, do inglês, *Graphical User Interface*) ou de processamento de imagens, são exemplos desse tipo de aplicação. Mesmo quando existe uma imagem que possa servir de modelo, uma comparação direta (pixel-a-pixel) entre ela e o resultado da execução do programa em teste, em geral, não produz o resultado esperado. Pequenas variações de forma, cor, textura em relação à imagem modelo não representam necessariamente falhas. A Figura 2.30 mostra uma interface gráfica na qual os resultados apresentados são os mesmo mas as aparências diferem devido a diferenças no ambiente de execução.

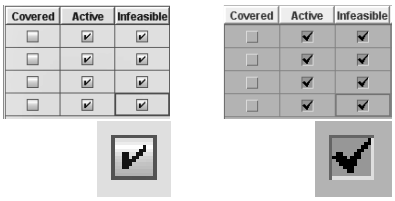


Figura 2.30. Mesma GUI com diferentes aparências

Para GUIs, a abordagem mais comum para se construírem oráculos de teste é o monitoramento dos eventos e dos estados que os componentes da interface assumem durante a execução. Por exemplo, Memon et al. [Memon et al. 2000] criaram um modelo formal para descrever uma GUI e o seu comportamento. Esse modelo baseia-se em objetos, suas propriedades e ações que fazem com que elas se alterem. Esse modelo é então usado para criar um oráculo, de tal forma que, dado um caso de teste, o oráculo pode computar o estado esperado para a GUI. Um monitor de execução coleta o estado real dos componentes da interface e compara-o com o estado esperado para decidir se o programa (ou a GUI) se comportou como desejado.

Em geral, os oráculos para programas GUI usam essa abordagem de monitorar, durante a execução, o comportamento dos componentes da interface. Uma execução pode ser monitorada e gravada para ser usada como modelo para execuções posteriores do mesmo caso de teste. Tal abordagem tem suas vantagens como no caso em que a interface gráfica altera-se entre uma execução e outra. Por exemplo, a mudança no posicionamento de um componente não influencia o comportamento do oráculo pois o estado relevante do componente não deve alterar-se.

Por outro lado, essa abordagem tem desvantagens. A mais séria é que ela pode ser somente utilizada para GUIs, não servindo para programas que geram outro tipo de imagens. A segunda desvantagem é que ela depende, para poder monitorar a execução da GUI, do apoio da biblioteca ou *toolkit* usado para desenvolver a interface, fazendo com que seja necessário desenvolverem-se diferentes oráculos para diferentes tipos de interfaces. E terceiro, essa abor-

dagem não é eficiente quando deseja-se avaliar aspectos visuais da interface como forma, tamanho e posicionamento. É o caso, por exemplo, de verificar se uma determinada página da Web apresenta-se no formato esperado para um determinado navegador.

Recentemente, foi proposta a utilização de técnicas de CBIR (Content Based Image Retrival) para construção de oráculos gráficos [Oliveira et al. 2008]. CBIR é definido como qualquer tecnologia que ajude a organizar arquivos digitais de imagens por meio do seu conteúdo visual [Datta et al. 2008]. De forma geral, os sistemas de CBIR são constituídos por programas computacionais que visam a localizar em uma base de imagens aquelas mais similares a uma imagem de consulta, de acordo com um ou mais critérios fornecidos. Os critérios de similaridade são obtidos a partir da extração de características da imagem como cor, textura e forma. Os sistemas automatizados de CBIR envolvem várias áreas da Computação, sendo as principais: Processamento de Imagens e Banco de Dados. Um sistema de CBIR é composto basicamente por três partes: extratores, funções de similaridades e estruturas de indexação.

