TÉCNICAS DE TESTE DE SOFTWARE

Teste de software é um elemento crítico da garantia de qualidade de software e representa a revisão final da especificação, projeto e geração de código.

A crescente visibilidade do software como um elemento do sistema e os "custos" de atendimento associados com uma falha são forças motivadoras para o teste rigoroso e bem planejado. Não é raro uma organização de desenvolvimento de software gastar entre 30 e 40% do esforço total do projeto no teste. A rigor, o teste de software que envolve vidas (p. ex., controle de vôo, monitoramento de reatores nucleares) pode custar de três a cinco vezes mais do que todos os outros passos de engenharia de software combinados!

O que é?

Uma vez gerado o código fonte, o software deve ser testado para descobrir (e corrigir) tantos erros quanto possível antes de ser entregue ao seu cliente. Sua meta é projetar uma série de casos de teste que têm uma grande probabilidade de encontrar erros - mas como? É aí que as técnicas de teste de software entram em cena. Essas técnicas fornecem diretrizes sistemáticas para projetar testes que (1) exercitam a lógica interna dos componentes de software e (2) exercitam os domínios de entrada e saída do programa para descobrir erros na função, comportamento e desempenho do programa.

Quem faz?

Durante os primeiros estágios de teste um engenheiro de software realiza todos os testes. No entanto, à medida que o processo de teste progride, especialistas podem ser envolvidos.

Por que é importante?

Revisões e outras atividades de SQA podem e efetivamente descobrem erros, mas elas não são suficientes. Toda vez que o programa é executado, o cliente o testa! Assim, você tem que executar um programa antes que ele chegue ao cliente com o objetivo específico de encontrar e remover todos os erros. Para encontrar o maior número possível de erros, testes devem ser conduzidos sistematicamente e casos de teste devem ser projetados usando técnicas disciplinadas.

Quais são os passos?

O software é testado de duas perspectivas diferentes: (1) a lógica interna do programa é exercitada usando técnicas de projeto de casos de teste (de "caixa-branca"). Requisitos de software são exercitados usando técnicas de projeto de casos de teste "caixa-preta". Em ambos os casos, o objetivo é encontrar o maior número de erros com a menor quantidade de esforço e tempo.

Qual é o produto do trabalho? Um conjunto de casos de teste planejado para exercitar tanto a lógica interna quanto os requisitos externos é projetado e documentado, os resultados esperados são definidos e os resultados reais são registrados.

Como garanto que fiz corretamente?

Modifique seu ponto de vista, ao começar o teste tente "quebrar" arduamente o software! Projete casos de teste de um modo disciplinado e revise os casos de teste que você cria, quanto ao rigor.

1. FUNDAMENTOS DO TESTE DE SOFTWARE

O teste expõe uma anomalia interessante para o engenheiro de software. Durante as primeiras atividades de engenharia de software, o engenheiro tenta construir um software a partir de um conceito abstrato até um produto tangível. Depois vem o teste. O engenheiro cria uma série de casos de testes, que são destinados a "demolir" o software que foi construído. De fato, o teste é um passo do processo de software que poderia ser visto (pelo menos psicologicamente) como destrutivo em vez de construtivo.

Engenheiros de software são por natureza pessoas construtivas. O teste exige que o desenvolvedor reveja noções preconcebidas da "corretividade" do software recém desenvolvido e supere um conflito de interesses que ocorre quando erros são descobertos.

1.1 Objetivos do teste

Num excelente livro sobre teste de software, Glen Myers enumera algumas regras que podem servir bem como objetivos do teste:

1. Teste é um processo de execução de um programa com a finalidade de encontrar um erro.

2. Um bom caso de teste é aquele que tem alta probabilidade de encontrar um erro ainda não descoberto.

3. Um teste bem-sucedido é aquele que descobre um erro ainda não descoberto.

Esses objetivos implicam uma dramática mudança de ponto de vista. Eles vão contra a visão comum de que um teste bem-sucedido é aquele no qual não são encontrados erros. Nosso objetivo é projetar testes que descobrem sistematicamente diferentes classes de erros e fazê-lo com uma quantidade mínima de tempo ou esforço.

Se o teste for conduzido de maneira bem-sucedida (de acordo com os objetivo declarados anteriormente), ele descobrirá erros no software. Como benefício secundário, o teste demonstra que as funções do software parecem estar funcionando do acordo com a especificação de que os requisitos de comportamento e desempenho parecem estar sendo satisfeitos. Além disso, os dados coletados à medida que o teste é conduzido fornecem uma boa indicação da confiabilidade do software e alguma indicação da qualidade de todo o software.

1.2 Princípios do teste

Antes de aplicar métodos para projetar casos de teste efetivos, um engenheiro de software deve entender os princípios básicos que guiam o teste de software. Davi sugere um conjunto de princípios de teste:

- **Todos os testes devem ser relacionados aos requisitos do cliente**. Como já vimos, o objetivo do teste de software é descobrir erros. Segue-se que os defeitos mais indesejáveis (do ponto de vista do cliente) são aqueles que levam o programa a deixar de satisfazer seus requisitos.
- **Os testes devem ser planejados muito antes do início do teste**. O planejamento de teste pode começar tão logo o modelo de requisitos seja completado. A definição detalhada dos casos de teste pode começar tão logo o modelo de projeto tenha sido consolidado. Assim sendo, todos os testes podem ser planejados e projetados antes que qualquer código tenha sido gerado.
- **O princípio de Pareto se aplica ao teste de software**. Colocado simplesmente o princípio de Pareto implica que 80% de todos os erros descobertos durante teste vão provavelmente ser relacionados a 20% de todos os componentes do programa. O problema, sem dúvida, é isolar os componentes suspeitos e testá-los rigorosamente.
- **O teste deve começar "no varejo" e progredir até o teste "no atacado"**. Os primeiros testes planejados e executados geralmente concentram-se nos componentes individuais. À medida que o teste progride, o foco se desloca numa tentativa de encontrar erros em conjuntos integrados de componentes e finalmente em todo o sistema.
- **Teste completo não é possível**. A quantidade de permutações de caminho mesmo para um programa de tamanho moderado, é excepcionalmente grande. Por essa razão, é impossível executar todas as combinações de caminhos durante o teste. É possível, no entanto, cobrir adequadamente a lógica do programa e garantir que todas as condições no projeto, a nível de componente, tenham sido exercitadas.
- **Para ser mais efetivo, o teste deveria ser conduzido por terceiros**. Por mais efetivo, nós queremos dizer teste que tem a maior probabilidade de encontrar erros (o principal objetivo do teste). Por motivos explicitados anteriormente, o engenheiro de software que criou o sistema não é a pessoa adequada para conduzir todos os testes do software.

1.3 Testabilidade

Em circunstâncias ideais, um engenheiro de software projeta um programa de computador, um sistema ou um produto com "testabilidade" em mente. Isso permite aos indivíduos encarregados do teste projetar casos de teste efetivos mais facilmente. Mas o que é testabilidade? James Bach descreve testabilidade da seguinte maneira:

A testabilidade de software é simplesmente a facilidade com que ele [um programa de computador] pode ser testado. Como o teste é profundamente difícil, vale a pena saber o que pode ser feito para facilitá-lo. Algumas vezes os programadores querem fazer coisas que ajudem o processo de teste, e uma lista dos possíveis pontos de projeto, características, etc. pode ser útil para negociar com eles.

Há certas métricas que podem ser usadas para medir a testabilidade na maior parte dos seus aspectos. Algumas vezes, a testabilidade é usada para dizer quão adequadamente um conjunto particular de testes vai cobrir o produto. Também é usada pelos militares para dizer quão facilmente uma ferramenta pode ser verificada e reparada no campo. Esses dois significados não são os mesmos da testabilidade de software. A lista adiante fornece um conjunto de características que levam a um software testável.

Operabilidade. "Quanto melhor funciona, mais eficientemente pode ser testado."

- O sistema tem poucos defeitos (bugs),
- Nenhum defeito bloqueia a execução dos testes.
- O produto evolui em estágios funcionais (permite desenvolvimento e testes simultâneos).

Observabilidade. "O que você vê é o que você testa."

- Saída distinta é gerada para cada entrada.
- Estados e variáveis do sistema são visíveis ou consultáveis durante a execução.
- Os estados e variáveis anteriores do sistema são visíveis ou consultáveis (p. ex.registro de transações).
- Todos os fatores que afetam a saída são visíveis.
- Saída incorreta é facilmente identificada.
- Erros internos são automaticamente detectados através de mecanismos de a autoteste.
- Erros internos são automaticamente relatados.
- O código-fonte é acessível.

