

NOW COVERS TESTING FOR USABILITY, SMART PHONE APPS,
AND AGILE DEVELOPMENT ENVIRONMENTS

THE ART OF
SOFTWARE
TESTING

THIRD EDITION

3

GLENFORD J. MYERS
COREY SANDLER · TOM BADGETT

A ARTE DE PROGRAMAS TESTE

A ARTE DE PROGRAMAS TESTE

Terceira edição

GLENFORD J. MYERS
TOM BADGETT
COREY SANDLER



John Wiley & Sons, Inc.

Copyright # 2012 pela Word Association, Inc. Todos os direitos reservados.

Publicado por John Wiley & Sons, Inc., Hoboken, Nova Jersey.

Publicado simultaneamente no Canadá.

Nenhuma parte desta publicação pode ser reproduzida, armazenada em um sistema de recuperação ou transmitida de qualquer forma ou por qualquer meio, eletrônico, mecânico, fotocópia, gravação, digitalização ou qualquer outro, exceto conforme permitido pela Seção 107 ou 108 do 1976 United Lei de Direitos Autorais dos Estados Unidos, sem a permissão prévia por escrito do Editor, ou autorização através do pagamento da taxa por cópia apropriada ao Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 464-8600, ou na web em www.copyright.com. Solicitações de permissão ao Editor devem ser endereçadas ao Departamento de Permissões, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, ou online em www.wiley.com/go/permissions.

Límite de responsabilidade/exclusão de garantia: Embora o editor e o autor tenham usado seus melhores esforços na preparação deste livro, eles não fazem representações ou garantias com relação à precisão ou integridade do conteúdo deste livro e especificamente se isentam de quaisquer garantias implícitas de comercialização ou adequação a um propósito específico. Nenhuma garantia pode ser criada ou estendida por representantes de vendas ou materiais de vendas por escrito. Os conselhos e estratégias aqui contidos podem não ser adequados à sua situação. Você deve consultar um profissional quando apropriado. Nem o editor nem o autor serão responsáveis por qualquer perda de lucro ou quaisquer outros danos comerciais, incluindo, mas não limitado a danos especiais, incidentais, consequenciais ou outros.

Para obter informações gerais sobre nossos outros produtos e serviços ou para suporte técnico, entre em contato com nosso Departamento de Atendimento ao Cliente nos Estados Unidos em (800) 762-2974, fora dos Estados Unidos em (317) 572-3993 ou fax (317) 572- 4002.

A Wiley também publica seus livros em diversos formatos eletrônicos. Alguns conteúdos que aparecem impressos podem não estar disponíveis em livros eletrônicos. Para obter mais informações sobre os produtos Wiley, visite nosso site em www.wiley.com.

Dados de Catalogação na Publicação da Biblioteca do Congresso:

Myers, Glenford J., 1946- A
arte do teste de software / Glenford J. Myers, Corey Sandler, Tom Badgett. — 3^a edição.
pág. cm.
Inclui índice.

ISBN 978-1-118-03196-4 (pano); ISBN 978-1-118-13313-2 (ebk); ISBN 978-1-118-13314-9 (ebk); ISBN 978-1-118-13315-6 (ebk)

1. Software de computador—Teste. 2. Depuração em informática. I. Sandler, Corey, 1950-II.
Badget, Tom. III. Título.
QA76.76.T48M894 2011
005.1 4—dc23

2011017548

Impresso nos Estados Unidos da América

10 9 8 7 6 5 4 3 2 1

Conteúdo

Prefácio	vii
Introdução	ix
1 Um Teste de Autoavaliação	1
2 A Psicologia e Economia do Teste de Software	5
3 Inspeções, orientações e revisões do programa	19
4 Projeto de Caso de Teste	41
5 Módulo (Unidade) Teste	85
6 Testes de ordem superior	113
7 Teste de Usabilidade (Usuário)	143
8 Depuração	157
9 Testes no ambiente ágil	175
10 Testando aplicativos da Internet	193
11 Teste de aplicativos móveis	213
Apêndice Exemplo de Aplicação de Teste Extremo	227
Índice	233

Prefácio

Em 1979, Glenford Myers publicou um livro que se tornou um clássico.

A Arte do Teste de Software resistiu ao teste do tempo — 25 anos na lista de livros disponíveis da editora. Este fato por si só é um testemunho da natureza sólida, essencial e valiosa de seu trabalho.

Durante esse mesmo período, os autores desta edição (a terceira) de *The Art of Software Testing* publicaram, coletivamente, mais de 200 livros, a maioria deles sobre temas de software de computador. Alguns desses títulos venderam muito bem e, como este, passaram por várias versões. *Fix Your Own PC*, de Corey Sandler, por exemplo, está em sua oitava edição no momento em que este livro vai para a impressão; e os livros de Tom Badgett sobre Microsoft PowerPoint e outros títulos do Office passaram por quatro ou mais edições. No entanto, ao contrário do livro de Myers, nenhum deles permaneceu atual por mais de alguns anos.

Qual é a diferença? Os livros mais recentes cobriam mais transitórios tópicos—sistemas operacionais, software de aplicativos, segurança, tecnologia de comunicação e configurações de hardware. Mudanças rápidas na tecnologia de hardware e software de computador durante as décadas de 1980 e 1990 exigiram mudanças e atualizações frequentes nesses tópicos.

Também durante esse período foram publicados centenas de livros sobre teste de software. Eles também adotaram uma abordagem mais transitória do assunto. A arte do teste de software por si só deu à indústria um guia duradouro e fundamental para um dos tópicos mais importantes da computação: como você garante que todo o software que você produz faça o que foi projetado para fazer e - tão importante quanto - não faz o que não deveria fazer?

A edição que você está lendo hoje mantém a filosofia fundamental estabelecida por Myers há mais de três décadas. Mas atualizamos os exemplos para incluir linguagens de programação mais atuais e abordamos tópicos que ainda não eram tópicos quando Myers escreveu a primeira edição: programação Web, comércio eletrônico, programação e teste Extreme (Agile) e aplicativos de teste para dispositivos móveis dispositivos.

viii Prefácio

Ao longo do caminho, nunca perdemos de vista o fato de que um novo clássico deve permanecer fiel às suas raízes, então nossa versão também oferece uma filosofia de teste de software e um processo que funciona em plataformas de hardware e software atuais e futuras imprevisíveis. Esperamos que a terceira edição de *The Art of Software Testing* também abranja uma geração de designers e desenvolvedores de software.

Introdução

Na época em que este livro foi publicado pela primeira vez, em 1979, era uma regra geral que, em um projeto de programação típico, aproximadamente 50% do tempo decorrido e mais de 50% do custo total foram gastos no teste do programa ou sistema que está sendo desenvolvido.

Hoje, um terço de século e duas atualizações de livros depois, o mesmo acontece. Existem novos sistemas de desenvolvimento, linguagens com ferramentas integradas e programadores acostumados a desenvolver mais rapidamente. Mas o teste continua a desempenhar um papel importante em qualquer projeto de desenvolvimento de software.

Dados esses fatos, você pode esperar que, a essa altura, os testes de programas tenham sido refinados em uma ciência exata. Isso está longe de ser o caso. Dentro Na verdade, parece ser menos conhecido sobre teste de software do que sobre qualquer outro aspecto do desenvolvimento de software. Além disso, os testes têm sido um assunto fora de moda; foi assim quando este livro foi publicado pela primeira vez e, infelizmente, isso não mudou. Hoje existem mais livros e artigos sobre teste de software – o que significa que, pelo menos, o tópico tem maior visibilidade do que quando este livro foi publicado pela primeira vez – mas o teste permanece entre as “artes obscuras” do desenvolvimento de software.

Isso seria motivo mais do que suficiente para atualizar este livro sobre a arte do teste de software, mas temos motivações adicionais. Em várias ocasiões, ouvimos professores e assistentes de ensino dizerem: “Nossos alunos se formam e entram na indústria sem nenhum conhecimento substancial de como testar um programa. Além disso, raramente temos algum conselho a oferecer em nossos cursos introdutórios sobre como um aluno deve testar e depurar seus exercícios.”

Assim, o objetivo desta edição atualizada de *The Art of Software Testing* é o mesmo de 1979 e 2004: preencher essas lacunas de conhecimento para o programador profissional e o estudante de ciência da computação. Como o título indica, o livro é uma discussão prática, e não teórica, do assunto, completa com linguagem atualizada e discussões de processo.

x Introdução

Embora seja possível discutir o teste de programa em uma veia teórica, este livro pretende ser um manual prático, "com os dois pés no chão".

Assim, muitos assuntos relacionados a testes de programas, como a ideia de matemática provando a exatidão de um programa, foram propositadamente excluídos.

O Capítulo 1 "atribui" um pequeno teste de autoavaliação que todo leitor deveria fazer antes de continuar lendo. Acontece que a informação prática mais importante que você deve entender sobre o teste do programa é um conjunto de questões filosóficas e econômicas; eles são discutidos no Capítulo 2. O Capítulo 3 apresenta o importante conceito de inspeções ou inspeções de código não baseadas em computador. Em vez de focar a atenção nos aspectos processuais ou gerenciais desse conceito, como fazem a maioria dessas discussões, este capítulo o aborda de um ponto de vista técnico de como encontrar erros.

O leitor alerta perceberá que o componente mais importante no pacote de truques de um testador de programa é o conhecimento de como escrever casos de teste eficazes; este é o assunto do Capítulo 3. O Capítulo 4 discute o teste de módulos individuais ou sub-rotinas, seguido no Capítulo 5 pelo teste de entidades maiores. O Capítulo 6 aborda o conceito de teste de usuário ou usabilidade, um componente de teste de software que sempre foi importante, mas é ainda mais relevante hoje devido ao advento de softwares mais complexos direcionados a um público cada vez maior. O Capítulo 7 oferece alguns conselhos práticos sobre depuração de programas, enquanto o Capítulo 8 aprofunda os conceitos de teste de programação extrema com ênfase no que veio a ser chamado de "ambiente ágil". O Capítulo 9 mostra como usar outros recursos de teste de programa, que são detalhados em outras partes deste livro, com programação da Web, incluindo sistemas de comércio eletrônico e os novos e altamente interativos sites de redes sociais. O Capítulo 10 descreve como testar software desenvolvido para o ambiente móvel.

Dirigimos este livro para três públicos principais. Primeiro, o programador profissional. Embora esperemos que nem tudo neste livro seja uma informação nova para esse público, acreditamos que isso aumentará o conhecimento do profissional sobre técnicas de teste. Se o material permitir que este grupo detecte apenas mais um bug em um programa, o preço do livro terá sido recuperado muitas vezes.

O segundo público é o gerente de projeto, que se beneficiará das informações práticas do livro sobre o gerenciamento do processo de teste.

O terceiro público é o estudante de programação e ciência da computação, e nosso objetivo para eles é duplo: expô-los aos problemas de

teste de programa e fornecer um conjunto de técnicas eficazes. Para este terceiro grupo, sugerimos que o livro seja utilizado como um suplemento em cursos de programação, de modo que os alunos sejam expostos ao tema de teste de software logo no início de sua formação.

1 Uma Autoavaliação

Teste

Desde que essenciais dificuldades fáceis de operar veio há mais de 30 anos, testes de software

O teste de software é mais difícil devido à vasta gama de linguagens de programação, sistemas operacionais e plataformas de hardware que evoluíram nas décadas seguintes. E enquanto relativamente poucas pessoas usavam computadores na década de 1970, hoje praticamente ninguém pode completar um dia de trabalho sem usar um computador. Não apenas computadores existem em sua mesa, mas um "computador" e, consequentemente, software, está presente em quase todos os dispositivos que usamos. Apenas tente pensar nos dispositivos hoje em que a sociedade depende e que não são acionados por software. Claro que existem alguns - martelos e carrinhos de mão

vêm à mente, mas a grande maioria usa algum tipo de software para operar.

O software é difundido, o que aumenta o valor de testá-lo. As próprias máquinas são centenas de vezes mais poderosas e menores do que os primeiros dispositivos, e o conceito atual de "computador" é muito mais amplo e difícil de definir. Televisores, telefones, sistemas de jogos e automóveis contêm computadores e software de computador e, em alguns casos, podem até ser considerados computadores.

Portanto, o software que escrevemos hoje toca potencialmente milhões de pessoas, seja permitindo-lhes fazer seu trabalho de forma eficaz e eficiente, ou causando-lhes uma frustração incalculável e custando-lhes na forma de trabalho perdido ou perda de negócios. Isso não quer dizer que o software seja mais importante hoje do que era quando a primeira edição deste livro foi publicada, mas é seguro dizer que os computadores – e o software que os conduz – certamente afetam mais pessoas e mais empresas agora do que nunca. antes da.

2 A arte do teste de software

O teste de software também é mais fácil, em alguns aspectos, porque o conjunto de software e sistemas operacionais é muito mais sofisticado do que no passado, fornecendo rotinas intrínsecas e bem testadas que podem ser incorporadas a aplicativos sem a necessidade de um programador desenvolver eles do zero. Interfaces gráficas de usuário (GUIs), por exemplo, podem ser construídas a partir de bibliotecas de uma linguagem de desenvolvimento e, como são objetos pré-programados que foram depurados e testados anteriormente, a necessidade de testá-los como parte de um aplicativo personalizado é muito reduzida.

E, apesar da infinidade de tipos de teste de software disponíveis no mercado hoje, muitos desenvolvedores parecem ter uma atitude contrária aos testes extensivos. Melhores ferramentas de desenvolvimento, GUIs pré-testadas e a pressão de prazos apertados em um ambiente de desenvolvimento cada vez mais complexo podem levar a evitar todos os protocolos de teste, exceto os mais óbvios. Enquanto os impactos de baixo nível de bugs podem incomodar apenas o usuário final, os piores impactos podem resultar em grandes perdas financeiras ou até mesmo causar danos às pessoas. Os procedimentos neste livro podem ajudar designers, desenvolvedores e gerentes de projeto a entender melhor o valor de testes abrangentes e fornecer diretrizes para ajudá-los a atingir as metas de teste exigidas.

O teste de software é um processo, ou uma série de processos, projetado para garantir que o código do computador faça o que foi projetado para fazer e, inversamente, que não faça nada não intencional. O software deve ser previsível e consistente, não apresentando surpresas aos usuários. Neste livro, veremos muitas abordagens para atingir esse objetivo.

Agora, antes de começarmos o livro, gostaríamos que você fizesse um pequeno exame. Queremos que você escreva um conjunto de casos de teste – conjuntos específicos de dados – para testar adequadamente um programa relativamente simples. Crie um conjunto de dados de teste para o programa – dados que o programa deve manipular corretamente para ser considerado um programa bem-sucedido. Segue uma descrição do programa:

O programa lê três valores inteiros de uma caixa de diálogo de entrada. Os três valores representam os comprimentos dos lados de um triângulo. O programa exibe uma mensagem informando se o triângulo é escaleno, isósceles ou equilátero.

Lembre-se que um triângulo escaleno é aquele em que não há dois lados iguais, enquanto um triângulo isósceles tem dois lados iguais e um triângulo equilátero tem três lados de igual comprimento. Além disso, os ângulos opostos ao

lados iguais em um triângulo isósceles também são iguais (segue também que os lados opostos a ângulos iguais em um triângulo são iguais), e todos os ângulos em um triângulo equilátero são iguais.

Avalie seu conjunto de casos de teste usando-o para responder ao seguinte perguntas. Dê a si mesmo um ponto para cada resposta sim.

1. Você tem um caso de teste que representa um triângulo escaleno válido?
(Observe que casos de teste como 1, 2, 3 e 2, 5, 10 não garantem uma resposta sim porque um triângulo com essas dimensões não é válido.)
2. Você tem um caso de teste que representa um triângulo equilátero válido?
3. Você tem um caso de teste que representa um triângulo isósceles válido?
(Observe que um caso de teste representando 2, 2, 4 não contaria porque não é um triângulo válido.)
4. Você tem pelo menos três casos de teste que representam triângulos isósceles válidos, de modo que você tentou todas as três permutações de dois lados iguais (como 3, 3, 4; 3, 4, 3; e 4, 3, 3) ?
5. Você tem um caso de teste em que um lado tem valor zero?
6. Você tem um caso de teste em que um lado tem um valor negativo?
7. Você tem um caso de teste com três inteiros maiores que zero tal que a soma de dois dos números seja igual ao terceiro? (Ou seja, se o programa dissesse que 1, 2, 3 representa um triângulo escaleno, ele conteria um bug.)
8. Você tem pelo menos três casos de teste na categoria 7 de tal forma que você tentou todas as três permutações onde o comprimento de um lado é igual à soma dos comprimentos dos outros dois lados (por exemplo, 1, 2, 3; 1 , 3, 2; e 3, 1, 2)?
9. Você tem um caso de teste com três inteiros maiores que zero tal que a soma de dois dos números seja menor que o terceiro (como 1, 2, 4 ou 12, 15, 30)?
10. Você tem pelo menos três casos de teste na categoria 9 de tal forma que tentou todas as três permutações (por exemplo, 1, 2, 4; 1, 4, 2; e 4, 1, 2)?
11. Você tem um caso de teste em que todos os lados são zero (0, 0, 0)?
12. Você tem pelo menos um caso de teste especificando valores não inteiros (como 2,5, 3,5, 5,5)?
13. Você tem pelo menos um caso de teste especificando o número errado de valores (dois ao invés de três inteiros, por exemplo)?
14. Para cada caso de teste, você especificou a saída esperada do programa além dos valores de entrada?

4 A arte do teste de software

É claro que um conjunto de casos de teste que satisfaça essas condições não garante que você encontrará todos os erros possíveis, mas como as questões de 1 a 13 representam erros que realmente ocorreram em diferentes versões deste programa, um teste adequado desse programa deve expor pelo menos esses erros.

Agora, antes de se preocupar com sua pontuação, considere o seguinte: em nossa experiência, programadores profissionais altamente qualificados pontuam, em média, apenas 7,8 de 14 possíveis. Se você se saiu melhor, parabéns; se não, estamos aqui para ajudar.

O objetivo do exercício é ilustrar que testar mesmo um programa trivial como este não é uma tarefa fácil. Dado que isso é verdade, considere a dificuldade de testar um sistema de controle de tráfego aéreo de 100.000 declarações, um compilador ou mesmo um programa de folha de pagamento comum. O teste também se torna mais difícil com as linguagens orientadas a objetos, como Java e Cþþ. Por exemplo, seus casos de teste para aplicativos criados com essas linguagens devem expor erros associados à instanciação de objetos e gerenciamento de memória.

Ao trabalhar com este exemplo, pode parecer que testar minuciosamente um programa complexo do mundo real seria impossível. Não tão! Embora a tarefa possa ser assustadora, o teste de programa adequado é uma parte muito necessária — e alcançável — do desenvolvimento de software, como você aprenderá neste livro.

2

A Psicologia e a Economia da Teste de software

O teste de software é uma tarefa técnica, sim, mas também envolve algumas implicações psicológicas.

Em um mundo ideal, gostaríamos de testar todas as permutações possíveis de um programa. Na maioria dos casos, no entanto, isso simplesmente não é possível. Mesmo um programa aparentemente simples pode ter centenas ou milhares de combinações possíveis de entrada e saída. Criar casos de teste para todas essas possibilidades é impraticável. O teste completo de um aplicativo complexo levaria muito tempo e exigiria muitos recursos humanos para ser economicamente viável.

Além disso, o testador de software precisa da atitude adequada (talvez "visão" seja uma palavra melhor) para testar com sucesso um aplicativo de software. Em alguns casos, a atitude do testador pode ser mais importante do que o próprio processo em si. Portanto, iniciaremos nossa discussão sobre teste de software com essas questões antes de nos aprofundarmos na natureza mais técnica do tópico.

A psicologia dos testes

Uma das principais causas de testes de aplicativos ruins é o fato de que a maioria dos programadores começa com uma definição falsa do termo. Eles podem dizer:

"Teste é o processo de demonstrar que os erros não estão presentes." "O objetivo do teste é mostrar que um programa executa corretamente suas funções pretendidas." "Teste é o processo de estabelecer confiança de que um programa faz o que é suposto fazer."

6 A Arte do Teste de Software

Essas definições estão de cabeça para baixo.

Quando você testa um programa, deseja adicionar algum valor a ele. Agregar valor por meio de testes significa aumentar a qualidade ou a confiabilidade do programa.

Aumentar a confiabilidade do programa significa encontrar e remover erros.

Portanto, não teste um programa para mostrar que ele funciona; em vez disso, comece com a suposição de que o programa contém erros (uma suposição válida para quase todos os programas) e, em seguida, teste o programa para encontrar o maior número possível de erros.

Assim, uma definição mais apropriada é esta:

Teste é o processo de executar um programa com a intenção de encontrar erros.

Embora isso possa soar como um jogo de semântica sutil, é realmente uma distinção importante. Compreender a verdadeira definição de teste de software pode fazer uma grande diferença no sucesso de seus esforços.

Os seres humanos tendem a ser altamente orientados para objetivos, e estabelecer o objetivo adequado tem um efeito psicológico importante sobre eles. Se nosso objetivo é demonstrar que um programa não tem erros, então seremos direcionados subconscientemente para esse objetivo; ou seja, tendemos a selecionar dados de teste com baixa probabilidade de causar falha no programa. Por outro lado, se nosso objetivo é demonstrar que um programa possui erros, nossos dados de teste terão maior probabilidade de encontrar erros. A última abordagem agregará mais valor ao programa do que a anterior.

Essa definição de teste tem inúmeras implicações, muitas das quais estão espalhadas por todo este livro. Por exemplo, isso implica que o teste é um processo destrutivo, até mesmo sádico, o que explica por que a maioria das pessoas acha difícil. Isso pode ir contra a nossa vontade; com boa sorte, a maioria de nós tem uma visão de vida construtiva, em vez de destrutiva. A maioria das pessoas tende a fazer objetos em vez de rasgá-los. A definição também tem implicações sobre como os casos de teste (dados de teste) devem ser projetados e quem deve e quem não deve testar um determinado programa.

Outra maneira de reforçar a definição adequada de teste é analisar o uso das palavras "bem-sucedido" e "fracassado" — em particular, seu uso pelos gerentes de projeto na categorização dos resultados dos casos de teste. A maioria dos gerentes de projeto refere-se a um caso de teste que não encontrou um erro como "execução de teste bem-sucedida", enquanto um teste que descobre um novo erro é geralmente chamado de "fracasso".

Mais uma vez, isso está de cabeça para baixo. "Sem sucesso" denota algo indesejável ou decepcionante. Para o nosso modo de pensar, um bem construído e

A Psicologia e Economia do Teste de Software 7

o teste de software executado é bem-sucedido quando encontra erros que podem ser corrigidos.

Esse mesmo teste também é bem-sucedido quando eventualmente estabelece que não há mais erros a serem encontrados. O único teste malsucedido é aquele que não examina adequadamente o software; e, na maioria dos casos, um teste que não encontrou erros provavelmente seria considerado malsucedido, já que o conceito de um programa sem erros é basicamente irrealista.

Um caso de teste que encontra um novo erro dificilmente pode ser considerado malsucedido; em vez disso, provou ser um investimento valioso. Um caso de teste malsucedido é aquele que faz com que um programa produza o resultado correto sem encontrar nenhum erro.

Considere a analogia de uma pessoa que visita um médico por causa de uma sensação geral de mal-estar. Se o médico faz alguns exames laboratoriais que não localizam o problema, não chamamos os exames laboratoriais de "bem-sucedidos"; foram testes malsucedidos, pois o patrimônio líquido do paciente foi reduzido pelas altas taxas de laboratório, o paciente ainda está doente e o paciente pode questionar a capacidade do médico como diagnosticador. No entanto, se um exame laboratorial determinar que o paciente tem úlcera péptica, o exame é bem-sucedido porque o médico já pode iniciar o tratamento adequado. Assim, a profissão médica parece usar essas palavras no sentido apropriado. A analogia, claro, é que devemos pensar no programa, quando começamos a testá-lo, como o paciente doente.

Um segundo problema com definições como "teste é o processo de demonstrar que os erros não estão presentes" é que tal objetivo é impossível de ser alcançado para praticamente todos os programas, mesmo programas triviais.

Mais uma vez, estudos psicológicos nos dizem que as pessoas têm um desempenho ruim quando partem para uma tarefa que sabem ser inviável ou impossível. Por exemplo, se você fosse instruído a resolver as palavras cruzadas no New York Times de domingo em 15 minutos, provavelmente alcançaria pouco ou nenhum progresso após 10 minutos, porque, se você for como a maioria das pessoas, estaria resignado ao fato de que a tarefa parece impossível. Se lhe pedissem uma solução em quatro horas, no entanto, poderíamos esperar ver mais progresso nos 10 minutos iniciais. Definir o teste de programa como o processo de descoberta de erros em um programa torna uma tarefa viável, superando assim esse problema psicológico.

Um terceiro problema com as definições comuns como "teste é o processo de demonstrar que um programa faz o que deveria fazer" é que programas que fazem o que deveriam fazer ainda podem conter erros. Ou seja, um erro está claramente presente se um programa não faz o que deveria fazer; mas os erros também estão presentes se um programa faz o que não deveria fazer. Considere o programa triangular do Capítulo 1. Mesmo se

8 A Arte do Teste de Software

pudesse demonstrar que o programa distingue corretamente entre todos os triângulos escalenos, isósceles e equiláteros, o programa ainda estaria em erro se fizer algo que não deveria fazer (como representar 1, 2, 3 como um triângulo escaleno ou dizendo que 0, 0, 0 representa um equilátero triângulo). É mais provável que descubramos a última classe de erros se ver o teste de programa como o processo de encontrar erros do que se o víssemos como o processo de mostrar que um programa faz o que deve fazer.

Para resumir, o teste de programa é visto mais apropriadamente como o processo destrutivo de tentar encontrar os erros em um programa (cuja presença é assumido). Um caso de teste bem-sucedido é aquele que promove o progresso nessa direção, fazendo com que o programa falhe. Claro, você eventualmente quer usar teste de programa para estabelecer algum grau de confiança de que um programa faz o que deve fazer e não faz o que não deve fazer, mas esse objetivo é melhor alcançado por uma exploração diligente de erros.

Considere alguém se aproximando de você com a alegação de que "meu programa é perfeito" (ou seja, livre de erros). A melhor maneira de estabelecer alguma confiança neste alegação é tentar refutá-la, isto é, tentar encontrar imperfeições em vez de apenas confirme se o programa funciona corretamente para algum conjunto de dados de entrada.

A economia do teste

Dada nossa definição de teste de programa, um próximo passo apropriado é determinar se é possível testar um programa para encontrar todos os seus erros. Nós mostrará que a resposta é negativa, mesmo para programas triviais. Dentro Em geral, é impraticável, muitas vezes impossível, encontrar todos os erros em um programa. Este problema fundamental terá, por sua vez, implicações para a economia de testes, suposições que o testador terá que fazer sobre o programa e a maneira como os casos de teste são projetados.

Para combater os desafios associados aos testes econômicos, você deve estabelecer algumas estratégias antes de começar. Duas das estratégias mais prevalentes incluem teste de caixa preta e teste de caixa branca, que exploraremos nas próximas duas seções.

Teste de caixa preta

Uma estratégia de teste importante é o teste de caixa preta (também conhecido como teste orientado a dados ou orientado a entrada/saída). Para usar este método, veja o programa como uma caixa preta. Seu objetivo é estar completamente desocupado com o

A Psicologia e Economia do Teste de Software 9

comportamento interno e estrutura do programa. Em vez disso, concentre-se em encontrar circunstâncias nas quais o programa não se comporte de acordo com suas especificações.

Nesta abordagem, os dados de teste são derivados exclusivamente das especificações (ou seja, sem tirar proveito do conhecimento da estrutura interna do programa).

Se você quiser usar essa abordagem para encontrar todos os erros no programa, o critério é o teste de entrada exaustivo, fazendo uso de todas as condições de entrada possíveis como um caso de teste. Por quê? Se você tentou três casos de teste de triângulo equilátero para o programa triângulo, isso de forma alguma garante a detecção correta de todos os triângulos equiláteros. O programa pode conter uma verificação especial para os valores 3842, 3842, 3842 e denotar tal triângulo como um triângulo escaleno. Como o programa é uma caixa preta, a única maneira de ter certeza de detectar a presença de tal instrução é tentar todas as condições de entrada.

Para testar o programa de triângulos exaustivamente, você teria que criar casos de teste para todos os triângulos válidos até o tamanho inteiro máximo da linguagem de desenvolvimento. Isso em si é um número astronômico de casos de teste, mas não é de forma alguma exaustivo: não encontraria erros onde o programa dissesse que 3, 4, 5 é um triângulo escaleno e que 2, A, 2 é um triângulo isósceles.

Para ter certeza de encontrar todos esses erros, você deve testar usando não apenas todas as entradas válidas, mas todas as entradas possíveis. Portanto, para testar exaustivamente o programa triângulo, você teria que produzir virtualmente um número infinito de casos de teste, o que, obviamente, não é possível.

Se isso parece difícil, o teste exaustivo de entrada de programas maiores é ainda mais problemático. Considere tentar um teste de caixa preta exaustivo de um compilador C++. Você não apenas teria que criar casos de teste representando todos os programas C++ válidos (novamente, virtualmente um número infinito), mas você teria que criar casos de teste para todos os programas C++ inválidos (um número infinito) para garantir que o compilador os detectasse como sendo inválidos. Ou seja, o compilador precisa ser testado para garantir que não faça o que não deveria fazer — por exemplo, compilar com sucesso um programa sintaticamente incorreto.

O problema é ainda mais oneroso para programas baseados em transações, como aplicativos de banco de dados. Por exemplo, em um aplicativo de banco de dados como um sistema de reservas de companhias aéreas, a execução de uma transação (como uma consulta de banco de dados ou uma reserva para um voo de avião) depende do que aconteceu nas transações anteriores. Portanto, você não apenas teria que tentar todas as transações válidas e inválidas exclusivas, mas também todas as possíveis sequências de transações.

10 A arte do teste de software

Esta discussão mostra que o teste de entrada exaustivo é impossível. Duas implicações importantes disso: (1) Você não pode testar um programa para garantir que ele esteja livre de erros; e (2) uma consideração fundamental no teste do programa é a economia. Assim, como o teste exaustivo está fora de questão, o objetivo deve ser maximizar o rendimento do investimento em teste, maximizando o número de erros encontrados por um número finito de casos de teste.

Fazer isso envolverá, entre outras coisas, ser capaz de espiar dentro do programa e fazer certas suposições razoáveis, mas não herméticas, sobre o programa (por exemplo, se o programa triângulo detectar 2, 2, 2 como um triângulo equilátero, parece razoável que fará o mesmo para 3, 3, 3). Isso fará parte da estratégia de design do caso de teste no Capítulo 4.

Teste de caixa branca

Outra estratégia de teste, o teste de caixa branca (ou orientado por lógica), permite examinar a estrutura interna do programa. Essa estratégia deriva dados de teste de um exame da lógica do programa (e muitas vezes, infelizmente, negligenciando a especificação).

O objetivo neste ponto é estabelecer para esta estratégia o teste de entrada analógico para exaustivo na abordagem caixa preta. Fazer com que cada instrução do programa seja executada pelo menos uma vez pode parecer a resposta, mas não é difícil mostrar que isso é altamente inadequado. Sem entrar em detalhes aqui, já que esse assunto é discutido com mais profundidade no Capítulo 4, o analógico é geralmente considerado um teste de caminho exaustivo. Ou seja, se você executar, por meio de casos de teste, todos os caminhos possíveis de fluxo de controle através do programa, possivelmente o programa foi completamente testado.

Há duas falhas nesta afirmação, no entanto. Uma é que o número de caminhos lógicos únicos através de um programa pode ser astronomicamente grande. Para ver isso, considere o programa trivial representado na Figura 2.1. O diagrama é um gráfico de fluxo de controle. Cada nó ou círculo representa um segmento de instruções que são executadas sequencialmente, possivelmente terminando com uma instrução de ramificação. Cada aresta ou arco representa uma transferência de controle (ramificação) entre segmentos. O diagrama, então, descreve um programa de 10 a 20 instruções que consiste em um loop DO que itera até 20 vezes. Dentro do corpo do loop DO há um conjunto de instruções IF aninhadas. Determinar o número de caminhos lógicos exclusivos é o mesmo que determinar o número total de maneiras únicas de se mover do ponto a para o ponto b (assumindo que todas as decisões no programa são independentes umas das outras). Este número é aproximadamente 1014, ou

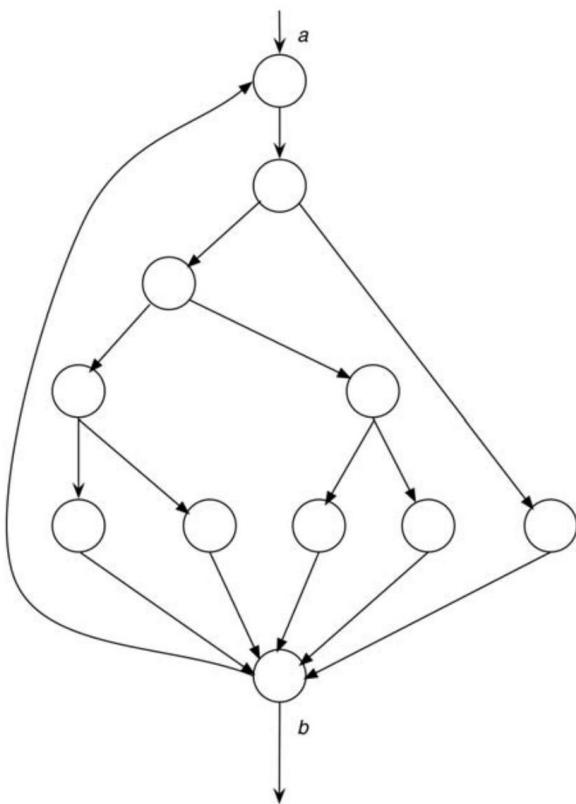


FIGURA 2.1 Gráfico de fluxo de controle de um programa pequeno.

100 trilhões. É calculado a partir de $520 \times 519 \times \dots \times 51$ do número de , onde 5 é o caminhos através do corpo do loop. A maioria das pessoas tem dificuldade em visualizar tal número, então considere desta forma: Se você pudesse escrever, executar e verificar um caso de teste a cada cinco minutos, levaria aproximadamente 1 bilhão de anos para tentar todos os caminhos. Se você fosse 300 vezes mais rápido, completando um teste uma vez por segundo, você poderia completar o trabalho em 3,2 milhões de anos, mais ou menos alguns anos bissexto e séculos.

Obviamente, em programas reais, cada decisão não é independente de todas as outras decisões, o que significa que o número de caminhos de execução possíveis seria um pouco menor. Por outro lado, os programas reais são muito maiores do que o programa simples representado na Figura 2.1. Portanto, o teste de caminho exaustivo, como o teste de entrada exaustivo, parece ser impraticável, se não impossível.

12 A Arte do Teste de Software

A segunda falha na afirmação "teste de caminho exaustivo significa um teste completo" é que cada caminho em um programa pode ser testado, mas o programa ainda pode estar carregado de erros. Há três explicações para isso.

A primeira é que um teste de caminho exaustivo não garante de forma alguma que um programa corresponda à sua especificação. Por exemplo, se lhe pedissem para escrever um rotina de classificação de ordem ascendente, mas erroneamente produziu uma rotina de classificação de ordem decrescente, o teste de caminho exaustivo seria de pouco valor; a programa ainda tem um bug: é o programa errado, pois não atende especificação.

Em segundo lugar, um programa pode estar incorreto devido à falta de caminhos. Exaustivo o teste de caminho, é claro, não detectaria a ausência de caminhos necessários.

Terceiro, um teste de caminho exaustivo pode não revelar erros de sensibilidade de dados. Há muitos exemplos de tais erros, mas um simples deve ser suficiente.

Suponha que em um programa você tenha que comparar dois números para convergência, isto é, para ver se a diferença entre os dois números é menor que algum valor predeterminado. Por exemplo, você pode escrever um estado Java IF

ment como

```
se (ab<c)
    System.out.println("ab<c");
```

Claro, a instrução contém um erro porque deve comparar c ao valor absoluto de ab. A detecção deste erro, no entanto, depende sobre os valores usados para a e b e não seriam necessariamente detectados por apenas executando todos os caminhos através do programa.

Em conclusão, embora o teste de entrada exaustivo seja superior ao teste de caminho exaustivo, nenhum deles se mostra útil porque ambos são inviáveis.

Talvez, então, existam maneiras de combinar elementos de caixa-preta e testes de caixa branca para derivar uma estratégia de teste razoável, mas não hermética. Este assunto é aprofundado no Capítulo 4.

Princípios de teste de software

Continuando com a premissa principal deste capítulo, que as considerações mais importantes no teste de software são questões de psicologia, podemos identificar um conjunto de princípios ou diretrizes de teste vitais. A maioria desses princípios pode parecer óbvia, mas muitas vezes são negligenciados. Tabela 2.1 resume esses princípios importantes, e cada um é discutido em mais detalhes nos parágrafos que se seguem.