Os extratores são métodos computacionais que retiram características das imagens a partir de algoritmos que analisam cores, formas, texturas ou outros aspectos relacionados à imagem como um todo ou a parte dela. Por exemplo, definindo-se as cores como uma classe particular de interesse, um extrator específico poderia retornar o valor do contraste de um determinado trecho da imagem, obtido por meio do cálculo da média de cores de um objeto presente neste trecho da imagem, dividida pela média de cores do fundo da imagem. As características extraídas são, em geral, transformadas em um valor que, posteriormente, pode ser comparado com o valor obtido para a mesma característica de outra imagem. Por exemplo, o extrator que calcula o contraste de um determinado trecho da imagem pode retornar o valor zero quando o contraste é nulo ou outro valor no intervalo entre zero e um.

O conjunto de características extraídas de uma imagem forma o seu vetor de características, que é utilizado na sua indexação e recuperação. O conjunto de características em si não é suficiente para determinar o resultado da recuperação. Outro elemento que influencia nos resultados da busca é a escolha de medidas de similaridade entre as imagens. Para aplicar uma consulta por similaridade sobre os vetores de características é necessário usar uma função de distância para o cálculo da similaridade, definindo-se a função de similaridade. Uma função de distância é um algoritmo que compara dois vetores de características e retorna um valor não negativo. Quanto menor o valor retornado, maior é a semelhança entre a imagem modelo e a imagem procurada.

Utilizar CBIR no contexto de oráculos de teste significa desenvolver um sistema que seja capaz de comparar uma imagem de referência com uma imagem produzida pelo programa em teste e estabelecer quão próximo ou distante elas são, de acordo com características visuais e conceitos de similaridade definidos. Para que isso não precise ser feito de forma completa-

mente manual, criando-se “do zero” um oráculo para cada aplicação, definiu-se um *framework* para geração de oráculos gráficos, usando os conceitos de CBIR [Oliveira et al. 2009]. A arquitetura do *framework* O-FIm (*Oracles for Images*) é mostrada na Figura 2.31.

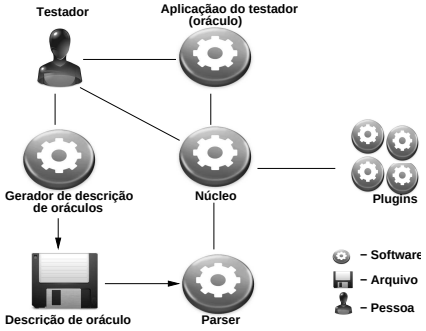


Figura 2.31. Arquitetura do O-FIm

O *framework* é implementado em Java e permite a criação de oráculos também nessa linguagem. O núcleo controla a instanciação do oráculo de teste e permite ao testador instalar novos *plugins* no sistema. Um *plugin* é uma classe Java que implementa a interface de um extrator de característica ou de uma função de similaridade. Para cada novo oráculo que se implemente, o testador deve fornecer os *plugins* que lhe sejam adequados.

Para automatizar o processo de criação de oráculos, o O-FIm provê um *parser* que analisa um arquivo de descrição e instancia automaticamente um oráculo, usando os *plugins* instalados. Assim, com apenas algumas linhas é possível criar-se um oráculo. Um exemplo de descritor de oráculos é mostrado na Figura 2.32. Ainda, o *framework* disponibiliza uma ferramenta no formato de um “*wizard”* que permite ao testador a geração de forma interativa do arquivo de descrição do oráculo.

```
Similarity Euclidiana
Extractor MyExtractor { color = "red" alpha = 78
                        rectangle = [100 100 30 40]
                      }
Extractor OurExtractor { rectangle = [0 0 128 64] scale = 1.33 }
Precision = 0.46
```

Figura 2.32. Um exemplo de descrição de oráculo

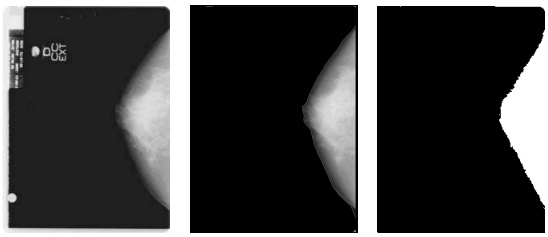
Esses conceitos foram aplicados em um estudo de caso envolvendo parte de um sistema CAD (*Computer Aided Diagnosis*) para a segmentação da região da mama em imagens mamográficas. Um dos problemas para se decidir se o programa “encontrou” a região correspondente à mama na imagem é que isso depende da interpretação do testador ou de algum especialista (radiologista) que deve acompanhar a execução dos testes. Isso pode ser inconveniente por diversas razões mas principalmente porque a disponibilidade de tal profissional nem sempre existe.