Controlabilidade. "Quanto mais você pode controlar o software, mais o teste pode ser automatizado e otimizado."

- Todas as saídas possíveis podem ser geradas por alguma combinação de entradas
- Todo o código é executável através de alguma combinação de entradas.
- Estados e variáveis do software e do hardware podem ser controlados diretamente pelo engenheiro de teste.
- Formatos de entrada e saída são consistentes e estruturados.
- Testes podem ser especificados, automatizados e reproduzidos convenientemente.

Decomponibilidade. "Controlando o escopo do teste, podemos isolar problemas mais rapidamente e realizar retestagem mais racionalmente."

- O sistema de software é construído a partir de módulos independentes.
- Módulos de software podem ser testados independentemente.

Simplicidade. "Quanto menos há a testar, mais rapidamente podemos testá-lo."

- Simplicidade funcional (p. ex., o conjunto de características é o mínimo necessário para satisfazer os requisitos).
- Simplicidade estrutural (p. ex., a arquitetura é modularizada para limitar a propagação de

defeitos).

- Simplicidade do código (p. ex., um padrão de código é adotado para facilita inspeção e a manutenção).

Estabilidade. "Quanto menos modificações, menos interrupções no teste."

- Modificações no software não são freqüentes.
- Modificações no software são controladas.
- Modificações no software não invalidam os testes existentes.
- O software se recupera bem das falhas.

Compreensibilidade. "Quanto mais informações temos, mais racionalmente vamos testar."

- O projeto é bem compreendido.
- As dependências entre componentes internos, externos e compartilhados bem compreendidas.
- Modificações no projeto são informadas.
- A documentação técnica é acessível instantaneamente.
- A documentação técnica é bem organizada.
- A documentação técnica é específica e detalhada.
- A documentação técnica é precisa.

Os atributos sugeridos por Bach podem ser usados por um engenheiro de software para desenvolver uma configuração de software (i. e., programas, dados e documentos) que é fácil de testar.

E sobre os testes propriamente ditos? Kaner, Falk e Nguyen sugerem os seguintes atributos para um "bom" teste:

1. Um bom teste tem uma alta probabilidade de encontrar um erro. Para alcançar essa meta, o testador deve entender o software e tentar desenvolver uma imagem mental de como o software pode falhar. O ideal é que as classes de falhas sejam investigadas. Por exemplo, uma classe de falhas em potencial, numa interface gráfica com o usuário (graphical user interface, GUI), é uma falha em reconhecer a posição correta do mouse. Um conjunto de testes seria projetado para exercitar o mouse numa tentativa de demonstrar erro no reconhecimento da sua posição.

2. Um bom teste não é redundante. O tempo e recursos para testes são limitados. Não tem sentido conduzir um teste que tenha a mesma finalidade de outro. Cada teste deve ter uma finalidade diferente (mesmo que sutil).

3. Um bom teste deve ser "de boa cepa". Em um grupo de testes que têm um objetivo semelhante, as limitações de tempo e recursos podem ser abrandadas no sentido da execução de apenas um subconjunto desses testes. Em tais casos, o teste que tenha maior probabilidade de revelar toda uma classe de erros deve ser usado.

4. Um bom teste não deve ser muito simples nem muito complexo. Apesar de ser possível combinar algumas vezes uma série de testes num caso de teste, os efeitos colaterais possíveis, associados com essa abordagem, podem mascarar erros. Em geral, cada teste deve ser executado separadamente.

2. PROJETO DE CASOS DE TESTE

O projeto de testes para software e outros produtos que passam por engenharia pode ser tão desafiante quanto o projeto inicial do produto propriamente dito. No entanto, por motivos já discutidos, os engenheiros de software freqüentemente tratam o teste como algo que se lembraram depois, desenvolvendo casos de testes que podem "parecer bons" mas têm pouca garantia de ser completos. Recordando os objetivos do teste, devemos projetar testes que tenham mais probabilidade de encontrar o maior número de erros com uma quantidade mínima de tempo e esforço.

Uma rica variedade de métodos de projeto de casos de teste tem sido desenvolvida para o software. Esses métodos fornecem ao desenvolvedor uma abordagem sistemática ao teste. Mais importante, os métodos fornecem um mecanismo que pode ajudar a totalidade dos testes e fornecem uma probabilidade mais alta para descobrir erro no software.

Qualquer produto que passa por engenharia (e muitas outras coisas) pode se testado por uma das duas maneiras: (1) sabendo a função especificada que o produto foi projetado para realizar, podem ser realizados testes que demonstram que cada função está plenamente operacional, enquanto ao mesmo tempo procuram erros em cada função; (2) sabendo como é o trabalho interno de um produto, podem se realizados testes para garantir que "todas as engrenagens combinam", isto é, que as operações internas são realizadas de acordo com as especificações e que todos o componentes internos foram adequadamente exercitados. A primeira abordagem de teste é chamada teste caixa-preta e a segunda, teste caixa-branca.

Quando o software para computador é considerado, teste caixa-preta refere-se a testes que são conduzidos na interface do software. Apesar de serem projetados par descobrir erros, testes caixa-preta são usados para demonstrar que as funções do software estão operacionais, que a entrada é adequadamente aceita e a saída corretamente produzida, e que a integridade da informação externa (p. ex., uma base de dados) é mantida. Um teste caixa-preta examina algum aspecto fundamental do sistema, se preocupando pouco com a estrutura lógica interna do software.

Um teste caixa-branca de software é baseado num exame rigoroso do detalhe procedimental. Caminhos lógicos internos ao software são testados, definindo caso de testes que exercitam conjuntos específicos de condições e/ou ciclos. O "estado do programa" pode ser examinado em vários pontos para determinar se o estado esperado ou enunciado corresponde ao estado real.

À primeira vista poderia parecer que um teste caixa-branca bastante rigoroso levaria a "programas 100% corretos". Tudo que precisaríamos seria definir todos os caminhos lógicos, desenvolver casos de teste para exercitá-los e avaliar os resultados, isto é, gerar casos de teste para exercitar a lógica do programa exaustivamente. Infelizmente um teste completo apresenta certos problemas logísticos. Mesmo para pequenos programas, o número de possíveis caminhos lógicos pode ser muito grande. Por exemplo, considere um programa de 100 linhas em linguagem C. Depois de algumas declarações básicas de dados, o programa contém dois ciclos aninhados, que executam de 1 a 20 vezes cada um, dependendo das condições especificadas na entrada. Dentro do ciclo interior, quatro construções se-então-senão são necessárias. Há aproximadamente 10^{14} caminhos possíveis que podem ser executados nesse programa!

Um teste caixa-branca não deve, no entanto, ser descartado como não-prático. Um número limitado de caminhos lógicos importantes pode ser selecionado e exercitado. Estruturas de dados importantes podem ser submetidas a prova quanto à validade. Os atributos, tanto dos testes caixa-branca quanto dos caixa-preta, podem ser combinados para obter uma abordagem que valida a interface do software e garantir seletivamente que o funcionamento interno do software está correto.

3. TESTE CAIXA-BRANCA

O teste caixa-branca, algumas vezes chamado teste caixa de vidro, é um método de projeto de casos de teste que usa a estrutura de controle do projeto procedimental para derivar casos de teste. Usando métodos de teste caixa-branca, o engenheiro de software pode derivar casos de teste que (1) garantam que todos os caminhos independentes de um módulo tenham sido exercitados pelo menos uma vez, (2) exercitam todas as decisões lógicas em seus lados verdadeiro e falso, (3) executam todos os ciclos nos seus limites e dentro de seus intervalos operacionais, e (4) exercitam as estruturas de dados internas para garantir sua validade.

Uma pergunta razoável pode ser feita neste ponto: "Por que gastar tempo e energia preocupando-se com (e testando) minúcias lógicas, quando poderíamos empregar melhor o esforço garantindo que os requisitos do programa foram satisfeitos?" Dito de outro modo, por que não gastamos toda nossa energia em testes caixa-preta? A resposta está na natureza dos defeitos de software:

- Erros lógicos e pressupostos incorretos são inversamente proporcionais à probabilidade de que um caminho de programa vai ser executado. Os erros tendem a penetrar no nosso trabalho quando projetamos e implementamos funções, condições ou controle que estão fora da função principal. Um processamento cotidiano tende a ser bem entendido (e bem examinado), enquanto um processamento "de casos especiais" tende a cair pelas frestas.