TABELA 2.1 Diretrizes de Teste do Programa Vital

Princípio Número	Princípio
1	Uma parte necessária de um caso de teste é uma definição do saída ou resultado.
2	Um programador deve evitar tentar testar seu próprio programa.
3	Uma organização de programação não deve testar seus próprios programas.
4	Qualquer processo de teste deve incluir uma inspeção completa do resultados de cada teste.
5	Os casos de teste devem ser escritos para condições de entrada que são inválidas e inesperado, bem como para aqueles que são válidos e esperados.
6	Examinando um programa para ver se ele não faz o que deveria fazer é apenas metade da batalha; a outra metade é ver se o programa faz o que não deveria fazer.
7	Evite casos de teste descartáveis, a menos que o programa seja realmente um programa descartável.
8	Não planeje um esforço de teste sob a suposição tácita de que não erros serão encontrados.
9	A probabilidade da existência de mais erros em uma seção de um programa é proporcional ao número de erros já encontrados naquela seção.
10	O teste é uma tarefa extremamente criativa e intelectualmente desafiadora.

Princípio 1: Uma parte necessária de um caso de teste é uma definição do saída ou resultado esperado.

Este princípio, embora óbvio, quando negligenciado é a causa de um dos erros mais frequentes em testes de programas. Novamente, é algo que se baseia na psicologia humana. Se o resultado esperado de um caso de teste não foi pré-definido, as chances são de que um mas errôneo, o resultado será interpretado como um resultado correto devido a o fenômeno do "olho vendo o que quer ver". palavras, apesar da definição destrutiva adequada de teste, há ainda um desejo subconsciente de ver o resultado correto. Uma maneira de

14 A Arte do Teste de Software

combater isso é encorajar um exame detalhado de todos os resultados especificando com precisão, com antecedência, a saída esperada do programa.

Portanto, um caso de teste deve consistir em dois componentes:

1. Uma descrição dos dados de entrada para o programa.
2. Uma descrição precisa da saída correta do programa para esse conjunto de dados de entrada.

Um problema pode ser caracterizado como um fato ou grupo de fatos para que não temos explicação aceitável, que parecem incomuns, ou que não se encaixam com nossas expectativas ou preconceitos. Deveria ser óbvio que algumas crenças anteriores são necessárias para que algo apareça problemático. Se não há expectativas, não pode haver surpresas.

Princípio 2: Um programador deve evitar tentar testar seu próprio programa.

Qualquer escritor sabe – ou deveria saber – que é uma má ideia tentar editar ou revisar seu próprio trabalho. Eles sabem o que o peça deve dizer, portanto, pode não reconhecer quando diz o contrário. E eles realmente não querem encontrar erros em seu próprio trabalho. O mesmo se aplica aos autores de software.

Outro problema surge com uma mudança de foco em um projeto de software. Depois que um programador projetou e codificou construtivamente um programa, é extremamente difícil mudar de repente a perspectiva para olhar no programa com um olhar destrutivo.

Como muitos proprietários sabem, remover o papel de parede (um destrutivo processo) não é fácil, mas é quase insuportavelmente deprimente se fosse suas mãos que penduraram o papel em primeiro lugar. Da mesma forma, a maioria dos programadores não pode testar efetivamente seus próprios programas porque eles não conseguem mudar as engrenagens mentais para tentar expor erros. Além disso, um programador pode inconscientemente evitar encontrar erros por medo de represálias de colegas ou de um supervisor, um cliente, ou o proprietário do programa ou sistema que está sendo desenvolvido.

Além dessas questões psicológicas, há um segundo problema significativo: o programa pode conter erros devido à incompreensão do programador quanto à declaração ou especificação do problema. Se este for o caso, é provável que o programador carregue o mesmo mal-entendido em testes de seu próprio programa.

Isso não significa que seja impossível para um programador testar seu próprio programa. Em vez disso, implica que o teste é mais eficaz e bem-sucedido se outra pessoa o fizer. No entanto, como iremos

discutido com mais detalhes no Capítulo 3, os desenvolvedores podem ser membros valiosos da equipe de teste quando a especificação do programa e o próprio código do programa estão sendo avaliados.

Observe que este argumento não se aplica à depuração (corrigindo erros conhecidos); a depuração é mais eficientemente executada pelo programador original.

Princípio 3: Uma organização de programação não deve testar seus próprios programas.

O argumento aqui é semelhante ao feito no princípio anterior. Um projeto ou organização de programação é, em muitos sentidos, uma organização viva com problemas psicológicos semelhantes aos de programadores individuais. Além disso, na maioria dos ambientes, uma organização de programação ou um gerente de projeto é amplamente medido em a capacidade de produzir um programa em uma determinada data e por um determinado custo. Uma razão para isso é que é fácil medir objetivos de tempo e custo, enquanto é extremamente difícil quantificar a confiabilidade de um programa. Portanto, é difícil para uma organização de programação ser objetivo ao testar seus próprios programas, pois o processo de teste, se abordado com a definição adequada, pode ser visto como decrescente a probabilidade de cumprir o cronograma e os objetivos de custo.

Novamente, isso não quer dizer que é impossível para um programador organização para encontrar alguns de seus erros, porque as organizações não fazer isso com algum grau de sucesso. Pelo contrário, implica que é mais econômico que os testes sejam realizados por uma parte objetiva e independente.

Princípio 4: Qualquer processo de teste deve incluir uma inspeção completa dos resultados de cada teste.

Este é provavelmente o princípio mais óbvio, mas, novamente, é algo que muitas vezes é esquecido. Vimos vários experimentos que mostram que muitos sujeitos falharam em detectar certos erros, mesmo quando os sintomas desses erros foram claramente observáveis nas listas de saída. Dito de outra forma, os erros encontrados em testes posteriores eram frequentemente perdido nos resultados de testes anteriores.

Princípio 5: Os casos de teste devem ser escritos para condições de entrada que são inválidas e inesperadas, bem como para aquelas que são válidas e esperado.

Há uma tendência natural ao testar um programa para se concentrar nas condições de entrada válidas e esperadas, negligenciando o

16 A Arte do Teste de Software

condições inválidas e inesperadas. Por exemplo, essa tendência aparece com frequência no teste do programa triângulo no Capítulo 1.

Poucas pessoas, por exemplo, alimentam o programa com os números 1, 2, 5 para garantir que o programa não interprete erroneamente isso como um triângulo equalateral em vez de um triângulo escaleno. Além disso, muitos erros que são descobertos de repente no software de produção aparecem quando ele é usado de alguma maneira nova ou inesperada. É difícil, se não impossível, definir todos os casos de uso para teste de software. Portanto, casos de teste que representam condições de entrada inesperadas e inválidas parecem ter um rendimento de detecção de erro maior do que casos de teste para condições de entrada válidas.

Princípio 6: Examinar um programa para ver se ele não faz o que deveria fazer é apenas metade da batalha; a outra metade é ver se o programa faz o que não deveria fazer.

Este é um corolário do princípio anterior. Os programas devem ser examinados quanto a efeitos colaterais indesejados. Por exemplo, um programa de folha de pagamento que produz os contracheques corretos ainda é um programa errôneo se também produzir cheques extras para empregados inexistentes ou se sobreescrivar o primeiro registro do arquivo de pessoal.

Princípio 7: Evite casos de teste descartáveis, a menos que o programa seja realmente um programa descartável.

Este problema é visto mais frequentemente com sistemas interativos para testar programas. Uma prática comum é sentar em um terminal e inventar casos de teste em tempo real e, em seguida, enviar esses casos de teste por meio do programa. A grande questão é que os casos de teste representam um investimento valioso que, nesse ambiente, desaparece após a conclusão do teste. Sempre que o programa tiver que ser testado novamente (por exemplo, após corrigir um erro ou fazer uma melhoria), os casos de teste devem ser reinventados. Na maioria das vezes, como essa reinvenção exige uma quantidade considerável de trabalho, as pessoas tendem a evitá-la. Portanto, o reteste do programa raramente é tão rigoroso quanto o teste original, o que significa que, se a modificação fizer com que uma parte funcional do programa falhe, esse erro geralmente não é detectado. Salvar casos de teste e executá-los novamente após alterações em outros componentes do programa é conhecido como teste de regressão.

Princípio 8: Não planeje um esforço de teste sob a suposição tácita que nenhum erro será encontrado.

Este é um erro que os gerentes de projeto costumam cometer e é um sinal do uso da definição incorreta de teste - ou seja, a suposição de que

teste é o processo de mostrar que o programa funciona corretamente. Mais uma vez, a definição de teste é o processo de executar um programa com a intenção de encontrar erros. E deve ser óbvio de nossas discussões anteriores que é impossível desenvolver um programa que seja completamente livre de erros. Mesmo após testes extensivos e correção de erros, é seguro assumir que os erros ainda existem; eles simplesmente ainda não foram encontrados.

Princípio 9: A probabilidade da existência de mais erros em uma seção de um programa é proporcional ao número de erros já encontrados nessa seção.

Este fenômeno é ilustrado na Figura 2.2. À primeira vista esse conceito pode parecer absurdo, mas é um fenômeno presente em muitos programas. Por exemplo, se um programa consiste em dois módulos, classes ou subrotinas, A e B, e cinco erros foram encontrados no módulo A, e apenas um erro foi encontrado no módulo B, e se o módulo A não foi submetido propositalmente a um teste mais rigoroso, então este princípio nos diz que a probabilidade de mais erros no módulo A é maior do que a probabilidade de mais erros no módulo B.

Outra maneira de afirmar este princípio é dizer que os erros tendem a vir em grupos e que, no programa típico, algumas seções parecem ser muito mais propensas a erros do que outras, embora ninguém tenha fornecido uma boa explicação de por que isso ocorre. O fenômeno é útil porque nos dá uma visão ou feedback no processo de teste. Se uma seção específica de um programa parece ser muito mais propensa a erros do que outras seções, esse fenômeno nos diz

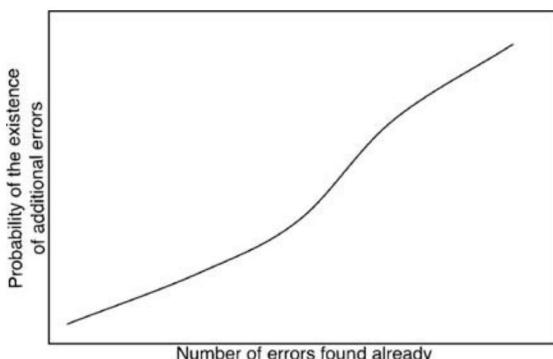


FIGURA 2.2 A surpreendente relação entre os erros remanescentes e os erros encontrados.

18 A Arte do Teste de Software

que, em termos de rendimento do nosso investimento em testes, testes adicionais os esforços são mais bem concentrados nesta seção propensa a erros.

Princípio 10: Testar é uma tarefa extremamente criativa e intelectualmente desafiadora.

Provavelmente é verdade que a criatividade necessária para testar um grande programa excede a criatividade necessária na concepção desse programa.

Já vimos que é impossível testar um programa o suficiente para garantir a ausência de todos os erros. Metodologias discutidas posteriormente neste livro ajudam você a desenvolver um conjunto razoável de testes

casos para um programa, mas essas metodologias ainda exigem uma quantidade significativa de criatividade.

Resumo

À medida que avança neste livro, tenha em mente estes importantes princípios de teste:

Teste é o processo de executar um programa com a intenção de encontrar erros.

O teste é mais bem-sucedido quando não é realizado pelo(s) desenvolvedor(es).

Um bom caso de teste é aquele que tem uma alta probabilidade de detectar um erro não descoberto.

Um caso de teste bem-sucedido é aquele que detecta um erro não descoberto.

Testes bem-sucedidos também incluem a definição cuidadosa da saída esperada como entrada.

Testes bem-sucedidos incluem estudar cuidadosamente os resultados dos testes.

3

Inspeções do programa, Percursos, e Avaliações

Por muitos anos, a maioria de nós na comunidade de programação trabalhou de acordo com as suposições de que os programas são escritos exclusivamente para execução em máquina, e não são destinados a serem lidos por pessoas, e que a única maneira de testar um programa é executá-lo em uma máquina. Essa atitude começou a mudar no início da década de 1970 por meio dos esforços dos desenvolvedores de programas que viram pela primeira vez o valor da leitura de código como parte de um regime abrangente de teste e depuração.

Hoje, nem todos os testadores de aplicativos de software leem código, mas o conceito de estudar código de programa como parte de um esforço de teste certamente é amplamente aceito. Vários fatores podem afetar a probabilidade de que um determinado esforço de teste e depuração inclua pessoas realmente lendo o código do programa: o tamanho ou a complexidade do aplicativo, o tamanho da equipe de desenvolvimento, o cronograma para o desenvolvimento do aplicativo (se o cronograma é relaxado ou intenso, por exemplo) e, claro, o histórico e a cultura da equipe de programação.

Por essas razões, discutiremos o processo de teste não baseado em computador ("teste humano") antes de nos aprofundarmos nas técnicas de teste mais tradicionais baseadas em computador. As técnicas de teste humano são bastante eficazes para encontrar erros - tanto que todo projeto de programação deve usar uma ou mais dessas técnicas. Você deve aplicar esses métodos entre o momento em que o programa é codificado e quando os testes baseados em computador começam. Você também pode desenvolver e aplicar métodos análogos

20 A arte do teste de software

em estágios iniciais do processo de programação (como no final de cada estágio de projeto), mas estão fora do escopo deste livro.

Antes de começarmos a discussão das técnicas de teste em humanos, tome nota deste ponto importante: Como o envolvimento de humanos resulta em métodos menos formais do que provas matemáticas conduzidas por um computador, você pode se sentir cético de que algo tão simples e informal possa ser útil.

Exatamente o oposto é verdadeiro. Essas técnicas informais não atrapalham o teste bem-sucedido; em vez disso, eles contribuem substancialmente para a produtividade e confiabilidade de duas maneiras principais.

Em primeiro lugar, é geralmente reconhecido que quanto mais cedo os erros forem encontrados, menores serão os custos de correção dos erros e maior a probabilidade de corrigi-los corretamente. Em segundo lugar, os programadores parecem experimentar uma mudança psicológica quando os testes baseados em computador começam. As pressões induzidas internamente parecem aumentar rapidamente e há uma tendência a querer "corrigir esse maldito bug o mais rápido possível". do que eles fazem ao corrigir um erro encontrado anteriormente.

Inspeções e Percursos

Os três principais métodos de teste humano são inspeções de código, orientações e testes de usuário (ou usabilidade). Cobrimos os dois primeiros, que são métodos orientados a código, neste capítulo. Esses métodos podem ser usados em praticamente qualquer estágio de desenvolvimento de software, depois que um aplicativo for considerado completo ou quando cada módulo ou unidade estiver completo (consulte o Capítulo 5 para obter mais informações sobre teste de módulo). Discutimos os testes de usuário em detalhes no Capítulo 7.

Os dois métodos de inspeção de código têm muito em comum, então discutiremos suas semelhanças juntos. Suas diferenças são enumeradas nas seções subsequentes.

Inspeções e orientações envolvem uma equipe de pessoas lendo ou inspecionando visualmente um programa. Com qualquer um dos métodos, os participantes devem realizar algum trabalho preparatório. O clímax é um "encontro de mentes", em uma conferência de participantes. O objetivo da reunião é encontrar erros, mas não encontrar soluções para os erros - ou seja, testar, não depurar.

As inspeções e orientações de código têm sido amplamente utilizadas há algum tempo. Na nossa opinião, a razão do seu sucesso está relacionada com alguns dos princípios identificados no Capítulo 2.

Em um passo a passo, um grupo de desenvolvedores – com três ou quatro sendo o número ideal – realiza a revisão. Apenas um dos participantes é o autor do programa. Portanto, a maioria dos testes de programas é conduzida por pessoas que não o autor, que segue o princípio de teste 2, que afirma que um indivíduo geralmente é ineficaz em testar seu próprio programa. (Consulte o Capítulo 2, Tabela 2.1 e a discussão subsequente para todos os 10 princípios de teste de programa.)

Uma inspeção ou passo a passo é uma melhoria em relação ao antigo processo de verificação de mesa (pelo qual um programador lê seu próprio programa antes de testá-lo). Inspeções e orientações são mais eficazes, novamente porque outras pessoas além do autor do programa estão envolvidas na processo.

Outra vantagem dos walkthroughs, resultando em custos mais baixos de depuração (correção de erros), é o fato de que, quando um erro é encontrado, ele geralmente está localizado precisamente no código, em oposição ao teste de caixa preta, onde você recebe apenas um resultado inesperado. Além disso, esse processo frequentemente expõe um lote de erros, permitindo que os erros sejam corrigidos posteriormente em massa. Os testes baseados em computador, por outro lado, normalmente expõem apenas um sintoma do erro (por exemplo, o programa não termina ou o programa imprime um resultado sem sentido), e os erros geralmente são detectados e corrigidos um a um.

Esses métodos de teste humano geralmente são eficazes em encontrar de 30 a 70 por cento dos erros de projeto lógico e de codificação em programas típicos. Eles não são eficazes, no entanto, na detecção de erros de projeto de alto nível, como erros cometidos no processo de análise de requisitos. Observe que uma taxa de sucesso de 30 a 70 por cento não significa que até 70 por cento de todos os erros possam ser encontrados. Lembre-se do Capítulo 2 que nunca podemos saber o número total de erros em um programa. Assim, o que isso significa é que esses métodos são eficazes em encontrar até 70% de todos os erros encontrados até o final do processo de teste.

Claro, uma possível crítica a essas estatísticas é que os processos humanos encontram apenas os erros “fáceis” (aqueles que seriam triviais de encontrar com testes baseados em computador) e que os erros difíceis, obscuros ou complicados podem ser encontrados apenas por testes baseados em computador. No entanto, alguns testadores usando essas técnicas descobriram que os processos humanos tendem a ser mais eficazes do que os processos de teste baseados em computador para encontrar certos tipos de erros, enquanto o oposto é verdadeiro para outros tipos de erros (por exemplo, variáveis não inicializadas versus divisão por zero erros).

22 A Arte do Teste de Software

A implicação é que inspeções/passeios e testes baseados em computador são complementares; a eficiência da detecção de erros sofrerá se um ou outro não estiver presente.

Finalmente, embora esses processos sejam inestimáveis para testar novos programas, eles são de valor igual ou até maior no teste de modificações em programas. Em nossa experiência, modificar um programa existente é um processo mais propenso a erros (em termos de erros por instrução escrita) do que escrever um novo programa. Portanto, as modificações do programa também devem ser submetidas a esses processos de teste, bem como às técnicas de teste de regressão.

Inspeções de código

Uma inspeção de código é um conjunto de procedimentos e técnicas de detecção de erros para leitura de código de grupo. A maioria das discussões sobre inspeções de código se concentra nos procedimentos, formulários a serem preenchidos e assim por diante. Aqui, após um breve resumo do procedimento geral, nos concentraremos nas técnicas reais de detecção de erros.

Equipe de inspeção

Uma equipe de inspeção geralmente é composta por quatro pessoas. O primeiro dos quatro desempenha o papel de moderador, que neste contexto equivale ao de um engenheiro de controle de qualidade. Espera-se que o moderador seja um programador competente, mas não é o autor do programa e não precisa conhecer os detalhes do programa. Os deveres do moderador incluem:

Distribuição de materiais e agendamento da sessão de inspeção.

Liderando a sessão.

Gravando todos os erros encontrados.

Garantir que os erros sejam posteriormente corrigidos.

O segundo membro da equipe é o programador. Os membros restantes da equipe geralmente são o designer do programa (se for diferente do programador) e um especialista em testes. O especialista deve ser bem versado em teste de software e estar familiarizado com os erros de programação mais comuns, que discutiremos mais adiante neste capítulo.

Agenda de Inspeção

Vários dias antes da sessão de inspeção, o moderador distribui a listagem do programa e as especificações do projeto para os outros participantes.

Espera-se que os participantes se familiarizem com o material antes da sessão.

Durante a sessão, ocorrem duas atividades:

1. O programador narra, enunciado por enunciado, a lógica do programa.

Durante o discurso, outros participantes devem fazer perguntas, que devem ser buscadas para determinar se existem erros. É provável que o programador, e não os outros membros da equipe, encontre muitos dos erros identificados durante esta narração. Em outras palavras, o simples ato de ler um programa em voz alta para uma audiência parece ser uma técnica de detecção de erros notavelmente eficaz.

2. O programa é analisado em relação às listas de verificação de erros de programação historicamente comuns (essa lista de verificação é discutida na próxima seção).

O moderador é responsável por garantir que as discussões prossigam em linhas produtivas e que os participantes concentrem sua atenção em encontrar erros, não em corrigi-los. (O programador corrige os erros após a sessão de inspeção.)

Após a conclusão da sessão de inspeção, o programador recebe uma lista dos erros descobertos. Se mais do que alguns erros forem encontrados, ou se algum dos erros exigir uma correção substancial, o moderador pode tomar providências para reinspecionar o programa depois que esses erros forem corrigidos. Essa lista subsequente de erros também é analisada, categorizada e usada para refinar a lista de verificação de erros para melhorar a eficácia de futuras inspeções.

Como dito, esse processo de inspeção geralmente se concentra em descobrir erros, não em corrigi-los. Dito isso, algumas equipes podem descobrir que, quando um problema menor é descoberto, duas ou três pessoas, incluindo o programador responsável pelo código, podem propor alterações de design para lidar com esse caso especial. A discussão desse problema menor pode, por sua vez, concentrar a atenção do grupo nessa área específica do projeto. Durante a discussão da melhor maneira de alterar o design para lidar com esse pequeno problema, alguém pode notar um segundo problema. Agora que o grupo viu dois problemas relacionados ao mesmo aspecto do design, os comentários provavelmente

24 A Arte do Teste de Software

vem grosso e rápido, com interrupções a cada poucas frases. Em poucos minutos, toda essa área do projeto pode ser explorada a fundo, e qualquer problemas tornados óbvios.

A hora e o local da inspeção devem ser planejados para evitar interrupções externas. A quantidade ideal de tempo para a sessão de inspeção parece ser de 90 a 120 minutos. A sessão é mentalmente desgastante experiência, portanto, sessões mais longas tendem a ser menos produtivas. A maioria das inspeções ocorre a uma taxa de aproximadamente 150 instruções de programa por hora. Por essa razão, programas grandes devem ser examinados em múltiplas inspeções, cada uma lidando com um ou vários módulos ou sub-rotinas.

Agenda Humana

Observe que para que o processo de inspeção seja eficaz, o grupo de teste deve adotar uma atitude adequada. Se, por exemplo, o programador visualizar o inspeção como um ataque ao seu caráter e adota uma postura defensiva, o processo será ineficaz. Em vez disso, o programador deve deixar seu ego na porta e colocar o processo em uma luz positiva e construtiva, tendo em mente que o objetivo da inspeção é encontrar erros no programa e, assim, melhorar a qualidade do trabalho. Por esta razão, a maioria das pessoas recomenda que os resultados de uma inspeção sejam assunto confidencial, compartilhado apenas entre os participantes. Em particular, se gerentes de alguma forma fazem uso dos resultados da inspeção (para assumir ou implicar que o programador é ineficiente ou incompetente, por exemplo), o propósito do processo pode ser frustrado.

Benefícios colaterais do processo de inspeção

O processo de inspeção tem vários efeitos colaterais benéficos, além de sua principal efeito de encontrar erros. Por um lado, o programador geralmente recebe feedback valioso sobre estilo de programação, escolha de algoritmos, e técnicas de programação. Os outros participantes ganham de forma semelhante sendo exposto a erros e estilo de programação de outro programador. Em geral, esse tipo de teste de software ajuda a reforçar uma abordagem de equipe para este projeto em particular e a projetos que envolvam esses participantes em geral. Reduzir o potencial de evolução de um relacionamento adversário, em favor de uma abordagem cooperativa e de equipe aos projetos, pode levar a mais desenvolvimento de programas eficiente e confiável.

Finalmente, o processo de inspeção é uma forma de identificar antecipadamente as seções mais propensas a erros do programa, ajudando a focar a atenção mais diretamente nessas seções durante os processos de teste baseados em computador (número 9 dos princípios de teste apresentados no Capítulo 2).

Uma lista de verificação de erros para inspeções

Uma parte importante do processo de inspeção é o uso de uma lista de verificação para examinar o programa em busca de erros comuns. Infelizmente, algumas listas de verificação se concentram mais em questões de estilo do que em erros (por exemplo, "Os comentários são precisos e significativos?" e "Os blocos de código if-else e os grupos do-while estão alinhados?"), e o erro as verificações são muito nebulosas para serem úteis (como, "O código atende aos requisitos de design?"). A lista de verificação nesta seção, dividida em seis categorias, foi compilada após muitos anos de estudo de erros de software. É amplamente independente de linguagem, o que significa que a maioria dos erros pode ocorrer com qualquer linguagem de programação. Você pode complementar esta lista com erros peculiares à sua linguagem de programação e com erros detectados após a conclusão do processo de inspeção.

Erros de referência de dados

Uma variável referenciada tem um valor não definido ou não inicializado?

Este é provavelmente o erro de programação mais frequente, ocorrendo em uma ampla variedade de circunstâncias. Para cada referência a um item de dados (variável, elemento de array, campo em uma estrutura), tente "provar" formalmente que o item tem um valor naquele ponto.

Para todas as referências de matriz, cada valor subscrito está dentro dos limites definidos da dimensão correspondente?

Para todas as referências de array, cada subscrito tem um valor inteiro?

Isso não é necessariamente um erro em todas as linguagens, mas, em geral, trabalhar com referências de array não inteiros é uma prática perigosa.

Para todas as referências por meio de ponteiros ou variáveis de referência, a memória referenciada está alocada no momento? Isso é conhecido como o problema da "referência pendente". Ocorre em situações em que o tempo de vida de um ponteiro é maior que o tempo de vida da memória referenciada. Uma instância ocorre onde um ponteiro referencia uma variável local dentro de um procedimento, o valor do ponteiro é atribuído a um parâmetro de saída ou uma variável global, o procedimento retorna (liberando o referenciado)

26 A Arte do Teste de Software

localização) e, posteriormente, o programa tenta usar o valor do ponteiro. De maneira semelhante à verificação dos erros anteriores, tente provar informalmente que, em cada referência usando uma variável de ponteiro, existe a memória referenciada.

- Quando uma área de memória tem nomes de alias com atributos diferentes, o valor de dados nesta área tem os atributos corretos quando referenciado? através de um desses nomes? Situações a serem procuradas são o uso de a instrução EQUIVALENCE em Fortran e a cláusula REDEFINES em COBOL. Como exemplo, um programa Fortran contém uma variável real A e uma variável inteira B; ambos são aliases para a mesma área de memória usando uma instrução EQUIVALENCE . Se o programa armazena um valor em A e então referencia a variável B, um erro provavelmente está presente, pois a máquina usaria a representação de bit de ponto flutuante na área de memória como um inteiro.

Barra lateral 3.1: História do COBOL e Fortran

COBOL e Fortran são linguagens de programação mais antigas que alimentaram o desenvolvimento de software empresarial e científico para gerações de hardware de computador, sistemas operacionais e programadores.

COBOL (um acrônimo para COMMON Business Oriented Language) foi definido pela primeira vez por volta de 1959 ou 1960, e foi projetado para dar suporte ao desenvolvimento de aplicativos de negócios em computadores de classe mainframe. A especificação original incluía aspectos de outras linguagens existentes na época. Grandes fabricantes de computadores e representantes do governo federal participaram desse esforço para criar uma linguagem de programação orientada para negócios que pudesse ser executada em uma variedade de plataformas de hardware e sistemas operacionais.

Os padrões da linguagem COBOL foram revisados e atualizados ao longo dos anos. Em 2002, COBOL estava disponível para a maioria das plataformas operacionais atuais e versões orientadas a objetos que suportavam o ambiente de desenvolvimento .NET.

No momento da redação deste artigo, a versão mais recente do COBOL é Visual COBOL 2010.

Fortran (originalmente FORTRAN, mas as referências modernas geralmente seguem a sintaxe de maiúsculas/minúsculas) é um pouco mais antiga que COBOL,

com especificações iniciais definidas no início e meados da década de 1950. Assim como o COBOL, o Fortran foi projetado para tipos específicos de desenvolvimento de aplicativos de mainframe, mas nas áreas de gerenciamento científico e numérico. O nome deriva de um sistema IBM existente na época, Mathematical FORmula TRANslating System. Embora o Fortran original contivesse apenas 32 instruções, ele marcou uma melhoria significativa em relação à programação em nível de assembly que o precedeu.

A versão atual na data de publicação deste livro é o Fortran 2008, formalmente aprovado pelos comitês de padrões apropriados em 2010. Assim como o COBOL, a evolução do Fortran adicionou suporte para uma ampla variedade de plataformas de hardware e sistema operacional. No entanto, o Fortran provavelmente é mais usado no desenvolvimento atual - bem como na manutenção de sistemas mais antigos - do que o COBOL.

- ❑ O valor de uma variável tem um tipo ou atributo diferente do que o compilador espera? Essa situação pode ocorrer quando um programa C ou C++ lê um registro na memória e o referencia usando uma estrutura, mas a representação física do registro difere da definição da estrutura.
- ❑ Há algum problema de endereçamento explícito ou implícito se, no computador em uso, as unidades de alocação de memória forem menores que as unidades de memória endereçável? Por exemplo, em alguns ambientes, cadeias de bits de comprimento fixo não necessariamente começam nos limites de byte, mas abordam apenas os limites ponto a byte. Se um programa computar o endereço de uma string de bits e posteriormente se referir à string por meio desse endereço, a localização incorreta da memória poderá ser referenciada. Essa situação também pode ocorrer ao passar um argumento de cadeia de bits para uma sub-rotina.
- ❑ Se forem usadas variáveis de ponteiro ou de referência, o local de memória referenciado tem os atributos que o compilador espera? Um exemplo de tal erro é quando um ponteiro C++ sobre o qual uma estrutura de dados é baseada recebe o endereço de uma estrutura de dados diferente.
- ❑ Se uma estrutura de dados é referenciada em vários procedimentos ou sub-rotinas, a estrutura é definida de forma idêntica em cada procedimento?
- ❑ Ao indexar em uma string, os limites da string estão fora de um em operações de indexação ou em referências subscriptas a arrays?

28 A Arte do Teste de Software

- Para linguagens orientadas a objetos, todos os requisitos de herança são atendidos na classe de implementação?

Erros de declaração de dados

- Todas as variáveis foram declaradas explicitamente? A falha em fazê-lo não é necessariamente um erro, mas é, no entanto, uma fonte comum de problemas. Por exemplo, se uma sub-rotina de programa recebe um parâmetro de array e não define o parâmetro como um array (como em uma instrução DIMENSION), uma referência ao array (como C¹A(I)) é interpretada como uma chamada de função, levando a máquina a tentar executar a matriz como um programa. Além disso, se uma variável não for declarada explicitamente em um procedimento ou bloco interno, entende-se que a variável é compartilhada com o bloco delimitador?
- Se todos os atributos de uma variável não forem declarados explicitamente na declaração, os padrões são bem compreendidos? Por exemplo, os atributos padrão recebidos em Java geralmente são uma fonte de surpresa quando não declarados corretamente.
- Onde uma variável é inicializada em uma declaração declarativa, ela é inicializada corretamente? Em muitas linguagens, a inicialização de arrays e strings é um tanto complicada e, portanto, propensa a erros.
- Cada variável é atribuída ao comprimento e tipo de dados corretos?
- A inicialização de uma variável é consistente com seu tipo de memória? Por exemplo, se uma variável em uma sub-rotina Fortran precisar ser reinicializada toda vez que a sub-rotina for chamada, ela deverá ser inicializada com uma instrução de atribuição em vez de uma instrução DATA .
- Existem variáveis com nomes semelhantes (por exemplo, VOLT e VOLTS)? Isso não é necessariamente um erro, mas deve ser visto como um aviso de que os nomes podem ter sido confundidos em algum lugar do programa.

Erros de computação

- Existem cálculos usando variáveis com tipos de dados inconsistentes (como não aritméticos)?
- Existem cálculos de modo misto? Um exemplo é ao trabalhar com variáveis de ponto flutuante e inteiro. Tais ocorrências não são necessariamente erros, mas devem ser exploradas com cuidado para garantir que as regras de conversão da linguagem sejam compreendidas.

Considere o seguinte trecho de Java mostrando o erro de arredondamento que pode ocorrer ao trabalhar com inteiros:

```
int x = 1;  
int y = 2;  
int z = 0;  
z = x/y;  
System.out.println ("z = " + z);
```

RESULTADO:

z = 0

- Existem cálculos usando variáveis com os mesmos dados tipo, mas de comprimentos diferentes?
- O tipo de dados da variável de destino de uma atribuição é menor que o tipo de dados ou um resultado da expressão à direita?
- É possível uma expressão de overflow ou underflow durante o cálculo de uma expressão? Ou seja, o resultado final pode parecer ter valor válido, mas um resultado intermediário pode ser muito grande ou muito pequeno para os tipos de dados da linguagem de programação.
- É possível que o divisor em uma operação de divisão seja zero?
- Se a máquina subjacente representar variáveis no formato base 2, são há quaisquer sequências da imprecisão resultante? Ou seja, 10.0,1 é raramente igual a 1.0 em uma máquina binária.
- Onde aplicável, o valor de uma variável pode sair do intervalo significativo? Por exemplo, as declarações que atribuem um valor à variável PROBABILITY podem ser verificadas para garantir que o valor atribuído será sempre positivo e não maior que 1,0.
- Para expressões contendo mais de um operador, são as suposições sobre a ordem de avaliação e precedência dos operadores correto?
- Existem usos inválidos de aritmética inteira, particularmente divisões? Por exemplo, se i é uma variável inteira, se a expressão $2i/2$ depende de i ter um valor par ou ímpar e se a multiplicação ou divisão é realizada primeiro.

Erros de comparação

- Existem comparações entre variáveis com diferentes tipos de dados, como comparar uma cadeia de caracteres com um endereço, data ou número?

30 A Arte do Teste de Software

- Existem comparações de modo misto ou comparações entre variáveis de diferentes comprimentos? Em caso afirmativo, certifique-se de que as regras de conversão sejam bem compreendidas.
- Os operadores de comparação estão corretos? Os programadores freqüentemente confundem tais relações como no máximo, pelo menos, maior que, não menor que e menor que ou igual.
- Cada expressão booleana declara o que deve declarar? Os programadores muitas vezes cometem erros ao escrever expressões lógicas envolvendo e, ou, e não.
- Os operandos de um operador booleano são booleanos? Os operadores de comparação e booleanos foram misturados erroneamente? Isso representa outra classe frequente de erros. Exemplos de alguns erros típicos são ilustrados aqui:

Se você quiser determinar se i está entre 2 e 10, a expressão $2 < i < 10$ está incorreta. Em vez disso, deve ser $(2 < i) \&\& (i < 10)$.

Se você deseja determinar se i é maior que x ou y , $i > x \&\& i > y$ está incorreto. Em vez disso, deve ser $(i > x) \&\& (i > y)$.

Se você quiser comparar três números para igualdade, $\text{if}(a \% 1 \% b \% 1 \% c)$ faz algo bem diferente.

Se você quiser testar a relação matemática $x > y > z$, a expressão correta é $(x > y) \&\& (y > z)$.

- Existem comparações entre números fracionários ou de ponto flutuante que são representados em base 2 pela máquina subjacente? Esta é uma fonte ocasional de erros devido ao truncamento e base-2 aproximações de números de base 10.
- Para expressões contendo mais de um operador booleano, as suposições sobre a ordem de avaliação e a precedência dos operadores estão corretas? Ou seja, se você vir uma expressão como $\text{if}((a \% 1 \% 2) \&\& (b \% 1 \% 2) \&\& (c \% 1 \% 3))$, está bem entendido se o and ou o or é executado primeiro?
- A maneira como o compilador avalia as expressões booleanas afeta o programa?
Por exemplo, a declaração

$$\text{if}(x \% 1 \% 0 \&\& (x/y) > z)$$

pode ser aceitável para compiladores que terminam o teste assim que um lado de e é falso, mas pode causar um erro de divisão por zero com outros compiladores.

Erros de fluxo de controle

- Se o programa contém uma ramificação de vários caminhos, como um GOTO calculado, a variável de índice pode exceder o número de possibilidades de ramificação? Por exemplo, na declaração

```
GOTO(200.300.400),i
```

terei sempre o valor de 1 , 2 ou 3?

- Cada loop terminará eventualmente? Elabore uma prova ou argumento informal mostrando que cada loop terminará.
- O programa, módulo ou sub-rotina terminará eventualmente?
- É possível que, devido às condições de entrada, um loop nunca seja executado? Se sim, isso representa um descuido? Por exemplo, se você tivesse o seguinte loop for e while encabeçado pelas seguintes instruções:

```
for (i%4;x;i<%z;ibp){  
    ...  
}  
  
ou . . .  
  
while (NOTFOUND) {  
    ...  
}
```

o que acontece se x for maior que z ou se NOTFOUND for inicialmente falso?

- Para um loop controlado tanto por iteração quanto por uma condição booleana (por exemplo, um loop de busca), quais são as consequências da queda do loop? Por exemplo, para o loop de pseudocódigo encabeçado por

```
DO I%1 para TABLESIZE WHILE (NOTFOUND)
```

o que acontece se NOTFOUND nunca se tornar falso?

- Existem erros isolados, como uma iteração a mais ou a menos? Este é um erro comum em loops baseados em zero. Você muitas vezes esquecerá de contar 0 como um número. Por exemplo, se você deseja criar um código Java para um loop que itera 10 vezes, o seguinte estaria errado, pois ele executa 11 iterações:

```
for (int i%0;i<%10;i+b){  
    System.out.println(i);  
}
```

32 A Arte do Teste de Software

Correto, o loop é iterado 10 vezes:

```
for (int i%40; i<10;i+b) {  
    System.out.println(i);  
}
```

- Se a linguagem contém um conceito de grupos de instruções ou blocos de código (por exemplo, do-while ou {...}), existe um while explícito para cada grupo e as instâncias de do correspondem aos seus grupos apropriados? Existe um colchete de fechamento para cada colchete aberto? A maioria dos compiladores modernos reclamará de tais incompatibilidades.
- Existem decisões não exaustivas? Por exemplo, se os valores esperados de um parâmetro de entrada são 1, 2 ou 3, a lógica assume que deve ser 3 se não for 1 ou 2? Se sim, a suposição é válida?