Para automatizar esse processo, uma opção é a criação de um conjunto de teste no qual cada saída esperada é dada por uma imagem na qual a região de interesse é marcada manualmente (Figura 2.33). No estudo de caso foram utilizadas 30 imagens de referência, nas quais a região da mama foi marcada manualmente. Depois, construiu-se um oráculo de teste utilizando o ambiente O-FIm com a função de similaridade Euclidiana e três extratores de caracteristicas:

área: conta o número de pixels dentro da região identificada como a mama.

perímetro: conta o número de pixels que foram marcados como parte da borda da mama.

assinatura: calcula um número que identifica o formato do contorno da área da mama em função da sua regularidade. Para isso, executa as seguintes ações: localiza o centro da mama na última coluna da imagem, mede a distância deste ponto até o contorno da mama, considerando intervalos em graus pré-estabelecidos e calcula o desvio-padrão entre as medidas obtidas. O desvio padrão informa o quanto uma borda é irregular. Para um círculo perfeito, por exemplo, este valor é zero.



(a) Imagem original (b) Imagem marcada (c) Imagem processada

Figura 2.33. Exemplos de imagens usadas no estudo de caso.

Utilizando esse oráculo foi possível experimentar seu comportamento com diversos *thresholds* diferentes, ou seja, valores que deveriam ser considerados

como limite para que a distância medida entre a imagem de referência e a imagem segmentada pelo programa fosse considerada excessiva, indicando portanto uma falha na execução do programa.

Mostrou-se assim uma alternativa que consideramos promissora para a automatização do teste de um tipo de aplicação que se enquadra na definição feita por Weyuker [Weyuker 1982]. Nela, as saídas se encontram em um formato complexo, a comparação não pode ser feita diretamente e deve existir um certo grau de flexibilidade ao se considerar o resultado de uma execução como correto ou incorreto.

2.8. Depuração

A atividade de depuração é tipicamente entendida como um corolário do teste bem sucedido [Agrawal et al. 1995], isto é, aquele que detecta a presença de defeitos em um software. Por isso, a informação gerada durante o teste é essencial para o sucesso da depuração. Ela é utilizada de maneira simplificada por técnicas tradicionais que partem da informação de que um dado caso de teste falha e de maneira sofisticada por técnicas que utilizam informação de cobertura de teste para identificar trechos do software suspeitos de conter o defeito.

A seguir, descrevemos as principais técnicas de depuração enfatizando o tipo de informação de teste utilizada. A discussão apresentada é baseado no capítulo sobre depuração de Chaim et al. [Chaim et al. 2007] e será focada nas técnicas de localização de defeitos.

2.8.1. Depuração baseada em rastreamento e inspeção

A depuração baseada em rastreamento e inspeção é a mais usada na prática. O sucesso dessa técnica deve-se a três fatores. Seu uso requer apenas treinamento básico, o custo em tempo de depuração é razoável e a sua disponibilidade é ampla (presente em qualquer ambiente de programação). Este tipo de depuração envolve o rastreamento de eventos e a inspeção do estado do programa no momento em que ocorre um evento.

Na sua forma mais básica, a depuração baseada em rastreamento e inspeção é realizada com a ajuda de comandos de escrita. O programador coloca em pontos estratégicos do programa comandos para imprimir os valores de determinadas variáveis. O objetivo é rastrear um determinado ponto do programa durante a execução (evento) e inspecionar o valor das variáveis escolhidas (estado parcial do programa). Com os depuradores simbólicos [Adams e Muchnick 1986], o rastreamento de alguns tipos de eventos e a inspeção do estado do programa ficaram extremamente facilitados. Por exemplo, usando os comandos de um depurador simbólico, o programador pode parar a execução do programa no ponto desejado ("*breakpoint*"), acessar os valores das variáveis, verificar a seqüência de chamadas de procedimentos ("*call stack*"), atribuir novos valores para variáveis etc.