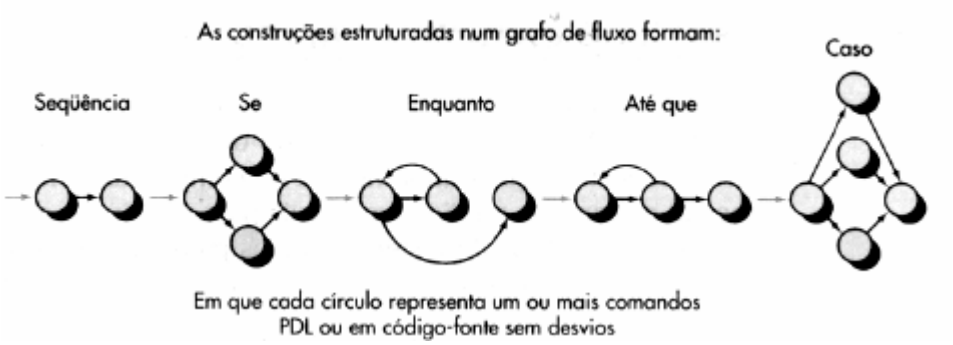
- Freqüentemente acreditamos que um caminho lógico não é provável de ser executado quando, na realidade, ele pode ser executado em base regular. O fluxo lógico de um programa é algumas vezes contra-intuitivo, significando que nossos pressupostos inconscientes sobre o fluxo de controle e dados podem nos levar a cometer erros de projeto que são descobertos apenas quando o teste de caminhos começa.
- Erros tipográficos são aleatórios. Quando um programa é traduzido em código-fonte, numa linguagem de programação, é provável que ocorram alguns erros de digitação. Muitos serão descobertos por mecanismos de verificação de sintaxe e ortografia, mas outros podem continuar não detectados até que o teste comece. É provável que um erro tipográfico exista tanto num caminho lógico obscuro quanto num caminho principal.

Cada uma dessas razões fornece argumento para a condução de testes caixa-branca. O teste caixa-preta, independentemente de quão rigoroso, pode não encontrar as espécies de erro mencionadas aqui. O teste caixa-branca tem muito mais probabilidade de descobri-los.

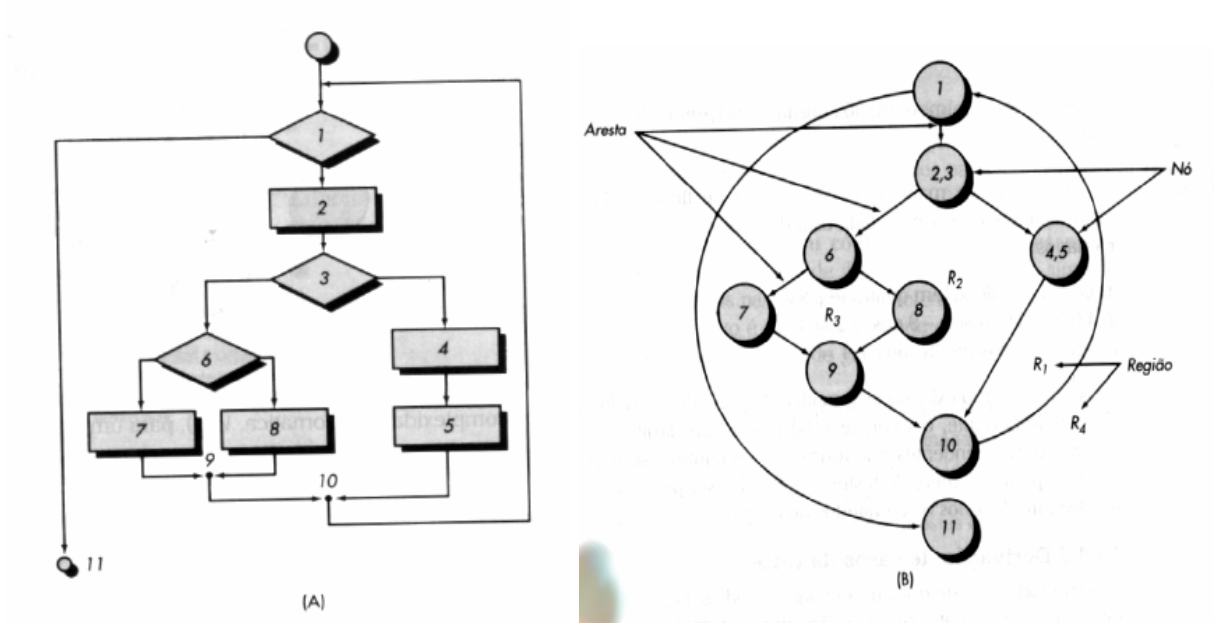
4. TESTE DE CAMINHO BÁSICO

Teste de caminho básico é uma técnica de teste caixa-branca proposta inicialmente por Tom McCabe. O método de caminho básico permite ao projetista de casos de teste originar uma medida da complexidade lógica de um projeto procedimental e usar essa medida como guia para definir um conjunto básico de caminhos de execução. Casos de testes derivados para exercitar o conjunto básico executam garantidamente cada comando do programa pelo menos uma vez durante o teste.

4.1 Notação de grafo de fluxo



Antes que o método de caminho básico possa ser introduzido, uma notação simples para a representação do fluxo de controle, chamada de grafo de fluxo (ou grafo de programa) deve ser introduzida. O grafo de fluxo mostra o fluxo de controle lógico usando a notação ilustrada na Fig. Cada construção estruturada tem um símbolo correspondente de grafo de fluxo.



Para ilustrar o uso de um grafo de fluxo, consideramos a representação de projeto procedimental na Fig. 2A. Aqui, um fluxograma é usado para mostrar a estrutura de controle do programa. A Fig. 2B mapeia o fluxograma num grafo de fluxo correspondente (considerando que nenhuma condição composta está contida nos losangos de decisão do fluxograma. Com referência à Fig. 2B, cada círculo, chamado de nó do grafo de fluxo, representa um ou mais comandos procedimentais. Uma seqüência de caixas de processamento e um losango de decisão podem ser mapeados em um único nó. As setas do grafo de fluxo, chamadas de arestas ou ligações, representam o fluxo de controle e são análogas às setas de fluxograma. Uma aresta deve terminar em nó mesmo que o nó não represente nenhum comando procedimental (p. ex., veja o símbolo para a construção se-então-senão). As áreas limitadas por arestas e nós são chamadas regiões. Ao contar regiões, incluímos a área fora do grafo como uma região.

.4.2 Complexidade ciclomática

Complexidade ciclomática é a métrica de software que fornece uma medida quantitativa da complexidade lógica de um programa. Quando usada no contexto do método de teste de caminho básico, o valor calculado para a complexidade ciclomática define o número de caminhos independentes no conjunto base de um programa e nos fornece um limite superior para a quantidade de testes que deve ser conduzida para garantir que todos os comandos tenham sido executados pelo menos uma vez.

Um caminho independente é qualquer caminho ao longo do programa que introduz pelo menos um novo conjunto de comandos de processamento ou uma nova condição. Quando enunciado em termos de grafo de fluxo, um caminho independente deve incluir pelo menos uma aresta que não tenha sido atravessada antes do caminho ser definido. Por exemplo, um conjunto de caminhos independentes para o grafo de fluxo mostrado na Fig. 17.2B é

Caminho 1: 1-11

Caminho 2: 1-2-3-4-5-10-1-11

Caminho 3: 1-2-3-6-8-9-10-1-11

Caminho 4: 1-2-3-6-7-9-10-1-11

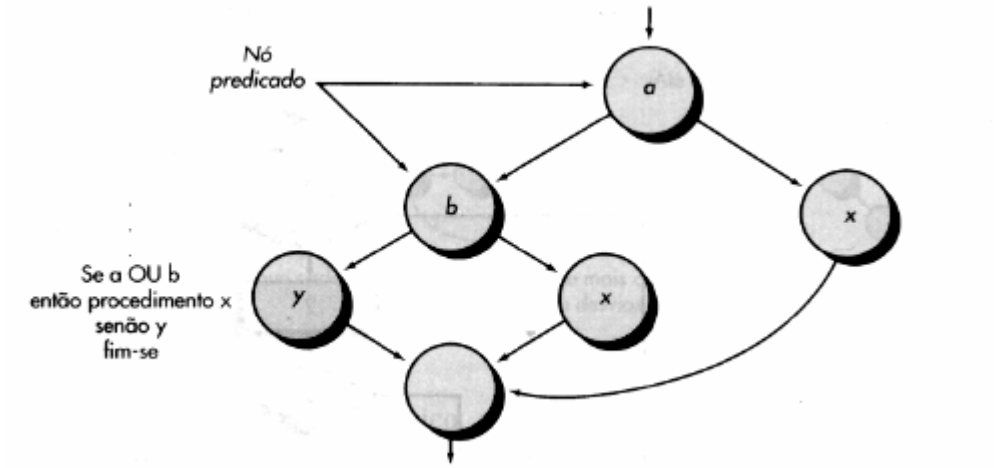
Note que cada novo caminho introduziu uma nova aresta.