Erros de interface

- O número de parâmetros recebidos por este módulo é igual ao número de argumentos enviados por cada um dos módulos de chamada? Além disso, a ordem está correta?
 - Os atributos (por exemplo, tipo de dados e tamanho) de cada parâmetro correspondem aos atributos de cada argumento correspondente?
 - O sistema de unidades de cada parâmetro corresponde ao sistema de unidades de cada argumento correspondente? Por exemplo, o parâmetro é expresso em graus, mas o argumento é expresso em radianos?
 - O número de argumentos passados por este módulo para outro módulo é igual ao número de parâmetros esperados por esse módulo?
 - Os atributos de cada argumento passado para outro módulo correspondem aos atributos do parâmetro correspondente nesse módulo?
 - O sistema de unidades de cada argumento passado para outro módulo corresponde ao sistema de unidades do parâmetro correspondente nesse módulo?
 - Se funções internas forem invocadas, o número, os atributos e a ordem dos argumentos estão corretos?
 - Se um módulo ou classe tiver vários pontos de entrada, há algum parâmetro referenciado que não esteja associado ao ponto de entrada atual?
- Esse erro existe na segunda instrução de atribuição no seguinte programa PL/1:

UMA: PROCEDIMENTO (W,X);
 W%4Xþ1;
 RETORNA
 B: ENTRADA (Y,Z);
 Y%4XþZ;
 FIM;

- Uma sub-rotina altera um parâmetro que deve ser apenas um valor de entrada?
- Se as variáveis globais estiverem presentes, elas têm a mesma definição e atributos em todos os módulos que as referenciam?
- As constantes são sempre passadas como argumentos? Em algumas implementações do Fortran, uma declaração como

LIGUE SUBX(J,3)

é perigoso, pois se a sub-rotina SUBX atribuir um valor ao seu segundo parâmetro, o valor da constante 3 será alterado.

Erros de entrada/saída

- Se os arquivos forem declarados explicitamente, seus atributos estão corretos?
- Os atributos na instrução OPEN do arquivo estão corretos?
- A especificação de formato está de acordo com as informações na instrução de E/S? Por exemplo, em Fortran, cada instrução FORMAT concorda (em termos de número e atributos dos itens) com a instrução READ ou WRITE correspondente ?
- Há memória suficiente disponível para armazenar o arquivo que seu programa irá ler?
- Todos os arquivos foram abertos antes do uso?
- Todos os arquivos foram fechados após o uso?
- As condições de fim de arquivo são detectadas e tratadas corretamente?
- As condições de erro de E/S são tratadas corretamente?
- Existem erros ortográficos ou gramaticais em algum texto impresso ou exibido pelo programa?
- O programa trata corretamente os erros "Arquivo não encontrado"?

Outras verificações

- Se o compilador produzir uma listagem de identificadores de referência cruzada, examine-a em busca de variáveis que nunca são referenciadas ou são referenciadas apenas uma vez.

34 A Arte do Teste de Software

- Se o compilador produzir uma listagem de atributos, verifique os atributos de cada variável para garantir que nenhum atributo padrão inesperado tenha sido atribuído.
- Se o programa compilou com sucesso, mas o computador produziu uma ou mais mensagens de "aviso" ou "informativas", verifique cada uma com cuidado. As mensagens de aviso são indicações de que o compilador suspeita que você está fazendo algo de validade questionável: Revise todas essas suspeitas. Mensagens informativas podem listar variáveis não declaradas ou usos de linguagem que impedem a otimização do código.
- O programa ou módulo é suficientemente robusto? Ou seja, ele verifica a validade de sua entrada?
- Está faltando uma função no programa?

Esta lista de verificação está resumida nas Tabelas 3.1 e 3.2.

Passo a passo

O passo a passo do código, assim como a inspeção, é um conjunto de procedimentos e técnicas de detecção de erros para leitura de código de grupo. Tem muito em comum com o processo de inspeção, mas os procedimentos são ligeiramente diferentes e uma técnica diferente de detecção de erros é empregada.

Assim como a inspeção, o passo a passo é uma reunião ininterrupta de uma a duas horas de duração. A equipe passo a passo é composta por três a cinco pessoas. Uma dessas pessoas desempenha um papel semelhante ao do moderador no processo de fiscalização; outra pessoa desempenha o papel de secretária (uma pessoa que registra todos os erros encontrados); e uma terceira pessoa desempenha o papel de testador. As sugestões sobre quem devem ser as três a cinco pessoas variam. Claro, o programador é uma dessas pessoas. Sugestões para os outros participantes incluem:

- Um programador altamente experiente
- Um especialista em linguagem de programação
- Um novo programador (para dar uma visão nova e imparcial)
- A pessoa que eventualmente manterá o programa
- Alguém de um projeto diferente
- Alguém da mesma equipe de programação que o programador

TABELA 3.1 Resumo da Lista de Verificação de Erros de Inspeção, Parte I

Referência de dados	Computação
1. Variável não definida usada?	1. Cálculos em não aritmética variáveis?
2. Subscritos dentro dos limites?	2. Cálculos de modo misto?
3. Subscritos não inteiros?	3. Cálculos sobre variáveis de comprimentos diferentes?
4. Referências pendentes?	4. Tamanho do alvo menor que o tamanho de valor atribuído?
5. Atributos corretos ao criar alias?	5. Estouro de resultado intermediário ou underflow?
6. Os atributos de registro e estrutura correspondem?	6. Divisão por zero?
7. Computando endereços de strings de bits? Passando argumentos de cadeia de bits?	7. Imprecisões de base 2?
8. Os atributos de armazenamento baseados estão corretos?	8. Valor da variável fora de intervalo significativo?
9. As definições de estrutura correspondem procedimentos?	9. Precedência do operador Entendido?
10. Erros isolados na indexação ou operações de subscrição?	10. Divisões inteiras corretas?
11. Requisitos de herança atendidos?	
Declaração de dados	Comparação
1. Todas as variáveis declaradas?	1. Comparações entre variáveis inconsistentes?
2. Atributos padrão entendidos?	2. Comparações de modo misto?
3. Arrays e strings inicializados corretamente?	3. As relações de comparação estão corretas?
4. Comprimentos, tipos e armazenamento corretos aulas atribuídas?	4. Expressões booleanas corretas?
5. Inicialização consistente com armazenamento classe?	5. Comparação e Booleano expressões misturadas?
6. Alguma variável com nomes semelhantes?	6. Comparações de frações de base 2 valores?
	7. A precedência do operador foi compreendida?
	8. Avaliação do compilador de Boolean expressões compreendidas?

36 A Arte do Teste de Software

TABELA 3.2 Resumo da Lista de Verificação de Erros de Inspeção, Parte II

Controle de fluxo	Entrada/Saída
1. Ramos Multiway ultrapassados? 2. Cada loop terminará? 3. O programa terminará? 4. Algum loop é ignorado devido às condições de entrada? 4. O tamanho do buffer corresponde ao tamanho do registro? 5. Possíveis falhas de loop estão corretas? 6. Erros de iteração off-by-one? 7. As instruções DO/END correspondem? 8. Alguma decisão não exaustiva? 9. Algum erro textual ou gramatical nas informações de saída?	1. Os atributos do arquivo estão corretos? 2. As declarações OPEN estão corretas? 3. Especificação do formato corresponde à instrução de E/S? 5. Arquivos abertos antes do uso? 6. Arquivos fechados após o uso? 7. Condições de fim de arquivo manipulado? 8. Erros de E/S tratados?
Interfaces	Outras verificações
1. Número de parâmetros de entrada igual ao número de argumentos? 2. Os atributos de parâmetro e argumento correspondem? 3. O sistema de unidades de parâmetros e argumentos corresponde? 3. Qualquer aviso ou mensagens informativas? 4. Número de argumentos transmitidos aos módulos chamados igual ao número de parâmetros? 5. Atributos dos argumentos transmitidos aos módulos chamados iguais aos atributos dos parâmetros? 6. Sistema de unidades de argumentos transmitidos aos módulos chamados igual a sistema de unidades de parâmetros? 7. Número, atributos e ordem dos argumentos para funções internas corretas? 8. Alguma referência a parâmetros não associados ao ponto de entrada atual? 9. Argumentos somente de entrada alterados? 10. Definições de variáveis globais consistentes entre os módulos?	1. Alguma variável não referenciada na listagem de referência cruzada? 2. Lista de atributos o que era esperado? 4. Entrada verificada quanto à validade? 5. Função ausente? 11. Constantes passadas como argumentos?

O procedimento inicial é idêntico ao do processo de inspeção: Os participantes recebem os materiais com vários dias de antecedência, para permitir tempo para se prepararem para o programa. No entanto, o procedimento no encontro é diferente. Em vez de simplesmente ler o programa ou usar listas de verificação de erros, os participantes "jogam computador". A pessoa designada como o testador chega à reunião armado com um pequeno conjunto de papel de teste casos—conjuntos representativos de entradas (e saídas esperadas) para o programa ou módulo. Durante a reunião, cada caso de teste é executado mentalmente; ou seja, os dados de teste são "percorridos" pela lógica do programa. O estado do programa (ou seja, os valores das variáveis) é monitorado em papel ou quadro branco.

Claro, os casos de teste devem ser simples por natureza e poucos em número, porque as pessoas executam programas a uma taxa muitas ordens de magnitude mais lenta do que uma máquina. Portanto, os próprios casos de teste não desempenhar um papel crítico; em vez disso, eles servem como um veículo para começar e para questionar o programador sobre sua lógica e suposições. Na maioria dos walkthroughs, os erros são encontrados durante o processo de questionando o programador do que são encontrados diretamente pelos casos de teste eles mesmos.

Assim como na inspeção, a atitude dos participantes é crítica. Os comentários devem ser direcionados ao programa e não ao programador. Em outras palavras, os erros não são vistos como fraquezas na pessoa que os cometeu. Pelo contrário, eles são vistos como inerentes à dificuldade de o desenvolvimento do programa.

O passo a passo deve ter um processo de acompanhamento semelhante ao descrito para o processo de inspeção. Além disso, os efeitos colaterais observados nas inspeções (identificação de seções propensas a erros e educação em erros, estilo e técnicas) também se aplicam ao processo passo a passo.

Verificação de mesa

Um terceiro processo de detecção de erro humano é a prática mais antiga de checagem de mesa. Uma verificação de mesa pode ser vista como uma inspeção de uma pessoa ou passo a passo: uma pessoa lê um programa, verifica-o em relação a uma lista de erros, e/ou percorre os dados de teste através dele.

Para a maioria das pessoas, a verificação de mesa é relativamente improdutiva. Uma razão é que é um processo completamente indisciplinado. Uma segunda razão, e mais importante, é que ela contraria o princípio de teste 2 (veja o Capítulo 2),

38 A Arte do Teste de Software

que afirma que as pessoas geralmente são ineficazes em testar seus próprios programas. Por esse motivo, você pode deduzir que a verificação de mesa é melhor executada por uma pessoa que não seja o autor do programa (por exemplo, dois programadores podem trocar programas em vez de verificar a mesa), mas mesmo isso é menos eficaz do que o passo a passo ou a inspeção processo de ção. O motivo é o efeito sinérgico do passo a passo ou equipe de inspeção. A sessão de equipe promove um ambiente saudável de competição; as pessoas gostam de se exibir encontrando erros. Em um processo de verificação de mesa, não há ninguém para quem você possa se exibir, impedindo esse efeito aparentemente valioso. Em suma, a verificação de mesa pode ser mais valiosa do que não fazer nada, mas é muito menos eficaz do que a inspeção ou o passo a passo.

Classificações de pares

O último processo de revisão humana não está associado ao teste do programa (ou seja, seu objetivo não é encontrar erros). No entanto, incluímos este processo aqui porque está relacionado com a ideia de leitura de código.

A classificação por pares é uma técnica de avaliação de programas anônimos em termos de qualidade geral, capacidade de manutenção, extensibilidade, usabilidade e clareza. O objetivo da técnica é fornecer autoavaliação do programador.

Um programador é selecionado para atuar como administrador do processo. O administrador, por sua vez, seleciona cerca de 6 a 20 participantes (6 é o mínimo para preservar o anonimato). Espera-se que os participantes tenham experiências semelhantes (por exemplo, não agrupe programadores de aplicativos Java com programadores de sistemas em linguagem assembly). Cada participante é convidado a selecionar dois de seus próprios programas para serem revisados. Um programa deve ser representativo do que o participante considera seu melhor trabalho; o outro deve ser um programa que o programador considere de qualidade inferior.

Uma vez coletados os programas, eles são distribuídos aleatoriamente aos participantes. Cada participante recebe quatro programas para revisar. Dois dos programas são os programas "melhores" e dois são os programas "mais pobres", mas o revisor não é informado sobre qual é qual. Cada participante gasta 30 minutos revisando cada programa e, em seguida, preenche um formulário de avaliação. Depois de analisar todos os quatro programas, cada participante avalia a qualidade relativa dos quatro programas. O formulário de avaliação pede ao revisor que

responda, em uma escala de 1 a 10 (1 significa definitivamente sim e 10 significa definitivamente não), questões como:

- O programa foi fácil de entender?
- O design de alto nível era visível e razoável?
- O design de baixo nível era visível e razoável?
- Seria fácil para você modificar este programa?
- Você estaria orgulhoso de ter escrito este programa?

O revisor também é solicitado para comentários gerais e melhorias sugeridas.

Após a revisão, os participantes recebem os formulários de avaliação anônimos para seus dois programas contribuídos. Eles também recebem um resumo estatístico mostrando a classificação geral e detalhada de seus programas originais em todo o conjunto de programas, bem como uma análise de como suas classificações de outros programas comparadas com as classificações de outros revisores do mesmo programa. O objetivo do processo é permitir que os programadores avaliem suas habilidades de programação. Como tal, o processo parece ser útil em ambientes industriais e de sala de aula.

Resumo

Este capítulo discutiu uma forma de teste que os desenvolvedores não costumam considerar: teste de código humano. A maioria das pessoas assume que, como os programas são escritos para execução em máquina, as máquinas também devem testar os programas. Essa suposição é inválida. As técnicas de teste em humanos são muito eficazes para revelar erros. Na verdade, a maioria dos projetos de programação deve incluir as seguintes técnicas de teste humano:

Inspeções de código usando listas de verificação

Passo a passo do grupo

Verificação de mesa

Revisões por pares

Outra forma de teste humano é o teste de usuário ou usabilidade, uma técnica de caixa preta que avalia o software de uma perspectiva prática do usuário final.

Abordamos esse tópico em detalhes no Capítulo 7.

4 Projeto de Caso de Teste

Indo além das questões psicológicas discutidas no Capítulo 2, a consideração mais importante no teste de programa é o design e a criação de casos de teste eficazes.

Testes, por mais criativos e aparentemente completos, não podem garantir a ausência de todos os erros. O design do caso de teste é tão importante porque o teste completo é impossível. Dito de outra forma, um teste de qualquer programa deve ser necessariamente incompleto. A estratégia óbvia, então, é tentar fazer os testes o mais completos possível.

Dadas as restrições de tempo e custo, a principal questão do teste se torna:

Qual subconjunto de todos os casos de teste possíveis tem a maior probabilidade de detectar a maioria dos erros?

O estudo de metodologias de projeto de casos de teste fornece respostas para essa pergunta.

Em geral, a metodologia menos eficaz de todas é o teste de entrada aleatória – o processo de testar um programa selecionando, aleatoriamente, algum subconjunto de todos os valores de entrada possíveis. Em termos de probabilidade de detectar a maioria dos erros, uma coleção de casos de teste selecionados aleatoriamente tem pouca chance de ser um subconjunto ótimo, ou mesmo próximo ao ótimo. Portanto, neste capítulo, queremos desenvolver um conjunto de processos de pensamento que permitem selecionar dados de teste de forma mais inteligente.

42 A Arte do Teste de Software

O Capítulo 2 mostrou que testes exaustivos de caixa preta e caixa branca são, em geral, impossível; ao mesmo tempo, sugeriu que uma estratégia de teste razoável poderia apresentar elementos de ambos. Esta é a estratégia desenvolvida neste capítulo. Você pode desenvolver um teste razoavelmente rigoroso usando certas metodologias de projeto de caso de teste orientadas a caixa preta e, em seguida, complementando esses casos de teste examinando a lógica do programa, usando caixa branca métodos.

As metodologias discutidas neste capítulo são:

Caixa preta	Caixa branca
Particionamento equivalente	Cobertura do extrato
Análise de valor de limite	Cobertura da decisão
Gráficos de causa e efeito	Cobertura de condição
Erro ao adivinhar	Cobertura de decisão/condição
	Cobertura de várias condições

Embora discutiremos esses métodos separadamente, recomendamos que você usa uma combinação da maioria, se não de todos, para projetar um teste rigoroso de um programa, uma vez que cada método tem pontos fortes e fracos distintos. Um método pode encontrar erros que outro método ignora, por exemplo.

Ninguém jamais prometeu que o teste de software seria fácil. Para citar um velho sábio, "Se você achava que projetar e codificar esse programa era difícil, você ainda não vi nada."

O procedimento recomendado é desenvolver casos de teste usando os métodos da caixa preta e, em seguida, desenvolver casos de teste complementares, conforme necessário, com métodos de caixa branca. Discutiremos a caixa branca mais conhecida métodos primeiro.

Teste de caixa branca

O teste de caixa branca está preocupado com o grau em que os casos de teste exercem ou cobrem a lógica (código-fonte) do programa. Como vimos no Capítulo 2, o teste final da caixa branca é a execução de cada caminho no programa; mas o teste de caminho completo não é uma meta realista para um programa com laços.

Teste de cobertura lógica

Se você se afastar completamente do teste de caminho, pode parecer que uma meta válida seria executar todas as instruções do programa pelo menos uma vez. Infelizmente, este é um critério fraco para um teste de caixa branca razoável. Este conceito é ilustrado na Figura 4.1. Suponha que esta figura represente um pequeno programa a ser testado. O trecho de código Java equivalente segue:

```
public void foo(int A,int B,int X) { if(A>1 &&
B!=0) {
    X=X/A;
} if(A!=2 jj X>1) {
    X=X+1;
}
```

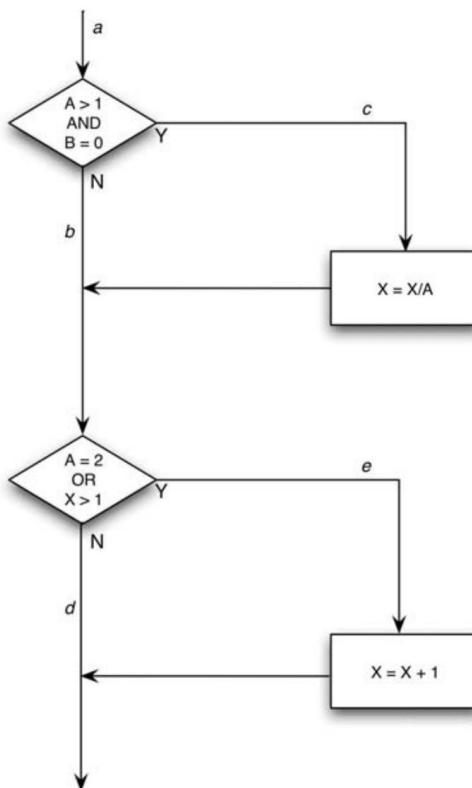


FIGURA 4.1 Um pequeno programa a ser testado.

Você pode executar cada instrução escrevendo um único caso de teste que percorre o caminho ace. Ou seja, definindo $A \neq 2$, $B \neq 0$ e $X \neq 3$ no ponto a, cada instrução seria executada uma vez (na verdade, X poderia receber qualquer valor inteiro >1).

Infelizmente, este critério é bastante pobre. Por exemplo, talvez a primeira decisão deva ser um ou em vez de um e. Nesse caso, esse erro não seria detectado. Talvez a segunda decisão devesse ter declarado $X > 0$; este erro não seria detectado. Além disso, há um caminho pelo programa no qual X permanece inalterado (o caminho abd). Se isso fosse um erro, passaria despercebido. Em outras palavras, o critério de cobertura da declaração é tão fraco que geralmente é inútil.

Um critério de cobertura lógica mais forte é conhecido como cobertura de decisão ou cobertura de ramificação. Esse critério afirma que você deve escrever casos de teste suficientes para que cada decisão tenha um resultado verdadeiro e um resultado falso pelo menos uma vez. Em outras palavras, cada direção de ramificação deve ser percorrida pelo menos uma vez. Exemplos de instruções de ramificação ou decisão são instruções switch-case, do-while e if-else . As instruções GOTO de vários caminhos se qualificam em algumas linguagens de programação, como Fortran.

A cobertura de decisão geralmente pode satisfazer a cobertura de instrução. Como cada instrução está em algum subcaminho que emana de uma instrução de ramificação ou do ponto de entrada do programa, cada instrução deve ser executada se todas as direções de ramificação forem executadas. Existem, no entanto, pelo menos três exceções:

Programas sem decisões.

Programas ou sub-rotinas/métodos com múltiplos pontos de entrada. Uma determinada instrução pode ser executada apenas se o programa for inserido em um ponto de entrada específico.

Declarações dentro das unidades ON. Atravessar todas as direções de ramificação não necessariamente fará com que todas as unidades ON sejam executadas.

Como consideramos a cobertura de declaração uma condição necessária, a cobertura de decisão, um critério aparentemente melhor, deve ser definida para incluir a cobertura de declaração. Assim, a cobertura de decisão requer que cada decisão tenha um resultado verdadeiro e um falso, e que cada afirmação seja executada pelo menos uma vez. Uma maneira alternativa e mais fácil de expressar isso é que cada decisão tem um resultado verdadeiro e um falso, e que cada ponto de entrada (incluindo unidades ON) seja invocado pelo menos uma vez.

Essa discussão considera apenas decisões ou ramificações bidirecionais e deve ser modificada para programas que contêm decisões de vários caminhos. Exemplos são programas Java contendo instruções switch-case , programas Fortran contendo instruções IF aritméticas (três vias) ou instruções GOTO computadas ou aritméticas e programas COBOL contendo instruções GOTO alteradas ou instruções GO-TO-DEPENDING-ON . Para tais programas, o critério é exercitar cada resultado possível de todas as decisões pelo menos uma vez e invocar cada ponto de entrada do programa ou sub-rotina pelo menos uma vez.

Na Figura 4.1, a cobertura de decisão pode ser atendida por dois casos de teste cobrindo os caminhos ace e abd ou, alternativamente, acd e abe. Se escolhermos a última alternativa, as duas entradas do caso de teste são A \leq 3, B \leq 0, X \leq 3 e A \geq 2, B \geq 1 e X \geq 1.

A cobertura de decisão é um critério mais forte do que a cobertura de declaração, mas ainda é bastante fraca. Por exemplo, há apenas 50% de chance de explorarmos o caminho onde x não é alterado (ou seja, somente se escolhermos a primeira alternativa). Se a segunda decisão estivesse errada (se deveria ter dito X <1 em vez de X >1), o erro não seria detectado pelos dois casos de teste no exemplo anterior.

Um critério que às vezes é mais forte do que a cobertura de decisão é a cobertura de condição. Nesse caso, você escreve casos de teste suficientes para garantir que cada condição em uma decisão assuma todos os resultados possíveis pelo menos uma vez. Mas, como na cobertura de decisão, isso nem sempre leva à execução de cada instrução, portanto, uma adição ao critério é que cada ponto de entrada do programa ou sub-rotina, bem como as unidades ON, sejam invocados pelo menos uma vez. Por exemplo, a declaração de ramificação:

FAÇA K \leq 0 a 50 ENQUANTO (J \leq K $<$ QUEST)

contém duas condições: K é menor ou igual a 50, e J \leq K é menor que QUEST? Portanto, casos de teste seriam necessários para as situações K $<\leq$ 50, K >50 (para atingir a última iteração do loop), J \leq K $<$ QUEST e J \leq K \geq QUEST.

A Figura 4.1 tem quatro condições: A >1 , B \leq 0, A \geq 2 e X >1 . Portanto, casos de teste suficientes são necessários para forçar as situações em que A >1 , A \leq 1, B \leq 0 e B \geq 0 estão presentes no ponto a e onde A \geq 2, A <2 , X >1 e X \leq 1 são presente no ponto b. Um número suficiente de casos de teste que satisfaçam o critério e os caminhos percorridos por cada um são:

A \geq 2, B \leq 0, X \leq 4 ás
A \leq 1, B \geq 1, X \geq 1 adb

46 A Arte do Teste de Software

Observe que, embora o mesmo número de casos de teste tenha sido gerado para este exemplo, a cobertura de condição geralmente é superior à cobertura de decisão, pois pode (mas nem sempre) fazer com que cada condição individual em uma decisão seja executada com ambos os resultados, enquanto a cobertura de decisão não. Por exemplo, na mesma declaração de ramificação

FAÇA K%0 a 50 ENQUANTO (JpK<QUEST)

é uma ramificação de duas vias (execute o corpo do loop ou pule-o). Se você estiver usando o teste de decisão, o critério pode ser satisfeito deixando o loop correr de K%0 a 51, sem nunca explorar a circunstância em que a cláusula WHILE se torna falsa. Com o critério de condição, entretanto, um caso de teste seria necessário para gerar um resultado falso para as condições JpK<QUEST.

Embora o critério de cobertura de condição pareça, à primeira vista, satisfazer o critério de cobertura de decisão, nem sempre o faz. Se a decisão SE(A & B) estiver sendo testada, o critério de cobertura de condição permitiria escrever dois casos de teste – A é verdadeiro, B é falso e A é falso, B é verdadeiro – mas isso não causaria o ENTÃO cláusula do IF a ser executado. Os testes de cobertura de condição para o exemplo anterior cobriram todos os resultados da decisão, mas isso foi apenas por acaso. Por exemplo, dois casos de teste alternativos

A%1, B%0, X%3
A%2, B%1, X%1

cobrem todos os resultados de condição, mas apenas dois dos quatro resultados de decisão (ambos cobrem o caminho abe e, portanto, não exercem o resultado verdadeiro da primeira decisão e o resultado falso da segunda decisão).

A saída óbvia para esse dilema é um critério chamado cobertura de decisão/condição. Requer casos de teste suficientes para que cada condição em uma decisão receba todos os resultados possíveis pelo menos uma vez, cada decisão receba todos os resultados possíveis pelo menos uma vez e cada ponto de entrada seja invocado pelo menos uma vez.

Um ponto fraco da cobertura de decisão/condição é que, embora possa parecer exercer todos os resultados de todas as condições, frequentemente não o faz, porque certas condições mascaram outras condições. Para ver isso, examine a Figura 4.2. O fluxograma nesta figura é a maneira como um compilador geraria código de máquina para o programa da Figura 4.1. As decisões multicondicionais no programa fonte foram divididas em decisões e ramificações individuais porque a maioria das máquinas não possui uma única instrução que toma decisões multicondicionais. Uma cobertura de teste mais completa, então,

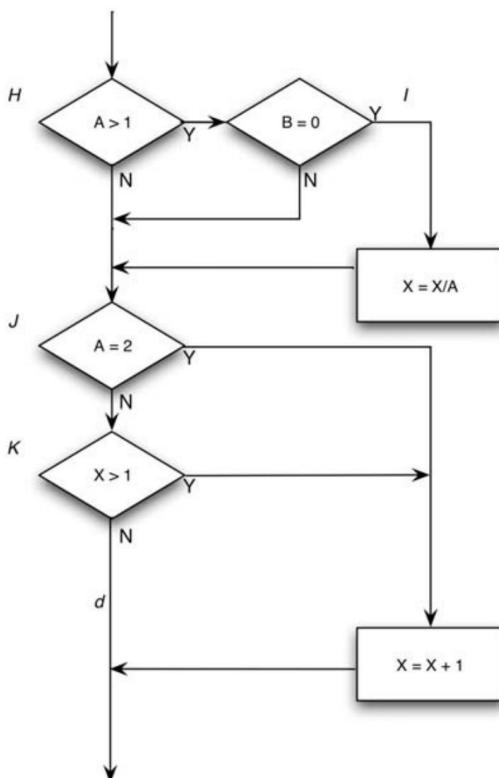


FIGURA 4.2 Código de Máquina para o Programa da Figura 4.1.

parece ser o exercício de todos os resultados possíveis de cada decisão primitiva. Os dois casos de teste de cobertura de decisão anteriores não isto; eles falham em exercer o resultado falso da decisão H e o resultado verdadeiro da decisão K.

A razão, como mostrado na Figura 4.2, é que os resultados das condições no e as expressões ou podem mascarar ou bloquear a avaliação de outras condições. Por exemplo, se uma condição e for falsa, nenhuma das condições na expressão precisam ser avaliadas. Da mesma forma, se uma condição ou for verdadeira, nenhuma das condições subsequentes precisa ser avaliada. Por isso, erros em expressões lógicas não são necessariamente revelados pela condição critérios de cobertura e de cobertura de decisão/condição.

Um critério que cobre este problema, e mais alguns, é a condição múltipla cobertura. Este critério requer que você escreva casos de teste suficientes para que todas as combinações possíveis de resultados de condição em cada decisão, e todas

48 A Arte do Teste de Software

pontos de entrada, são invocados pelo menos uma vez. Por exemplo, considere a seguinte sequência de pseudocódigo.

```
NOTFOUND%4TRUE;
DO I%1 para TABSIZE WHILE (NOTFOUND); /*PESQUISAR TABELA*
... procurando lógica ... ;
FIM
```

As quatro situações a serem testadas são:

1. I<%TABSIZ e NOTFOUND são verdadeiros.
2. I<%TABSIZ e NOTFOUND é falso (encontrar a entrada antes bater na ponta da mesa).
3. I>TABSIZ e NOTFOUND são verdadeiros (atingir o final da tabela sem encontrar a entrada).
4. I>TABSIZ e NOTFOUND é falso (a entrada é a última no tabela).

Deve ser fácil ver que um conjunto de casos de teste que satisfaça o critério de condição múltipla também satisfaz a cobertura de decisão, cobertura de condição, e critérios de cobertura de decisão/condição.

Voltando à Figura 4.1, os casos de teste devem abranger oito combinações:

- | | |
|---------------|---------------|
| 1. A>1, B%0 | 5. A%2, X>1 |
| 2. A>1, B<>0 | 6. A%2, X<%1 |
| 3. A<%1, B%0 | 7. A<>2, X>1 |
| 4. A<%1, B<>0 | 8. A<>2, X<%1 |

Nota Lembre-se do trecho de código Java apresentado anteriormente que os casos de teste 5 a 8 expressam valores no ponto da segunda instrução if . Desde X pode ser alterado acima desta instrução if , os valores necessários nesta instrução if devem ser copiados através da lógica para encontrar a entrada correspondente valores.

Essas combinações a serem testadas não implicam necessariamente que oito casos são necessários. Na verdade, eles podem ser cobertos por quatro casos de teste. Os valores de entrada do caso de teste e as combinações que eles cobrem são os seguintes:

A%2, B%0, X%4 Coberturas 1, 5

A%2, B%1, X%1 Coberturas 2, 6

A₁/1, B₁/0, X₁/2 Coberturas 3, 7

A₁/1, B₁/1, X₁/1 Coberturas 4, 8

O fato de haver quatro casos de teste e quatro caminhos distintos na Figura 4.1 é apenas coincidência. Na verdade, esses quatro casos de teste não cobrem todos os caminhos; eles perdem o caminho acd. Por exemplo, você precisaria de oito casos de teste para a seguinte decisão:

```
if(x<1&y && comprimento(z)<1&0 && FLAG)
    { j<1;
senão
    i<1;
}
```

embora contenha apenas dois caminhos. No caso de loops, o número de casos de teste exigidos pelo critério de condição múltipla é normalmente muito menor que o número de caminhos.

Em resumo, para programas contendo apenas uma condição por decisão, um critério de teste mínimo é um número suficiente de casos de teste para: (1) invocar todos os resultados de cada decisão pelo menos uma vez e (2) invocar cada ponto de entrada (como ponto de entrada ou unidade ON) pelo menos uma vez, para garantir que todas as instruções sejam executadas pelo menos uma vez. Para programas contendo decisões com múltiplas condições, o critério mínimo é um número suficiente de casos de teste para invocar todas as combinações possíveis de resultados de condição em cada decisão e todos os pontos de entrada no programa, pelo menos uma vez.

(A palavra "possível" é inserida porque algumas combinações podem ser consideradas impossíveis de criar.)

Teste de caixa preta

Conforme discutimos no Capítulo 2, o teste de caixa preta (orientado por dados ou orientado por entrada/saída) é baseado nas especificações do programa. O objetivo é encontrar áreas em que o programa não se comporte de acordo com suas especificações.

Particionamento equivalente

O Capítulo 2 descreveu um bom caso de teste como aquele que tem uma probabilidade razoável de encontrar um erro; também afirmou que um teste exaustivo de entrada de um programa é impossível. Portanto, ao testar um programa, você está limitado a um

50 A Arte do Teste de Software

pequeno subconjunto de todas as entradas possíveis. Claro, então, você deseja selecionar o subconjunto "certo", ou seja, o subconjunto com a maior probabilidade de encontrar a maioria dos erros.

Uma maneira de localizar esse subconjunto é perceber que um caso de teste bem selecionado também deve ter duas outras propriedades:

1. Reduz, em mais de um, o número de outros casos de teste que devem ser desenvolvidos para atingir algum objetivo predefinido de teste "razoável".
2. Abrange um grande conjunto de outros casos de teste possíveis. Ou seja, ele nos diz algo sobre a presença ou ausência de erros além desse conjunto específico de valores de entrada.

Essas propriedades, embora pareçam semelhantes, descrevem duas considerações distintas. A primeira implica que cada caso de teste deve invocar tantas considerações de entrada diferentes quanto possível para minimizar o número total de casos de teste necessários. A segunda implica que você deve tentar particionar o domínio de entrada de um programa em um número finito de classes de equivalência tal que você possa razoavelmente supor (mas, é claro, não ter certeza absoluta) que um teste de um valor representativo de cada class é equivalente a um teste de qualquer outro valor. Ou seja, se um caso de teste em uma classe de equivalência detecta um erro, espera-se que todos os outros casos de teste na classe de equivalência encontrem o mesmo erro. Por outro lado, se um caso de teste não detectou um erro, esperaríamos que nenhum outro caso de teste na classe de equivalência se enquadrasse em outra classe de equivalência, uma vez que as classes de equivalência podem se sobrepor.

Essas duas considerações formam uma metodologia de caixa preta conhecida como particionamento de equivalência. A segunda consideração é usada para desenvolver um conjunto de condições "interessantes" a serem testadas. A primeira consideração é então usada para desenvolver um conjunto mínimo de casos de teste cobrindo essas condições.

Um exemplo de uma classe de equivalência no programa de triângulos do Capítulo 1 é o conjunto "três números de igual valor com valores inteiros maiores que zero". Um teste de um elemento do conjunto, é improvável que um erro seja encontrado por um teste de outro elemento do conjunto. Em outras palavras, nosso tempo de teste é melhor gasto em outro lugar: em diferentes classes de equivalência.

O design do caso de teste por particionamento de equivalência prossegue em duas etapas: (1) identificar as classes de equivalência e (2) definir os casos de teste.

Condição externa	Classes de equivalência válidas	Classes de equivalência inválidas

FIGURA 4.3 Um Formulário para Enumerar Classes de Equivalência.

Identificando as Classes de Equivalência As classes de equivalência são identificadas tomando cada condição de entrada (geralmente uma sentença ou frase na especificação) e dividindo-a em dois ou mais grupos. Você pode usar a tabela na Figura 4.3 para fazer isso. Observe que dois tipos de classes de equivalência são identificados: classes de equivalência válidas representam entradas válidas para o programa e classes de equivalência inválidas representam todos os outros estados possíveis da condição (ou seja, valores de entrada errôneos). Assim, estamos aderindo ao princípio 5, discutido no Capítulo 2, que afirma que você deve focar a atenção em condições inválidas ou inesperadas.

Dada uma entrada ou condição externa, identificar as classes de equivalência é em grande parte um processo heurístico. Siga estas orientações:

1. Se uma condição de entrada especificar um intervalo de valores (por exemplo, "a contagem de itens pode ser de 1 a 999"), identifique uma classe de equivalência válida ($1 < \text{contagem de itens} < 999$) e duas classes de equivalência inválidas (contagem de itens < 1 e contagem de itens > 999).
2. Se uma condição de entrada especificar o número de valores (por exemplo, "de um a seis proprietários podem ser listados para o automóvel"), identifique uma classe de equivalência válida e duas classes de equivalência inválidas (nenhum proprietário e mais de seis proprietários) .
3. Se uma condição de entrada especifica um conjunto de valores de entrada, e há motivos para acreditar que o programa trata cada um de forma diferente ("tipo

de veículo deve ser ÔNIBUS, CAMINHÃO, TÁXICA, PASSAGEIRO ou MOTO"), identificar uma classe de equivalência válida para cada uma e uma classe de equivalência inválida ("TRAILER", por exemplo).

4. Se uma condição de entrada especificar uma situação "obrigatória", como "o primeiro caractere do identificador deve ser uma letra", identifique uma classe de equivalência válida (é uma letra) e uma classe de equivalência inválida (não é uma carta).