Uma característica interessante de ser incluída nos depuradores simbólicos é a execução em reverso, isto é, permitir que o programador execute o

programa em sentido contrário. Há uma longa série de iniciativas de inclusão dessa característica em ferramentas experimentais [Balzer 1969]. Entretanto, não há depuradores simbólicos disponíveis comercialmente com essa função. O fator limitante é que, mesmo nas implementações mais eficientes, é ainda exigida uma grande quantidade de memória para registrar as alterações tanto no estado do programa como nos arquivos de entrada e saída. Estes dois requisitos dificultam a sua utilização em programas reais. Algoritmos eficientes para execução bidirecional na depuração de programas foram propostos [Boothe 2000]. Porém, eles ainda precisam ser avaliados quanto à sua escalabilidade. As ferramentas de rastreamento e inspeção utilizam informação básica de teste: se um caso de teste falhou ou não.

2.8.2. Fatiamento de programas

Fatiamento ("*slicing*") de programas [Korel e Laski 1988, Weiser 1984] é uma técnica cujo objetivo é selecionar fatias ("*slices*") do programa. Uma fatia é um conjunto de comandos que afetam os valores de uma ou mais variáveis em um determinado ponto do programa. As variáveis e o ponto do programa definem o critério de fatiamento. A fatia pode ser determinada estaticamente ou dinamicamente. No primeiro caso, os comandos selecionados podem afetar as variáveis no ponto especificado para alguma possível entrada do programa. No segundo caso, os comandos selecionados efetivamente afetam os valores das variáveis no ponto especificado para uma determinada entrada.

Esta técnica, porém, possui alguns problemas. O primeiro é o tamanho das fatias, tanto estáticas (especialmente) como dinâmicas. Para programas grandes, o número de comandos que afetam um critério de fatiamento pode também ser grande, o que torna a técnica pouco atrativa. O segundo problema da técnica é o seu custo. O fatiamento dinâmico de programas requer o rastreamento dos comandos e das posições de memória de maneira a identificar precisamente os comandos que afetam um critério de fatiamento. Este requisito impõe um custo muito grande em tempo de depuração para programas que possuem longas execuções [Nishimatsu et al. 1999].

Novas técnicas de fatiamento dinâmico [Zhang e Gupta 2004] conseguiram obter fatias de programas de longa duração a custo bastante reduzido. No entanto, elas realizam um pré-processamento dos dados coletados durante a execução (comandos executados e posições de memória acessadas) que pode demorar dezenas de minutos. Uma vez realizado esse pré-processamento, diferentes fatias de uma mesma execução podem ser obtidas em segundos.

Com o objetivo de reduzir o custo de obtenção das fatias, Agrawal et al. [Agrawal et al. 1995] propõem a determinação de uma fatia do programa a partir dos resultados obtidos do teste estrutural do programa. O conjunto de comandos associados aos requisitos de teste estruturais (e.g., nós, ramos, c-usos e p-usos) executados por um caso de teste particular é chamado de fatia de execução.

A vantagem das fatias baseadas em informação de teste é que a maior parte do custo para sua obtenção já foi amortizado durante o teste do programa. Entretanto, também nesse caso, há uma grande probabilidade de essas fatias incluírem um número elevado de comandos.

Para reduzir o espaço de busca para localização do defeito, heurísticas utilizando fatias têm sido propostas [Agrawal et al. 1995, Collofello e Cousins 1987]. A idéia é realizar operações com as fatias para determinar um conjunto menor de comandos com grande probabilidade de conter o defeito. As heurísticas mais simples realizam operações de intersecção e união de fatias.