O caminho 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 não é considerado como caminho independente, porque é simplesmente uma combinação de caminhos já especificados e não atravessa nenhuma nova aresta.

Os caminhos 1, 2, 3 e 4 constituem um conjunto-base para o grafo de fluxo da Fig. 17.2B. Isto é, se testes podem ser projetados para forçar a execução desses caminhos (conjunto-base), todo comando do programa terá sido garantidamente executado pelo menos uma vez e cada condição terá sido executada no seu lado verdadeiro e no seu lado falso. Deve-se notar que o conjunto-base não é único. De fato, diversos conjuntos-base diferentes podem ser derivados para um dado projeto procedimental.

Como sabemos quantos caminhos procurar? O cálculo da complexidade ciclomática fornece a resposta.

A complexidade ciclomática tem fundamentação na teoria dos grafos e nos dá uma métrica de software extremamente útil. A complexidade é calculada por uma de três maneiras:



1. O número de regiões do grafo de fluxo corresponde à complexidade ciclomática.

2. A complexidade ciclomática, V(G), para um grafo de fluxo, G, é definida como

V(G) = E - N + 2 → em que E é o número de arestas (edges-E) do grafo de fluxo e N é o número de nós (nodes-N) do grafo de fluxo.

3. A complexidade ciclomática, V(G), para um grafo de fluxo, G, é também definida como

V(G) = P + 1 → em que P é o número de nós predicados (predicate nodes-P) contido no grafo de fluxo G.

Com referência mais uma vez ao grafo de fluxo da Fig. 2B, a complexidade ciclomática pode ser calculada usando cada um dos algoritmos anteriormente mencionados:

1. O grafo de fluxo tem quatro regiões.

2. V(G) = 11 arestas - 9 nós + 2 = 4.

3. V(G) = 3 nós predicados + 1 = 4.

Assim, a complexidade ciclomática do grafo de fluxo da Fig. 2B é 4.

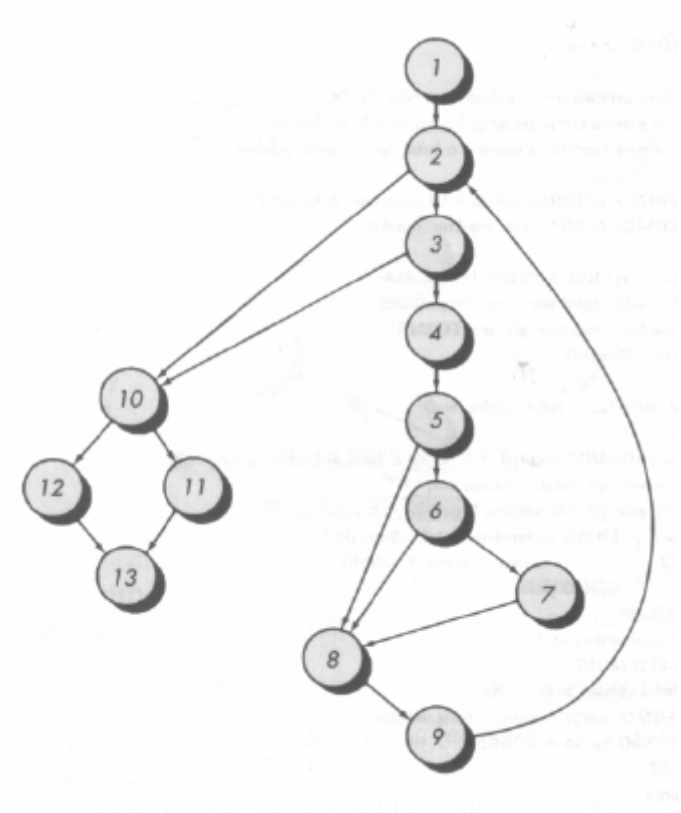
Mais importante, o valor de V(G) nos dá um limite superior para o número de caminhos independentes que formam o conjunto-base e, por implicação, um limite superior para o número de testes que precisa ser projetado e executado para garantir a cobertura de todos os comandos do programa.

4.3 Derivação de casos de teste

O método de teste do caminho básico pode ser aplicado a um projeto procedimental ou ao código-fonte. Os seguintes passos podem ser aplicados para originar o conjunto-base:

1. Usando o projeto ou código como base, desenhe o grafo de fluxo correspondente. Um grafo de fluxo é criado usando os símbolos e regras de construção apresentados. Um grafo de fluxo é criado numerando os comandos que vão ser mapeados nos nós correspondentes do grafo de fluxo.

2. Determine a complexidade ciclomática do grafo de fluxo resultante. Deve-se notar que V(G) pode ser determinado sem desenvolver um grafo de fluxo, contando todos os comandos condicionais (para o procedimento média, as condições compostas contam como duas) e somando 1. Referindo-se à Fig. 3:



V(G) = 6 regiões

V(G) = 17 arestas - 13 nós + 2 = 6

V(G) = 5 nós predicados + 1 = 6

3. Determine um conjunto-base de caminhos linearmente independentes. O valor de V(G) fornece um número de caminhos linearmente independentes na estrutura de controle do programa. No caso do procedimento média, esperamos especificar seis caminhos:

caminho 1: 1-2-10-11-13

caminho 2: 1-2-10-12-13

caminho 3: 1-2-3-10-11-13

caminho 4: 1-2-3-4-5-8-9-2-...

caminho 5: 1-2-3-4-5-6-8-9-2-...

caminho 6: 1-2-3-4-5-6-7-8-9-2-...

As reticências (...) após os caminhos 4, 5 e 6 indicam que qualquer caminho através do resto da estrutura de controle é aceitável. Freqüentemente vale a pena identificar os nós predicados como ajuda para a derivação dos casos de teste. Nesse caso, os nós 2, 3, 5, 6 e 10 são nós predicados.

4. Prepare casos de teste que vão forçar a execução de cada caminho do conjunto-base. Os dados devem ser escolhidos de modo que as condições nos nós predicados sejam adequadamente ajustadas à medida que cada caminho é testado.

Cada caso de teste é executado e comparado aos resultados esperados. Uma vez completados todos os casos de teste, o testador pode estar certo de que todos os comandos do programa foram executados pelo menos uma vez.

5. TESTE DE ESTRUTURA DE CONTROLE

A técnica de teste de caminho básico é uma de várias técnicas de teste da estrutura de controle. Apesar do teste de caminho básico ser simples e altamente efetivo, não é suficiente por si só.

5.1 Teste de condição

Teste de condição é um método de projeto de caso de teste que exercita as condições lógicas contidas em um módulo de programa. Uma condição simples é uma variável booleana ou uma expressão relacional, possivelmente precedida por um operador NÃO (-).

O método de teste de condição focaliza o teste de cada condição do programa. As estratégias de teste de condição têm geralmente duas vantagens. Primeiro, a medida da cobertura de teste de uma condição é simples. Segundo, a cobertura de teste das condições de um programa dá diretrizes para a geração de testes adicionais para o programa.

A finalidade do teste de condição é detectar não apenas erros nas condições de um programa, mas também outros erros do programa. Se um conjunto de testes para um programa P é efetivo para detectar erros nas condições contidas em P, é provável que esse conjunto de teste seja também efetivo para detectar outros erros em P. Além disso, se uma estratégia de teste é efetiva para detectar erros numa condição, então provável que essa estratégia também seja efetiva para detectar erros no programa.

Diversas estratégias para teste de condição têm sido propostas. O teste de desvio e provavelmente a estratégia de teste de condição mais simples. Para uma condição composta C, os ramos verdadeiro e falso de C e cada condição simples de C precisam ser executadas pelo menos uma vez.

O teste de domínio requer que três ou quatro testes sejam derivados para uma expressão relacional.

5.2 Teste de fluxo de dados

O método de teste de fluxo de dados seleciona caminhos de teste de um programa de acordo com a localização das definições e do uso das variáveis no programa. Diversas estratégias de teste de fluxo de dados foram estudadas e comparadas.

Como os comandos de um programa estão relacionados uns com os outros, de acordo com as definições e usos das variáveis, a abordagem de teste de fluxo de dado é efetiva para detecção de erros. No entanto, os problemas de medida da cobertura de teste e de seleção dos caminhos de teste para o teste de fluxo de dados são mais difíceis do que os problemas correspondentes para o teste de condição.