Se houver alguma razão para acreditar que o programa não trata os elementos de uma classe de equivalência de forma idêntica, divida a classe de equivalência em classes de equivalência menores. Ilustraremos um exemplo desse processo em breve.

Identificando os Casos de Teste O segundo passo é o uso de classes de equivalência para identificar os casos de teste. O processo é como se segue:

1. Atribua um número único a cada classe de equivalência.
2. Até que todas as classes de equivalência válidas tenham sido cobertas (incorporadas em) casos de teste, escreva um novo caso de teste cobrindo o maior número possível de classes de equivalência válidas descobertas.
3. Até que seus casos de teste tenham coberto todas as classes de equivalência inválidas, escreva um caso de teste que cubra uma, e apenas uma, das classes de equivalência inválidas descobertas.

A razão pela qual os casos de teste individuais cobrem casos inválidos é que certas verificações de entrada errônea mascaram ou substituem outras verificações de entrada errônea. Por exemplo, se a especificação indicar "digite o tipo de livro (HARDCOVER, SOFTCOVER ou LOOSE) e o valor (1-999)," o caso de teste, (XYZ 0), expressando duas condições de erro (tipo e valor do livro inválidos) provavelmente não exercerá o cheque do valor, pois o programa pode dizer

"XYZ É TIPO DE LIVRO DESCONHECIDO" e não se preocupe em examinar os restos da entrada.

Um exemplo

Como exemplo, suponha que estamos desenvolvendo um compilador para um subconjunto da linguagem Fortran e desejamos testar a verificação de sintaxe da instrução DIMENSION . A especificação está listada abaixo. (Nota: isso não é

a declaração completa do Fortran DIMENSION ; foi editado consideravelmente para torná-lo tamanho de livro didático. Não se iluda pensando que o teste de programas reais é tão fácil quanto os exemplos deste livro.) Na especificação, os itens em itálico indicam unidades sintáticas para as quais entidades específicas devem ser substituídas em declarações reais; colchetes são usados para indicar itens de opção; e uma reticência indica que o item anterior pode aparecer várias vezes em sucessão.

Uma instrução DIMENSION é usada para especificar as dimensões das matrizes. A forma da instrução DIMENSION é

DIMENSION anúncio[,anúncio]...

onde ad é um descritor de matriz do formulário

n(d[,d]...)

onde n é o nome simbólico da matriz e d é um declarador de dimensão. Os nomes simbólicos podem ter de uma a seis letras ou dígitos, sendo que o primeiro deve ser uma letra. Os números mínimo e máximo de declarações de dimensão que podem ser especificados para uma matriz são um e sete, respectivamente. A forma de um declarador de dimensão é

[lb:]ub

onde lb e ub são os limites de dimensão inferior e superior. Um limite pode ser uma constante no intervalo de 65534 a 65535 ou o nome de uma variável inteira (mas não um nome de elemento de matriz). Se lb não for especificado, assume-se que é 1. O valor de ub deve ser maior ou igual a lb. Se lb for especificado, seu valor pode ser negativo, 0 ou positivo. Como para todas as instruções, a instrução DIMENSION pode ser continuada em várias linhas.

O primeiro passo é identificar as condições de entrada e, a partir delas, localizar as classes de equivalência. Estes são tabulados na Tabela 4.1. Os números na tabela são identificadores únicos das classes de equivalência.

O próximo passo é escrever um caso de teste cobrindo uma ou mais classes de equivalência válidas. Por exemplo, o caso de teste

DIMENSÃO A(2)

abrange as classes 1, 4, 7, 10, 12, 15, 24, 28, 29 e 43.

TABELA 4.1 Classes de Equivalência

Condição de entrada	Equivalência válida Aulas	Equivalência inválida Aulas
Número de matriz descriptores	um (1), > um (2)	nenhum (3)
Tamanho do nome da matriz	1–6 (4)	0 (5), >6 (6)
Nome da matriz	tem letras (7), tem dígitos (8)	Tem algo mais (9)
O nome do array começa com a letra sim (10)		não (11)
Número de dimensões	1–7 (12)	0 (13), >7 (14)
O limite superior é	constante (15), variável inteira (16)	nome do elemento da matriz (17), outra coisa (18)
Nome da variável inteira	tem a letra (19), tem dígitos (20)	tem outra coisa (21)
A variável inteira começa com carta	sim (22)	não (23)
Constante	–65534–65535 (24)	<–65534 (25), >65535 (26)
Limite inferior especificado	sim (27), não (28)	
Limite superior para inferior vinculado	maior que (29), igual (30)	menor que (31)
Limite inferior especificado	negativo (32), zero (33), > 0 (34)	
O limite inferior é	constante (35), variável inteira (36)	nome do elemento da matriz (37), outra coisa (38)
O limite inferior é	um (39)	ub>1 (40), ub<1 (41)
Várias linhas	sim (42), não (43)	

O próximo passo é elaborar um ou mais casos de teste cobrindo o restante classes de equivalência válidas. Um caso de teste do formulário

DIMENSÃO A 12345 (I,9,J4XXXX,65535,1,KLM,
X,1000, BBB(-65534:100,0:1000,10:10, I:65535)

abrange as demais classes. As classes de equivalência de entrada inválidas e um caso de teste representando cada um, são:

- (3): DIMENSÃO (5):
DIMENSÃO (10)
- (6): DIMENSÃO A234567(2)
- (9): DIMENSÃO A.1(2)
- (11): DIMENSÃO 1A(10)
- (13): DIMENSÃO B (14):
DIMENSÃO B(4,4,4,4,4,4,4,4)
- (17): DIMENSÃO B(4,A(2))
- (18): DIMENSÃO B(4,,7)
- (21): DIMENSÃO C(I.,10)
- (23): DIMENSÃO C(10,1J)
- (25): DIMENSÃO D(- 65535:1)
- (26): DIMENSÃO D(65536)
- (31): DIMENSÃO D(4:3)
- (37): DIMENSÃO D(A(2):4)
- (38): D(:4)
- (43): DIMENSÃO D(0)

Assim, as classes de equivalência foram cobertas por 17 casos de teste. Você pode querer considerar como esses casos de teste se comparariam a um conjunto de casos de teste derivados de maneira ad hoc.

Embora o particionamento de equivalência seja muito superior a uma seleção aleatória de casos de teste, ele ainda apresenta deficiências. Ele ignora certos tipos de casos de teste de alto rendimento, por exemplo. As próximas duas metodologias, análise de valor limite e gráficos de causa e efeito, cobrem muitas dessas deficiências.

Análise de valor de limite

A experiência mostra que os casos de teste que exploram as condições de contorno têm um retorno maior do que os casos de teste que não exploram. As condições de fronteira são aquelas situações diretamente sobre, acima e abaixo das bordas das classes de equivalência de entrada e classes de equivalência de saída. A análise de valor limite difere da partição de equivalência em dois aspectos:

1. Em vez de selecionar qualquer elemento em uma classe de equivalência como sendo representativo, a análise de valor de contorno exige que um ou mais elementos sejam selecionados de modo que cada aresta da classe de equivalência seja objeto de um teste.
2. Em vez de apenas focar a atenção nas condições de entrada (espaço de entrada), os casos de teste também são derivados considerando o espaço de resultado (classes de equivalência de saída).

56 A Arte do Teste de Software

É difícil apresentar um "livro de receitas" para a análise de valor de fronteira, pois requer um grau de criatividade e uma certa especialização em relação ao problema em questão. (Portanto, como muitos outros aspectos do teste, é mais um estado de espírito do que qualquer outra coisa.) No entanto, algumas diretrizes gerais são necessárias:

1. Se uma condição de entrada especificar um intervalo de valores, escreva casos de teste para as extremidades do intervalo e casos de teste de entrada inválida para situações logo além das extremidades. Por exemplo, se o domínio válido de um valor de entrada for -1,0 a 1,0, escreva casos de teste para as situações -1,0, 1,0, -1,001 e 1,001.
2. Se uma condição de entrada especificar um número de valores, escreva casos de teste para o número mínimo e máximo de valores e um abaixo e além desses valores. Por exemplo, se um arquivo de entrada pode conter de 1 a 255 registros, escreva casos de teste para 0, 1, 255 e 256 registros.
3. Use a diretriz 1 para cada condição de saída. Por exemplo, se um programa de folha de pagamento calcular a dedução mensal do FICA e se o mínimo for \$ 0,00 e o máximo for \$ 1.165,25, escreva casos de teste que causem a dedução de \$ 0,00 e \$ 1.165,25. Além disso, veja se é possível inventar casos de teste que possam causar uma dedução negativa ou uma dedução de mais de \$ 1.165,25.

Observe que é importante examinar os limites do espaço de resultados porque nem sempre os limites dos domínios de entrada representam o mesmo conjunto de circunstâncias que os limites dos intervalos de saída (por exemplo, considere uma subrotina senoidal).. Além disso, nem sempre é possível gerar um resultado fora do intervalo de saída; no entanto, vale a pena considerar a possibilidade.

4. Use a diretriz 2 para cada condição de saída. Se um sistema de recuperação de informações exibe os resumos mais relevantes com base em uma solicitação de entrada, mas nunca mais de quatro resumos, escreva casos de teste de modo que o programa exiba zero, um e quatro resumos e escreva um caso de teste que possa causar o programa exibir erroneamente cinco resumos.
5. Se a entrada ou saída de um programa for um conjunto ordenado (um arquivo sequencial, por exemplo, ou uma lista linear ou uma tabela), concentre a atenção no primeiro e no último elemento do conjunto.
6. Além disso, use sua criatividade para procurar outros limites condições.

O programa de análise de triângulos do Capítulo 1 pode ilustrar a necessidade de análise de valor de contorno. Para que os valores de entrada representem um triângulo, eles devem ser números inteiros maiores que 0, onde a soma de quaisquer dois é maior que o terceiro. Se você estivesse definindo partições equivalentes, você poderia definir uma onde esta condição é atendida e outra onde a soma de dois dos inte

gers não é maior que o terceiro. Portanto, dois casos de teste possíveis podem ser 3–4–5 e 1–2–4. No entanto, perdemos um erro provável. Isto é, se uma expressão no programa fosse codificada como $A \neq B > C$ em vez de $A \neq B > C$, o programa nos diria erroneamente que 1–2–3 representa um triângulo escaleno válido. Portanto, a diferença importante entre a análise de valor de contorno e a partição de equivalência é que a análise de valor de contorno explora situações dentro e ao redor das bordas das partições de equivalência.

Como exemplo de uma análise de valor limite, considere a seguinte especificação de programa:

O MTEST é um programa que avalia exames de múltipla escolha. A entrada é um arquivo de dados chamado OCR, com vários registros com 80 caracteres. De acordo com a especificação do arquivo, o primeiro registro é um título usado como título em cada relatório de saída. O próximo conjunto de registros descreve as respostas corretas no exame. Esses registros contêm um "2" como o último caractere na coluna 80. No primeiro registro desse conjunto, o número de perguntas é listado nas colunas 1 a 3 (um valor de 1 a 999).

As colunas 10–59 contêm as respostas corretas para as questões 1–50 (qualquer caractere é válido como resposta). Os registros subsequentes contêm, nas colunas 10–59, as respostas corretas para as questões 51–100, 101–150 e assim por diante.

O terceiro conjunto de registros descreve as respostas de cada aluno; cada um desses registros contém um "3" na coluna 80. Para cada aluno, o primeiro registro contém o nome ou número do aluno nas colunas 1–

9 (qualquer caractere); as colunas 10–59 contêm as respostas do aluno para as questões 1–50. Se o teste tiver mais de 50 perguntas, os registros subsequentes do aluno conterão as respostas 51–100, 101–150 e assim por diante, nas colunas 10–59. O número máximo de alunos é 200. Os dados de entrada são ilustrados na Figura 4.4. Os quatro registros de saída são:

1. Um relatório, ordenado por identificador de aluno, mostrando a nota de cada aluno (porcentagem de acertos) e classificação.
2. Um relatório semelhante, mas classificado por grau.

58 A Arte do Teste de Software

		Título			
1					80
1	Nº de perguntas		Respostas corretas 1-50		2
1	3 4	9 10		59 60	79 80
1			Respostas corretas 51–100		2
1		9 10		59 60	79 80
1	Identificador do aluno		Respostas corretas 1-50		3
1		9 10		59 60	79 80
1			Respostas corretas 51–100		3
1		9 10		59 60	79 80
1	Identificador do aluno		Respostas corretas 1-50		3
1		9 10		59 60	79 80

FIGURA 4.4 Entrada para o Programa MTEST.

3. Um relatório indicando a média, mediana e desvio padrão dos graus.
4. Um relatório, ordenado por número de pergunta, mostrando a porcentagem de idade dos alunos que responderam corretamente a cada pergunta.

Podemos começar lendo metódicamente a especificação, procurando condições de entrada. A primeira condição de entrada de limite é um arquivo de entrada vazio. A segunda condição de entrada é o registro de título; condições de contorno são registro de título ausente e os títulos mais curtos e mais longos possíveis. Nas próximas condições de entrada são a presença de registros de respostas corretas e a campo de número de perguntas no primeiro registro de resposta. A classe de equivalência

pois o número de perguntas não é de 1 a 999, porque algo especial acontece a cada múltiplo de 50 (ou seja, são necessários vários registros). Uma partição razoável disso em classes de equivalência é 1–50 e 51–999.

Portanto, precisamos de casos de teste em que o campo de número de perguntas seja definido como 0, 1, 50, 51 e 999. Isso cobre a maioria das condições de limite para o número de registros de respostas corretas; no entanto, três situações mais interessantes são a ausência de registros de respostas e ter um registro de resposta a mais e um a menos (por exemplo, o número de perguntas é 60, mas há três registros de resposta em um caso e um registro de resposta no outro caso).

Os casos de teste exclusivos identificados até agora são:

1. Arquivo de entrada
vazio
2. Registro de título
ausente
3. Título de 1
caractere
4. Título de 80 caracteres
5. Exame de 1 questão
6. Exame de 50 questões
7. Exame de 51 questões
8. Exame de 999 questões
9. Exame de 0 questões
10. Campo de número de questões com valor não numérico
11. Não há registros de respostas corretas após o título
12. Muitos registros de respostas corretas
13. Poucos registros de respostas corretas

As próximas condições de entrada estão relacionadas às respostas dos alunos. Os casos de teste de valor limite aqui parecem ser:

14. 0 alunos
15. 1 aluno
16. 200 alunos
17. 201 alunos
18. Um aluno tem um registro de resposta, mas há duas respostas corretas
registros.
19. O aluno acima é o primeiro aluno do arquivo.
20. O aluno acima é o último aluno do arquivo.
21. Um aluno tem dois registros de respostas, mas há apenas um registro de respostas corretas.

60 A Arte do Teste de Software

22. O aluno acima é o primeiro aluno do arquivo.
23. O aluno acima é o último aluno do arquivo.

Você também pode derivar um conjunto útil de casos de teste examinando os limites de saída, embora alguns dos limites de saída (por exemplo, relatório vazio 1) sejam cobertos pelos casos de teste existentes. As condições de contorno dos relatórios 1 e 2 são:

- 0 alunos (o mesmo que o teste 14)
- 1 aluno (o mesmo que o teste 15)
- 200 alunos (o mesmo que o teste 16)

24. Todos os alunos recebem a mesma nota.
25. Todos os alunos recebem uma nota diferente.
26. Alguns alunos, mas não todos, recebem a mesma nota (para ver se as classificações são computadas corretamente).
27. Um aluno recebe uma nota 0.
28. Um aluno recebe uma nota 10.
29. Um aluno tem o menor valor de identificador possível (para verificar a classificação).
30. Um aluno tem o valor identificador mais alto possível.
31. O número de alunos é tal que o relatório é grande o suficiente para caber em uma página (para ver se uma página estranha é impressa).
32. O número de alunos é tal que todos os alunos, exceto um, cabem em um página.

As condições de contorno do relatório 3 (média, mediana e desvio padrão) são:

33. A média está no máximo (todos os alunos têm nota perfeita).
34. A média é 0 (todos os alunos recebem nota 0).
35. O desvio padrão está no máximo (um aluno recebe um 0 e o outro recebe 100).
36. O desvio padrão é 0 (todos os alunos recebem a mesma nota).

Os testes 33 e 34 também cobrem os limites da mediana. Outro caso de teste útil é a situação em que há 0 alunos (procurando uma divisão por 0 no cálculo da média), mas isso é idêntico ao caso de teste 14.

Um exame do relatório 4 produz os seguintes testes de valor limite:

37. Todos os alunos respondem corretamente à questão 1.
38. Todos os alunos respondem incorretamente à questão 1.
39. Todos os alunos respondem corretamente à última pergunta.
40. Todos os alunos respondem incorretamente à última pergunta.
41. O número de perguntas é tal que o relatório é grande o suficiente para caber em uma página.

42. O número de perguntas é tal que todas as perguntas, exceto uma, cabem em uma página.

Um programador experiente provavelmente concordaria neste ponto que muitos desses 42 casos de teste representam erros comuns que podem ter sido cometidos no desenvolvimento deste programa, mas a maioria desses erros provavelmente não seria detectada se um método de geração de casos de teste aleatório ou ad hoc fosse usado. A análise de valor limite, se praticada corretamente, é um dos métodos de projeto de caso de teste mais úteis. No entanto, muitas vezes é usado de forma ineficaz porque a técnica, na superfície, parece simples. Você deve entender que as condições de contorno podem ser muito sutis e, portanto, a identificação delas requer muita reflexão.

Representação gráfica de causa e efeito

Um ponto fraco da análise de valor limite e partição de equivalência é que eles não exploram combinações de circunstâncias de entrada. Por exemplo, talvez o programa MTEST da seção anterior falhe quando o produto do número de questões e o número de alunos excede algum limite (o programa fica sem memória, por exemplo). O teste de valor limite não detectaria necessariamente tal erro.

O teste de combinações de entrada não é uma tarefa simples porque mesmo se você particionar por equivalência as condições de entrada, o número de combinações geralmente é astronômico. Se você não tiver uma maneira sistemática de selecionar um subconjunto de condições de entrada, provavelmente selecionará um subconjunto arbitrário de condições, o que pode levar a um teste ineficaz.

A representação gráfica de causa-efeito auxilia na seleção, de forma sistemática, de um conjunto de casos de teste de alto rendimento. Tem um efeito colateral benéfico ao apontar incompletude e ambiguidades na especificação.

Um grafo causa-efeito é uma linguagem formal na qual uma linguagem natural especificação é traduzida. O gráfico na verdade é um circuito lógico digital (um rede lógica combinatória), mas em vez de notação eletrônica padrão, uma notação um pouco mais simples é usada. Nenhum conhecimento de eletrônica é necessário além da compreensão da lógica booleana (ou seja, dos operadores lógicos e, ou, e não).

O processo a seguir é usado para derivar casos de teste:

1. A especificação é dividida em partes viáveis. Isso é necessário porque os gráficos de causa e efeito tornam-se difíceis de manejá-los quando usados em especificações grandes. Por exemplo, ao testar um sistema de comércio eletrônico, uma solução viável peça pode ser a especificação para escolher e verificar um único item colocado em um carrinho de compras. Ao testar um design de página da Web, você pode teste uma única árvore de menu ou até mesmo uma sequência de navegação menos complexa.
2. As causas e efeitos na especificação são identificados. Uma causa é um condição de entrada distinta ou uma classe de equivalência de condições de entrada. Um efeito é uma condição de saída ou uma transformação do sistema (um efeito prolongado que uma entrada tem no estado do programa ou sistema). Por exemplo, se uma transação fizer com que um arquivo ou registro de banco de dados seja atualizada, a alteração é uma transformação do sistema; uma confirmação mensagem seria uma condição de saída.
Você identifica causas e efeitos lendo a palavra de especificação por palavra e sublinhando palavras ou frases que descrevem causas e efeitos.
Uma vez identificados, cada causa e efeito recebe um número único.
3. O conteúdo semântico da especificação é analisado e transformado em um gráfico booleano ligando as causas e os efeitos. Isto é o gráfico causa-efeito.
4. O gráfico é anotado com restrições que descrevem combinações de causas e/ou efeitos que são impossíveis por causa da sintaxe ou do ambiente restrições mentais.
5. Ao rastrear metodicamente as condições de estado no gráfico, você converte o gráfico em uma tabela de decisão de entrada limitada. Cada coluna da tabela representa um caso de teste.
6. As colunas na tabela de decisão são convertidas em casos de teste.

A notação básica para o gráfico é mostrada na Figura 4.5. Pense em cada um nó como tendo o valor 0 ou 1; 0 representa o estado "ausente" e 1 representa o estado "presente".

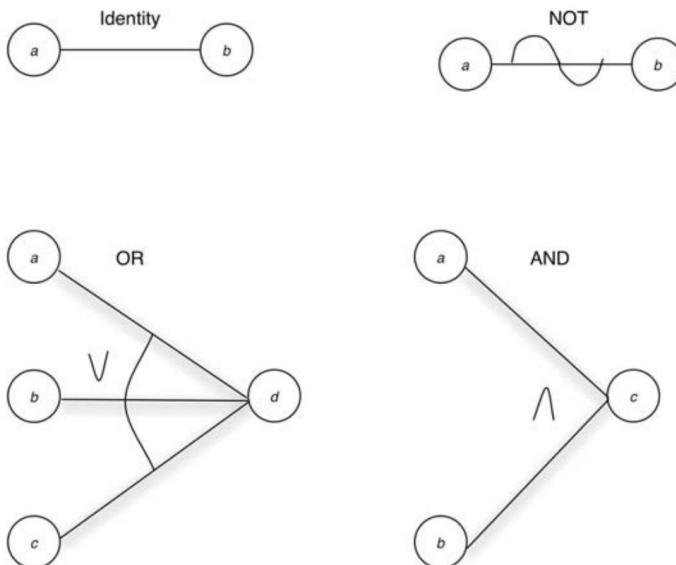


FIGURA 4.5 Símbolos básicos do gráfico de causa-efeito.

A função identidade afirma que se a é 1, b é 1; senão b é 0.

A função not afirma que se a é 1, b é 0, senão b é 1.

A função ou afirma que se a ou b ou c é 1, d é 1; senão d é 0.

A função and afirma que se a e b são 1, c é 1; senão c é 0.

As duas últimas funções (ou e e) podem ter qualquer número de entradas.

Para ilustrar um pequeno gráfico, considere a seguinte especificação:

O caractere na coluna 1 deve ser um "A" ou um "B". O caractere em a coluna 2 deve ser um dígito. Nessa situação, a atualização do arquivo é feita. Se o primeiro caractere estiver incorreto, a mensagem X12 será emitida. Se o segundo caractere não for um dígito, a mensagem X13 é emitida.

As causas são:

1—caractere na coluna 1 é "A" 2—

caractere na coluna 1 é "B" 3—

caractere na coluna 2 é um dígito

64 A Arte do Teste de Software

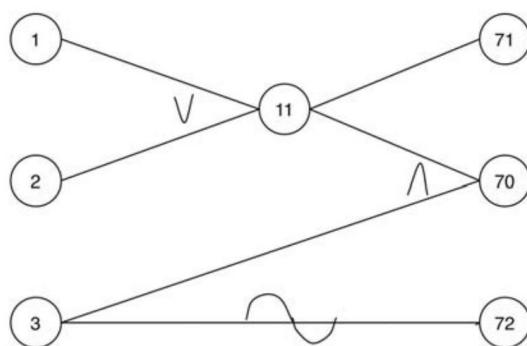


FIGURA 4.6 Exemplo de gráfico de causa-efeito.

e os efeitos são:

70—atualização feita

71—mensagem X12 é emitida

72—mensagem X13 é emitida

O gráfico causa-efeito é mostrado na Figura 4.6. Observe o nó intermediário 11 que foi criado. Você deve confirmar que o gráfico representa a especificação definindo todos os estados possíveis das causas e verificando se os efeitos estão definidos com os valores corretos. Para leitores familiarizados com diagramas lógicos, a Figura 4.7 é o circuito lógico equivalente.

Embora o gráfico da Figura 4.6 represente a especificação, ele contém uma combinação impossível de causas — é impossível que ambas as causas 1 e 2 sejam definidas como 1 simultaneamente. Na maioria dos programas, certas combinações de causas são impossíveis devido a considerações sintáticas ou ambientais (um caractere não pode ser um "A" e um "B" simultaneamente).

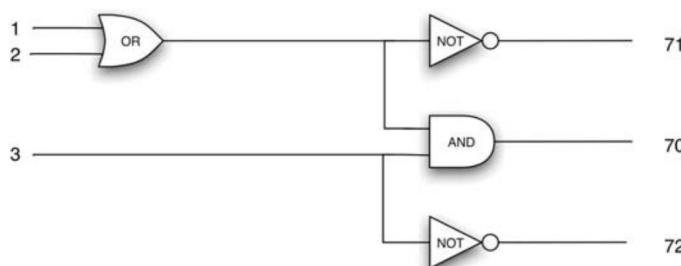


FIGURA 4.7 Diagrama Lógico Equivalente à Figura 4.6.

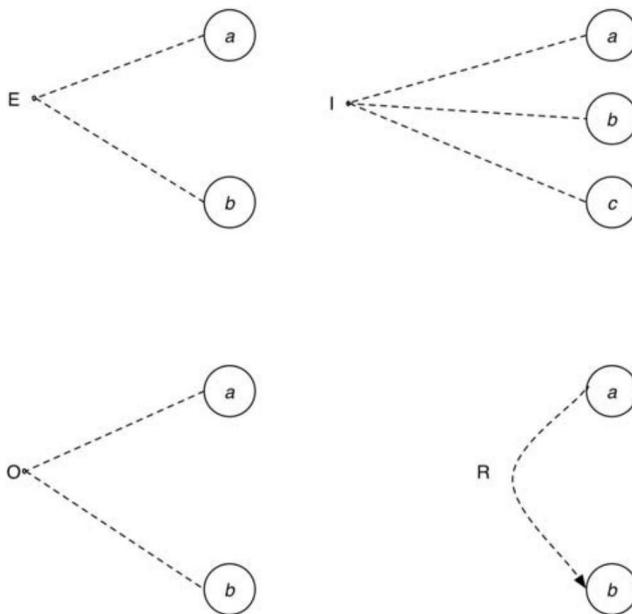


FIGURA 4.8 Símbolos de restrição.

Para explicar isso, a notação na Figura 4.8 é usada. A restrição E afirma que deve ser sempre verdade que, no máximo, um de a e b pode ser 1 (a e b não podem ser 1 simultaneamente). A restrição I afirma que pelo menos um de a, b e c deve ser sempre 1 (a, b e c não podem ser 0 simultaneamente).

A restrição O afirma que um, e apenas um, de a e b deve ser 1. A restrição R afirma que para a ser 1, b deve ser 1 (ou seja, é impossível que a seja 1 e b seja 0).

Frequentemente há a necessidade de uma restrição entre os efeitos. O M contrastrain na Figura 4.9 afirma que se o efeito a é 1, o efeito b é forçado a 0.

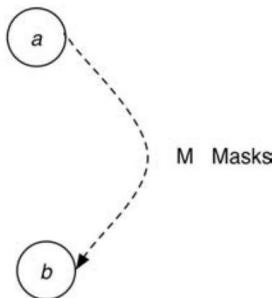


FIGURA 4.9 Símbolo para restrição "Máscaras".

66 A Arte do Teste de Software

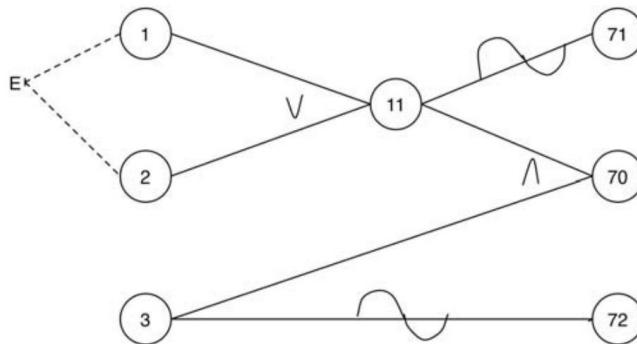


FIGURA 4.10 Exemplo de gráfico de causa-efeito com restrição "exclusiva".

Voltando ao exemplo simples anterior, vemos que é fisicamente impossível que as causas 1 e 2 estejam presentes simultaneamente, mas é possível que nenhuma delas esteja presente. Portanto, elas estão ligadas à restrição E, conforme mostrado na Figura 4.10.

Para ilustrar como o gráfico de causa e efeito é usado para derivar casos de teste, use a seguinte especificação para um comando de depuração em um interativo sistema.

O comando DISPLAY é usado para visualizar a partir de uma janela de terminal o conteúdo dos locais de memória. A sintaxe do comando é mostrada em Figura 4.11. Os colchetes representam operandos opcionais alternativos. Letras maiúsculas representam palavras-chave de operandos. As letras minúsculas representam valores dos operandos (os valores reais devem ser substituídos). Os operandos sublinhados representam os valores padrão (ou seja, o valor usado quando o operando é omitido).

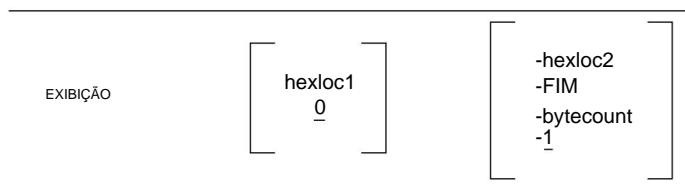


FIGURA 4.11 Sintaxe do Comando DISPLAY.

O primeiro operando (hexloc1) especifica o endereço do primeiro byte cujo conteúdo deve ser exibido. O endereço pode ter de um a seis dígitos hexadecimais (0–9, A–F) de comprimento. Se não for especificado, o endereço 0 é assumido. O endereço deve estar dentro do intervalo de memória real da máquina.

O segundo operando especifica a quantidade de memória a ser exibida. Se hexloc2 for especificado, ele define o endereço do último byte no intervalo de locais a serem exibidos. Pode ter de um a seis dígitos hexadecimais de comprimento. O endereço deve ser maior ou igual ao endereço inicial (hexloc1). Além disso, hexloc2 deve estar dentro do intervalo de memória real da máquina. Se END for especificado, a memória será exibida até o último byte real na máquina. Se for especificado bytecount , define o número de bytes de memória a serem exibidos (começando com o local especificado em hexloc1). O operando bytecount é um inteiro hexadecimal (um a seis dígitos). A soma de bytecount e hexloc1 não deve exceder o tamanho real da memória mais 1 e bytecount deve ter um valor de pelo menos 1.

Quando o conteúdo da memória é exibido, o formato de saída na tela é uma ou mais linhas do formato

```
xxxxxx % palavra1 palavra2 palavra3 palavra4
```

onde xxxxxx é o endereço hexadecimal da palavra1. Um número inteiro de palavras (sequências de quatro bytes, onde o endereço do primeiro byte na palavra é um múltiplo de 4) é sempre exibido, independentemente do valor de hexloc1 ou da quantidade de memória a ser exibida. Todas as linhas de saída sempre conterão quatro palavras (16 bytes). O primeiro byte do intervalo exibido cairá na primeira palavra.

As mensagens de erro que podem ser produzidas são

M1 é uma sintaxe de comando inválida.

A memória M2 solicitada está além do limite de memória real.

A memória M3 solicitada é um intervalo zero ou negativo.

Como exemplos:

EXIBIÇÃO

exibe as primeiras quatro palavras na memória (endereço inicial padrão de 0, contagem de bytes padrão de 1);

MOSTRADOR 77F

exibe a palavra que contém o byte no endereço 77F e as três palavras subsequentes;

MOSTRADOR 77F-407A

exibe as palavras que contêm os bytes no intervalo de endereços 775-407A;

MOSTRADOR 77F.6

exibe as palavras contendo os seis bytes começando no local 77F; e

EXIBIR 50FF-END

exibe as palavras contendo os bytes no intervalo de endereços 50FF até o final da memória.

O primeiro passo é uma análise cuidadosa da especificação para identificar os causas e efeitos. As causas são as seguintes:

1. O primeiro operando está presente.
2. O operando hexloc1 contém apenas dígitos hexadecimais.
3. O operando hexloc1 contém de um a seis caracteres.
4. O operando hexloc1 está dentro da faixa de memória real da máquina.
5. O segundo operando é END.
6. O segundo operando é hexloc.
7. O segundo operando é bytecount.
8. O segundo operando é omitido.
9. O operando hexloc2 contém apenas dígitos hexadecimais.
10. O operando hexloc2 contém de um a seis caracteres.
11. O operando hexloc2 está dentro da faixa de memória real da máquina.
12. O operando hexloc2 é maior ou igual ao operando hexloc1.
13. O operando bytecount contém apenas dígitos hexadecimais.
14. O operando bytecount contém de um a seis caracteres.

15. bytecount β hexloc1 $<\frac{1}{4}$ tamanho da memória β
1. 16. bytecount $>\frac{1}{4}$ 1.
17. A faixa especificada é grande o suficiente para exigir várias linhas de saída.
18. O início do intervalo não cai em um limite de palavra.

Cada causa recebeu um número único arbitrário. Observe que quatro causas (5 a 8) são necessárias para o segundo operando porque o segundo operando pode ser (1) END, (2) hexloc2, (3) byte-count, (4) ausente e (5) nenhum dos o de cima. Os efeitos são os seguintes:

91. A mensagem M1 é exibida.
92. A mensagem M2 é exibida.
93. A mensagem M3 é exibida.
94. A memória é exibida em uma linha.
95. A memória é exibida em várias linhas.
96. O primeiro byte do intervalo exibido cai em um limite de palavra.
97. O primeiro byte do intervalo exibido não cai em um limite de palavra.

O próximo passo é o desenvolvimento do gráfico. Os nós de causa são listados verticalmente no lado esquerdo da folha de papel; os nós de efeito são listados verticalmente no lado direito. O conteúdo semântico da especificação é cuidadosamente analisado para interligar as causas e os efeitos (ou seja, para mostrar em que condições um efeito está presente).

A Figura 4.12 mostra uma versão inicial do gráfico. O nó intermediário 32 representa um primeiro operando sintaticamente válido; o nó 35 representa um segundo operando sintaticamente válido. O nó 36 representa um comando sintaticamente válido. Se o nó 36 for 1, o efeito 91 (a mensagem de erro) não aparecerá. Se o nó 36 for 0, o efeito 91 está presente.

O gráfico completo é mostrado na Figura 4.13. Você deve explorá-lo com cuidado para se convencer de que ele reflete com precisão a especificação.

Se a Figura 4.13 fosse usada para derivar os casos de teste, muitos casos de teste impossíveis de criar seriam derivados. A razão é que certas combinações de causas são impossíveis por causa de restrições sintáticas. Por exemplo, as causas 2 e 3 não podem estar presentes a menos que a causa 1 esteja presente. A causa 4 não pode estar presente a menos que ambas as causas 2 e 3 estejam presentes. A Figura 4.14 contém o gráfico completo com as condições de restrição. Observe que, no máximo, uma das causas 5, 6, 7 e 8 pode estar presente. Todas as outras restrições de causa são a condição requerer. Observe que a causa 17 (várias linhas de saída) requer o não da causa 8

70 A Arte do Teste de Software

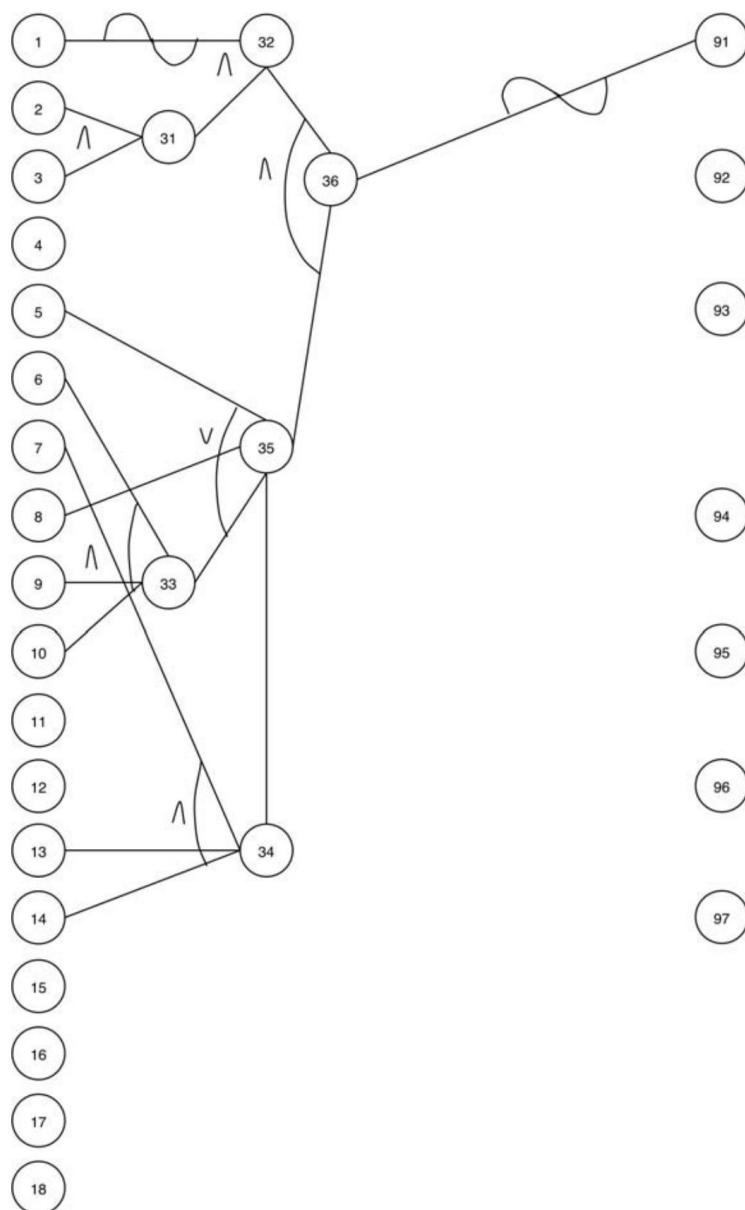


FIGURA 4.12 Início do Gráfico para o Comando DISPLAY.