As heurísticas mais sofisticadas estabelecem um *ranking* com os comandos do programa executados pelos casos de teste. Nesse *ranking*, os comandos que ocorrem mais frequentemente em fatias de casos de teste que manifestam falhas são mais bem classificados do que aqueles que não ocorrem tão frequentemente. Essa idéia foi inicialmente proposta por Collofello e Cousins [Collofello e Cousins 1987] para evitar que um defeito localizado em um trecho do código executado tanto por casos de teste que falham como que não falham fosse eliminado na operação de subtração da técnica de recorte. Essa mesma ideia foi retomada posteriormente por Jones e colegas [Jones et al. 2002].

O uso de heurísticas impõe alguns riscos. Apesar de a intuição ser de que há grande probabilidade do trecho de código selecionado conter o defeito, há também a chance de que ele seja excluído durante as operações envolvendo conjuntos (e.g., intersecção, subtração) ou durante a criação do *ranking*. Nesse último caso, o trecho selecionado pode incluir os efeitos do defeito, mas excluir o próprio. Além disso, as heurísticas que operam sobre fatias possuem as mesmas restrições inerentes à técnica utilizada para obtê-las.

A informação de teste utilizada são os dados de cobertura de teste estrutural, quando são utilizados as fatias de execução. Os demais tipos de fatias utilizam somente a informação se um teste passa ou falha.

Depuração algorítmica

A técnica de depuração algorítmica foi originalmente desenvolvida para programas sem efeitos colaterais escritos em Prolog [Shapiro 1983]. Esta técnica é baseada em um processo interativo durante o qual o programador fornece conhecimento a respeito do comportamento esperado do programa ao sistema de depuração e este, por sua vez, guia o programador durante o processo de localização do defeito [Fritzson et al. 1992].

A técnica funciona da seguinte maneira. Para localização do defeito, o caso de teste que manifestou uma falha deve ser executado sob a supervisão do sistema baseado em depuração algorítmica. O sistema cria, então, uma árvore de execução na qual os nós representam invocações dos procedimentos do programa. Esses nós contêm o nome do procedimento e os valores de

entrada e saída utilizados na invocação. O sistema, então, visita, partindo do procedimento de mais alto nível, os nós da árvore de execução perguntando ao programador se os valores dos parâmetros de entrada e saída estão corretos. Se sim, o processo continua visitando os próximos nós de mesmo nível. Caso contrário, o processo visita os nós dos procedimentos de nível inferior invocados pelo procedimento de nível superior. Esse processo termina quando é identificado um procedimento no qual os parâmetros de entrada estão corretos e os de saída incorretos ou que não faz chamada a nenhum outro procedimento ou, se o faz, os valores dos parâmetros de entrada e saída dos procedimentos invocados estão corretos.

Caso o programador responda corretamente às questões colocadas pelo sistema de depuração algorítmica, o procedimento onde está o defeito é identificado. Entretanto, a aplicabilidade dessa técnica em programas reais é reduzida. Entre as dificuldades para sua utilização estão: o número de perguntas que o programador deve responder, o tratamento de efeitos colaterais, especialmente os causados pelo uso de ponteiros e o espaço (não-limitado) requerido pela árvore de execução cujo tamanho depende do número de invocações dos procedimentos.

A definição da técnica não prevê o uso de informação detalhada de teste mas ela pode ser útil à depuração algorítmica. Há implementações [Fritzson et al. 1992] da técnica que verificam em uma base de dados se os valores dos parâmetros de entrada e saída já foram executados por algum caso de teste que não falhou ou se pertencem a uma categoria que contém apenas este tipo de caso de teste. Este tipo de informação pode auxiliar na redução do número de perguntas. Outras técnicas [Silva e Chitil 2006] utilizam fatiamento para reduzir o número de perguntas a serem respondidas.

Depuração delta

A depuração delta foi criada inicialmente por Zeller [Zeller 2002] com o objetivo de minimizar a entrada de casos de teste que provocam a ocorrência de falhas. Um problema típico de utilização da depuração delta é determinar qual dos comandos provoca a falha de carregamento de uma página HTML. A depuração delta soluciona essa questão fazendo o teste de carregamento de páginas que são resultantes da diferença entre a página que provocou a falha e uma página que não provocou (no caso, uma página vazia).