5.3 Teste de ciclo

Ciclos são a pedra fundamental da grande maioria de todos algoritmos implementados em software. E, no entanto, freqüentemente nós lhes damos pouca atenção ao conduzir testes de software.

Teste de ciclo é uma técnica de teste caixa-branca que focaliza exclusivamente a validade de construções de ciclo. Quatro diferentes classes de ciclos podem ser definidas: ciclos simples, concatenados, aninhados e desestruturados (Fig. 8).

Ciclos simples. O seguinte conjunto de testes pode ser aplicado a ciclos simples em que n é o número máximo de passagens permitidas pelo ciclo.

1. Pule o ciclo completamente.
2. Apenas uma passagem pelo cicio.
3. Duas passagens pelo ciclo.
4. m passagens pelo ciclo em que m < n.
5. n - 1, n, n + 1 passagens pelo ciclo.

Ciclos aninhados. Se tivéssemos que estender a abordagem de teste de ciclos simples para ciclos aninhados, o número de testes possíveis iria crescer geometricamente, medida que o nível de aninhamento crescesse. Isso iria resultar num número de testes impraticável. Beizer sugere uma abordagem que vai ajudar a reduzir o número de testes:

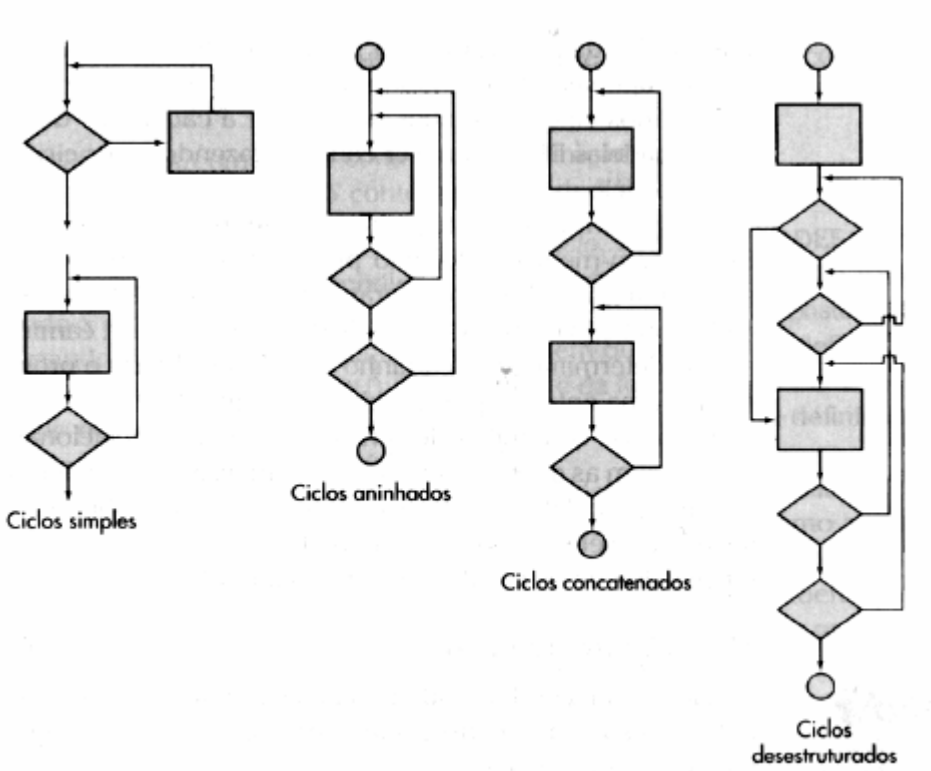
1. Comece no ciclo mais interno. Ajuste todos os outros ciclos para os valores mínimos

2. Conduza testes de ciclo simples para o ciclo mais interno enquanto mantém os ciclos externos nos seus valores mínimos do parâmetro de iteração (p. ex., contador de ciclo). Adicione

outros testes para valores fora do intervalo ou excluídos.

3. Trabalhe em direção ao exterior, conduzindo testes para o ciclo seguinte, mas mantendo todos os outros ciclos externos nos valores mínimos e os outros ciclos aninhados em valores "típicos".

4. Continue até que todos os ciclos tenham sido testados.



Ciclos concatenados. Esses ciclos podem ser testados usando a abordagem definida para ciclos simples, se cada um dos ciclos é independente do outro. No entanto, se dois ciclos são concatenados e o contador de ciclo, para o ciclo 1, é usado como vala inicial para o ciclo 2, então os ciclos não são independentes. Quando os ciclos não são independentes, a abordagem aplicada a ciclos aninhados é recomendada.

Ciclos desestruturados. Sempre que possível essa classe de ciclos deve ser reprojetada para refletir o uso de construções de programação estruturada.

6. TESTE CAIXA-PRETA

Um teste caixa-preta, também chamado de teste comportamental, focaliza os requisitos funcionais do software. Isto é, o teste caixa-preta permite ao engenheiro de software derivar conjuntos de condições de entrada que vão exercitar plenamente todos os requisitos funcionais de um programa. O teste caixa-preta não é uma alternativa às técnica caixa-branca. Ao contrário, é uma abordagem complementar, que mais provavelmente descobrirá uma classe diferente de erros do que os métodos caixa-branca.

O teste caixa-preta tenta encontrar erros das seguintes categorias:

(1) funções incorretas ou omitidas,

(2) erros de interface,

(3) erros de estrutura de dados ou de acesso a base de dados externa,

(4) erros de comportamento ou desempenho e

(5) erros de iniciação e término.

Diferente do teste caixa-branca, que é realizado no início do processo de teste, o teste caixa-preta tende a ser aplicado durante os últimos estágios do teste. Como o teste caixa-preta despreza, de propósito, a estrutura de controle, atenção é focalizada no domínio da informação. Os

testes são projetados para responder às seguintes questões:

- Como a validade funcional é testada?
- Como o comportamento e o desempenho do sistema são testados?
- Que classes de entrada vão constituir bons casos de teste?
- O sistema é particularmente sensível a certos valores de entrada?
- Como são isolados os limites de uma classe de dados?
- Que taxas e volumes de dados o sistema pode tolerar?
- Que efeito as combinações específicas de dados vão ter na operação do sistema

Aplicando técnicas caixa-preta, derivamos um conjunto de casos de teste que satisfaz os seguintes critérios:

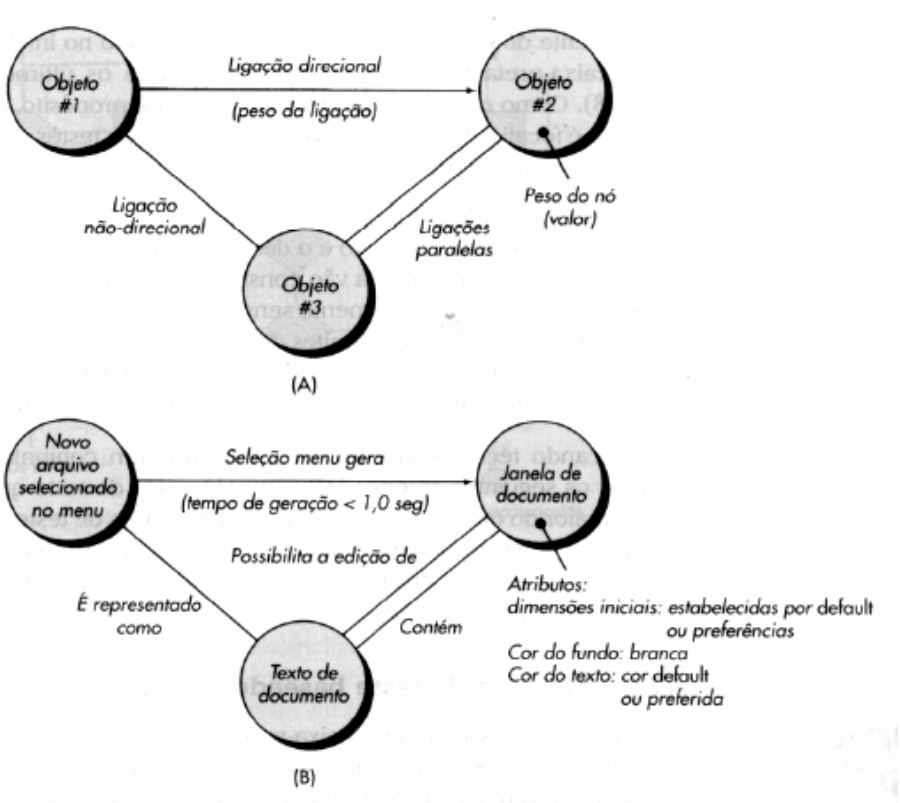
(1) casos de teste que reduzem, de um valor que é maior do que 1, o número adicional de casos de teste que precisam ser projetados para atingir um teste razoável e

(2) casos de teste que nos dizem algo sobre a presença ou ausência de classes de erros, ao invés de um erro associado somente com o teste específico em mãos.