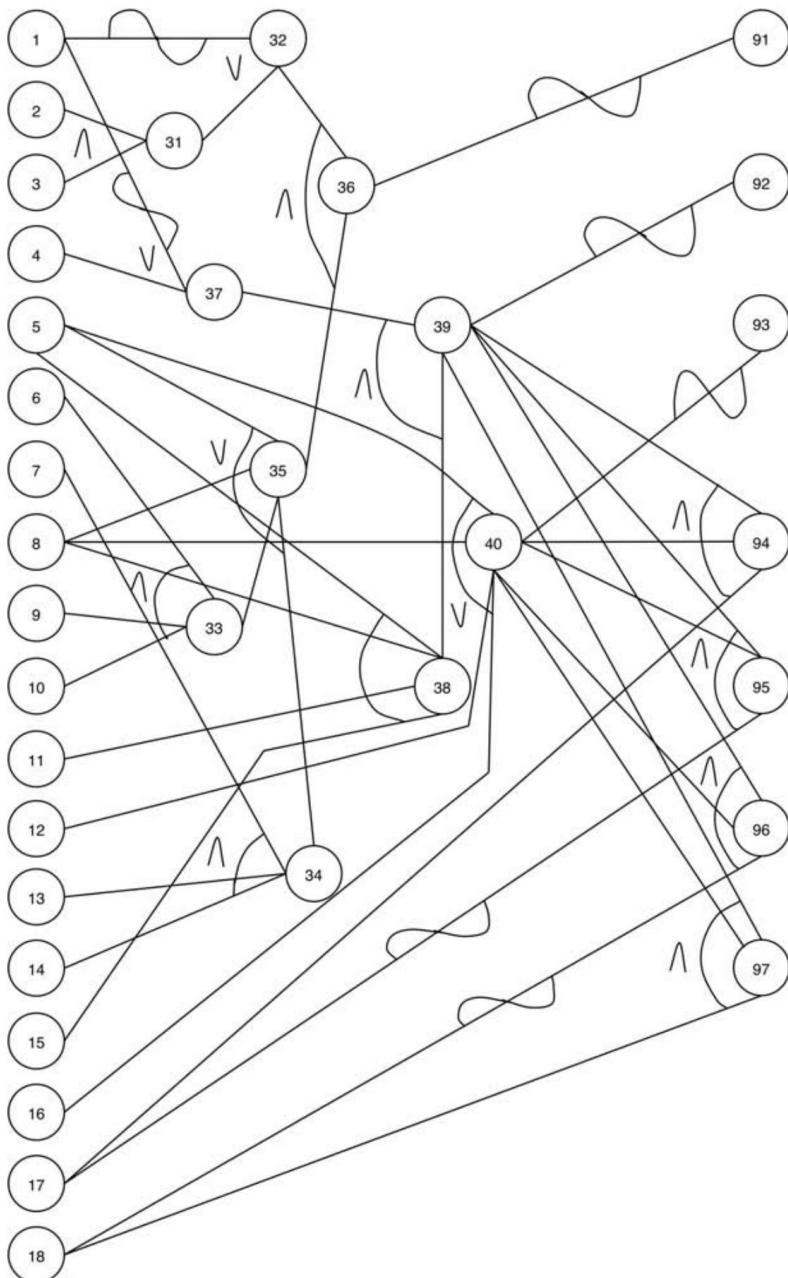


FIGURA 4.13 Gráfico completo de causa-efeito sem restrições.

72 A Arte do Teste de Software

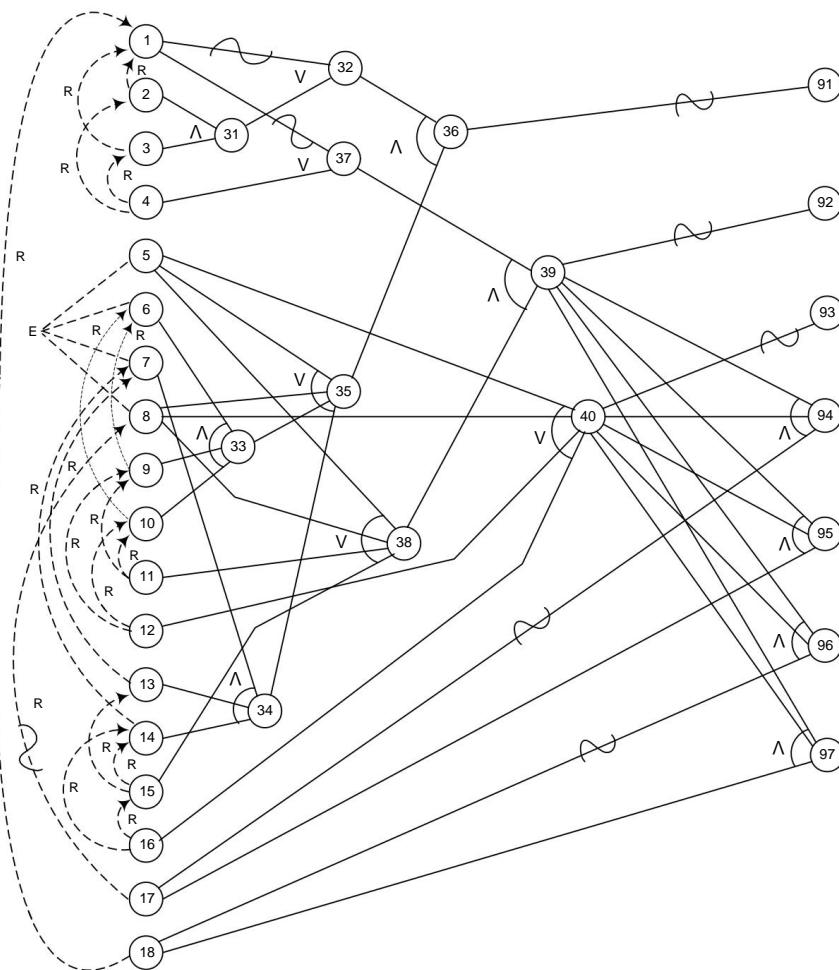


FIGURA 4.14 Gráfico completo de causa-efeito do comando DISPLAY.

(o segundo operando é omitido); a causa 17 pode estar presente somente quando a causa 8 está ausente. Novamente, você deve explorar as condições de restrição com cuidado.

O próximo passo é a geração de uma tabela de decisão de entrada limitada. Para leitores familiarizados com tabelas de decisão, as causas são as condições e os efeitos são as ações. O procedimento utilizado é o seguinte:

1. Selecione um efeito para ser o estado atual (1).
2. Voltando ao gráfico, encontre todas as combinações de causas (sujeitas às restrições) que definirão esse efeito como 1.
3. Crie uma coluna na tabela de decisão para cada combinação de causas.

4. Para cada combinação, determine os estados de todos os outros efeitos e coloque-os em cada coluna.

Na execução da etapa 2, as considerações são as seguintes:

1. Ao rastrear de volta através de um nó ou cuja saída deve ser 1, nunca defina mais de uma entrada para ou para 1 simultaneamente. Isso é chamado de sensibilização de caminho. Seu objetivo é evitar a falha na detecção de certos erros por causa de uma causa mascarando outra causa.
2. Ao rastrear de volta através de um nó e cuja saída deve ser 0, todas as combinações de entradas que levam à saída 0 devem, é claro, ser enumeradas. No entanto, se você estiver explorando a situação em que uma entrada é 0 e uma ou mais das outras são 1, não é necessário enumerar todas as condições sob as quais as outras entradas podem ser 1.
3. Ao rastrear de volta através de um nó e cuja saída deve ser 0, apenas uma condição em que todas as entradas são zero precisa ser enumerada. (Se e está no meio do gráfico de tal forma que suas entradas vêm de outros nós intermediários, pode haver um número excessivamente grande de situações em que todas as suas entradas são 0.)

Essas considerações complicadas estão resumidas na Figura 4.15, e A Figura 4.16 é usada como exemplo.

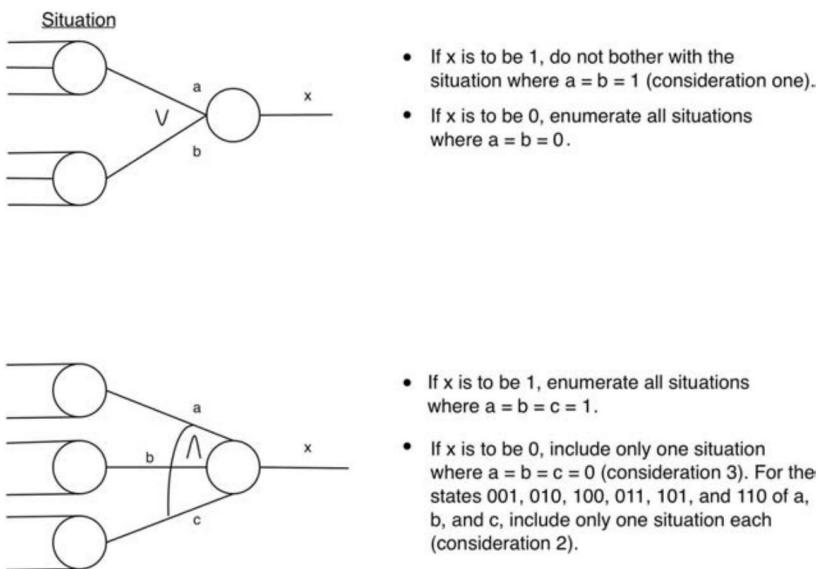


FIGURA 4.15 Considerações usadas ao traçar o gráfico.

74 A Arte do Teste de Software

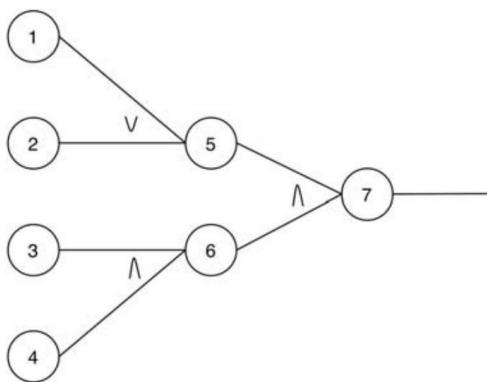


FIGURA 4.16 Exemplo de gráfico para ilustrar as considerações de rastreamento.

Suponha que queremos localizar todas as condições de entrada que causam a saída state seja 0. A consideração 3 afirma que devemos listar apenas uma circunstância onde os nós 5 e 6 são 0. A consideração 2 afirma que para o estado onde o nó 5 é 1 e o nó 6 é 0, devemos listar apenas uma circunstância onde o nó 5 é 1, em vez de enumerar todas as maneiras possíveis que o nó 5 pode ser 1. Da mesma forma, para o estado em que o nó 5 é 0 e o nó 6 é 1, temos deve listar apenas uma circunstância em que o nó 6 é 1 (embora haja apenas um neste exemplo). A consideração 1 afirma que onde o nó 5 deve ser definido como 1, não devemos definir os nós 1 e 2 como 1 simultaneamente. Daí, nós chegaria a cinco estados dos nós 1 a 4; por exemplo, os valores:

0	0	0	0	$(5 \setminus 0, 6 \setminus 0)$
1	0	0	0	$(5 \setminus 1, 6 \setminus 0)$
1	0	0	1	$(5 \setminus 1, 6 \setminus 0)$
1	0	1	0	$(5 \setminus 1, 6 \setminus 0)$
0	0	1	1	$(5 \setminus 0, 6 \setminus 1)$

em vez dos 13 estados possíveis de nós 1 a 4 que levam a um 0 estado de saída.

Essas considerações podem parecer caprichosas, mas têm um propósito importante: diminuir os efeitos combinados do gráfico. Eles eliminam situações que tendem a ser casos de teste de baixo rendimento. Se casos de teste de baixo rendimento não forem eliminados, um grande gráfico de causa e efeito produzirá um número de casos de teste. Se o número de casos de teste for muito grande para ser prático,

você selecionará algum subconjunto, mas não há garantia de que o baixo rendimento casos de teste serão os eliminados. Por isso, é melhor eliminá-los durante a análise do gráfico.

Vamos agora converter o gráfico de causa-efeito da Figura 4.14 na tabela de decisão. O efeito 91 será selecionado primeiro. O efeito 91 está presente se o nó 36 estiver 0. O nó 36 é 0 se os nós 32 e 35 são 0,0; 0,1; ou 1,0; e considerações 2 e 3 aplicam-se aqui. Ao rastrear as causas e considerar as restrições entre as causas, você pode encontrar as combinações de causas que levam a efeito 91 estar presente, embora fazê-lo seja um processo trabalhoso.

A tabela de decisão resultante, sob a condição de que o efeito 91 esteja presente, é mostrado na Figura 4.17 (colunas 1 a 11). Colunas (testes) 1 a 3 representam as condições em que o nó 32 é 0 e o nó 35 é 1. Colunas 4 a 10 representam as condições em que o nó 32 é 1 e o nó 35 é 0.

Usando a consideração 3, apenas uma situação (coluna 11) de 21 possíveis situações em que os nós 32 e 35 são 0 são identificadas. Os espaços em branco na tabela representam situações de "não me importo" (ou seja, o estado da causa é irrelevante) ou indicam que o estado de uma causa é óbvio por causa dos estados de outras causas dependentes (por exemplo, na coluna 1, sabemos que as causas 5, 7 e 8 devem ser 0 porque eles existem em uma situação de "no máximo um" com causa 6).

As colunas 12 a 15 representam as situações em que o efeito 92 está presente. As colunas 16 e 17 representam as situações em que o efeito 93 está presente. A Figura 4.18 representa o restante da tabela de decisão.

O último passo é converter a tabela de decisão em 38 casos de teste. Um conjunto de 38 casos de teste estão listados aqui. O número ou números ao lado de cada caso de teste designar os efeitos que se espera que estejam presentes. Suponha que o último localização na memória na máquina que está sendo usada é 7FFF.

1	EXIBIÇÃO 234AF74-123	(91)
2	MOSTRADOR 2ZX4-3000	(91)
3	MOSTRAR HHHHHHHH-2000	(91)
4	EXIBIR 200 200	(91)
5	DISPLAY 0-22222222	(91)
6	EXIBIR 1-2X	(91)
7	MOSTRADOR 2-ABCDEFGHI	(91)
8	MOSTRADOR 3.1111111	(91)
9	MOSTRAR 44.\$42	(91)
10	EXIBIR 100\$\$\$\$\$\$\$\$	(91)

76 A Arte do Teste de Software

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
2	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
3	0	1	0	1			1	1	1	0	1			1		1	1	
4													1	0	0	1	1	
5				0											1			
6	1	1	1	0	1	1					1	1	1	1		1	1	
7				0				1	1	1	1							
8				0														
9	1	1	1	1	0	0					0	1				1		
10	1	1	1	0	1	0					1	1				1		
11											0				0	1		
12																0		
13							1	0	0	1							1	
14							0	1	0	1							1	
15											0							
16																0		
17																		
18																		
91	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
92	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0
93	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
94	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
95	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
96	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
97	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

FIGURA 4.17 Primeira Metade da Tabela de Decisão Resultante.

	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	3534	36	37	38	
1	1	1	1	1	1	0	0	0	10		1	1	1	1	11		0	0	0	11	1
2		1	1	11						1	1	1			1				1	1	1
3	1	1	1	1						1	1	1			1				1	1	1
4	1	1	1	1						1	1	1	1		1				1	1	1
5	1					1					1	1	1								
6			1						1	1	1	1									
7				1						1	1	1	1								
8	1	1	1																		
9				1						1	1	1	1								
10				1						1	1	1	1								
11				1						1	1	1	1								
12				1						1	1	1	1								
13				1						1	1	1	1								
14				1						1	1	1	1								
15				1						1	1	1	1								
16				1						1	1	1	1								
17	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1		1		1		
18	1	1	1	1	0	0	0	0	1	1					1	0	0	0		0	
91	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
92	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
93	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
94	1	1	1	1	1	1	1				1	1	1	0	0		0	0	0	0	0
95	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1		1		1		
96	0	0	0	1	1	1				1	1	1	0	0	1	1		1		1	
97	1	1	1	1	0	0	0	0	1	1					1	0	0	0	0	0	0

FIGURA 4.18 Segunda Metade da Tabela de Decisão Resultante.

- 11 EXIBIR 10000000-M (91)
- 12 MOSTRADOR FF-8000 (92)
- 13 EXIBIR FFF.7001 (92)
- 14 DISPLAY 8000-END (92)
- 15 MOSTRADOR 8000–8001 (92)
- 16 VISOR AA-A9 (93)
- 17 EXIBIR 7.000,0 (93)
- 18 EXIBIÇÃO 7FFF9-END (94, 97)

19	MOSTRADOR 1	(94, 97)
20	EXIBIÇÃO 21-29	(94, 97)
21	MOSTRADOR 4021.A	(94, 97)
22	EXIBIR - FIM	(94, 96)
23	EXIBIÇÃO	(94, 96)
24	MOSTRAR -F	(94, 96)
25	EXIBIR .E	(94, 96)
26	EXIBIÇÃO 7FF8-END	(94, 96)
27	MOSTRADOR 6000	(94, 96)
28	VISOR A0-A4	(94, 96)
29	MOSTRADOR 20.8	(94, 96)
30	DISPLAY 7001-END	(95, 97)
31	EXIBIÇÃO 5-15	(95, 97)
32 w	MOSTRADOR 4FF.100	(95, 97)
33	EXIBIR - FIM	(95, 96)
34	MOSTRAR -20	(95, 96)
35	MOSTRAR .11	(95, 96)
36	DISPLAY 7000-END	(95, 96)
37	EXIBIÇÃO 4-14	(95, 96)
38	EXIBIR 500.11	(95, 96)

Observe que onde dois ou mais casos de teste diferentes são invocados, para a maioria parte, o mesmo conjunto de causas, diferentes valores para as causas foram selecionados para melhorar ligeiramente o rendimento dos casos de teste. Observe também que, devido à tamanho real de armazenamento, caso de teste 22 é impossível (produzirá efeito 95 em vez de 94, conforme observado no caso de teste 33). Assim, 37 casos de teste foram identificados.

Comentários A representação gráfica de causa-efeito é um método sistemático de gerar casos representando combinações de condições. A alternativa seria fazer uma seleção ad hoc de combinações; mas ao fazê-lo, é provável que você ignoraria muitos dos casos de teste "interessantes" identificados pelo gráfico de causa e efeito.

Como o gráfico de causa e efeito requer a tradução de uma especificação em uma rede lógica booleana, ela oferece uma perspectiva diferente e uma visão adicional da especificação. Na verdade, o desenvolvimento de uma causa

gráfico de efeito é uma boa maneira de descobrir ambiguidades e incompletude nas especificações. Por exemplo, o leitor astuto pode ter notado que este processo descobriu um problema na especificação do comando DISPLAY . A especificação afirma que todas as linhas de saída contêm quatro palavras.

Isso não pode ser verdade em todos os casos; ele não pode ocorrer para os casos de teste 18 e 26 porque o endereço inicial está a menos de 16 bytes do final da memória.

Embora o gráfico de causa e efeito produza um conjunto de casos de teste úteis, normalmente não produz todos os casos de teste úteis que podem ser identificados. Por exemplo, no exemplo não dissemos nada sobre verificar se os valores de memória exibidos são idênticos aos valores na memória e determinar se o programa pode exibir todos os valores possíveis em um local de memória. Além disso, o gráfico causa-efeito não explora adequadamente as condições de contorno. Claro, você pode tentar cobrir as condições de contorno durante o processo. Por exemplo, em vez de identificar a causa única

$\text{hexloc2} > \frac{1}{4} \text{hexloc1}$

você pode identificar duas causas:

$\text{hexloc2} \frac{1}{4} \text{hexloc1}$
 $\text{hexloc2} > \text{hexloc1}$

O problema em fazer isso, no entanto, é que complica tremendamente o gráfico e leva a um número excessivamente grande de casos de teste. Por esta razão, é melhor considerar uma análise de valor de limite separada. Por exemplo, as seguintes condições de contorno podem ser identificadas para a especificação DISPLAY :

1. hexloc1 tem um dígito
2. hexloc1 tem seis dígitos
3. hexloc1 tem sete dígitos 4.
 $\text{hexloc1} \frac{1}{4} 0$ 5. $\text{hexloc1} \frac{1}{4}$
7FFF 6. $\text{hexloc1} \frac{1}{4} 8000$ 7.
- hexloc2 tem um dígito 8.
- hexloc2 tem seis dígitos 9.
- hexloc2 tem sete dígitos 10.
- $\text{hexloc2} \frac{1}{4} 0$

11. hexloc2 ¼ 7FFF
12. hexloc2 ¼ 8000
13. hexloc2 ¼ hexloc
14. hexloc2 ¼ hexloc1 þ 1
15. hexloc2 ¼ hexloc1 1
16. bytecount tem um dígito
17. bytecount tem seis dígitos
18. bytecount tem sete dígitos 19.
bytecount ¼ 1 20. hexloc1 þ
bytecount ¼ 8000 21. hexloc1 þ
bytecount ¼ 8001 22. display 16
bytes (uma linha) 23. display 17
bytes (duas linhas)

Observe que isso não implica que você escreveria 60 (37 þ 23) casos de teste. Como o gráfico de causa-efeito nos dá margem de manobra na seleção de valores específicos para operandos, as condições de contorno podem ser combinadas nos casos de teste derivados do gráfico de causa-efeito. Neste exemplo, reescrevendo alguns dos 37 casos de teste originais, todas as 23 condições de contorno podem ser cobertas sem nenhum caso de teste adicional. Assim, chegamos a um conjunto pequeno, mas potente, de casos de teste que satisfazem ambos os objetivos.

Observe que o gráfico de causa e efeito é consistente com vários dos princípios de teste do Capítulo 2. A identificação da saída esperada de cada caso de teste é uma parte inerente da técnica (cada coluna na tabela de decisão indica os efeitos esperados). Observe também que isso nos encoraja a procurar efeitos colaterais indesejados. Por exemplo, a coluna (teste) 1 especifica que devemos esperar que o efeito 91 esteja presente e que os efeitos 92 a 97 estejam ausentes.

O aspecto mais difícil da técnica é a conversão do gráfico em tabela de decisão. Esse processo é algorítmico, o que implica que você pode automatizá-lo escrevendo um programa; existem vários programas comerciais para ajudar na conversão.

Erro ao adivinhar

Tem-se observado com frequência que algumas pessoas parecem ser naturalmente hábeis em testes de programas. Sem usar nenhuma metodologia específica, como limite

análise de valor de gráficos de causa e efeito, essas pessoas parecem ter um talento especial para farejar erros.

Uma explicação para isso é que essas pessoas estão praticando - subconscientemente com mais frequência do que não - uma técnica de projeto de caso de teste que poderia ser chamada de adivinhação de erros. Dado um programa específico, eles supõem – tanto por intuição quanto por experiência – certos tipos prováveis de erros e então escrevem casos de teste para expor esses erros.

É difícil fornecer um procedimento para a técnica de adivinhação de erros, uma vez que é em grande parte um processo intuitivo e ad hoc. A idéia básica é enumerar uma lista de possíveis erros ou situações propensas a erros e então escrever casos de teste com base na lista. Por exemplo, a presença do valor 0 na entrada de um programa é uma situação propensa a erros. Portanto, você pode escrever casos de teste para os quais valores de entrada específicos têm um valor 0 e para os quais valores de saída específicos são forçados a 0. Além disso, onde um número variável de entradas ou saídas pode estar presente (por exemplo, o número de entradas em um lista a ser pesquisada), os casos de "nenhum" e "um" (por exemplo, lista vazia, lista contendo apenas uma entrada) são situações propensas a erros. Outra ideia é identificar casos de teste associados a suposições que o programador possa ter feito ao ler a especificação (ou seja, fatores que foram omitidos da especificação, seja por acidente ou porque o redator achou que eram óbvios).

Uma vez que um procedimento para adivinhação de erros não pode ser dado, a próxima melhor alternativa é discutir o espírito da prática, e a melhor maneira de fazer isso é apresentando exemplos. Se você estiver testando uma sub-rotina de classificação, as seguintes situações são a serem exploradas:

- A lista de entrada está vazia.
- A lista de entrada contém uma entrada.
- Todas as entradas na lista de entrada têm o mesmo valor.
- A lista de entrada já está classificada.

Em outras palavras, você enumera os casos especiais que podem ter sido negligenciados quando o programa foi projetado. Se estiver testando uma sub-rotina de pesquisa binária, você pode tentar as situações em que: (1) há apenas uma entrada na tabela que está sendo pesquisada; (2) o tamanho da tabela é uma potência de 2 (por exemplo, 16); e (3) o tamanho da tabela é um menor e um maior que uma potência de 2 (por exemplo, 15 ou 17).

82 A Arte do Teste de Software

Considere o programa MTEST na seção sobre análise de valor limite. Os seguintes testes adicionais vêm à mente ao usar a técnica de adivinhação de erros:

O programa aceita "em branco" como resposta?

Um registro tipo 2 (resposta) aparece no conjunto de registros tipo 3 (aluno).

Um registro sem 2 ou 3 na última coluna aparece como diferente do registro inicial (título).

Dois alunos têm o mesmo nome ou número.

Como a mediana é calculada de forma diferente dependendo se há um número par ou ímpar de itens, teste o programa para um número par de alunos e um número ímpar de alunos.

O campo número de perguntas tem um valor negativo.

Testes de adivinhação de erros que vêm à mente para o comando DISPLAY da seção anterior são as seguintes:

DISPLAY 100- (segundo operando parcial)

DISPLAY 100. (segundo operando parcial)

DISPLAY 100-10A 42 (extra operando)

DISPLAY 000-0000FF (zeros à esquerda)

A estratégia

As metodologias de projeto de casos de teste discutidas neste capítulo podem ser combinadas em uma estratégia geral. A razão para combiná-los já deve ser óbvia: cada um contribui com um conjunto específico de casos de teste úteis, mas nenhum deles por si só contribui com um conjunto completo de casos de teste. Uma estratégia razoável é a seguinte:

1. Se a especificação contiver combinações de condições de entrada, inicie com gráficos de causa e efeito.
2. Em qualquer caso, use a análise de valor limite. Lembre-se de que esta é uma análise dos limites de entrada e saída. A análise do valor limite produz um conjunto de condições de teste suplementares, mas, conforme observado na seção sobre gráficos de causa e efeito, muitas ou todas elas podem ser incorporadas aos testes de causa e efeito.

3. Identifique as classes de equivalência válidas e inválidas para a entrada e saída e complemente os casos de teste identificados acima, se necessário.
4. Use a técnica de adivinhação de erros para adicionar casos de teste adicionais.
5. Examine a lógica do programa em relação ao conjunto de casos de teste. Use o critério de cobertura de decisão, cobertura de condição, idade de cobertura de decisão/condição ou critério de cobertura de várias condições (sendo o último o mais completo). Se o critério de cobertura não foi atendido pelos casos de teste identificados nas quatro etapas anteriores, e se atender ao critério não for impossível (ou seja, certas combinações de condições podem ser impossíveis de criar devido à natureza do programa), adicione casos de teste suficientes para fazer com que o critério seja satisfeito.

Novamente, o uso dessa estratégia não garante que todos os erros sejam encontrados, mas foi considerado um compromisso razoável. Além disso, representa uma quantidade considerável de trabalho árduo, mas, como dissemos no início deste capítulo, ninguém jamais afirmou que testar programas é fácil.

Resumo

Depois de concordar que o teste de software agressivo é uma adição valiosa aos seus esforços de desenvolvimento, a próxima etapa é projetar casos de teste que exercitarião seu aplicativo o suficiente para produzir resultados de teste satisfatórios. Na maioria dos casos, considere uma combinação de metodologias de caixa preta e caixa branca para garantir que você tenha projetado testes rigorosos de programa.

As técnicas de projeto de caso de teste discutidas neste capítulo incluem:

Cobertura lógica. Testes que exercitam todos os resultados do ponto de decisão pelo menos uma vez e garantem que todas as instruções ou pontos de entrada sejam executados pelo menos uma vez.

Particionamento equivalente. Define condições ou classes de erro para ajudar a reduzir o número de testes finitos. Assume que um teste de um valor representativo dentro de uma classe também testa todos os valores ou condições dentro dessa classe.

Análise de valor limite. Testa cada condição de aresta de uma classe de equivalência; também considera classes de equivalência de saída, bem como classes de entrada.

Gráficos de causa-efeito. Produz representações gráficas booleanas de resultados de casos de teste em potencial para auxiliar na seleção eficiente e completa casos de teste.

84 A Arte do Teste de Software

Erro ao adivinhar. Produz casos de teste com base no conhecimento intuitivo e especializado dos membros da equipe de teste para definir possíveis erros de software para facilitar o design eficiente do caso de teste.

Testes extensivos e aprofundados não são fáceis; nem o projeto de caso de teste mais extenso garantirá que todos os erros sejam descobertos. Dito isso, os desenvolvedores dispostos a ir além dos testes superficiais, que dedicarão tempo suficiente ao design do caso de teste, analisarão cuidadosamente os resultados do teste e agirão decisivamente sobre as descobertas, serão recompensados com um software funcional e confiável, razoavelmente livre de erros.

5

Teste do Módulo (Unidade)

Até este ponto ignoramos amplamente a mecânica de testes e o ambiente de programa que está sendo testado. No entanto, como grandes programas (digamos, de 500 instruções ou mais de 50 classes) requerem tratamento de teste especial, neste capítulo, consideraremos um passo inicial na estruturação do teste de um grande programa: teste de módulo. Os capítulos 6 e 7 enumeram as etapas restantes.

Teste de módulo (ou teste de unidade) é um processo de teste de subprogramas, subrotinas, classes ou procedimentos individuais em um programa. Mais especificamente, em vez de testar inicialmente o programa como um todo, testar é primeiro focado nos blocos de construção menores do programa. As motivações para fazer isso são três. Primeiro, o teste de módulo é uma maneira de gerenciar os elementos combinados de teste, uma vez que a atenção está focada inicialmente em unidades do programa. Em segundo lugar, o teste de módulo facilita a tarefa de depuração (o processo de identificar e corrigir um erro descoberto), uma vez que, quando um erro for encontrado, ele é conhecido por existir em um módulo específico. Finalmente, o teste de módulo introduz o paralelismo no processo de teste de programa, apresentando-nos a oportunidade de testar vários módulos simultaneamente.

O objetivo do teste de módulo é comparar a função de um módulo com alguma especificação funcional ou de interface que define o módulo. Para enfatizar novamente o objetivo de todos os processos de teste, o objetivo aqui não é mostrar que o módulo atende sua especificação, mas que o módulo contradiz a especificação. Neste capítulo, abordamos o teste de módulo de três pontos de vista:

1. A maneira como os casos de teste são projetados.
2. A ordem em que os módulos devem ser testados e integrados.
3. Orientação sobre a realização dos testes.

Projeto de Caso de Teste

Você precisa de **dois tipos de informações ao projetar casos de teste** para um teste de módulo: **uma especificação para o módulo e o código-fonte do módulo.** **A especificação normalmente define os parâmetros de entrada e saída do módulo e sua função.**

O teste de módulo é amplamente orientado para a caixa branca. Uma razão é que, à medida que você testa entidades maiores, como programas inteiros (que será o caso de processos de teste subsequentes), o teste de caixa branca se torna menos viável. Uma segunda razão é que os processos de teste subsequentes são orientados para encontrar diferentes tipos de erros (por exemplo, erros não necessariamente associados à lógica do programa, como o programa não atender aos requisitos de seus usuários). Portanto, o procedimento de projeto de caso de teste para um teste de módulo é o seguinte:

Analise a lógica do módulo usando um ou mais métodos de caixa branca e, em seguida, complemente esses casos de teste aplicando métodos de caixa preta à especificação do módulo.

Os métodos de projeto de casos de teste que usaremos foram definidos no Capítulo 4; nós irá ilustrar seu uso em um módulo de teste aqui através de um exemplo.

Suponha que desejamos testar um módulo chamado BÔNUS e sua função é adicionar \$ 2.000 ao salário de todos os funcionários do departamento ou departamentos com a maior receita de vendas. No entanto, se o salário atual de um funcionário qualificado for de US\$ 150.000 ou mais, ou se o funcionário for um gerente, o salário deverá ser aumentado em apenas US\$ 1.000.

As entradas do módulo são mostradas nas tabelas da Figura 5.1. Se o módulo executar sua função corretamente, ele retornará um código de erro 0. Se o funcionário ou a tabela de departamentos não contiverem entradas, ele retornará um código de erro 1. Se não encontrar nenhum funcionário em um departamento elegível, retornará um erro código de 2.

O código-fonte do módulo é mostrado na Figura 5.2. Os parâmetros de entrada ESIZE e DSIZE contêm o número de entradas nas tabelas de funcionários e departamentos. Observe que, embora o módulo seja escrito em PL/1, a discussão a seguir é amplamente independente da linguagem; as técnicas são aplicáveis a programas codificados em outras linguagens. Além disso, como a lógica PL/1 no módulo é bastante simples, praticamente qualquer leitor, mesmo aqueles não familiarizados com PL/1, deve ser capaz de entendê-la.

Tabela de funcionários

Tabela de departamentos

FIGURA 5.1 Tabelas de Entrada para o Módulo BÔNUS.

BÔNUS: PROCEDIMENTO (EMPTAB,DEPTTAB,ESIZE,DSIZE,ERRCODE);

DECLARAR 1 EMPTAB (*),

2 NOME CARACTERÍSTICA(6),

2 CARACTERÍSTICAS DE CÓDIGO(1),

2 DEP CHAR(3),

2 DECIMAL FIXO DE SALÁRIO(7,2);

DECLARAR 1 DEPTTAB (*),

2 DEP CHAR(3).

2 VENDAS DECIMAL FIXO(8,2);

DECLARE (ESIZE,DSIZE) BINÁRIO FIXO;

DECLARE ERRCODE DECIMAL FIXO(1);

```
DECLARE MAXSALES FIXED DECIMAL(8,2) INIT(0); /*MAX. VENDAS EM DEPTTAB*/ DECLARAR  
(I,J,K) BINÁRIO FIXO; /*CONTADORES*/ DECLARE ENCONTRARDEBITO FUNCIONAR SELEÇÃO VAL  
SINC DECIMAL FIXO(7,2) INIT(200,00); /*INCREMENTO PADRÃO*/ DECLARE LINC FIXED  
DECIMAL(7,2) INIT(100,00); /*INCREMENTO INFERIOR*/ DECLARAR O SALÁRIO FIXO  
DECIMAL(7,2) INIT(15000,00); /*LIMITE DE SALÁRIO*/
```

```
DECLARE MGR CHAR(1) INIT('M');
```

(contínuo)

FIGURA 5.2 Módulo BÔNUS.

88 A Arte do Teste de Software

```

1 ERRCODE=0;
2 SE(TAMANHO<=0)|(TAMANHO<=0)
3     ENTÃO CÓDIGO DE ERRO=1;                                /*EMPTAB OU DEPTTAB ESTÃO VAZIOS*/
4     MAIS FAÇA;
5         FAÇO I = 1 PARA DIMENSIONAR;                      /* ENCONTRAR MAXSALES E MAXDEPTS*/
6             IF(VENDAS(I)>=MAXVENDAS) THEN MAXVENDAS=VENDAS(I);
7             FIM;
8             DO J = 1 PARA DIMENSIONAR;
9                 IF(VENDAS(J)=MAXVENDAS)                         /*DEPARTAMENTO ELEGÍVEL*/
10                ENTÃO FAÇA;
11                ENCONTRADO='0'B;
12                FAÇA K = 1 PARA TAMANHO;
13                IF(EMPTAB.DEPT(K)=DEPTTAB.DEPT(J))
14                ENTÃO FAÇA;
15                ENCONTRADO='1'B;
16                SE(SALÁRIO(K)>=SALÁRIO)|CÓDIGO(K)=MGR)
17                ENTÃO SALÁRIO(K)=SALÁRIO(K)+LINC;
18                OUTRO SALÁRIO(K)=SALÁRIO(K)+SINC;
19                FIM;
20            FIM;
21            SE(-ENCONTRADO) ENTÃO CÓDIGO DE ERRO=2;
22        FIM;
23    FIM;
24 FIM;
25 FIM;

```

FIGURA 5.2 (continuação)

Barra lateral 5.1: Plano de fundo PL/1

Leitores novos no desenvolvimento de software podem não estar familiarizados com PL/1 e pensar que é uma linguagem "morta". É verdade, provavelmente há muito pouco novo desenvolvimento usando PL/1, mas manutenção de sistemas existentes continua, e as construções PL/1 ainda são uma boa maneira de aprender sobre os procedimentos de programação.

PL/1, que significa Programming Language One, foi desenvolvido na década de 1960 pela IBM para fornecer um ambiente de desenvolvimento semelhante ao inglês para suas máquinas de classe mainframe, começando com o IBM System/360. Nessa época da história do computador, muitos programadores estavam migrando para linguagens especializadas como COBOL, projetada para desenvolvimento de aplicativos de negócios, e Fortran, projetada para aplicações científicas. (Consulte a Barra Lateral 3.1 no Capítulo 3 para obter um pouco de conhecimento sobre esses idiomas.)