Considere-se uma página que contenha metade do texto da página que provoca a falha. Se a falha continua a ocorrer, então o trecho que induz a falha está na metade incluída. Se, por outro lado, a falha não ocorre mais, então o trecho omitido é o que contém a falha. O caminho inverso pode ser também percorrido partindo da página que não provoca a falha e incluindo trechos da página errônea. Fazendo esse processo repetidamente pode-se chegar ao comando HTML que provoca a ocorrência da falha.

O algoritmo da depuração delta é inspirada na busca binária, porém, tratando situações em que a página carregada produz um resultado indeterminado. A depuração delta pode ser utilizada para identificar outras circunstâncias causadoras de falhas. Por exemplo, considere-se que um programa teve 10.000 linhas alteradas e uma falha ocorre quando executado por um determinado caso de teste que antes das modificações não falhava. Qual dessas 10.000 linhas causa a falha? Fazendo a execução repetida de versões do programa com diferentes conjuntos de linhas modificadas e utilizando o algoritmo de depuração delta pode-se identificar as linhas que dão origem à falha.

Cleve e Zeller [Cleve e Zeller 2005] propuseram a utilização da depuração delta para identificação de trechos do programa que causam da ocorrência da falha. Isso é realizado comparando-se o estado do programa em execuções que manifestam a falha e execuções que não manifestam. São feitas alterações no estado do programa utilizando o algoritmo de depuração delta de forma a identificar os comandos que causam a manifestação da falha. Os resultados obtidos são promissores, porém, o custo para utilizar esta técnica em programas com longas execuções ainda é uma questão em aberto.

Referências bibliográficas

[Adams e Muchnick 1986] Adams, E. e Muchnick, S. S. (1986). Dbxtool: A window-based symbolic debugger for sun workstation. *Software Practice and Experience*, 16(7):653–669.

[Agrawal et al. 1995] Agrawal, H., Horgan, J. R., London, S. e Wong, W. E. (1995). Fault localization using execution slices and dataflow tests. In *Proceedings 6th International Symposium on Software Reliability Engineering (ISSRE 1995)*, pp. 143–151, Toulouse, France. Los Alamitos, CA. IEEE Computer Society.

[Andrea 2008] Andrea, J. (2008). Envisioning the next generation of functional testing tools. *IEEE Software*, 23:58–65.

[Balzer 1969] Balzer, R. M. (1969). Exdams: Extensible debugging and monitoring system. In *Spring Joint Computer Conference*, pp. 567–589, Reston, VA, U.S.A. AFIPS Press.

[Baresi e Young 2001] Baresi, L. e Young, M. (2001). Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A.

[Beck 2002] Beck, K. (2002). *Test-Driven Development By Example*. Addison Wesley, New York, 1a. edição.

[Boothe 2000] Boothe, B. (2000). Efficient algorithms for bidirectional debugging. In *Proceedings 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 299–310, Vancouver, Britith Columbia, Canada. New York, NY. ACM. SIGPLAN Notices 35(5) (May 2000).

[Chaim et al. 2007] Chaim, M. L., Jino, M. e Maldonado, J. C. (2007). *Introdução ao Teste de Software*, capítulo Depuração, pp. 293–314. Editora Campus.

[Chu et al. 1997] Chu, H.-D., Dobson, J. E. e Liu, I.-C. (1997). Fast: a framework for automating statistics-based testing. *Software Quality Control*, 6(1):13–36.

[Cleve e Zeller 2005] Cleve, H. e Zeller, A. (2005). Locating causes of program failures. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*.

[Collofello e Cousins 1987] Collofello, J. S. e Cousins, L. (1987). Toward automatic software fault localization through decision-to-decision path analysis. In *Proceedings of the AFIP 1987 National Computer Conference*, pp. 539–544, Chicago.