6.1 Métodos de teste baseados em grafo

O primeiro passo no teste caixa-preta é entender os objetos que estão modelados no software e as relações que conectam esses objetos. Uma vez que isso tenha sido conseguido, o passo seguinte é definir uma série de testes que verifica se "todos os objetos têm a relação esperada uns com os outros". Dito de outro modo, teste de software começa criando um grafo dos objetos importantes e de suas relações e depois estabelecendo uma série de testes que vão cobrir o grafo de modo que cada objeto e relação sejam exercitados e que os erros sejam descobertos.

Para executar esses passos, o engenheiro de software começa criando um grafo - uma coleção de nós que representam objetos; ligações que representam as relações entre objetos; pesos de nó que descrevem as propriedades de um nó (p. ex., um valor de dados específico ou comportamento de um estado); e pesos de ligação que descrevem algumas características de uma ligação.



A representação simbólica de um grafo é mostrada na Fig. 9A. Os nós são representados

com círculos conectados por ligações que assumem algumas formas diferentes. Uma ligação direcionada (representada por uma seta) indica que a relação se move em apenas uma direção. Uma ligação bidirecional, também chamada de ligação simétrica, implica que a relação se aplica em ambas as direções. Ligações paralelas são usadas quando diversas relações diferentes são estabelecidas entre nós de grafo.

Como exemplo simples, considere uma parte de um grafo para uma aplicação de processamento de texto (Fig. 9B) em que

Objeto #1 = arquivo novo selecionado no menu

Objeto #2 = janela de documento

Objeto #3 = texto de documento

Com referência à Figura, uma seleção de menu de arquivo novo gera uma janela de documento. O peso do nó de janela de documento fornece uma lista de atributos da janela, que devem ser esperados quando a janela é gerada. O peso da ligação indica que a janela deve ser gerada em menos que 1,0 segundo. Uma ligação não-direcional estabelece uma relação simétrica entre arquivo novo selecionado no menu e texto de documento, e ligações paralelas indicam relações entre janela de documento e texto de documento. Na verdade, um grafo muito mais detalhado deveria ser gerado como precursor do projeto de casos de teste. O engenheiro de software origina depois casos de teste percorrendo o grafo e cobrindo cada uma das relações mostradas. Esse casos de teste são projetados numa tentativa de encontrar erros em quaisquer da relações.

Beizer descreve alguns métodos de teste de comportamento que podem fazer uso de grafos:

Modelagem de fluxo de transação. Os nós representam passos em alguma transação (p. ex., os passos necessários para fazer uma reserva numa linha aérea, usando um serviço on-line), e as ligações representam as conexões lógicas entre os passos (p. ex., entrada.informação.vôo é seguida por processamento. validação/disponibilidade). O diagrama de fluxo de dados pode ser usado para ajudar a criar grafos desse tipo.

Modelagem de estado finito. Os nós representam diferentes estados do software observáveis pelo usuário (p. ex., cada uma das "telas" que aparece, enquanto um funcionário, que dá entrada em pedidos, recebe um pedido por telefone) e a ligações representam as transições que ocorrem para ir de um estado para outro estado (p. ex., informação do pedido é verificada durante pesquisa da disponibilidade de estoque e é seguida por entrada informação-faturamento-cliente). O Diagrama de transição de estados pode ser usado para ajudar na criação de grafos desse tipo.

Modelagem do fluxo de dados. Os nós são objetos de dados e as ligações são a transformações que ocorrem para traduzir um objeto de dados em outro. Por exemplo, o nó imposto.retido (IR) é calculado a partir à salário.bruto (SB) usando a relação IR = 0,62 x SB.

Modelagem de tempo. Os nós são objetos de programa e as ligações são a conexões seqüenciais entre esses objetos. Os pesos das ligações são usados para especificar os tempos de execução necessários, enquanto o programa é executado.

O teste baseado em grafos começa com a definição de todos os nós e pesos de nós, isto é, objetos e atributos são identificados. O modelo de dados pode ser usado como ponto de partida, mas é importante notar que muitos nós podem ser objetos de programa (não representados explicitamente no modelo de dados). Para da uma indicação dos pontos de partida e parada de um grafo, é útil definir nós de entrada e de saída.

Uma vez identificados os nós, as ligações e os pesos das ligações devem se estabelecidos. Em geral, as ligações devem ser denominadas, apesar de as ligações que representam fluxo de controle entre objeto de programa não precisarem se denominadas.

Em muitos casos, o modelo de gratos pode ter ciclos (i. e., um caminho ao longo do grafo no qual um ou mais nós são encontrados mais que uma vez). O teste de ciclo pode também ser aplicado no nível comportamental (caixa-preta). O grafo vai ajudar na identificação dos ciclos que precisam ser testados.

Cada relação é estudada separadamente de modo que casos de teste possam ser derivados. A transitividade das relações seqüenciais é estudada para determinar como o impacto das relações se propaga pelos objetos definidos num grafo. A transitividade pode ser ilustrada considerando-se três objetos, X, Y, e Z. Considere as seguintes relações:

X é necessário para calcular Y

Y é necessário para calcular Z

Conseqüentemente, uma relação transitiva pode ser estabelecida entre X e Z:

X é necessário para calcular Z

Com base nessa relação transitiva, testes para encontrar erros no cálculo de Z devem considerar diversos valores, tanto de X quanto de Y.

A simetria de uma relação (ligação do grafo) é também uma diretriz importante para o projeto de casos de teste. Se uma ligação é realmente bidirecional (simétrica), é importante testar essa característica. A capacidade de desfazer em muitas aplicações de computadores pessoais implementa simetria limitada. Isto é, UNDO permite que uma ação seja negada depois de ter sido completada. Isso deve ser rigorosamente testado e todas as exceções (i. e., lugares em que o UNDO não pode ser usado) devem ser registradas. Finalmente, cada nó do grafo deve ter uma relação que leva de volta a si próprio; em essência, um ciclo "sem ação" ou "ação nula". Essas relações reflexivas devem também ser testadas.

À medida que tem início o projeto de casos de teste, o primeiro objetivo é conseguir cobertura de nós. Por isso, queremos dizer que devem ser projetados testes para demonstrar que nenhum nó foi omitido inadvertidamente e que os pesos dos nós (atributos de objeto) estão corretos.

A seguir, cobertura de ligações é abordada. Cada relação é testada com base nas suas propriedades. Por exemplo, uma relação simétrica é testada para demonstrar que é de fato bidirecional. Uma relação transitiva é testada para demonstrar que a transitividade está presente. Uma relação reflexiva é testada para garantir que um ciclo nulo está presente. Quando os pesos das ligações tiverem sido especificados, são criados testes para demonstrar que esses pesos são válidos. Finalmente, o teste de ciclo é invocado.

6.2 Particionamento de equivalência

O particionamento de equivalência é um método de teste caixa-preta que divide o domínio de entrada de um programa em classes de dados, das quais casos de teste podem ser derivados. Um caso de teste ideal descobre sozinho uma classe de erros (p. ex., processamento incorreto de todos os dados de caracteres), que poderia de outra forma exigir que muitos casos fossem executados antes que um erro geral fosse observado. O particionamento de equivalência busca definir um caso de teste que descobre classes de erros, reduzindo assim o número total de casos de testes que precisam ser desenvolvidos.

Projeto de casos de teste para particionamento de equivalência é baseado numa avaliação das classes de equivalência para uma condição de entrada. Usando conceitos introduzidos na seção anterior, se o conjunto de objetos pode ser ligado por relações que são simétricas, transitivas e reflexivas, uma classe de equivalência esta presente. Uma classe de equivalência representa um conjunto de estados válidos ou inválidos para condições de entrada. Tipicamente, uma condição de entrada é um valor numérico específico, um intervalo de valores, um conjunto de valores relacionados ou uma condição booleana. Classes de equivalência podem ser definidas de acordo com as seguintes diretrizes:

1. Se uma condição de entrada especifica um intervalo, uma classe de equivalência válida e duas inválidas são definidas.

2. Se uma condição de entrada exige um valor específico, uma classe de equivalência válida e duas inválidas são definidas.

3. Se uma condição de entrada especifica um membro de um conjunto, uma classes de equivalência válida e uma inválida são definidas.