Um dos principais objetivos dos projetistas de PL/1 era uma linguagem de desenvolvimento que pudesse competir com sucesso com COBOL e Fortran, proporcionando um ambiente de desenvolvimento que fosse mais fácil de aprender com uma linguagem mais natural. Todos os objetivos iniciais do PL/1 provavelmente nunca foram alcançados, mas esses primeiros designers obviamente fizeram sua lição de casa, porque o PL/1 foi refinado e atualizado ao longo dos anos e ainda está em uso em alguns ambientes hoje.

Em meados da década de 1990, o PL/1 foi estendido a outras plataformas de computador, incluindo OS/2, Linux, UNIX e Windows. O novo suporte ao sistema operacional trouxe extensões de idioma para fornecer mais flexibilidade e funcionalidade.

Independentemente de qual técnica de cobertura lógica você usa, a primeira etapa é listar as decisões condicionais no programa. Os candidatos neste programa são todos os comandos IF e DO . Ao inspecionar o programa, podemos ver que todas as instruções DO são iterações simples, e cada limite de iteração será igual ou maior que o valor inicial (o que significa que cada corpo do loop sempre será executado pelo menos uma vez); e a única maneira de sair de cada loop é através da instrução DO . Assim, as instruções DO neste programa não precisam de atenção especial, pois qualquer caso de teste que faça com que uma instrução DO seja executada acabará fazendo com que ela ramifique em ambas as direções (ou seja, entrar no corpo do loop e pular o corpo do loop). Portanto, as afirmações que devem ser analisadas são:

- 2 SE (TAMANHO<10) j (TAMANHO<10)
- 6 SE (VENDAS(I)>1 VENDAS MÁXIMAS)
- 9 SE (VENDAS(J)>1 VENDAS MÁXIMAS)
- 13 SE (EMPTAB.DEPT(K)>1 DEPTTAB.DEPT(J))
- 16 SE (SALÁRIO(K)>1 SALÁRIO) j (CÓDIGO(K)>1MGR)
- 21 SE (-ENCONTRADO) ENTÃO CÓDIGO DE ERRO=2

TABELA 5.1 Situações Correspondentes aos Resultados da Decisão

Decisão Verdadeiro Resultado	Resultado Falso
2 ESIZE ou DSIZE 0	ESIZE e DSIZE>0
6 Sempre ocorrerá pelo menos uma vez. Encomende o DEPTTAB para que um	departamento com vendas mais baixas ocorra depois de um departamento com vendas mais altas.
9 Sempre ocorrerá pelo menos uma vez. Todos os departamentos não têm as	mesmas vendas.
13 Há um funcionário em um departamento elegível.	Há um funcionário que não está em um departamento elegível.
16 Um funcionário elegível é um gerente ou recebe LSALARY ou mais.	Um funcionário elegível não é gerente e ganha menos de SALÁRIO.
21 Todos os departamentos elegíveis não contêm funcionários.	Um departamento elegível contém pelo menos um funcionário.

Dado o pequeno número de decisões, provavelmente deveríamos optar pela cobertura multicondicional, mas examinaremos todos os critérios de cobertura lógica (exceto cobertura de instrução, que sempre é muito limitada para ser útil) para ver seus efeitos.

Para satisfazer o critério de cobertura de decisão, precisamos de casos de teste suficientes para invocar ambos os resultados de cada uma das seis decisões. As situações de entrada necessárias para invocar todos os resultados de decisão estão listadas na Tabela 5.1. Desde dois de os resultados sempre ocorrerão, existem 10 situações que precisam ser forçadas por casos de teste. Observe que para construir a Tabela 5.1, as circunstâncias do resultado da decisão tiveram que ser rastreadas através da lógica do programa para determinar as circunstâncias de entrada correspondentes apropriadas. Por exemplo, a decisão 16 não é invocada por nenhum funcionário que cumpra as condições; o funcionário deve estar em um departamento elegível.

As 10 situações de interesse na Tabela 5.1 podem ser invocadas pelos dois casos de teste mostrados na Figura 5.3. Observe que cada caso de teste inclui uma definição da saída esperada, de acordo com os princípios discutidos no Capítulo 2.

Embora esses dois casos de teste atendam ao critério de cobertura de decisão, deve ser óbvio que pode haver muitos tipos de erros no módulo que não são detectados por esses dois casos de teste. Por exemplo, os casos de teste não exploram as circunstâncias em que o código de erro é 0, um funcionário é um gerente ou a tabela de departamentos está vazia (DSIZE<%0).

Caso de teste	Entrada	Saída esperada																														
1	TAMANHO = 0 Todas as outras entradas são irrelevantes	CÓDIGO DE ERRO = 1 ESIZE, DSIZE, EMPTAB e DEPTTAB são inalterados																														
2	ESIZE = DSIZE = 3 EMPATAB <table border="1"> <tr><td>JONES</td><td>E</td><td>D42</td><td>21.000,00</td></tr> <tr><td>SMITH</td><td>E</td><td>D32</td><td>14.000,00</td></tr> <tr><td>LORIN</td><td>E</td><td>D42</td><td>10.000,00</td></tr> </table> DEPTTAB <table border="1"> <tr><td>D42</td><td>10.000,00</td></tr> <tr><td>D32</td><td>8.000,00</td></tr> <tr><td>D95</td><td>10.000,00</td></tr> </table>	JONES	E	D42	21.000,00	SMITH	E	D32	14.000,00	LORIN	E	D42	10.000,00	D42	10.000,00	D32	8.000,00	D95	10.000,00	CÓDIGO DE ERRO = 2 ESIZE, DSIZE e DEPTTAB permanecem inalterados EMPATAB <table border="1"> <tr><td>JONES</td><td>E</td><td>D42</td><td>21.100,00</td></tr> <tr><td>SMITH</td><td>E</td><td>D32</td><td>14.000,00</td></tr> <tr><td>LORIN</td><td>E</td><td>D42</td><td>10.200,00</td></tr> </table>	JONES	E	D42	21.100,00	SMITH	E	D32	14.000,00	LORIN	E	D42	10.200,00
JONES	E	D42	21.000,00																													
SMITH	E	D32	14.000,00																													
LORIN	E	D42	10.000,00																													
D42	10.000,00																															
D32	8.000,00																															
D95	10.000,00																															
JONES	E	D42	21.100,00																													
SMITH	E	D32	14.000,00																													
LORIN	E	D42	10.200,00																													

FIGURA 5.3 Casos de Teste para Satisfazer o Critério de Cobertura de Decisão.

Um teste mais satisfatório pode ser obtido usando o critério de cobertura de condição. Aqui precisamos de casos de teste suficientes para invocar ambos os resultados de cada condição nas decisões. As condições e situações de entrada necessárias para invocar todos os resultados estão listadas na Tabela 5.2. Como dois dos resultados sempre ocorrerão, existem 14 situações que devem ser forçadas pelos casos de teste. Novamente, essas situações podem ser invocadas por apenas dois casos de teste, conforme mostrado na Figura 5.4.

Os casos de teste na Figura 5.4 foram projetados para ilustrar um problema. Como eles invocam todos os resultados da Tabela 5.2, eles satisfazem o critério de cobertura de condição, mas provavelmente são um conjunto de casos de teste mais pobre do que os da Figura 5.3 em termos de satisfação do critério de cobertura de decisão.

A razão é que eles não executam todas as instruções. Por exemplo, a instrução 18 nunca é executada. Além disso, eles não realizam muito mais do que os casos de teste na Figura 5.3. Eles não causam a situação de saída ERRORCODE%0. Se a instrução 2 tiver configurado erroneamente ESIZE%0 e DSIZE%0, esse erro não será detectado. É claro que um conjunto alternativo de casos de teste pode resolver esses problemas, mas o fato é que os dois casos de teste na Figura 5.4 satisfazem o critério de cobertura de condição.

Usar o critério de cobertura de decisão/condição eliminaria a principal fraqueza nos casos de teste na Figura 5.4. Aqui, forneceríamos casos de teste suficientes para que todos os resultados de todas as condições e decisões fossem invocados pelo menos uma vez. Tornar Jones um gerente e Lorin um não gerente poderia conseguir isso. Isso teria o resultado de gerar ambos os resultados da decisão 16, fazendo com que executemos a instrução 18.

92 A Arte do Teste de Software

TABELA 5.2 Situações Correspondentes aos Resultados da Condição

Condição de Decisão		Resultado real	Resultado Falso
2	TAMANHO 0	TAMANHO 0	TAMANHO>0
2	TAMANHO 0	TAMANHO 0	DSIZE>0
6	VENDAS(I) VENDAS MÁXIMAS	Sempre ocorrerá pelo menos uma vez.	Encomende o DEPTTAB para que um departamento com vendas mais baixas ocorra depois de um departamento com vendas mais altas.
9	VENDAS(J)% VENDAS MÁXIMAS	Sempre ocorrerá pelo menos uma vez.	Todos os departamentos não têm as mesmas vendas.
13	EMPATAB.DEPT (K)% DEPTTAB. DEPT(J)	Há um funcionário	Há um funcionário que em um departamento elegível não está em um departamento elegível. departamento.
16	SALÁRIO(K) SALÁRIO	Um funcionário elegível ganha LSALÁRIO ou mais.	Um funcionário elegível ganha menos que LSALARY.
16	CÓDIGO(K)%MGR	Um funcionário elegível	Um funcionário elegível não é um gerente. é um gerente.
21	-ENCONTRADO	Um departamento elegível	contém pelo menos um funcionário.

Caso de teste	Entrada	Saída esperada																														
1	ESIZE = DSIZE = 0 Todas as outras entradas são irrelevantes	CÓDIGO DE ERRO = 1 ESIZE, DSIZE, EMPATAB e DEPTTAB permanecem inalterados																														
2	ESIZE = DSIZE = 3 EMPATAB <table border="1"> <tr><td>JONES</td><td>E</td><td>D42</td><td>21.000,00</td></tr> <tr><td>SMITH</td><td>E</td><td>D32</td><td>14.000,00</td></tr> <tr><td>LORIN</td><td>M</td><td>D42</td><td>10.000,00</td></tr> </table> DEPTTAB <table border="1"> <tr><td>D42</td><td>10.000,00</td></tr> <tr><td>D32</td><td>8.000,00</td></tr> <tr><td>D95</td><td>10.000,00</td></tr> </table>	JONES	E	D42	21.000,00	SMITH	E	D32	14.000,00	LORIN	M	D42	10.000,00	D42	10.000,00	D32	8.000,00	D95	10.000,00	CÓDIGO DE ERRO = 2 ESIZE, DSIZE e DEPTTAB permanecem inalterados EMPATAB <table border="1"> <tr><td>JONES</td><td>E</td><td>D42</td><td>21.000,00</td></tr> <tr><td>SMITH</td><td>E</td><td>D32</td><td>14.000,00</td></tr> <tr><td>LORIN</td><td>M</td><td>D42</td><td>10.100,00</td></tr> </table>	JONES	E	D42	21.000,00	SMITH	E	D32	14.000,00	LORIN	M	D42	10.100,00
JONES	E	D42	21.000,00																													
SMITH	E	D32	14.000,00																													
LORIN	M	D42	10.000,00																													
D42	10.000,00																															
D32	8.000,00																															
D95	10.000,00																															
JONES	E	D42	21.000,00																													
SMITH	E	D32	14.000,00																													
LORIN	M	D42	10.100,00																													

FIGURA 5.4 Casos de Teste para Satisfazer o Critério de Cobertura da Condição.

Um problema com isso, no entanto, é que essencialmente não é melhor do que os casos de teste na Figura 5.3. Se o compilador que está sendo usado parar de avaliar uma expressão ou assim que determinar que um operando é verdadeiro, essa modificação resultaria na expressão CODE(K)%MGR na instrução 16 nunca tendo um resultado verdadeiro. Portanto, se essa expressão fosse codificada incorretamente, os casos de teste não detectariam o erro.

O último critério a ser explorado é a cobertura multicondicional. Este critério requer casos de teste suficientes para que todas as combinações possíveis de condições em cada decisão sejam invocadas pelo menos uma vez. Isso pode ser feito trabalhando a partir da Tabela 5.2. As decisões 6, 9, 13 e 21 têm duas combinações cada uma; as decisões 2 e 16 têm quatro combinações cada. A metodologia para projetar os casos de teste é selecionar um que cubra o maior número possível de combinações, selecionar outro que cubra o maior número possível de combinações restantes e assim por diante. Um conjunto de casos de teste que satisfazem o critério de cobertura multicondicional é mostrado na Figura 5.5. O conjun-

Teste caso	Entrada	Saída esperada																																																
1	TAMANHO = 0 TAMANHO = 0 Todas as outras entradas são irrelevantes	CÓDIGO DE ERRO = 1 ESIZE, DSIZE, EMPTAB e DEPTTAB permanecem inalterados																																																
2	TAMANHO = 0 TAMANHO > 0 Todas as outras entradas são irrelevantes	O mesmo que acima																																																
3	TAMANHO > 0 TAMANHO = 0 Todas as outras entradas são irrelevantes	O mesmo que acima																																																
4	TAMANHO = 5 TAMANHO = 4 EMPATAB <table border="1"> <tr><td>JONES</td><td>M</td><td>D42</td><td>21.000,00</td></tr> <tr><td>AVISO</td><td>M</td><td>D95</td><td>12.000,00</td></tr> <tr><td>LORIN</td><td>E</td><td>D42</td><td>10.000,00</td></tr> <tr><td>TOY D95E</td><td>16.000,00</td><td></td><td></td></tr> <tr><td>SMITH D32E</td><td></td><td>14.000,00</td><td></td></tr> </table> DEPTTAB <table border="1"> <tr><td>D42</td><td>10.000,00</td></tr> <tr><td>D32</td><td>8.000,00</td></tr> <tr><td>D95</td><td>10.000,00</td></tr> <tr><td>D44</td><td>10.000,00</td></tr> </table>	JONES	M	D42	21.000,00	AVISO	M	D95	12.000,00	LORIN	E	D42	10.000,00	TOY D95E	16.000,00			SMITH D32E		14.000,00		D42	10.000,00	D32	8.000,00	D95	10.000,00	D44	10.000,00	CÓDIGO DE ERRO = 2 ESIZE, DSIZE e DEPTTAB permanecem inalterados EMPATAB <table border="1"> <tr><td>JONES</td><td>M</td><td>D42</td><td>21.100,00</td></tr> <tr><td>AVISO</td><td>M</td><td>D95</td><td>12.100,00</td></tr> <tr><td>LORIN</td><td>E</td><td>D42</td><td>10.200,00</td></tr> <tr><td>BRINQUEDO</td><td>E</td><td>D95</td><td>16.100,00</td></tr> <tr><td>SMITH</td><td>E</td><td>D32</td><td>14.000,00</td></tr> </table>	JONES	M	D42	21.100,00	AVISO	M	D95	12.100,00	LORIN	E	D42	10.200,00	BRINQUEDO	E	D95	16.100,00	SMITH	E	D32	14.000,00
JONES	M	D42	21.000,00																																															
AVISO	M	D95	12.000,00																																															
LORIN	E	D42	10.000,00																																															
TOY D95E	16.000,00																																																	
SMITH D32E		14.000,00																																																
D42	10.000,00																																																	
D32	8.000,00																																																	
D95	10.000,00																																																	
D44	10.000,00																																																	
JONES	M	D42	21.100,00																																															
AVISO	M	D95	12.100,00																																															
LORIN	E	D42	10.200,00																																															
BRINQUEDO	E	D95	16.100,00																																															
SMITH	E	D32	14.000,00																																															

FIGURA 5.5 Casos de Teste para Atender o Critério de Cobertura Multicondicional.

94 A Arte do Teste de Software

abrangente do que os conjuntos anteriores de casos de teste, o que implica que deveríamos ter selecionado esse critério no início.

É importante perceber que o módulo BONUS pode ter um número tão grande de erros que mesmo os testes que satisfaçam o critério de cobertura multicondicional não detectariam todos eles. Por exemplo, nenhum caso de teste gera a situação em que ERRORCODE é retornado com o valor 0; assim, se a instrução 1 estivesse faltando, o erro não seria detectado. Se LSALARY fosse inicializado erroneamente em \$ 150.000,01, o erro passaria despercebido. Se a declaração 16 indicasse SALÁRIO(K)>SALÁRIO em vez de SALÁRIO(K) >½SALÁRIO, este erro não seria encontrado. Além disso, se uma variedade de erros off-by-one (como não manipular a última entrada em DEPTTAB ou EMPTAB corretamente) seria detectada dependeria em grande parte do acaso.

Dois pontos devem ficar aparentes agora: primeiro, o critério multicondicional é superior aos outros critérios e, segundo, qualquer critério de cobertura lógica não é bom o suficiente para servir como o único meio de derivar testes de módulo. Portanto, o próximo passo é complementar os testes da Figura 5.5 com um conjunto de testes de caixa preta. Para isso, as especificações da interface do BONUS são mostradas a seguir:

BONUS, um módulo PL/1, recebe cinco parâmetros, simbolicamente referidos aqui como EMPTAB, DEPTTAB, ESIZE, DSIZE e ERRORCODE. Os atributos desses parâmetros são:

```

DECLARE 1 EMPTAB(*), /*ENTRADA E SAÍDA*/
        2 NOME DE PERSONAGEM(6),
        2 CARACTERES DE CÓDIGO(1),
        2 PERSONAGEM DE DEPARTAMENTO(3),
        2 DECIMAL FIXO DE SALÁRIO(7,2);
DECLARE 1 DEPTTAB(*), /*INPUT*/
        2 PERSONAGEM DE DEPARTAMENTO(3),
        2 VENDAS DECIMAL FIXO(8,2);
DECLARE (ESIZE, DSIZE) BINÁRIO FIXO; /*ENTRADA*/
DECLARE ERRCODE DECIMAL FIXO(1); /*RESULTADO*/

```

O módulo assume que os argumentos transmitidos possuem esses atributos. ESIZE e DSIZE indicam o número de entradas em EMPTAB e DEPTTAB, respectivamente. Nenhuma suposição deve ser feita sobre a ordem das entradas em EMPTAB e DEPTTAB. A função do módulo é incrementar o salário (EMPTAB.SALARY) dos funcionários do departamento ou departamentos com maior valor de vendas (DEPTTAB.SALES). Se um elegível

o salário atual do funcionário é de \$ 150.000 ou mais, ou se o empregado for um gerente (EMPTAB.CODE \neq 'M'), o incremento é de \$1.000; caso contrário, o incremento para o funcionário elegível é de \$ 2.000. O módulo assume que o salário incrementado caberá no campo EMPTAB.SALARY. Se ESIZE e DSIZE não forem maiores que 0, ERRCODE será definido como 1 e nenhuma ação adicional será tomada. Em todos os outros casos, a função é completamente executada. No entanto, se for constatado que um departamento de vendas máximas não tem funcionário, o processamento continua, mas ERRCODE terá o valor 2; caso contrário, é definido como 0.

Esta especificação não é adequada para gráficos de causa e efeito (não há um conjunto discernível de condições de entrada cujas combinações devam ser exploradas); assim, será utilizada a análise de valor de contorno. Os limites de entrada identificados são os seguintes:

1. EMPTAB tem 1 entrada.
2. EMPTAB tem o número máximo de entradas (65.535).
3. EMPTAB tem 0 entradas.
4. DEPTTAB tem 1 entrada.
5. O DEPTTAB possui 65.535 entradas.
6. DEPTTAB tem 0 entradas.
7. Um departamento de vendas máximas tem 1 funcionário.
8. Um departamento de vendas máximas tem 65.535 funcionários.
9. Um departamento de vendas máximas não tem funcionários.
10. Todos os departamentos do DEPTTAB têm as mesmas vendas.
11. O departamento de vendas máximas é a primeira entrada no DEPTTAB.
12. O departamento de vendas máximas é a última entrada no DEPTTAB.
13. Um funcionário elegível é a primeira entrada no EMPTAB.
14. Um funcionário elegível é a última entrada no EMPTAB.
15. Um funcionário elegível é um gerente.
16. Um funcionário elegível não é um gerente.
17. Um funcionário elegível que não seja gerente tem um salário de \$ 149.999,99.
18. Um funcionário elegível que não seja gerente tem um salário de \$ 150.000.
19. Um funcionário elegível que não seja gerente tem um salário de \$ 150.000,01.

Os limites de saída são os seguintes:

20. ERRCODE \neq 0
21. ERRCODE \neq 1
22. ERRCODE \neq 2
23. O aumento salarial de um funcionário elegível é de \$ 299.999,99.

Uma outra condição de teste baseada na técnica de adivinhação de erros é a seguinte:

24. Um departamento de vendas máximas sem funcionários é seguido no DEPTTAB com outro departamento de vendas máximo com funcionários.

Isso é usado para determinar se o módulo termina erroneamente processamento da entrada quando encontra uma situação ERRCODE%2 .

Revendo essas 24 condições, os números 2, 5 e 8 parecem casos de teste impraticáveis. Como eles também representam condições que nunca ocorrerão (geralmente uma suposição perigosa a ser feita ao testar, mas aparentemente segura aqui), nós os excluímos. O próximo passo é comparar as 21 condições restantes com o conjunto atual de casos de teste (Figura 5.5) para determinar quais condições de contorno ainda não foram cobertas. Fazendo isso, vemos que as condições 1, 4, 7, 10, 14, 17, 18, 19, 20, 23 e 24 requerem casos de teste além daqueles da Figura 5.5.

O próximo passo é projetar casos de teste adicionais para cobrir as 11 condições de contorno. Uma abordagem é mesclar essas condições nos casos de teste existentes (ou seja, modificando o caso de teste 4 na Figura 5.5), mas isso não é recomendado porque isso pode inadvertidamente prejudicar a cobertura completa de multicondições dos casos de teste existentes. Portanto, a abordagem mais segura é adicionar casos de teste aos da Figura 5.5. Ao fazer isso, o objetivo é projetar o menor número de casos de teste necessário para cobrir as condições de contorno. Os três casos de teste na Figura 5.6 realizam isso. O caso de teste 5 cobre as condições 7, 10, 14, 17, 18, 19 e 20; o caso de teste 6 cobre as condições 1, 4 e 23; e o caso de teste 7 cobre a condição 24.

A premissa aqui é que a cobertura lógica, ou caixa branca, casos de teste em A Figura 5.6 forma um teste de módulo razoável para o procedimento BONUS.

Testes Incrementais

Ao realizar o processo de teste de módulo, há duas considerações principais: o projeto de um conjunto eficaz de casos de teste, que foi discutido na seção anterior, e a maneira pela qual os módulos são combinados para formar um programa de trabalho. A segunda consideração é importante porque tem as seguintes implicações:

A forma na qual os casos de teste do módulo são escritos

Os tipos de ferramentas de teste que podem ser usadas

Caso de teste	Entrada	Saída esperada																												
5	TAMANHO = 3 TAMANHO = 2 EMPATAB <table border="1"> <tr><td>ALIADO</td><td>E</td><td>D36</td><td>14.999,99</td></tr> <tr><td>MELHOR</td><td>E</td><td>D33</td><td>15.000,00</td></tr> <tr><td>CELTO</td><td>E</td><td>D33</td><td>15.000,01</td></tr> </table> DEPTTAB <table border="1"> <tr><td>D33</td><td>55.400,01</td></tr> <tr><td>D36</td><td>55.400,01</td></tr> </table>	ALIADO	E	D36	14.999,99	MELHOR	E	D33	15.000,00	CELTO	E	D33	15.000,01	D33	55.400,01	D36	55.400,01	CÓDIGO DE ERRO = 0 ESIZE, DSIZE e DEPTTAB são inalterado EMPATAB <table border="1"> <tr><td>ALIADO</td><td>E</td><td>D36</td><td>15.199,99</td></tr> <tr><td>MELHOR</td><td>E</td><td>D33</td><td>15.100,00</td></tr> <tr><td>CELTO</td><td>E</td><td>D33</td><td>15.100,01</td></tr> </table>	ALIADO	E	D36	15.199,99	MELHOR	E	D33	15.100,00	CELTO	E	D33	15.100,01
ALIADO	E	D36	14.999,99																											
MELHOR	E	D33	15.000,00																											
CELTO	E	D33	15.000,01																											
D33	55.400,01																													
D36	55.400,01																													
ALIADO	E	D36	15.199,99																											
MELHOR	E	D33	15.100,00																											
CELTO	E	D33	15.100,01																											
6	TAMANHO = 1 TAMANHO = 1 EMPATAB <table border="1"> <tr><td>CHEFE</td><td>M</td><td>D99 99.899,99</td><td></td></tr> </table> DEPTTAB <table border="1"> <tr><td>D99 99.000,00</td></tr> </table>	CHEFE	M	D99 99.899,99		D99 99.000,00	CÓDIGO DE ERRO = 0 ESIZE, DSIZE e DEPTTAB são inalterado EMPATAB <table border="1"> <tr><td>CHEFE</td><td>M</td><td>D99 99.999,99</td><td></td></tr> </table>	CHEFE	M	D99 99.999,99																				
CHEFE	M	D99 99.899,99																												
D99 99.000,00																														
CHEFE	M	D99 99.999,99																												
7	TAMANHO = 2 TAMANHO = 2 EMPATAB <table border="1"> <tr><td>DOLE</td><td>E</td><td>D67</td><td>10.000,00</td></tr> <tr><td>FORD</td><td>E</td><td>D22</td><td>33.333,33</td></tr> </table> DEPTTAB <table border="1"> <tr><td>D66</td><td>20.000,00</td></tr> <tr><td>D67</td><td>20.000,00</td></tr> </table>	DOLE	E	D67	10.000,00	FORD	E	D22	33.333,33	D66	20.000,00	D67	20.000,00	CÓDIGO DE ERRO = 2 ESIZE, DSIZE e DEPTTAB são inalterado EMPATAB <table border="1"> <tr><td>DOLE</td><td>E</td><td>D67</td><td>10.000,00</td></tr> <tr><td>FORD</td><td>E</td><td>D22</td><td>33.333,33</td></tr> </table>	DOLE	E	D67	10.000,00	FORD	E	D22	33.333,33								
DOLE	E	D67	10.000,00																											
FORD	E	D22	33.333,33																											
D66	20.000,00																													
D67	20.000,00																													
DOLE	E	D67	10.000,00																											
FORD	E	D22	33.333,33																											

FIGURA 5.6 Casos de Teste de Análise de Valor de Limite Complementar para BÔNUS.

A ordem em que os módulos são codificados e testados

O custo de gerar casos de teste

O custo de depuração (localização e reparação de erros detectados)

Em suma, então, é uma consideração de importância substancial. Nessa seção, discutimos duas abordagens, testes incrementais e não incrementais; a seguir, exploramos duas abordagens incrementais, de cima para baixo e desenvolvimento ou teste de baixo para cima.

A questão aqui ponderada é a seguinte: Você deve testar um programa testando cada módulo independentemente e, em seguida, combinando os módulos para

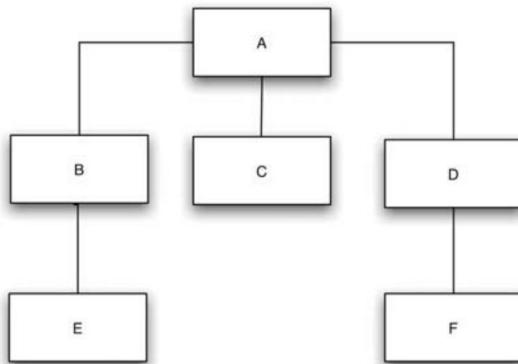


FIGURA 5.7 Exemplo de Programa de Seis Módulos.

forma o programa, ou você deve combinar o próximo módulo a ser testado com o conjunto de módulos testados anteriormente antes de ser testado? A primeira abordagem é chamada não-incremental, ou "big-bang", teste ou integração; a segunda abordagem é conhecida como teste incremental ou integração.

O programa da Figura 5.7 é usado como exemplo. Os retângulos representam os seis módulos (sub-rotinas ou procedimentos) do programa. As linhas conectando os módulos representam a hierarquia de controle do programa; ou seja, o módulo A chama os módulos B, C e D; o módulo B chama o módulo E; e assim por diante. O teste não incremental, a abordagem tradicional, é realizado da seguinte maneira. Primeiro, um teste de módulo é realizado em cada um dos seis módulos, testando cada módulo como uma entidade independente. Os módulos podem ser testados ao mesmo tempo ou em sucessão, dependendo do ambiente (por exemplo, instalações de computação interativa versus processamento em lote) e do número de pessoas envolvidas. Finalmente, os módulos são combinados ou integrados (por exemplo, "link editado") para formar o programa.

O teste de cada módulo requer um módulo de driver especial e um ou mais módulos stub. Por exemplo, para testar o módulo B, os casos de teste são primeiro projetados e então alimentados ao módulo B passando-lhe argumentos de entrada de um módulo driver, um pequeno módulo que deve ser codificado para "conduzir" ou transmitir casos de teste através do módulo em teste. (Alternativamente, uma ferramenta de teste pode ser usada.) O módulo driver também deve exibir, para o testador, os resultados produzidos por B. Além disso, como o módulo B chama o módulo E, algo deve estar presente para receber o controle quando B chamar E. Um módulo stub, um módulo especial com o nome "E" que deve ser codificado para simular a função do módulo E, faz isso.

Quando o teste do módulo de todos os seis módulos estiver concluído, os módulos são combinados para formar o programa.

A abordagem alternativa é o teste incremental. Em vez de testar cada módulo isoladamente, o próximo módulo a ser testado é combinado primeiro com o conjunto de módulos que já foram testados.

É prematuro fornecer um procedimento para testar incrementalmente o programa na Figura 5.7, porque há um grande número de abordagens incrementais possíveis. Uma questão chave é se devemos começar no topo ou no fundo do programa. No entanto, já que discutiremos essa questão na próxima seção, vamos supor, por enquanto, que estamos começando de baixo.

O primeiro passo é testar os módulos E, C e F, em paralelo (por três pessoas) ou em série. Observe que devemos preparar um driver para cada módulo, mas não um stub. O próximo passo é testar B e D; mas em vez de testá-los isoladamente, eles são combinados com os módulos E e F, respectivamente. Em outras palavras, para testar o módulo B, um driver é escrito, incorporando os casos de teste, e o par BE é testado. O processo incremental, adicionando o próximo módulo ao conjunto ou subconjunto de módulos testados anteriormente, continua até que o último módulo (módulo A neste caso) seja testado. Observe que esse procedimento poderia ter progredido alternativamente de cima para baixo.

Várias observações devem ser evidentes neste momento:

1. Testes não incrementais requerem mais trabalho. Para o programa da Figura 5.7, cinco drivers e cinco stubs devem ser preparados (assumindo que não precisamos de um módulo de driver para o módulo superior). O teste incremental de baixo para cima exigiria cinco pilotos, mas sem canhotos. Um teste incremental de cima para baixo exigiria cinco canhotos, mas nenhum motorista. Menos trabalho é necessário porque os módulos testados anteriormente são usados em vez dos módulos de driver (se você começar de cima) ou módulos stub (se você começar de baixo) necessários na abordagem não incremental.
2. Erros de programação relacionados a interfaces incompatíveis ou suposições incorretas entre os módulos serão detectados mais cedo quando o teste incremental for usado. A razão é que as combinações de módulos são testadas juntas em um momento inicial. No entanto, quando testes não incrementais são usados, os módulos não “vêem uns aos outros” até o final do processo.
3. Como resultado, a depuração deve ser mais fácil se o teste incremental for usado. Se assumirmos que existem erros relacionados a interfaces e suposições intermódulos (uma boa suposição, por experiência), então, se

teste não incremental foi usado, os erros não aparecerão até que todo o programa tenha sido combinado. Neste momento, podemos ter dificuldade em identificar o erro, pois ele pode estar em qualquer lugar dentro do programa. Por outro lado, se o teste incremental for usado, um erro desse tipo deve ser mais fácil de identificar, porque é provável que o erro esteja associado ao módulo adicionado mais recentemente.

4. Testes incrementais podem resultar em testes mais completos. Se você estiver testando o módulo B, o módulo E ou A (dependendo se você começou de baixo ou de cima) é executado como resultado. Embora E ou A devesse ter sido exaustivamente testado anteriormente, talvez executá-lo como resultado do teste do módulo de B invoque uma nova condição, talvez uma que represente uma deficiência no teste original de E ou A.

Por outro lado, se o teste não incremental for usado, o teste de B afetará apenas o módulo B. Em outras palavras, o teste incremental substitui os módulos testados anteriormente pelos stubs ou drivers necessários no teste não incremental. Como resultado, os módulos reais recebem mais exposição pela conclusão do último teste do módulo.

5. A abordagem não incremental parece usar menos tempo de máquina. Se o módulo A da Figura 5.7 estiver sendo testado usando a abordagem de baixo para cima, os módulos B, C, D, E e F provavelmente serão executados durante a execução de A. Em um teste não incremental de A, apenas stubs para B, C e E são executados. O mesmo vale para um teste incremental de cima para baixo. Se o módulo F está sendo testado, os módulos A, B, C, D e E podem ser executados durante o teste de F ; no teste não incremental de F, apenas o driver para F, mais o próprio F , execute. Assim, o número de instruções de máquina executadas durante uma execução de teste usando a abordagem incremental é aparentemente maior do que para a abordagem não incremental. Contrabalançando isso está o fato de que o teste não incremental requer mais drivers e stubs do que o teste incremental; tempo de máquina é necessário para desenvolver os drivers e stubs.

6. No início da fase de teste do módulo, há mais oportunidades para atividades paralelas quando o teste não incremental é usado (ou seja, todos os módulos podem ser testados simultaneamente). Isso pode ser significativo em um projeto grande (muitos módulos e pessoas), já que o número de funcionários de um projeto geralmente atinge seu pico no início da fase de teste do módulo.

Em resumo, as observações 1 a 4 são vantagens do teste incremental, enquanto as observações 5 e 6 são desvantagens. Dadas as tendências atuais

na indústria de computação (os custos de hardware têm diminuído e parecem destinados a continuar a fazê-lo, enquanto a capacidade de hardware aumenta, e os custos de mão de obra e as consequências de erros de software estão aumentando), e dado o fato de que quanto mais cedo um erro for encontrado, menor será o custo de repará-lo, você pode ver que as observações de 1 a 4 estão crescendo em importância, enquanto a observação 5 está se tornando menos importante. Observação 6 parece ser uma desvantagem fraca, se houver. Isso leva à conclusão que o teste incremental é superior.

Teste de cima para baixo versus teste de baixo para cima

Dada a conclusão da seção anterior - que o teste incremental é superior aos testes não incrementais - a seguir exploramos dois testes incrementais estratégias: testes de cima para baixo e de baixo para cima. Antes de entrar neles, no entanto, devemos esclarecer vários equívocos. Primeiro, os termos de cima para baixo testes, desenvolvimento de cima para baixo e design de cima para baixo geralmente são usados como sinônimos. Teste de cima para baixo e desenvolvimento de cima para baixo são sinônimos (eles representam uma estratégia de ordenar a codificação e teste de módulos), mas design de cima para baixo é algo bem diferente e independente. Um programa que foi projetado de cima para baixo pode ser testado incrementalmente em uma moda de cima para baixo ou de baixo para cima.

Em segundo lugar, os testes de baixo para cima (ou desenvolvimento de baixo para cima) são muitas vezes equivocadamente equiparados a testes não incrementais. A razão é que de baixo para cima o teste começa de maneira idêntica a um teste não incremental (ou seja, quando os módulos inferiores, ou terminais, são testados), mas como vimos na seção anterior, o teste de baixo para cima é uma estratégia incremental. Finalmente, como ambas as estratégias são incrementais, não repetiremos aqui as vantagens de testes incrementais; discutiremos apenas as diferenças entre os testes de cima para baixo e de baixo para cima.

Teste de cima para baixo

A estratégia de cima para baixo começa com o módulo superior, ou inicial, no programa. Depois disso, não há um único procedimento "certo" para selecionar o próximo módulo a ser testado incrementalmente; a única regra é que para ser elegível para ser o próximo módulo, pelo menos um dos módulos subordinados (chamando) do módulo deve ter sido testado anteriormente.

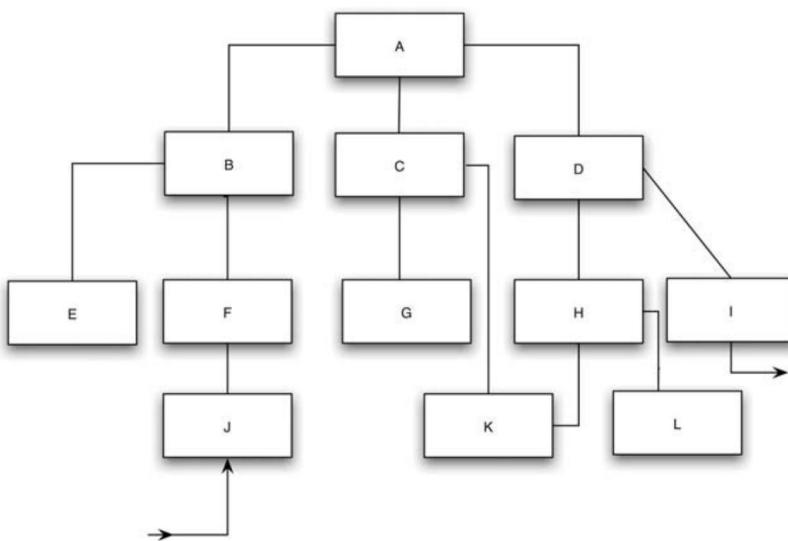


FIGURA 5.8 Exemplo de Programa de 12 Módulos.