[Crispin 2002] Crispin, L. (2002). The need for speed: Automating acceptance testing in an extreme programming environment. *UpGrade - The European Journal for the Informatics Professional*, 3:104–106.

[Crispin 2006] Crispin, L. (2006). Driving software quality: How test-driven development impacts software quality. *IEEE Software*, 23(6):70–71.

[Datta et al. 2008] Datta, R., Joshi, D., Li, J. e Wang, J. Z. (2008). Image retrieval: Ideas, influences, and trends of the new age. *ACM Comput. Surv.*, 40(2):1–60.

[Delamaro et al. 2007] Delamaro, M. E., Maldonado, J. C. e Jino, M. (2007). *Introdução ao teste de software*, capítulo Conceitos básicos. Editora Campus.

[Delamaro et al. 1993] Delamaro, M. E., Maldonado, J. C., Jino, M. e Chaim, M. L. (1993). Proteum: Uma ferramenta de teste baseada na análise de mutantes. In *Caderno de Ferramentas do VII Simpósio Brasileiro de Engenharia de Software*, pp. 31–33, Rio de Janeiro, RJ, Brasil.

[DeMillo et al. 1978] DeMillo, R. A., Lipton, R. J. e Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41.

[Dijkstra 1970] Dijkstra, E. W. (1970). Notes on structured programming. Relatório Técnico 70-WSK-03, Technological University of Eindhoven, The Netherlands.

[Fabbri et al. 2007] Fabbri, S. C. P., Vincenzi, A. M. R., Barbosa, E. F. e Maldonado, J. C. (2007). *Introdução ao teste de software*, capítulo Teste funcional. Editora Campus.

[Fritzson et al. 1992] Fritzson, P., Shahmehri, N., Kamkar, M. e Gyimothy, T. (1992). Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and Systems*, 1(4):303–322.

[Gotlieb et al. 1998] Gotlieb, A., Botella, B. e Rueher, M. (1998). Automatic test data generation using constraint solving techniques. *SIGSOFT Softw. Eng. Notes*, 23(2):53–62.

[Hajnal e Forgács 1998] Hajnal, A. e Forgács, I. (1998). An applicable test data generation algorithm for domain errors. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 63–72, New York, NY, USA. ACM.

[Hendricksons 2008] Hendricksons, E. (2008). Acceptance test driven development (ATDD): an overview. In *Software Testing Australia/New Zealand , 7.*, Wellington. STANZ.

[Jones et al. 2002] Jones, J. A., Harrold, M. J. e Stasko, J. (2002). Visualization of test information to assist fault localization. In *Proceedings of the 24th ACM/IEEE International Conference on Software Engineering*.

[Korel 1990] Korel, B. (1990). Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879.

[Korel 1996] Korel, B. (1996). Automated test data generation for programs with procedures. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 209–215, New York, NY, USA. ACM.

[Korel e Laski 1988] Korel, B. e Laski, J. (1988). Dynamic program slicing. *Information Processing Letters*, 29(3):155–163.

[Laprévote et al. 2009] Laprévote, A., Vincenzi, A. M. R., Chaim, M. L., Silva, M. A. G., Tosi, D., Hui.wu e Krysztofiak, T. (2009). Test suites and benchmarks for the chosen set of open source projects and artifacts. methodology for creating test suites and benchmarks for arbitrary systems – version 1. Working Document, QualiPSo Project — Quality Platform for Open Source Software.

[Mahmood 2007] Mahmood, S. (2007). A systematic review of automated test data generation techniques. Dissertação de mestrado, School of Engineering – Blekinge Institute of Technology, Ronneby, Sweden.

[McMinn 2004] McMinn, P. (2004). Search-based software test data generation: a survey. *STVR – Software Testing, Verification and Reliability*, 14(2):105–156.

[Memon et al. 2000] Memon, A. M., Pollack, M. E. e Soffa, M. L. (2000). Automated test oracles for guis. *SIGSOFT Softw. Eng. Notes*, 25(6):30–39.