4. Se uma condição de entrada é booleana, uma classe de equivalência válida e uma inválida são definidas.

Como exemplo, considere dados mantidos como parte de uma aplicação de automação bancária. O usuário pode ter acesso ao banco usando um computador pessoal, fornecer uma senha de seis dígitos e a seguir dar uma série de comandos digitado que disparam as várias funções bancárias. Durante a seqüência de admissão, o software fornecido para aplicação bancária aceita dados da forma:

- Código de área - branco ou número de três dígitos

- Prefixo - número de três dígitos não iniciado por 0 ou 1
- Sufixo - número de quatro dígitos
- Senha - cadeia alfanumérica de seis dígitos
- Comandos - cheque, depósito, pagamento de conta e similares...

As condições de entrada associadas a cada elemento da aplicação bancária podem ser especificadas como:

- Código de área - condição de entrada, booleana - o código de área pode ou não estar presente.
    - condição de entrada, intervalo - valores definidos entre 200 e 999, com exceções específicas.
- Prefixo: condição de entrada, intervalo - valor especificado > 200
    - condição de entrada, valor - com tamanho de três dígitos
- Senha: condição de entrada, booleana - uma senha pode estar ou não presente.
    - Condição de entrada, valor - cadeia de seis caracteres.
- Comando: condição de entrada, conjunto - contendo os comandos mencionados anteriormente.

Aplicando as diretrizes para derivação das classes de equivalência, casos de teste para cada item de dados do domínio de entrada podem ser desenvolvidos e executados. Casos de testes são selecionados de modo que o maior número de atributos de uma classe de equivalência é exercitado ao mesmo tempo.

6.3 Análise de valor-limite

Por motivos que não estão completamente claros, um grande número de erros tende a ocorrer nas fronteiras do domínio de entrada ao invés de no "centro". É por essa razão que a análise de valor-limite (boundary valise analysis, BVA) foi desenvolvida como técnica de teste. A análise de valor-limite leva à seleção de casos de teste os que exercitam os valores limítrofes.

A análise de valor-limite é uma técnica de projeto de casos de teste que completa o particionamento de equivalência. Em vez de selecionar qualquer elemento de uma classe de equivalência, a BVA leva à seleção de casos de teste nas "bordas" da classe. Em vez de focalizar somente as condições de entrada, a BVA deriva casos de teste também para o domínio de saída.

As diretrizes para a BVA são semelhantes em muitos aspectos às que são fornecidas para o particionamento de equivalência:

1. Se uma condição de entrada especifica um intervalo limitado pelos valores a e b, casos de testes devem ser projetados com os valores a e b e imediatamente acima e imediatamente abaixo de a e b.

2. Se uma condição de entrada especifica vários valores, casos de teste devem ser desenvolvidos para exercitar os números mínimo e máximo. Valores imediatamente acima e imediatamente abaixo do mínimo e do máximo também são testados.

3. Aplica as diretrizes 1 e 2 às condições de saída. Por exemplo, considere que uma tabela de temperatura versos pressão é esperada como saída de um programa de análise de engenharia. Casos de teste devem ser projetados para criar um relatório de saída que produza o número máximo (e mínimo) admissível de entradas na tabela.

4. Se as estruturas de dados internas do programa têm limites prescritos (p. ex., um vetor tem um limite definido de 100 entradas), certifique-se de projetar um caso de teste para exercitar a estrutura de dados no seu limite.

A maioria dos engenheiros de software realiza a BVA intuitivamente num certo grau, Aplicando essas diretrizes, o teste de limite vai ser mais completo, tendo assim uma maior probabilidade de detecção de erro.

6.4 Teste de comparação

Há algumas situações (p. ex., avïônica de aeronaves, sistemas de freio de automóveis) em que a confiabilidade do software é absolutamente crítica. Em tais aplicações software e hardware redundantes são freqüentemente usados para minimizar a possibilidade de erro. Quando um software redundante é desenvolvido, equipes de engenharia de software separadas desenvolvem versões independentes de uma aplicação usando a mesma especificação. Em tais situações, cada versão pode ser testada com os mesmos dados de teste para garantir que todas fornecem saída idêntica. Depois, todas as versões são executadas em paralelo com comparação em tempo real dos resultados para garantir consistência.

Usando as lições aprendidas dos sistemas redundantes, pesquisadores têm sugerido que versões independentes do software sejam desenvolvidas para aplicações críticas, mesmo quando uma única versão vai ser usada no sistema baseado em computador que será entregue. Essas versões independentes formam a base da técnica de teste caixa-preta chamada teste de comparação ou teste de emparelhamento.

Quando múltiplas implementações da mesma especificação tiverem sido produzidas, casos de teste projetados usando outras técnicas caixa-preta (p. ex., particionamento de equivalência) são fornecidos como entrada a cada versão do software. Se a saída de cada versão é a mesma, é considerado que todas as implementações estão certas. Se a saída é diferente, cada uma das aplicações é investigada para determinar se um defeito em uma ou mais versões é responsável pela diferença. Na maioria dos casos a comparação da saída pode ser realizada por uma ferramenta automatizada.

O teste de comparação não é a toda prova. Se a especificação a partir da qual todas as versões foram desenvolvidas estiver errada, todas as versões irão provavelmente refletir o erro. Além disso, se cada uma das versões independentes produzir resultados idênticos mas incorretos, o teste de comparação vai falhar na detecção do erro.

7. TESTE DE AMBIENTES, ARQUITETURAS E APLICAÇÕES ESPECIALIZADAS

À medida que o software para computador torna-se mais complexo, a necessidade de abordagens de testes especializadas tem crescido. Os métodos de teste caixa-branca e caixa-preta são aplicáveis em todos os ambientes, arquiteturas e aplicações, mas diretrizes e abordagens especiais para teste está algumas vezes disponíveis. Nesta seção, consideramos diretrizes para teste de ambientes, arquiteturas e aplicações especializadas que são comumente encontradas por engenheiros de software.

7.1 Teste de GUI

Interfaces gráficas com o usuário (graphical user interfaces, GUI) apresentam desafio interessantes para os engenheiros de software. Devido a componentes reusáveis fornecidos como parte de ambientes de desenvolvimento de GUI, a criação da interface com o usuário tornou-se menos demorada e mais precisa. Mas, ao mesmo tempo, a complexidade das GUI cresceu, levando a mais dificuldade no projeto e execução de casos de teste.

Como muitas GUI modernas têm a mesma aparência e funcionamento, uma série de testes padrão pode ser derivada. Grafos de modelagem de estados finitos podem se usados para derivar uma série de testes que tratam de objetos de dados e programas específicos, que são relevantes para a GUI.

Devido ao grande número de permutações associadas com as operações GUI, o teste deve ser conduzido usando ferramentas automatizadas. Uma grande variedade de ferramentas de teste de GUI apareceu no mercado nos últimos anos.

7.2 Teste de arquiteturas cliente/servidor

Arquiteturas clientes/servidor (client/server, C/S) representam um significativo desafio para os testadores de software. A natureza distribuída de ambientes cliente/servidor, os aspectos de

desempenho, associados com processamento de transações, a presença potencial de várias plataformas de hardware diferentes, as complexidades de redes de comunicação, a necessidade de atender muitos clientes por uma base de dados centralizada (ou em alguns casos distribuída) e os requisitos de coordenação impostos ao servidor, tudo se combina para tornar o teste de arquiteturas C/S e do software que nelas reside consideravelmente mais difícil do que de aplicações isoladas. De fato, estudos recentes na indústria indicam um significativo aumento de tempo e custo de teste quando ambientes C/S são desenvolvidos.

7.3 Teste da documentação e dispositivos de ajuda

O termo teste de software invoca imagens de grande número de casos de teste preparados para exercitar programas de computador e os dados que eles manipulam. O teste deve também ser estendido para o terceiro elemento da configuração de software - documentação.

Erros na documentação podem ser tão devastadores para aceitação do programa quanto erros nos dados ou código-fonte. Nada é mais frustrante que seguir exatamente um guia do usuário ou um documento de ajuda on-line e obter resultados ou comportamentos que não coincidam com aqueles previstos pela documentação. É por isso que o teste de documentação deve ser uma parte significativa de todo o plano de teste de software.

O teste de documentação pode ser abordado em duas fases. A primeira fase, revisão e inspeção, examina o documento quanto à clareza editorial. A segunda fase, teste ao vivo, usa a documentação em conjunto com o uso do programa real.