A Figura 5.8 é usada para ilustrar essa estratégia. A a L são os 12 módulos do programa. Suponha que o módulo J contenha a leitura de E/S do programa operações e o módulo I contém as operações de gravação.

O primeiro passo é testar o módulo A. Para isso, os módulos stub que representam B, C e D devem ser escritos. Infelizmente, a produção de stub módulos é muitas vezes incompreendido; como evidência, muitas vezes você pode ver declarações como "um módulo stub precisa apenas escrever uma mensagem dizendo 'chegamos até aqui'"; e, "em muitos casos, o módulo fictício (stub) simplesmente sai - sem fazer qualquer trabalho." Na maioria das situações, essas declarações são falsas. Desde o módulo A chama o módulo B, A espera que B realize algum trabalho; este trabalho mais provavelmente algum resultado (argumentos de saída) foi retornado para A. Se o stub simplesmente retorna o controle ou escreve uma mensagem de erro sem retornar um resultado, o módulo A falhará, não por causa de um erro em A, mas por causa de uma falha do stub para simular o módulo correspondente. Além disso, devolver um A saída "com fio" de um módulo stub geralmente é insuficiente. Por exemplo, considere a tarefa de escrever um stub representando uma rotina de raiz quadrada, um banco de dados rotina de pesquisa de tabela, uma rotina "obter registro de arquivo mestre correspondente" ou semelhante. Se o stub retornar uma saída fixa com fio, mas não tiver o valor específico esperado pelo módulo chamador durante essa chamada, o módulo chamador poderá falhar ou produzir um resultado confuso. Assim, a produção de stubs não é uma tarefa trivial.

Outra consideração é a forma como os casos de teste são apresentados ao programa, uma consideração importante que nem é mencionada na maioria das discussões sobre testes de cima para baixo. Em nosso exemplo, a pergunta é: Como você alimenta os casos de teste para o módulo A? O módulo superior em programas típicos não recebe argumentos de entrada nem executa operações de entrada/saída, portanto a resposta não é imediatamente óbvia. A resposta é que os dados de teste são alimentados ao módulo (módulo A nesta situação) de um ou mais de seus stubs.

Para ilustrar, suponha que as funções de B, C e D sejam as seguintes:

- B — Obter resumo do arquivo da transação.
- C — Determina se o status semanal atende à cota.
- D — Produzir relatório resumido semanal.

Um caso de teste para A, então, é um resumo de transação retornado do stub B. O Stub D pode conter instruções para gravar seus dados de entrada em uma impressora, permitindo que os resultados de cada teste sejam examinados.

Neste programa, existe outro problema. Presumivelmente, o módulo A chama o módulo B apenas uma vez; portanto, o problema é como alimentar mais de um caso de teste para A. Uma solução é desenvolver várias versões do stub B, cada uma com um conjunto diferente de dados de teste conectados a serem retornados a A. Para executar os casos de teste, o programa é executado várias vezes, cada vez com uma versão diferente do stub B. Outra alternativa é colocar os dados de teste em arquivos externos e fazer com que o stub B leia os dados de teste e os retorne para A. Em ambos os casos, tendo em mente o anterior discussão, você deve ver que o desenvolvimento de módulos stub é mais difícil do que muitas vezes parece ser. Além disso, muitas vezes é necessário, devido às características do programa, representar um caso de teste em vários stubs abaixo do módulo em teste (ou seja, onde o módulo recebe dados para serem executados chamando vários módulos).

Depois que A foi testado, um módulo real substitui um dos stubs e os stubs exigidos por esse módulo são adicionados. Por exemplo, a Figura 5.9 pode representar a próxima versão do programa.

Depois de testar o módulo superior, várias sequências são possíveis. Por exemplo, se estivermos realizando todas as sequências de teste, quatro exemplos das muitas sequências possíveis de módulos são:

1. ABCDEFGHIJKL
2. ABEFJCGKDHLI
3. ADHIKLCGBFJE
4. ABFJDIECGKHL

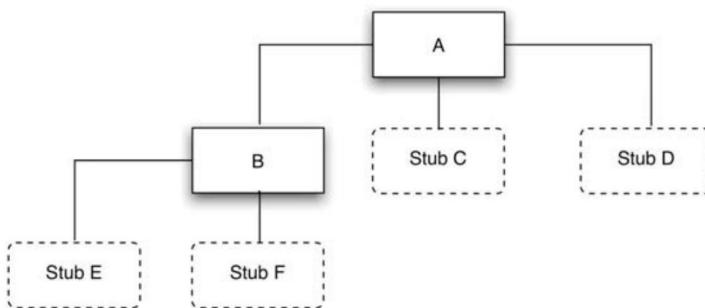


FIGURA 5.9 Segundo Passo no Teste Top-Down.

Se ocorrerem testes paralelos, outras alternativas são possíveis. Por exemplo, após o módulo A ter sido testado, um programador pode pegar o módulo A e teste a combinação AB; outro programador poderia testar AC; e um terceiro poderia testar AD. Em geral, não há melhor sequência, mas aqui estão duas orientações a considerar:

1. Se houver seções críticas do programa (talvez módulo G), projete a sequência de forma que essas seções sejam adicionadas o mais cedo possível. Uma "seção crítica" pode ser um módulo complexo, um módulo com um novo algoritmo, ou um módulo suspeito de ser propenso a erros.
2. Projete a sequência de forma que os módulos de E/S sejam adicionados o quanto antes possível.

A motivação para o primeiro deve ser óbvia, mas a motivação para a segunda merece uma discussão mais aprofundada. Lembre-se de que um problema com stubs é que alguns deles devem conter os casos de teste, e outros devem escrever seus entrada para uma impressora ou monitor. No entanto, assim que o módulo aceitar o entrada do programa é adicionada, a representação dos casos de teste é consideravelmente simplificado; sua forma é idêntica à entrada aceita pelo programa final (por exemplo, de um arquivo de transação ou de um terminal). Da mesma forma, uma vez adicionado o módulo que executa a função de saída do programa, a colocação de código em módulos stub para escrever resultados de casos de teste pode não ser mais necessário. Assim, se os módulos J e I são os módulos de E/S, e se o módulo G desempenha alguma função crítica, a sequência incremental pode ser

ABFJDICGEKHL

e a forma do programa após o sexto incremento seria a mostrada na Figura 5.10.

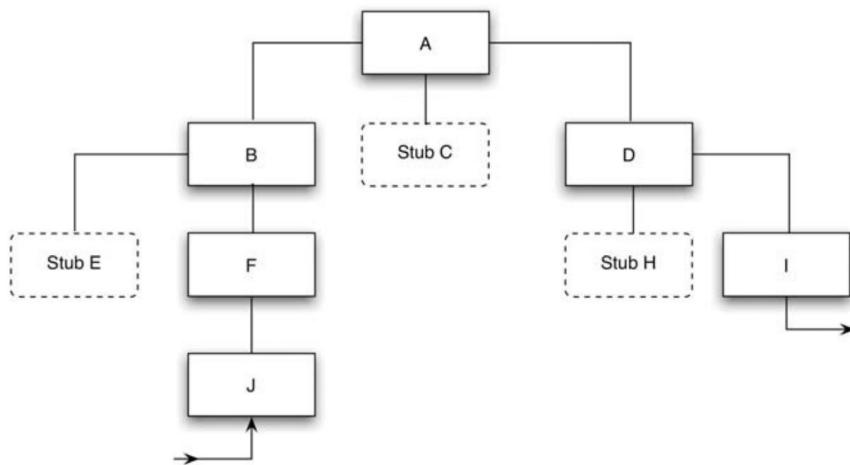


FIGURA 5.10 Estado Intermediário no Teste Top-Down.

Uma vez atingido o estado intermediário da Figura 5.10, a representação dos casos de teste e a inspeção dos resultados são simplificadas. Ele tem uma outra vantagem, pois você tem uma versão funcional do programa, ou seja, uma versão que executa operações reais de entrada e saída. No entanto, stubs ainda estão simulando alguns dos "interiores". Esta versão esquelética inicial:

- Permite encontrar erros e problemas de fator humano.
- Possibilita a demonstração do programa ao eventual usuário.
- Serve como evidência de que o design geral do programa é sólido.
- Serve como um impulsionador de moral.

Esses pontos representam a grande vantagem da estratégia top-down.

Por outro lado, a abordagem de cima para baixo tem algumas deficiências sérias. Suponha que nosso estado atual de teste seja o da Figura 5.10 e que nosso próximo passo seja substituir o stub H pelo módulo H. O que devemos fazer neste ponto (ou antes) é usar os métodos descritos anteriormente neste capítulo para projetar um conjunto de casos de teste para H. Observe, no entanto, que os casos de teste estão na forma de entradas de programa reais para o módulo J. Isso apresenta vários problemas. Primeiro, por causa dos módulos intermediários entre J e H (F, B, A e D), podemos achar impossível representar certos casos de teste para o módulo J que testa todas as situações predefinidas em H. Por exemplo, se H é o BONUS da Figura 5.2, pode ser impossível, devido à natureza do módulo D interveniente, criar alguns dos sete casos de teste das Figuras 5.5 e 5.6.

106 A Arte do Teste de Software

Em segundo lugar, devido à "distância" entre H e o ponto em que os dados de teste entram no programa, mesmo que fosse possível testar todas as situações, determinar quais dados alimentar J para testar essas situações em H é muitas vezes uma tarefa mental difícil.

Terceiro, como a saída exibida de um teste pode vir de um módulo que está a uma grande distância do módulo que está sendo testado, correlacionar a saída exibida com o que aconteceu no módulo pode ser difícil ou impossível. Considere adicionar o módulo E à Figura 5.10. Os resultados de cada caso de teste são determinados examinando a saída escrita pelo módulo I, mas por causa dos módulos intermediários, pode ser difícil deduzir a saída real de E (ou seja, os dados retornados para B).

A estratégia de cima para baixo, dependendo de como é abordada, pode ter mais dois problemas. As pessoas ocasionalmente sentem que a estratégia pode ser sobreposta à fase de projeto do programa. Por exemplo, se você estiver projetando o programa da Figura 5.8, você pode acreditar que, depois que os dois primeiros níveis forem projetados, os módulos A a D podem ser codificados e testados enquanto o projeto dos níveis inferiores progride. Como enfatizamos em outro lugar, essa geralmente é uma decisão imprudente. O design do programa é um processo iterativo, o que significa que, quando estamos projetando os níveis inferiores da estrutura de um programa, podemos descobrir mudanças ou melhorias desejáveis nos níveis superiores. Se os níveis superiores já foram codificados e testados, as melhorias desejáveis provavelmente serão descartadas, uma decisão imprudente a longo prazo.

Um problema final que geralmente surge na prática é não testar completamente um módulo antes de prosseguir para outro módulo. Isso ocorre por dois motivos: pela dificuldade de embutir dados de teste em módulos stub, e porque os níveis superiores de um programa geralmente fornecem recursos para os níveis inferiores. Na Figura 5.8, vimos que testar o módulo A pode exigir várias versões do stub para o módulo B. Na prática, há uma tendência de dizer: "Como isso representa muito trabalho, não executarei todos os casos de teste de A agora. Vou esperar até colocar o módulo J no programa, quando a representação dos casos de teste será mais fácil, e lembrar neste ponto de terminar o teste do módulo A." Claro, o problema aqui é que podemos esquecer de teste o restante do módulo A neste momento posterior. Além disso, como os níveis superiores geralmente fornecem recursos para uso dos níveis inferiores (por exemplo, abertura de arquivos), às vezes é difícil determinar se os recursos foram fornecidos corretamente (por exemplo, se um arquivo foi aberto com os atributos apropriados) até que o módulos inferiores que os utilizam são testados.

Teste de baixo para cima

O próximo passo é examinar a estratégia de teste incremental de baixo para cima. Na maioria das vezes, o teste de baixo para cima é o oposto do teste de cima para baixo; assim, as vantagens do teste de cima para baixo tornam-se as desvantagens do teste de baixo para cima, e as desvantagens do teste de cima para baixo tornam-se as vantagens do teste de baixo para cima. Por causa disso, a discussão sobre testes de baixo para cima é mais curta.

A estratégia bottom-up começa com os módulos terminais no programa (os módulos que não chamam outros módulos). Após esses módulos terem sido testados, novamente não há melhor procedimento para selecionar o próximo módulo a ser testado incrementalmente; a única regra é que para ser elegível para o próximo módulo, todos os módulos subordinados do módulo (os módulos que ele chama) devem ter sido testados previamente.

Voltando à Figura 5.8, o primeiro passo é testar alguns ou todos os módulos E, J, G, K, L e I, em série ou em paralelo. Para fazer isso, cada módulo precisa de um módulo de driver especial: um módulo que contém entradas de teste conectadas, chama o módulo que está sendo testado e exibe as saídas (ou compara as saídas reais com as saídas esperadas). Ao contrário da situação com stubs, várias versões de um driver não são necessárias, pois o módulo do driver pode chamar iterativamente o módulo que está sendo testado. Na maioria dos casos, os módulos de driver são mais fáceis de produzir do que os módulos stub.

Como foi o caso anterior, um fator que influencia a sequência de testes é a natureza crítica dos módulos. Se decidirmos que os módulos D e F são os mais crítico, um estado intermediário do teste incremental de baixo para cima pode ser o da Figura 5.11. Os próximos passos podem ser testar E e depois testar B, combinando B com os módulos E, F e J testados anteriormente.

Uma desvantagem da estratégia de baixo para cima é que não existe o conceito de um programa esquelético inicial. De fato, o programa de trabalho não existe até que o último módulo (módulo A) seja adicionado, e este programa de trabalho é o programa completo. Embora as funções de E/S possam ser testadas antes que todo o programa seja integrado (os módulos de E/S estão sendo usados na Figura 5.11), as vantagens do programa esquelético inicial não estão presentes.

Os problemas associados à impossibilidade, ou dificuldade, de criar todas as situações de teste na abordagem top-down não existem aqui. Se você pensar em um módulo de driver como uma sonda de teste, a sonda está sendo colocada diretamente no módulo que está sendo testado; não há módulos intermediários para se preocupar. Examinando outros problemas associados à abordagem de cima para baixo, você

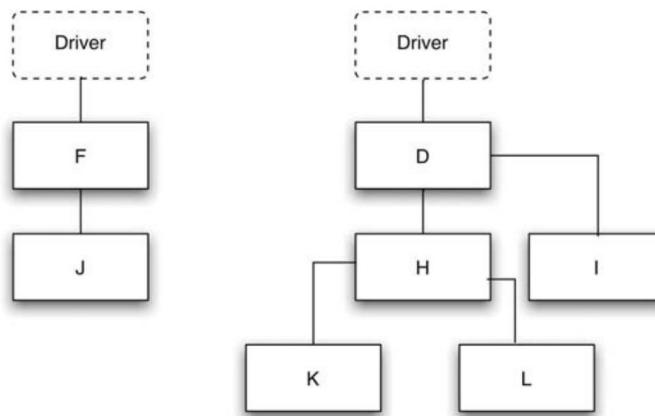


FIGURA 5.11 Estado Intermediário no Teste Bottom-Up.

não pode tomar a decisão imprudente de sobrepor projeto e teste, já que o teste de baixo para cima não pode começar até que a parte inferior do programa tenha sido projetada. Além disso, o problema de não completar o teste de um módulo antes iniciar outro, devido à dificuldade de codificação de dados de teste em versões de um stub, não existe ao usar o teste de baixo para cima.

Uma comparação

Seria conveniente se a questão de cima para baixo versus de baixo para cima fosse tão clara como a questão incremental versus não incremental, mas infelizmente não é. A Tabela 5.3 resume as vantagens e desvantagens relativas das duas abordagens (excluindo o discutido anteriormente

vantagens compartilhadas por ambos - as do teste incremental). A primeira vantagem de cada abordagem pode parecer o fator decisivo, mas há nenhuma evidência mostrando que grandes falhas ocorrem com mais frequência nos níveis superior ou inferior do programa típico. A maneira mais segura de tomar uma decisão é pesar os fatores na Tabela 5.3 em relação ao programa específico que está sendo testado. Na falta de tal programa aqui, as graves consequências da quarta desvantagem - de teste de cima para baixo e a disponibilidade de ferramentas de teste que eliminam a necessidade de drivers, mas não de canhotos - parece dar a estratégia de baixo para cima a borda.

Além disso, pode ser evidente que os testes de cima para baixo e de baixo para cima não são as únicas estratégias incrementais possíveis.

TABELA 5.3 Comparação de testes de cima para baixo e de baixo para cima

Teste de cima para baixo	
Vantagens	Desvantagens
1. Vantajosa quando grandes falhas ocorrem na parte superior do programa.	1. Os módulos stub devem ser produzidos. 2. Os módulos de stub geralmente são mais complicados do que parecem à primeira vista.
2. Uma vez que as funções de E/S são adicionadas, a representação de casos é mais fácil.	3. Antes que as funções de E/S sejam adicionadas, a representação de casos de teste em stubs pode ser difícil.
3. O programa esquelético precoce permite manifestações e eleva o moral.	4. As condições de teste podem ser impossíveis ou muito difíceis de criar. 5. A observação do resultado do teste é mais difícil.
	6. Leva à conclusão de que projeto e teste podem ser sobrepostos. 7. Adia a conclusão dos testes determinados módulos.
Desvantagens do Teste	
Vantagens 1.	Bottom-Up 1. Os
Vantajoso quando grandes falhas ocorrem na parte inferior do programa.	módulos de driver devem ser produzidos. 2. O programa como uma entidade não existe até que o último módulo seja adicionado.
2. As condições de teste são mais fáceis de criar.	
3. A observação dos resultados dos testes é mais fácil.	

Executando o Teste

A parte restante do teste do módulo é o ato de realmente realizar o teste. Um conjunto de dicas e diretrizes para fazer isso está incluído aqui.

Quando um caso de teste produz uma situação em que os resultados reais do módulo não correspondem aos resultados esperados, há duas explicações possíveis: o módulo contém um erro ou os resultados esperados estão incorretos (o caso de teste está incorreto). Para minimizar essa confusão, o conjunto de casos de teste

devem ser revisados ou inspecionados antes que o teste seja executado (ou seja, os casos de teste devem ser testados).

O uso de ferramentas de teste automatizadas pode minimizar parte do trabalho penoso do processo de teste. Por exemplo, existem ferramentas de teste que eliminam a necessidade de módulos de driver. As ferramentas de análise de fluxo enumeram os caminhos através de um programa, encontram instruções que nunca podem ser executadas (código "inalcançável") e identificam instâncias em que uma variável é usada antes de receber um valor.

Como foi a prática anterior neste capítulo, lembre-se de que uma definição do resultado esperado é uma parte necessária de um caso de teste. Ao executar um teste, lembre-se de procurar por efeitos colaterais (instâncias em que um módulo faz algo que não deveria fazer). Em geral, essas situações são difíceis de detectar, mas algumas delas podem ser encontradas verificando, após a execução do caso de teste, as entradas do módulo que não devem ser alteradas. Por exemplo, o caso de teste 7 na Figura 5.6 afirma que, como parte do resultado esperado, ESIZE, DSIZE e DEPTTAB devem permanecer inalterados. Ao executar este caso de teste, não apenas a saída deve ser examinada para obter o resultado correto, mas ESIZE, DSIZE e DEPTTAB devem ser examinados para determinar se foram alterados erroneamente.

Os problemas psicológicos associados a uma pessoa que tenta testar seus próprios programas também se aplicam aos testes de módulo. Em vez de testar seus próprios módulos, os programadores podem trocá-los; mais especificamente, o programador do módulo chamador é sempre um bom candidato para testar o módulo chamado. Observe que isso se aplica apenas a testes; a depuração de um módulo sempre deve ser realizada pelo programador original.

Evite casos de teste descartáveis; representá-los de tal forma que possam ser reutilizados no futuro. Lembre-se do fenômeno contra-intuitivo na Figura 2.2. Se um número anormalmente alto de erros for encontrado em um subconjunto dos módulos, é provável que esses módulos contenham ainda mais erros, ainda não detectados. Esses módulos devem ser submetidos a mais testes de módulo e, possivelmente, a um passo a passo ou inspeção de código adicional. Por fim, lembre-se de que o objetivo de um teste de módulo não é demonstrar que o módulo funciona corretamente, mas demonstrar a presença de erros no módulo.

Resumo

Neste capítulo, apresentamos alguns dos mecanismos de teste, especialmente no que se refere a programas grandes. Este é um processo de teste de componentes individuais do programa – sub-rotinas, subprogramas, classes e

procedimentos. No teste de módulo, você compara a funcionalidade do software com a especificação que define sua função pretendida. O teste de módulo ou unidade pode ser uma parte importante da caixa de ferramentas de um desenvolvedor para ajudar a obter um aplicativo confiável, especialmente com linguagens orientadas a objetos, como Java e C#. O objetivo no teste de módulo é o mesmo de qualquer outro tipo de teste de software: tentar mostrar como o programa contradiz a especificação. Além da especificação do software, você precisará do código-fonte de cada módulo para efetuar um teste de módulo.

O teste de módulo é basicamente um teste de caixa branca. (Consulte o Capítulo 4 para obter mais informações sobre procedimentos de caixa branca e projeto de casos de teste para teste.) Um projeto de teste de módulo completo incluirá estratégias incrementais, como técnicas de cima para baixo e de baixo para cima.

É útil, ao se preparar para um teste de módulo, revisar os princípios psicológicos e econômicos apresentados no Capítulo 2.

Mais um ponto: o software de teste de módulo é apenas o começo de um procedimento de teste exaustivo. Você precisará passar para testes de ordem superior, que abordamos no Capítulo 6, e testes de usuários, abordados no Capítulo 7.

6

Testes de ordem superior

Quando você termina de testar o módulo de um programa, você tem apenas iniciado o processo de teste. Isso é especialmente verdadeiro para programas grandes ou complexos. Considere este importante conceito:

Um erro de software ocorre quando o programa não faz o que o usuário final espera razoavelmente que ele faça.

Aplicando esta definição, mesmo se você pudesse realizar um teste de módulo absolutamente perfeito, você ainda não poderia garantir que encontrou todos os erros de software. Para completar o teste, então, alguma forma de teste adicional é necessária.

Chamamos essa nova forma de teste de ordem superior.

O desenvolvimento de software é, em grande parte, um processo de comunicação de informações sobre o programa eventual e de tradução dessas informações de uma forma para outra. Em essência, está se movendo do conceitual para o concreto. Por esse motivo, a grande maioria dos erros de software pode ser atribuída a falhas, erros e “ruídos” durante a comunicação e tradução de informações.

Essa visão do desenvolvimento de software é ilustrada na Figura 6.1, um modelo do ciclo de desenvolvimento de um produto de software. O fluxo do processo pode ser resumido em sete etapas:

1. Traduzir as necessidades do usuário do programa em um conjunto de requisitos escritos.

Esses são os objetivos do produto.

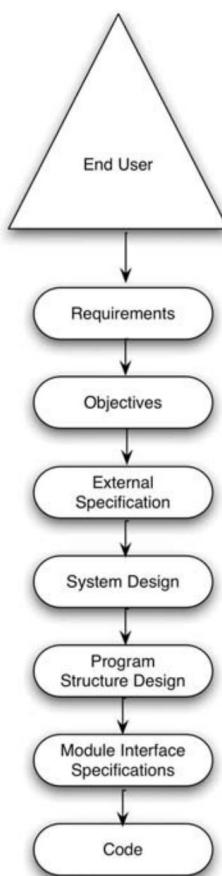


FIGURA 6.1 O Processo de Desenvolvimento de Software.

2. Traduzir os requisitos em objetivos específicos avaliando viabilidade, tempo e custo, resolvendo requisitos conflitantes e estabelecendo prioridades e compensações.
3. Traduzir os objetivos em uma especificação precisa do produto, visualizando o produto como uma caixa preta e considerando apenas suas interfaces e interações com o usuário final. Essa descrição é chamada de externa especificação.
4. Se o produto for um sistema como um sistema operacional, controle de voo sistema, sistema de banco de dados ou sistema de gerenciamento de pessoal de funcionários, em vez de um aplicativo (por exemplo, compilador, programa de folha de pagamento, processador de texto), o próximo processo é o projeto do sistema. Este passo

partitiona o sistema em programas, componentes ou subsistemas individuais e define suas interfaces.

5. Projete a estrutura do programa ou programas especificando a função de cada módulo, a estrutura hierárquica dos módulos e as interfaces entre os módulos.
6. Desenvolva uma especificação precisa que defina a interface e a função de cada módulo.
7. Traduzir, por meio de uma ou mais subetapas, a especificação da interface do módulo para o algoritmo do código-fonte de cada módulo.

Aqui está outra maneira de ver essas formas de documentação:

Os requisitos especificam por que o programa é necessário.

Os objetivos especificam o que o programa deve fazer e quão bem o programa deve fazê-lo.

Especificações externas definem a representação exata do programa aos usuários.

A documentação associada aos processos subsequentes especifica, em níveis crescentes de detalhes, como o programa é construído.

Dada a premissa de que as sete etapas do ciclo de desenvolvimento envolvem comunicação, compreensão e tradução de informações, e a premissa de que a maioria dos erros de software decorre de falhas no tratamento da informação, existem três abordagens complementares para prevenir e/ou detectar esses erros.

Primeiro, podemos introduzir mais precisão no processo de desenvolvimento para evitar muitos dos erros. Em segundo lugar, podemos introduzir, no final de cada processo, uma etapa de verificação separada para localizar o maior número possível de erros antes de prosseguir para o próximo processo. Essa abordagem é ilustrada na Figura 6.2. Por exemplo, a especificação externa é verificada comparando-a com a saída do estágio anterior (a declaração de objetivos) e retornando quaisquer erros descobertos ao processo de especificação externa. (Use os métodos de inspeção de código e passo a passo discutidos no Capítulo 3 na etapa de verificação no final do sétimo processo.)

A terceira abordagem é orientar processos de teste distintos para processos de desenvolvimento distintos. Ou seja, concentre cada processo de teste em uma etapa de tradução específica - portanto, em uma classe específica de erros. Essa abordagem é ilustrada na Figura 6.3.

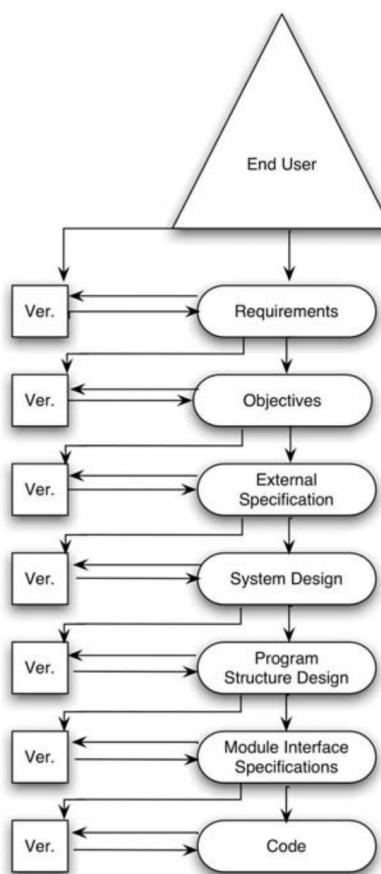


FIGURA 6.2 O Processo de Desenvolvimento com Verificação Intermediária Passos.

O ciclo de teste é estruturado para modelar o ciclo de desenvolvimento. Em outras palavras, você deve ser capaz de estabelecer uma correspondência direta entre os processos de desenvolvimento e teste. Por exemplo:

A finalidade de um teste de módulo é encontrar discrepâncias entre os módulos do programa e suas especificações de interface.

O propósito de um teste de função é mostrar que um programa não corresponder às suas especificações externas.

O objetivo de um teste de sistema é mostrar que o produto está de acordo com seus objetivos originais.

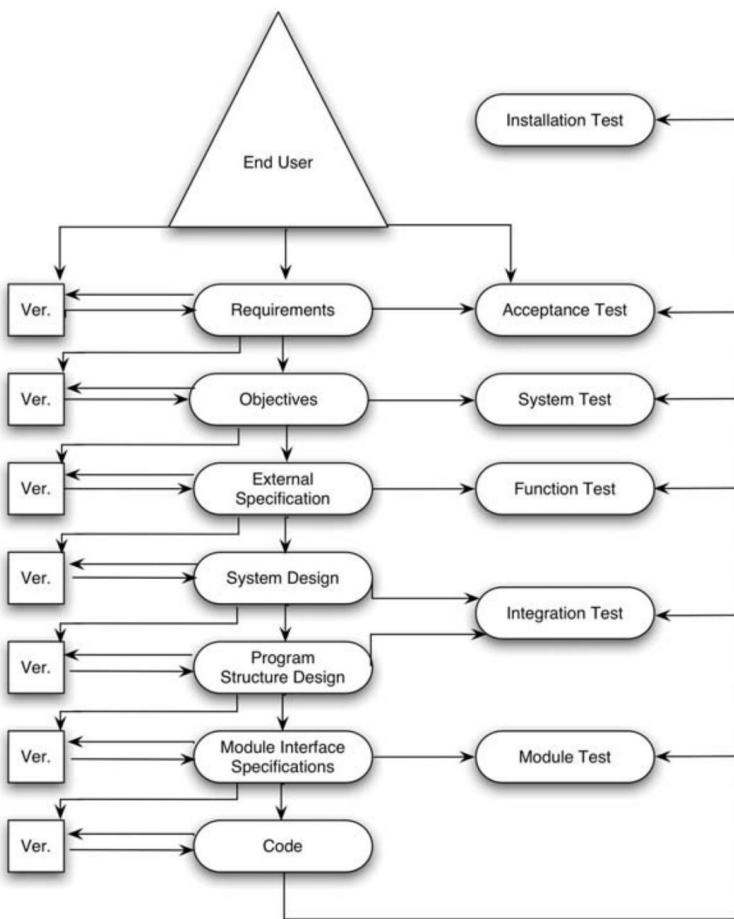


FIGURA 6.3 A correspondência entre desenvolvimento e teste Processos.

Observe como estruturamos estas declarações: "encontra discrepâncias", "não corresponde", "é inconsistente". Lembre-se de que o objetivo do teste de software é encontrar problemas (porque sabemos que haverá problemas!). Se você se propõe a provar que algum tipo de entrada funciona corretamente, ou assume que o programa é fiel às suas especificações e objetivos, seu teste ficará incompleto. Somente ao tentar provar que algum tipo de entrada funciona de forma inadequada, e assumir que o programa não é fiel às suas especificações e objetivos, seu teste estará completo. Este é um conceito importante que repetimos ao longo deste livro.

As vantagens dessa estrutura são que ela evita testes redundantes improdutivos e evita que você ignore turmas grandes de erros. Por exemplo, em vez de simplesmente rotular os testes do sistema como "o teste de todo o sistema" e possivelmente repetir testes anteriores, o teste do sistema é orientado para uma classe distinta de erros (aqueles cometidos durante a tradução dos objetivos para a especificação externa) e medido em relação a um tipo distinto de documentação no processo de desenvolvimento.

Os métodos de teste de ordem superior mostrados na Figura 6.3 são mais aplicáveis a produtos de software (programas escritos como resultado de um contrato ou destinados a uso amplo, em oposição a programas experimentais ou escrito para uso apenas pelo autor do programa). Programas não escritos como os produtos muitas vezes não têm requisitos e objetivos formais; por tal programas, o teste de função pode ser o único teste de ordem superior. Também a necessidade de testes de ordem superior aumenta junto com o tamanho do programa. A razão é que a proporção de erros de projeto (erros cometidos no processos de desenvolvimento) para erros de codificação é consideravelmente maior em grandes programas do que em pequenos programas.

Observe que a sequência de processos de teste na Figura 6.3 não implica necessariamente uma sequência de tempo. Por exemplo, como o teste do sistema não é definido como "o tipo de teste que você faz após o teste de função", mas sim como um tipo distinto de teste focado em uma classe distinta de erros, pode muito bem ser parcialmente sobreposto no tempo com outros processos de teste.

Neste capítulo, discutimos os processos de teste de função, sistema, aceitação e instalação. Omitimos o teste de integração porque muitas vezes ele não é considerado uma etapa de teste separada; e, quando o módulo incremental teste é usado, é uma parte implícita do teste do módulo.

Manteremos as discussões desses processos de teste breves, gerais, e, na maioria das vezes, sem exemplos porque técnicas específicas utilizadas nesses testes de ordem superior são altamente dependentes do programa específico sendo testado. Por exemplo, as características de um teste de sistema (os tipos de casos de teste, a maneira como os casos de teste são projetados, as ferramentas de teste usadas) para um sistema operacional diferirá consideravelmente de um teste de sistema de um compilador, um programa que controla um reator nuclear ou um programa de aplicativo de banco de dados.

Nas últimas seções deste capítulo, abordamos questões organizacionais e de planejamento, juntamente com a importante questão de determinar quando parar de testar.

Teste de função

Conforme indicado na Figura 6.3, o teste de função é um processo de tentativa de encontrar discrepâncias entre o programa e a especificação externa. Uma especificação externa é uma descrição precisa do comportamento do programa do ponto de vista do usuário final.

Exceto quando usado em programas pequenos, o teste de função normalmente é uma atividade de caixa preta. Ou seja, você confia no processo de teste de módulo anterior para atingir os critérios de cobertura lógica de caixa branca desejados.

Para executar um teste de função, você analisa a especificação para derivar um conjunto de casos de teste. Os métodos de particionamento de equivalência, análise de valor de contorno, representação gráfica de causa e efeito e métodos de adivinhação de erros descritos no Capítulo 4 são especialmente pertinentes ao teste de função. Na verdade, os exemplos do Capítulo 4 são exemplos de testes de função. As descrições da instrução Fortran DIMENSION , o programa de pontuação do exame e o comando DISPLAY são exemplos de especificações externas. Não são, contudo, exemplos completamente realistas; por exemplo, uma especificação externa real para o programa de pontuação incluiria uma descrição precisa do formato dos relatórios. (Nota: Como discutimos testes de função no Capítulo 4, não apresentamos exemplos de testes de função nesta seção.)

Muitas das diretrizes que fornecemos no Capítulo 2 também são particularmente pertinentes ao teste de função. Em particular, acompanhe quais funções exibiram o maior número de erros; essa informação é valiosa porque informa que essas funções provavelmente também contêm a preponderância de erros ainda não detectados. Além disso, lembre-se de focar uma quantidade suficiente de atenção em condições de entrada inválidas e inesperadas. (Lembre-se de que a definição do resultado esperado é uma parte vital de um caso de teste.)

Finalmente, como sempre, tenha em mente que o objetivo do teste de função é expor erros e discrepâncias com a especificação, não demonstrar que o programa corresponde à sua especificação externa.

Teste do sistema

O teste do sistema é o processo de teste mais incompreendido e mais difícil. O teste de sistema não é um processo de teste das funções do sistema ou programa completo, porque isso seria redundante com o processo de teste de função. Em vez disso, como mostrado na Figura 6.3, o teste do sistema tem um

propósito particular: comparar o sistema ou programa com seus objetivos originais. Dado este propósito, considere estas duas implicações:

1. O teste do sistema não se limita aos sistemas. Se o produto for um programa, teste de sistema é o processo de tentar demonstrar como o programa, como um todo, não atinge seus objetivos.
2. O teste do sistema, por definição, é impossível se não houver um conjunto de objetivos escritos e mensuráveis para o produto.

Ao procurar discrepâncias entre o programa e seus objetivos, focar nos erros de tradução cometidos durante o processo de concepção do especificação externa. Isso torna o teste do sistema um processo de teste vital, pois em termos do produto, do número de erros cometidos e da gravidade desses erros, esta etapa do ciclo de desenvolvimento geralmente é a mais propenso a erros.

Também implica que, ao contrário do teste de função, a especificação externa não pode ser usado como base para derivar os casos de teste do sistema, pois isso subverteria o propósito do teste do sistema. Por outro lado, o documento de objetivos não pode ser usado por si só para formular casos de teste, uma vez que não contém, por definição, descrições precisas das funções do programa interfaces externas. Resolvemos este dilema usando o user do programa documentação ou publicações - projete o teste do sistema analisando o Objetivos; formular casos de teste analisando a documentação do usuário. este tem o efeito colateral útil de comparar o programa com seus objetivos e a documentação do usuário, bem como comparar a documentação do usuário com os objetivos, conforme mostrado na Figura 6.4.

A Figura 6.4 ilustra por que o teste do sistema é o processo de teste mais difícil. A seta mais à esquerda na figura, comparando o programa com seus objetivos, é o objetivo central do teste do sistema, mas não há metodologias de projeto de casos de teste conhecidas. A razão para isso é que os objetivos declaram o que um programa deve fazer e quão bem o programa deve fazê-lo, mas eles fazem não informar a representação das funções do programa. Por exemplo, o objetivos para o comando DISPLAY especificados no Capítulo 4 podem ter leia a seguir:

Será fornecido um comando para visualizar, a partir de um terminal, o conteúdo de principais locais de armazenamento. Sua sintaxe deve ser consistente com a taxa de sintaxe de todos os outros comandos do sistema. O usuário deve ser capaz de especificar

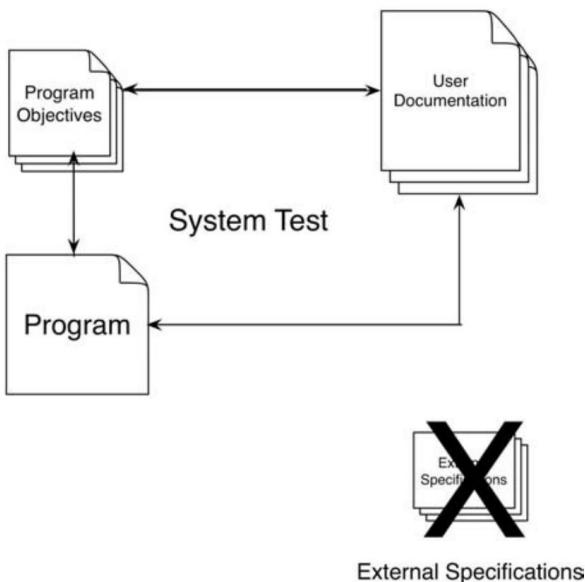


FIGURA 6.4 O Teste do Sistema.

um intervalo de locais, por meio de um intervalo de endereços ou um endereço e uma contagem.