[Meszaros 2007] Meszaros, G. (2007). *XUnit Test Patterns — Refactoring Test Code*. Addison-Wesley, Boston, 1a. edição.

[Meudec 2001] Meudec, C. (2001). Atgen: automatic test data generation using constraint logic programming and symbolic execution. *STVR – Software Testing, Verification and Reliability*, 11(2):81–96.

[Michael e McGraw 1998] Michael, C. e McGraw, G. (1998). Automated software test data generation for complex programs. In *ASE '98: Proceedings of the 13th IEEE international conference on Automated software engineering*, p. 136, Washington, DC, USA. IEEE Computer Society.

[Nguyen 2001] Nguyen, H. Q. (2001). *Testing Applications on the Web: Testing Planning for Internet-Based Systems*. John Wiley and Sons, Inc., New York, 1a. edição.

[Nishimatsu et al. 1999] Nishimatsu, A., Jihira, M., Kusumoto, S. e Inoue, K. (1999). Call-Mark Slicing: A efficient and economical way of reducing slice. In *Proceedings of the 21st International Conference on Software Engineering*, pp. 422–431, Los Angeles, CA, U.S.A. IEEE Computer Society Press.

[Offutt e Craft 1994] Offutt, A. J. e Craft, W. M. (1994). Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4(3):131–154.

[Oliveira et al. 2008] Oliveira, R. A. P., Delamaro, M. E. e Nunes, F. L. S. (2008). Estrutura para utilização de Recuperação de Imagem Baseada em Conteúdo em oráculos de teste de software com saída gráfica. In *Workshop de Visão Computacional (WVC 2008)*, Bauru – SP.

[Oliveira et al. 2009] Oliveira, R. A. P., Delamaro, M. E. e Nunes, F. L. S. (2009). O-FIm - Oracle for Images. In *Sessão de Ferramentas 2009 - XVI Sessão de Ferramentas - SBES (Simpósio Brasileiro de Engenharia de Software)*, pp. 1 – 6, Fortaleza – CE – Brasil.

[Pargas et al. 1999] Pargas, R. P., Harrold, M. J. e Peck, R. (1999). Test-data generation using genetic algorithms. *STVR – Software Testing, Verification and Reliability*, 9(4):263–282.

[Rapps e Weyuker 1982] Rapps, S. e Weyuker, E. J. (1982). Data flow analysis techniques for test data selection. In *VI International Conference on Software Engineering*, pp. 272–278, Tokio, Japan. IEEE Computer Society Press.

[Scott e Neil 2009] Scott, B. e Neil, T. (2009). *Designing Web Interfaces: Principles and Patterns for Rich Interactions*. O'Reilly Media, Sebastopol, CA, 1a. edição.

[Shapiro 1983] Shapiro, E. Y. (1983). *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachussetts.

[Silva e Chitil 2006] Silva, J. e Chitil, O. (2006). Combining algorithmic debugging and program slicing. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pp. 157–166, New York, NY, USA. ACM.

[Sommerville 2007] Sommerville, I. (2007). *Engenharia de Software*. Addison Wesley, São Paulo, SP, 8a. edição.

[Tidwell 2005] Tidwell, J. (2005). *Designing Interfaces: Patterns for Effective*

Interaction Design. O'Reilly Media, Sebastopol, CA, 1a edição.

[Tracey et al. 1998] Tracey, N., Clark, J. e Mander, K. (1998). Automated program flaw finding using simulated annealing. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 73–81, New York, NY, USA. ACM.

[Weiser 1984] Weiser, M. (1984). Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357.

[Weyuker 1982] Weyuker, E. J. (1982). On testing non-testable programs. *The Computer Journal*, 25(4).

[Zeller 2002] Zeller, A. (2002). Isolating cause-effect chains from computer programs. In *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pp. 1–10.

[Zhang e Gupta 2004] Zhang, X. e Gupta, R. (2004). Cost effective dynamic program slicing. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pp. 94–106, New York, NY, USA. ACM Press.