Surpreendentemente, um teste de documentação ao vivo pode ser abordado usando técnicas que são análogas aos vários métodos de teste caixa-preta. Testes baseados em gratos podem ser usados para descrever o uso do programa; particionamento de equivalência e análise de valor e limite podem ser usados para definir várias classes de entrada e interações associadas. O uso do programa é então rastreado através da documentação. As seguintes questões devem ser respondidas durante ambas as fases:

- A documentação descreve precisamente como escolher cada modo de uso?
- A descrição de cada seqüência de interação é precisa?
- Os exemplos são corretos?
- A terminologia, as descrições dos menus e as respostas do sistema são consistentes com o programa real?
- É relativamente fácil localizar diretrizes na documentação?
- A correção de defeitos pode ser realizada facilmente com a documentação?
- O sumário e o índice do documento são precisos e completos?
- O projeto do documento (leiaute, escolha de tipos, tabulação, gráficos) conduz ao entendimento e rápida assimilação da informação?
- As mensagens de erro do software mostradas para o usuário são todas descritas em mais detalhes no documento? As ações a serem tomadas como conseqüência de uma mensagem de erro são claramente delineadas?
- Se são usadas ligações de hipertexto, elas são precisas e completas?
- Se é usado hipertexto, o projeto de navegação é apropriado à informação exigida?

O único modo viável de responder a essas questões é ter um parceiro independente (p. ex., usuários selecionados) testando a documentação no contexto de uso do programa. Todas as discrepâncias são anotadas e as áreas de ambigüidade ou fraqueza do documento são definidas para potencial reescrita.

7.4 Teste de sistemas de tempo real

A natureza dependente de tempo e assíncrona, de muitas aplicações em tempo real, adiciona um elemento novo e potencialmente difícil à mistura do teste - tempo. O projetista de casos de teste não tem apenas que considerar casos de teste caixa branca e caixa-preta, mas também a manipulação de eventos (i. e., processamento de interrupções), a tempestividade dos dados e o paralelismo das tarefas (processos) que manipulam os dados. Em muitas situações, dados

de teste fornecidos vão resultar em processamento adequado, quando um sistema de tempo real está num estado, enquanto que os mesmos dados fornecidos quando o sistema está num estado diferente podem levar a erro.

Por exemplo, o software de tempo real que controla uma nova fotocopiadora aceite interrupções do operador (i. e., o operador da máquina aperta teclas de controle tais como RESET ou ESCUREÇA), sem erro, quando a máquina está fazendo cópias (no estado "copiando'). Essas mesmas interrupções do operador, se efetuadas quando ê máquina está no estado "emperrada" provocam a exibição de um código de diagnóstico indicando que a posição onde está emperrada foi perdida (um erro).

Além disso, a relação íntima que existe entre software de tempo real e seu ambiente de hardware pode também causar problemas de teste. Testes de software devem considerar o impacto de falhas do hardware no processamento do software. Tais falhas podem ser extremamente difíceis de simular realisticamente.

Métodos de projeto de casos de teste abrangentes para sistemas de tempo real ainda têm que evoluir. No entanto, uma estratégia global de quatro passos pode ser proposta:

Teste de tarefa. O primeiro passo no teste de software de tempo real é testar cada tarefa independentemente. Isto é, testes caixa-branca e caixa-preta são projetados e executados para cada tarefa. Cada tarefa é executada independentemente durante esses testes. O teste de tarefa descobre erros de lógica e de função, mas não de tempestividade ou de comportamento.

Teste comportamental. Usando modelos do sistema criados com ferramentas CASE, é possível simular o comportamento de um sistema de tempo real e examinar seu comportamento como conseqüência de eventos externos. Essas atividades de análise podem servir de base para o projeto de casos de testes que são conduzidos depois do software de tempo real ter sido construído. Usando uma técnica que é semelhante ao particionamento de equivalência, eventos (p. ex., interrupções, sinais de controle) são categorizados para teste. Por exemplo, eventos para a fotocopiadora poderiam ser interrupções do usuário (p. ex., reset contador), interrupções mecânicas (p. ex., papel emperrado), interrupções do sistema (p. ex., toner baixo) e modos de falha (p. ex., rolo superaquecido). Cada um desses eventos é testado individualmente e o comportamento do sistema executável é examinado para detectar erros que ocorrem como conseqüência do processamento associado a esses eventos. O comportamento do modelo do sistema (desenvolvido durante a atividade de análise) e o software executável podem ser comparados para verificar sua conformidade. Uma vez testada cada classe de eventos, os eventos são apresentados ao sistema em ordem aleatória e com freqüência aleatória. O comportamento do software é examinado para detectar erros de comportamento.

Testes intertarefas. Uma vez que erros em tarefas individuais e no comportamento do sistema foram isolados, o teste passa para erros relativos a tempo. Tarefas assíncronas que se comunicam umas com as outras são testadas com diferentes taxas de dados e carga de processamento para detectar se erros de sincronização intertarefas vão ocorrer. Além disso, tarefas que se comunicam por fila de mensagens ou depósito de dados são testadas para descobrir erros no tamanho dessas áreas de armazenamento.

Teste de sistema. O software e o hardware são integrados e todo um conjunto de testes de sistema é conduzido numa tentativa de descobrir erros na interface software/hardware. A maioria dos sistemas de tempo real processa interrupções. Assim, o teste da manipulação desses eventos booleanos é essencial.

Usando o diagrama de transição de estados e a especificação de controle, o testador desenvolve uma lista de todas as possíveis interrupções e do processamento que ocorre como conseqüência das interrupções. Então, são projetados testes para avaliar as seguintes características do sistema:

- As prioridades de interrupção estão atribuídas adequadamente e propriamente manipuladas?
- O processamento para cada interrupção foi conduzido corretamente?
- O desempenho (p. ex., tempo de processamento) de cada procedimento de manipulação de interrupção está de acordo com os requisitos?
- Um alto volume de interrupções chegando em tempos críticos cria problemas de função ou desempenho?

Além disso, áreas globais de dados, que são usadas para transferir informação como parte

do processamento de interrupções, devem ser testadas para avaliar o potencial de geração de efeitos colaterais.

8. RESUMO

O objetivo principal do projeto de casos de teste é originar um conjunto de testes que tenha a maior probabilidade de descobrir erros no software. Para alcançar esse objetivo, duas diferentes categorias de técnicas de projeto de casos de teste são usadas: teste caixa-branca e teste caixa-preta.

O teste caixa-branca focaliza a estrutura de controle do programa. Casos de testes são originados para garantir que todos os comandos do programa tenham sido executados pelo menos uma vez durante o teste e que todas as condições lógicas tenham sido exercitadas. O teste de caminho básico, uma técnica caixa-branca, faz uso de grafos de programa (ou matrizes de grafos) para originar um conjunto de testes linearmente independentes que vão garantir a cobertura. Os testes de condição e de fluxo de dados exercitam adicionalmente a lógica do programa e os testes de ciclo complementam outras técnicas caixa-branca, fornecendo um procedimento para exercitar ciclos com vários graus de complexidade.

Hetzel descreve o teste caixa-branca como "teste no varejo". Sua implicação é de que os testes caixa-branca que consideramos são tipicamente aplicados a pequenos componentes de programa (p. ex., módulos ou pequenos grupos de módulos). O teste caixa-preta, por outro lado, amplia nosso enfoque e pode ser chamado de "teste por atacado".

Os testes caixa-preta são projetados para validar os requisitos funcionais sem se prender ao funcionamento interno de um programa. As técnicas de teste caixa-preta focalizam o domínio de informação do software, originando casos pelo particionamento dos domínios de entrada e saída de um programa, de um modo que fornece rigorosa cobertura de teste. O particionamento de equivalência divide o domínio de entrada em classes de dados que provavelmente exercitam função específica do software. A análise de valor-limite investiga a habilidade do programa de manipular dados no limite de aceitabilidade. Teste de matriz ortogonal fornece um método eficiente e sistemático para testar sistemas com pequeno número de parâmetros de entrada.

Métodos especializados de teste abrangem uma grande variedade de capacidade do software e áreas de aplicação. O teste para interfaces gráficas com o usuário arquitetura cliente/servidor, documentação e facilidades de ajuda e sistemas de teme real requerem em cada caso diretrizes e técnicas especializadas.

Desenvolvedores de software experientes freqüentemente dizem "Ele é simplesmente transferido de você [o engenheiro de software] para seu cliente. Toda vez que o seu cliente usa o programa, um teste está sendo conduzido". Pela aplicação de casos de teste, o engenheiro de software pode conseguir um teste mais completo e conseqüentemente descobrir e corrigir o maior número de erros antes que os "testes do cliente" comecem.

Bibliografia

- Pressman, Roger S. Engenharia de Software MAKRON BOOKS 5º Edição - 2002
- Sommerville, Ian. Engenharia de Software. Addison Wesley, 6ª Edição - 2003