Padrões sensatos devem ser fornecidos para operandos de comando.

A saída deve ser exibida como várias linhas de várias palavras (em hexadecimal), com espaçamento entre as palavras. Cada linha deve conter o endereço da primeira palavra dessa linha. O comando é um comando "trivial", o que significa que sob cargas razoáveis do sistema, ele deve começar a exibir a saída dentro de dois segundos, e deve haver não haja tempo de atraso observável entre as linhas de saída. Uma programação erro no processador de comando deve, na pior das hipóteses, causar a falha do comando; o sistema e a sessão do usuário não devem ser afetados.

O processador de comandos não deve ter mais de um erro detectado pelo usuário após o sistema ser colocado em produção.

Dada a declaração de objetivos, não há metodologia identificável que produziria um conjunto de casos de teste, além da linha de orientação vaga, mas útil, de escrever casos de teste para tentar mostrar que o programa é consistente com cada sentença da declaração de objetivos. Daí, um abordagem diferente para o projeto de casos de teste é tomada aqui: Em vez de descrever uma metodologia, são discutidas categorias distintas de casos de teste do sistema. Por causa da ausência de uma metodologia, o teste do sistema requer um

quantidade de criatividade; de fato, o projeto de bons casos de teste de sistema requer mais criatividade, inteligência e experiência do que o necessário para projetar o próprio sistema ou programa.

A Tabela 6.1 lista 15 categorias de casos de teste, juntamente com uma breve descrição. Discutimos as categorias aqui. Não afirmamos que todas as 15 categorias se aplicam a todos os programas, mas para evitar negligenciar algo, recomendamos que você explore todas elas ao projetar casos de teste.

TABELA 6.1 15 Categorias de Casos de Teste

Categoria	Descrição
Instalação	Garantir que a funcionalidade nos objetivos seja implementada.
Volume	Sujeite o programa a volumes anormalmente grandes de dados para processo.
Estresse	Sujeitar o programa a cargas anormalmente grandes, geralmente processamento simultâneo.
Usabilidade	Determine quão bem o usuário final pode interagir com o programa.
Segurança	Tente subverter as medidas de segurança do programa.
atuação	Determinar se o programa atende a resposta e requisitos de rendimento.
Armazenar	Certifique-se de que o programa gerencie corretamente suas necessidades de armazenamento, tanto sistema e físico.
Configuração	Verifique se o programa funciona adequadamente no configurações recomendadas.
Compatibilidade/ Conversão	Determinar se novas versões do programa são compatível com versões anteriores.
Instalação	Certifique-se de que os métodos de instalação funcionem em todos os plataformas.
Confiabilidade	Determinar se o programa atende às especificações de confiabilidade como tempo de atividade e MTBF.
Recuperação	Teste se as instalações de recuperação do sistema funcionam conforme projetado.
Facilidade de manutenção/ Manutenção	Determine se o aplicativo fornece corretamente mecanismos para gerar dados sobre eventos que requerem suporte técnico.
Documentação	Valide a precisão de toda a documentação do usuário.
Procedimento	Determinar a precisão dos procedimentos especiais necessários para usar ou manter o programa.

Teste de instalações

O tipo mais óbvio de teste de sistema é determinar se cada instalação (ou função; mas a palavra "função" não é usada aqui para evitar confundir isso com teste de função) mencionada nos objetivos foi realmente implementado. O procedimento é escanear os objetivos frase por frase, e quando uma frase específica um quê (por exemplo, "a sintaxe deve ser consistente..." "o usuário deve ser capaz de especificar um intervalo de localizações... ' '), determinar que o programa satisfaz o "o quê". Esse tipo de teste geralmente pode ser executado sem um computador; uma comparação mental dos objetivos com a documentação do usuário às vezes é suficiente. Mesmo assim, uma lista de verificação é útil para garantir que você verifique mentalmente os mesmos objetivos na próxima vez que você realizar o teste.

Teste de volume

Um segundo tipo de teste de sistema é submeter o programa a grandes volumes de dados. Por exemplo, um compilador pode receber um programa fonte absurdamente grande para compilar. Um editor de ligação pode ser alimentado com um programa contendo milhares de módulos. Um simulador de circuito eletrônico pode receber um circuito contendo milhões de componentes. A fila de trabalhos de um sistema operacional pode ser preenchido até a capacidade. Se um programa deve manipular arquivos que abrangem vários volumes, são criados dados suficientes para fazer com que o programa mude de um volume para outro. Em outras palavras, o objetivo do teste de volume é mostrar que o programa não pode lidar com o volume de dados especificado em seu Objetivos.

Obviamente, o teste de volume pode exigir recursos significativos, portanto, em termos de tempo de máquina e pessoas, você não deve exagerar. Ainda, cada programa deve ser exposto a pelo menos alguns testes de volume.

Teste de estresse

O teste de estresse submete o programa a cargas pesadas ou estresses. Isto deveria não confundir com teste de volume; um estresse pesado é um volume de pico de dados, ou atividade, encontrados em um curto espaço de tempo. Uma analogia seria avaliando um datilógrafo: Um teste de volume determinaria se o datilógrafo poderia lidar com um rascunho de um grande relatório; um teste de estresse determinaria se o datilógrafo poderia digitar a uma taxa de 50 palavras por minuto.

Como o teste de estresse envolve um elemento de tempo, não é aplicável a muitos programas - por exemplo, um compilador ou uma folha de pagamento de processamento em lote programa. É aplicável, no entanto, a programas que operam sob diferentes cargas ou programas interativos, em tempo real e de controle de processos. Se um sistema de controle de tráfego aéreo deve rastrear até 200 aviões em seu setor, você pode fazer um teste de estresse simulando a presença de 200 aviões. Desde não há nada que impeça fisicamente um avião 201 de entrar no setor, um teste de estresse adicional exploraria a reação do sistema a esse inesperado avião. Um teste de estresse adicional pode simular a entrada simultânea de um grande número de aviões no setor.

Se um sistema operacional deve suportar um máximo de 15 trabalhos simultâneos, o sistema pode ficar sobre carregado ao tentar executar 15 trabalhos simultaneamente. Você pode enfatizar um simulador de aeronave de treinamento de pilotos determinar a reação do sistema a um trainee que força o leme para a esquerda, puxa o acelerador, abaixa os flaps, levanta o nariz, abaixa o trem de pouso, acende as luzes de pouso e inclina para a esquerda, tudo ao mesmo tempo. (Tais casos de teste podem exigir um piloto de quatro mãos ou, realisticamente, dois especialistas no cockpit.) Você pode testar um sistema de controle de processo fazendo com que todos os processos monitorados gerem sinais simultaneamente, ou um sistema de comutação telefônica roteando para ele um grande número de telefonemas simultâneos.

Aplicativos baseados na Web são assuntos comuns de testes de estresse. Aqui, você deseja garantir que seu aplicativo e hardware possam lidar com um volume de destino de usuários simultâneos. Você pode argumentar que pode ter milhões de pessoas acessando o site ao mesmo tempo, mas isso não é realista. Você precisa definir seu público e, em seguida, criar um teste de estresse para representar o número máximo de usuários que você acha que usarão seu site. (O Capítulo 10 fornece mais informações sobre como testar aplicativos baseados na Web.)

Da mesma forma, você pode enfatizar um aplicativo de dispositivo móvel - um telefone celular sistema operacional, por exemplo - lançando vários aplicativos que correr e permanecer residente, fazendo ou recebendo um ou mais telefones chamadas. Você pode lançar um programa de navegação GPS, um aplicativo que usa recursos de CPU e radiofrequência (RF) quase continuamente, então tentar usar outros aplicativos ou fazer chamadas telefônicas. (Capítulo 11 discute o teste de aplicativos móveis com mais detalhes.)

Embora muitos testes de estresse representem condições que o programa provavelmente experimentará durante sua operação, outros podem representar verdadeiramente situações "nunca ocorrerão"; mas isso não significa que esses testes sejam

nao é útil. Se essas condições impossíveis detectarem erros, o teste é valioso porque é provável que os mesmos erros também possam ocorrer em situações realistas e menos estressantes.

Testando usabilidade

Outra área importante do caso de teste é a usabilidade, ou teste de usuário. Embora essa técnica de teste tenha quase 30 anos, ela se tornou mais importante com o advento de mais softwares baseados em GUI e a profunda penetração de hardware e software de computador em todos os aspectos de nossa sociedade. Ao incumbir o usuário final final de um aplicativo de testar o software em um ambiente do mundo real, problemas potenciais podem ser descobertos que até mesmo o roteamento de teste automatizado mais agressivo provavelmente não encontraria. Essa área de teste de software é tão importante que a abordaremos com mais detalhes no próximo capítulo.

Teste de segurança

Em resposta à crescente preocupação da sociedade com a privacidade, muitos programas agora têm objetivos específicos de segurança. O teste de segurança é o processo de tentar criar casos de teste que subvertam as verificações de segurança do programa.

Por exemplo, você pode tentar formular casos de teste que contornam o mecanismo de proteção de memória de um sistema operacional. Da mesma forma, você pode tentar subverter os mecanismos de segurança de dados de um sistema de banco de dados. Uma maneira de conceber tais casos de teste é estudar problemas de segurança conhecidos em sistemas semelhantes e gerar casos de teste que tentem demonstrar problemas comparáveis no sistema que você está testando. Por exemplo, fontes publicadas em revistas, salas de bate-papo ou grupos de notícias frequentemente cobrem bugs conhecidos em sistemas operacionais ou outros sistemas de software. Ao pesquisar falhas de segurança em programas existentes que fornecem serviços semelhantes ao que você está testando, você pode criar casos de teste para determinar se seu programa sofre do mesmo tipo de problema.

Os aplicativos baseados na Web geralmente precisam de um nível mais alto de teste de segurança do que a maioria dos aplicativos. Isso é especialmente verdadeiro para sites de comércio eletrônico. Embora exista tecnologia suficiente, ou seja, criptografia, para permitir que os clientes concluam transações com segurança pela Internet, você não deve confiar na mera aplicação de tecnologia para garantir a segurança. Além disso, você precisará convencer sua base de clientes de que sua aplicação é segura,

ou corre o risco de perder clientes. Novamente, o Capítulo 10 fornece mais informações sobre testes de segurança em aplicativos baseados na Internet.

Teste de performance

Muitos programas têm objetivos específicos de desempenho ou eficiência, declarando propriedades como tempos de resposta e taxas de rendimento sob determinadas condições de carga de trabalho e configuração. Novamente, como o objetivo de um teste de sistema é demonstrar que o programa não atende aos seus objetivos, os casos de teste devem ser projetados para mostrar que o programa não atende aos seus objetivos de desempenho.

Teste de armazenamento

Da mesma forma, os programas ocasionalmente têm objetivos de armazenamento que informam, por exemplo, a quantidade de memória do sistema que o programa usa e o tamanho dos arquivos temporários ou de log. Você precisa verificar se seu programa pode controlar o uso da memória do sistema para que não afete negativamente outros processos em execução no host. O mesmo vale para arquivos físicos no sistema de arquivos. O preenchimento de uma unidade de disco pode causar um tempo de inatividade significativo. Você deve projetar casos de teste para mostrar que esses objetivos de armazenamento não foram atendidos.

Teste de configuração

Programas como sistemas operacionais, sistemas de gerenciamento de banco de dados e programas de mensagens suportam uma variedade de configurações de hardware, incluindo vários tipos e números de dispositivos de E/S e linhas de comunicação, ou diferentes tamanhos de memória. Muitas vezes, o número de configurações possíveis é muito grande para testar cada uma, mas no mínimo, você deve testar o programa com cada tipo de dispositivo de hardware e com a configuração mínima e máxima. Se o próprio programa puder ser configurado para omitir componentes do programa, ou se o programa puder ser executado em computadores diferentes, cada configuração possível do programa deverá ser testada.

Hoje, muitos programas são projetados para vários sistemas operacionais. Assim, ao testar tal programa, você deve fazê-lo em todos os sistemas operacionais para os quais foi projetado. Programas projetados para serem executados em um navegador da Web requerem atenção especial, pois existem vários navegadores da Web disponíveis e nem todos funcionam da mesma maneira. Dentro

Além disso, o mesmo navegador da Web funcionará de maneira diferente em sistemas operacionais diferentes.

Teste de compatibilidade/conversão

A maioria dos programas desenvolvidos não são completamente novos; muitas vezes são substitutos de algum sistema deficiente. Como tal, os programas geralmente têm objetivos específicos em relação à sua compatibilidade e procedimentos de conversão do sistema existente. Novamente, ao testar o programa em relação a esses objetivos, a orientação dos casos de teste é demonstrar que os objetivos de compatibilidade não foram atendidos e que os procedimentos de conversão não funcionam. Aqui você tenta gerar erros ao mover dados de um sistema para outro. Um exemplo seria atualizar um sistema de banco de dados.

Você deseja garantir que a nova versão seja compatível com seus dados existentes, assim como precisa validar que uma nova versão de um aplicativo de processamento de texto seja compatível com seus formatos de documento anteriores. Existem vários métodos para testar esse processo; no entanto, eles são altamente dependentes do sistema de banco de dados que você emprega.

Teste de instalação

Alguns tipos de sistemas de software têm procedimentos de instalação complicados. Testar o procedimento de instalação é uma parte importante do processo de teste do sistema. Isso é particularmente verdadeiro para um sistema de instalação automatizado que faz parte do pacote do programa. Um programa de instalação com defeito pode impedir que o usuário tenha uma experiência bem-sucedida com o sistema principal que você está testando. A primeira experiência de um usuário é quando ele instala o aplicativo. Se essa fase tiver um desempenho ruim, o usuário/cliente pode encontrar outro produto ou ter pouca confiança na validade do aplicativo.

Teste de confiabilidade

É claro que o objetivo de todos os tipos de teste é a melhoria da confiabilidade do programa, mas se os objetivos do programa contiverem declarações específicas sobre confiabilidade, testes de confiabilidade específicos podem ser planejados. Testar objetivos de confiabilidade pode ser difícil. Por exemplo, um sistema on-line moderno, como uma rede corporativa de longa distância (WAN) ou um provedor de serviços de Internet (ISP), geralmente tem um tempo de atividade desejado de 99,97% ao longo da vida útil do

128 A Arte do Teste de Software

sistema. Não há nenhuma maneira conhecida de testar esse objetivo dentro de um período de teste de meses ou mesmo anos. Os sistemas de software críticos de hoje têm padrões de confiabilidade ainda mais altos, e o hardware de hoje deve suportar esses objetivos. Você potencialmente pode testar programas ou sistemas com objetivos de tempo médio entre falhas (MTBF) mais modestos ou objetivos de erro operacional razoáveis (em termos de teste).

Um MTBF de não mais que 20 horas, ou um objetivo de que um programa não apresente mais de 12 erros únicos após ser colocado em produção, por exemplo, apresenta possibilidades de teste, particularmente para estatística, comprovação de programa ou baseado em modelo. metodologias de teste. Esses métodos estão além do escopo deste livro, mas a literatura técnica (online e não) oferece ampla orientação nessa área. Por exemplo, se essa área de teste de programa for de seu interesse, pesquise o conceito de asserções indutivas. O objetivo deste método é o desenvolvimento de um conjunto de teoremas sobre o programa em questão, cuja demonstração garante a ausência de erros no programa. O método começa escrevendo asserções sobre as condições de entrada do programa e resultados corretos. As asserções são expressas simbolicamente em um sistema lógico formal, geralmente o cálculo de predicados de primeira ordem. Você então localiza cada loop no programa e, para cada loop, escreve uma asserção declarando as condições invariantes (sempre verdadeiras) em um ponto arbitrário do loop. O programa agora foi particionado em um número fixo de caminhos de comprimento fixo (todos os caminhos possíveis entre um par de asserções). Para cada caminho, você pega a semântica das instruções do programa intervenientes para modificar a asserção e, eventualmente, chega ao final do caminho. Neste ponto, existem duas asserções no final do caminho: a original e a derivada da asserção na extremidade oposta.

Você então escreve um teorema afirmando que a afirmação original implica a afirmação derivada e tenta provar o teorema. Se os teoremas puderem ser provados, você pode assumir que o programa não contém erros - contanto que o programa termine. Uma prova separada é necessária para mostrar que o programa sempre terminará eventualmente.

Por mais complexo que pareça esse tipo de prova ou previsão de software, testes de confiabilidade e, de fato, o conceito de engenharia de confiabilidade de software (SRE) estão conosco hoje e são cada vez mais importantes para sistemas que devem manter tempos de atividade muito altos. Para ilustrar esse ponto, examine a Tabela 6.2 para ver o número de horas por ano que um sistema deve funcionar para dar suporte a vários requisitos de tempo de atividade. Esses valores devem indicar a necessidade de SRE.

TABELA 6.2 Horas por ano para vários requisitos de tempo de atividade

Requisitos de porcentagem de tempo de atividade	Horas de funcionamento por ano
100	8760,0
99,9	8751,2
98	8584,8
97	8497,2
96	8409,6
95	8322,0

Teste de recuperação

Programas como sistemas operacionais, sistemas de gerenciamento de banco de dados e programas de teleprocessamento geralmente têm objetivos de recuperação que indicam como o sistema é se recuperar de erros de programação, falhas de hardware e dados erros. Um objetivo do teste do sistema é mostrar que essas funções de recuperação não funcionam corretamente. Erros de programação podem ser injetados propositalmente em um sistema para determinar se ele pode se recuperar deles. Hardware falhas como erros de paridade de memória ou erros de dispositivo de E/S podem ser simulados. Erros de dados, como ruído em uma linha de comunicação ou um inválido ponteiro em um banco de dados pode ser criado propositalmente ou simulado para analisar a reação do sistema.

Um objetivo de projeto de tais sistemas é minimizar o tempo médio de recuperação (MTTR). O tempo de inatividade geralmente faz com que uma empresa perca receita porque o sistema está inoperante. Um objetivo do teste é mostrar que o sistema não cumpre o acordo de nível de serviço para MTTR. Muitas vezes, o MTTR têm um limite superior e inferior, portanto, seus casos de teste devem refletir esses limites.

Teste de manutenção/manutenção

O programa também pode ter objetivos para sua operacionalidade ou manter características de habilidade. Todos os objetivos desse tipo devem ser testados. Tais objetivos podem definir os auxílios de serviço a serem fornecidos com o sistema, incluindo programas de despejo de armazenamento ou diagnósticos, o tempo médio para depurar um problema aparente, os procedimentos de manutenção e a qualidade da documentação da lógica interna.

Teste de Documentação

Como ilustramos na Figura 6.4, o teste do sistema também se preocupa com a precisão da documentação do usuário. A principal forma de realizar este teste é usar a documentação para determinar a representação dos casos de teste de sistema anteriores. Isto é, uma vez que um caso particular de estresse é planejado, você usaria a documentação como um guia para escrever o caso de teste real. Além disso, a própria documentação do usuário deve ser objeto de uma inspeção (semelhante ao conceito de inspeção de código no Capítulo 3), para verificar precisão e clareza. Quaisquer exemplos ilustrados na documentação devem ser codificados em casos de teste e inseridos no programa.

Teste de procedimento

Finalmente, muitos programas são partes de sistemas maiores, não completamente automatizados, envolvendo procedimentos que as pessoas executam. Quaisquer procedimentos humanos prescritos, como aqueles para o operador do sistema, administrador de banco de dados ou usuário final, deve ser testado durante o teste do sistema.

Por exemplo, um administrador de banco de dados deve documentar procedimentos para backup e recuperação do sistema de banco de dados. Se possível, uma pessoa não associada à administração do banco de dados devem testar os procedimentos. No entanto, uma empresa deve criar os recursos necessários para testar adequadamente os procedimentos. Esses recursos geralmente incluem hardware e licenciamento de software adicional.

Executando o teste do sistema

Uma das considerações mais importantes na implementação do teste do sistema é determinar quem deve fazê-lo. Para responder a isso de forma negativa, (1) os programadores não devem realizar um teste de sistema; e (2) de todos os testes fases, esta é a que a organização responsável por desenvolver os programas definitivamente não devem funcionar.

O primeiro ponto decorre do fato de que uma pessoa executando um sistema teste deve ser capaz de pensar como um usuário final, o que implica uma compreensão completa das atitudes e do ambiente do usuário final e de como o programa será usado. Obviamente, então, se viável, um bom candidato a teste é um ou mais usuários finais. No entanto, como o fim típico usuário não terá a habilidade ou experiência para executar muitas das

categorias de testes descritas anteriormente, uma equipe de teste de sistema ideal pode ser composta por alguns especialistas profissionais em teste de sistema (pessoas que passam a vida realizando testes de sistema), um usuário final representativo ou dois, um engenheiro de fatores humanos e os principais analistas originais. ou designers do programa. Incluir os projetistas originais não viola o princípio 2 da Tabela 2.1, "Diretrizes de Teste de Programas Vitais", recomendando não testar seu próprio programa, já que o programa provavelmente passou por muitas mãos desde que foi concebido. Portanto, os designers originais não têm os incômodos vínculos psicológicos com o programa que motivou esse princípio.

O segundo ponto decorre do fato de que um teste de sistema é uma atividade do tipo "vale tudo, não há restrições". Novamente, a organização de desenvolvimento tem vínculos psicológicos com o programa que são contrários a esse tipo de atividade. Além disso, a maioria das organizações de desenvolvimento está mais interessada em que o teste do sistema prossiga da maneira mais suave possível e dentro do cronograma, portanto, não está realmente motivada a demonstrar que o programa não atende aos seus objetivos. No mínimo, o teste do sistema deve ser realizado por um grupo independente de pessoas com poucos ou nenhum vínculo com a organização de desenvolvimento.

Talvez a maneira mais econômica de realizar um teste de sistema (econômico em termos de encontrar o maior número de erros com uma determinada quantia de dinheiro, ou gastar menos dinheiro para encontrar o mesmo número de erros), seja subcontratar o teste a uma empresa separada. Falamos mais sobre isso na última seção deste capítulo.

Teste de aceitação

Voltando ao modelo geral do processo de desenvolvimento mostrado na Figura 6.3, você pode ver que o teste de aceitação é o processo de comparar o programa com seus requisitos iniciais e as necessidades atuais de seus usuários finais. É um tipo incomum de teste, pois geralmente é realizado pelo cliente ou usuário final do programa e normalmente não é considerado responsabilidade da organização de desenvolvimento. No caso de um programa contratado, a organização contratante (usuária) realiza o teste de aceitação comparando a operação do programa com o contrato original.

Como é o caso de outros tipos de teste, a melhor maneira de fazer isso é criar casos de teste que tentem mostrar que o programa não atende ao contrato; se esses casos de teste não forem bem-sucedidos, o programa será aceito. No

132 A Arte do Teste de Software

No caso de um produto de programa, como o sistema operacional de um fabricante de computador ou o sistema de banco de dados de uma empresa de software, o cliente sensato primeiro realiza um teste de aceitação para determinar se o produto satisfaz suas necessidades.

Embora o teste de aceitação final seja, de fato, responsabilidade do cliente ou usuário final, o desenvolvedor experiente conduzirá os testes do usuário durante o ciclo de desenvolvimento e antes de entregar o produto acabado ao usuário final ou cliente contratado. Consulte o Capítulo 7 para obter mais informações sobre teste de usuário ou usabilidade.

Teste de instalação

O processo de teste restante na Figura 6.3 é o teste de instalação. Sua posição na figura é um pouco incomum, pois não está relacionada, como todos os outros processos de teste, a fases específicas do processo de projeto. É um tipo incomum de teste porque seu objetivo não é encontrar erros de software, mas encontrar erros que ocorrem durante o processo de instalação.

Muitos eventos ocorrem durante a instalação de sistemas de software. Uma pequena lista de exemplos inclui o seguinte:

O usuário deve selecionar uma variedade de opções.

Arquivos e bibliotecas devem ser alocados e carregados.

As configurações de hardware válidas devem estar presentes.

Os programas podem precisar de conectividade de rede para se conectar a outros programas.

A organização que produziu o sistema deve desenvolver os testes de instalação, que devem ser entregues como parte do sistema e executados após a instalação do sistema. Entre outras coisas, os casos de teste podem verificar se um conjunto compatível de opções foi selecionado, se todas as partes do sistema existem, se todos os arquivos foram criados e possuem o conteúdo necessário e se a configuração de hardware é adequada.

Planejamento e Controle de Testes

Se você considerar que o teste de um sistema grande pode envolver escrever, executar e verificar dezenas de milhares de casos de teste, lidar com milhares de

de módulos, reparando milhares de erros e empregando centenas de pessoas em um período de um ano ou mais, é evidente que você enfrenta um imenso desafio de gerenciamento de projeto no planejamento, monitoramento e controle do processo de teste. Na verdade, o problema é tão grande que poderíamos dedicar um livro inteiro apenas ao gerenciamento de testes de software. A intenção desta seção é resumir algumas dessas considerações.

Conforme mencionado no Capítulo 2, o principal erro cometido com mais frequência no planejamento de um processo de teste é a suposição tácita de que nenhum erro será encontrado. O resultado óbvio desse erro é que os recursos planejados (pessoas, tempo de calendário e tempo de computador) serão grosseiramente subestimados, um problema notório na indústria de computação. Para agravar o problema, está o fato de que o processo de teste cai no final do ciclo de desenvolvimento, o que significa que as alterações de recursos são difíceis. Um segundo problema, talvez mais insidioso, é que a definição errada de teste está sendo usada, pois é difícil ver como alguém usando a definição correta de teste (o objetivo é encontrar erros) planejaria um teste usando a suposição de que não erros serão encontrados.

Como é o caso da maioria dos empreendimentos, o plano é a parte crucial da gestão do processo de teste. Os componentes de um bom plano de teste são os seguintes:

1. Objetivos. Os objetivos de cada fase de teste devem ser definidos.
2. Critérios de conclusão. Os critérios devem ser elaborados para especificar quando cada fase de teste será considerada completa. Este assunto é discutido na próxima seção.
3. Horários. Os horários do calendário são necessários para cada fase. Eles devem indicar quando os casos de teste serão projetados, escritos e executados. Algumas metodologias de software como Extreme Programming (discutidas no Capítulo 9) exigem que você projete os casos de teste e os testes de unidade antes do início da codificação do aplicativo.
4. Responsabilidades. Para cada fase, as pessoas que projetarão, escreverão, executarão e verificarão os casos de teste e as pessoas que repararão os erros descobertos devem ser identificadas. E, como em grandes projetos surgem inevitavelmente disputas sobre se determinados resultados de testes representam erros, um árbitro deve ser identificado.
5. Bibliotecas e padrões de casos de teste. Em um grande projeto, são necessários métodos sistemáticos de identificação, escrita e armazenamento de casos de teste.

6. Ferramentas. As ferramentas de teste necessárias devem ser identificadas, incluindo um plano para quem irá desenvolvê-los ou adquiri-los, como eles serão usados e quando eles serão necessários.
7. Hora do computador. Este é um plano para a quantidade de tempo de computador necessários para cada fase de teste. Incluiria servidores usados para compilar aplicativos, se necessário; máquinas desktop necessárias para testes de instalação; Servidores Web para aplicações baseadas na Web; em rede dispositivos, se necessário; e assim por diante.
8. Configuração de hardware. Se configurações ou dispositivos especiais de hardware são necessários, é necessário um plano que descreva os requisitos, como eles serão atendidos e quando serão necessários.
9. Integração. Parte do plano de teste é uma definição de como o programa serão reunidos (por exemplo, teste incremental de cima para baixo). Um sistema contendo os principais subsistemas ou programas pode ser montado de forma incremental, usando a abordagem de cima para baixo ou de baixo para cima, para instância, mas onde os blocos de construção são programas ou subsistemas, em vez de módulos. Se este for o caso, um plano de integração do sistema é necessário. O plano de integração do sistema define a ordem de integração, a capacidade funcional de cada versão do sistema e as responsabilidades de produzir "scaffolding", código que simula a função de componentes inexistentes.
10. Procedimentos de rastreamento. Você deve identificar meios para rastrear vários aspectos do progresso do teste, incluindo a localização de módulos propensos a erros e estimativa do progresso em relação ao cronograma, recursos, e critérios de conclusão.
11. Procedimentos de depuração. Você deve definir mecanismos para relatar erros detectados, acompanhar o andamento das correções e adicionar o correções no sistema. Cronogramas, responsabilidades, ferramentas e tempo/recursos do computador também devem fazer parte do plano de depuração.
12. Teste de regressão. O teste de regressão é realizado após fazer uma melhoria funcional ou reparo no programa. Seu propósito é determinar se a mudança regrediu outros aspectos do programa. Geralmente é executado executando novamente algum subconjunto de casos de teste do programa. O teste de regressão é importante porque as mudanças e correções de erros tendem a ser muito mais propensas a erros do que as código do programa original (da mesma forma que a maioria dos erros tipográficos em jornais são o resultado de editoriais de última hora)

alterações, em vez de alterações na cópia original). Um plano para testes de regressão – quem, como, quando – também é necessário.

Critérios de conclusão do teste

Uma das questões mais difíceis de responder ao testar um programa é determinar quando parar, pois não há como saber se o erro detectado é o último erro remanescente. Na verdade, em qualquer coisa que não seja um programa pequeno, não é razoável esperar que todos os erros sejam eventualmente detectados. Dado esse dilema, e dado o fato de que a economia diz que os testes devem terminar, você pode se perguntar se a pergunta deve ser respondida de maneira puramente arbitrária ou se existem alguns critérios úteis de interrupção.

Os critérios de conclusão normalmente usados na prática são sem sentido e contraproducente. Os dois critérios mais comuns são estes:

1. Pare quando o tempo programado para o teste expirar.
2. Pare quando todos os casos de teste forem executados sem detectar erros - ou seja, parar quando os casos de teste não forem bem-sucedidos.

O primeiro critério é inútil porque você pode satisfazê-lo não fazendo absolutamente nada. Não mede a qualidade do teste. O segundo critério é igualmente inútil porque também é independente da qualidade dos casos de teste. Além disso, é contraproducente porque subconscientemente o encoraja a escrever casos de teste com baixa probabilidade de detectar erros.

Conforme discutido no Capítulo 2, os humanos são altamente orientados para objetivos. Se lhe disserem que terminou uma tarefa quando os casos de teste não tiveram sucesso, você inconscientemente escreverá casos de teste que levam a esse objetivo, evitando os casos de teste úteis, de alto rendimento e destrutivos.

Existem três categorias de critérios mais úteis. A primeira categoria, mas não a melhor, é basear a conclusão no uso de metodologias específicas de projeto de casos de teste. Por exemplo, você pode definir a conclusão do teste do módulo da seguinte forma:

Os casos de teste são derivados de (1) satisfazer o critério de cobertura multicondicional e (2) uma análise de valor limite do módulo

especificação de interface, e todos os casos de teste resultantes são eventualmente malsucedidos.

Você pode definir o teste de função como concluído quando o seguinte condições são satisfeitas:

Os casos de teste são derivados de (1) gráficos de causa-efeito, (2) análise de valor de limite e (3) suposição de erro, e todos os casos de teste resultantes são eventualmente malsucedidos.

Embora esse tipo de critério seja superior aos dois mencionados anteriormente, ele apresenta três problemas. Em primeiro lugar, não é útil em uma fase de teste em que metodologias específicas não estejam disponíveis, como a fase de teste do sistema. Em segundo lugar, trata-se de uma medida subjetiva, pois não há como garantir que uma pessoa tenha utilizado uma determinada metodologia, como a análise de valor limite, de forma adequada e rigorosa. Terceiro, em vez de definir uma meta e deixar o testador escolher a melhor maneira de alcançá-la, ele faz o oposto; metodologias de projeto de caso de teste são ditadas, mas nenhum objetivo é dado. Portanto, esse tipo de critério é útil às vezes para algumas fases de teste, mas deve ser aplicado somente quando o testador provou suas habilidades no passado ao aplicar as metodologias de projeto de casos de teste com sucesso.

A segunda categoria de critérios – talvez a mais valiosa – é declarar os requisitos de conclusão em termos positivos. Como o objetivo do teste é encontrar erros, por que não fazer do critério de conclusão a detecção de um número predefinido de erros? Por exemplo, você pode declarar que um teste de módulo de um módulo específico não está completo até que três erros sejam descobertos. Talvez o critério de conclusão de um teste de sistema deva ser definido como a detecção e reparo de 70 erros, ou um tempo decorrido de três meses, o que ocorrer depois.

Observe que, embora esse tipo de critério reforce a definição de teste, ele apresenta dois problemas, ambos superáveis. Um problema é determinar como obter o número de erros a serem detectados.

A obtenção desse número requer as três estimativas a seguir:

1. Uma estimativa do número total de erros no programa.
2. Uma estimativa de qual porcentagem desses erros pode ser encontrada por meio de testes.

3. Uma estimativa de qual fração dos erros originou-se em processos de projeto específicos e durante quais fases de teste esses erros provavelmente serão detectados.

Você pode obter uma estimativa aproximada do número total de erros de várias maneiras. Um método é obtê-los através da experiência com programas anteriores. Além disso, existe uma variedade de módulos preditivos. Alguns deles exigem que você teste o programa por algum período de tempo, registre os tempos decorridos entre a detecção de erros sucessivos e insira esses tempos em parâmetros em uma fórmula. Outros módulos envolvem a propagação de erros conhecidos, mas não divulgados, no programa, testando o programa por um tempo e, em seguida, examinando a proporção de erros propagados detectados para erros não propagados detectados. Outro modelo emprega duas equipes de teste independentes cujos membros testam por um tempo, examinam os erros encontrados por cada uma e os erros detectados em comum por ambas as equipes e usam esses parâmetros para estimar o número total de erros. Outro método bruto para obter essa estimativa é usar médias de toda a indústria. Por exemplo, o número de erros que existem em programas típicos no momento em que a codificação é concluída (antes que um passo a passo ou inspeção de código seja empregado) é de aproximadamente 4 a 8 erros por 100 instruções de programa.

A segunda estimativa da lista anterior (a porcentagem de erros que podem ser encontrados de forma viável por meio de testes) envolve uma estimativa um tanto arbitrária, levando em consideração a natureza do programa e as consequências de erros não detectados.

Dada a atual escassez de informações sobre como e quando os erros são cometidos, a terceira estimativa é a mais difícil. Os dados existentes indicam que, em grandes programas, aproximadamente 40% dos erros são erros de codificação e de projeto lógico, e que o restante é gerado nos processos de projeto anteriores.

Para usar este critério, você deve desenvolver suas próprias estimativas que sejam pertinentes ao programa em questão. Um exemplo simples é apresentado aqui. Suponha que estamos prestes a começar a testar um programa de 10.000 instruções, que o número de erros remanescentes após a execução das inspeções de código é estimado em 5 por 100 instruções, e estabeleçemos, como objetivo, a detecção de 98% da codificação e do projeto lógico erros e 95 por cento dos erros de projeto. O número total de erros é, portanto, estimado em 500. Dos 500 erros, assumimos que 200 são erros de codificação e design lógico e

TABELA 6.3 Estimativa hipotética de quando os erros podem ser encontrados

	Erros de codificação e design lógico	Erros de projeto
Teste do módulo	65%	0%
Teste de funcionamento	30%	60%
Teste do sistema	3%	35%
Total	98%	95%

300 são falhas de projeto. Portanto, o objetivo é encontrar 196 codificação e design lógico erros e 285 erros de projeto. Uma estimativa plausível de quando os erros são provável de ser detectado é mostrado na Tabela 6.3.

Se agendarmos quatro meses para testes de função e três meses para teste do sistema, os três critérios de conclusão a seguir podem ser estabelecido:

1. O teste do módulo é concluído quando 130 erros são encontrados e corrigidos (65 por cento dos 200 erros estimados de codificação e design lógico).
2. O teste de função é concluído quando 240 erros (30 por cento de 200 mais 60 por cento de 300) são encontrados e corrigidos, ou quando quatro meses de teste de função foram concluídos, o que ocorrer mais tarde. A razão para a segunda cláusula é que se encontrarmos 240 erros rapidamente, é provavelmente uma indicação de que subestimamos o número total de erros e, portanto, não deve interromper o teste de função cedo.
3. O teste do sistema é concluído quando 111 erros são encontrados e corrigidos, ou quando três meses de teste do sistema forem concluídos, o que ocorrer mais tarde.

O outro problema óbvio com esse tipo de critério é o da superestimação. E se, no exemplo anterior, houver menos de 240 erros restante quando o teste de função começa? Com base no critério, podemos nunca complete a fase de teste de função.

Este é um problema estranho se você pensar bem: não temos o suficiente erros; o programa é bom demais. Você poderia rotulá-lo como não um problema porque é o tipo de problema que muitas pessoas gostariam de ter. Se isso acontecer ocorrer, um pouco de bom senso pode resolvê-lo. Se não encontrarmos 240 erros em quatro meses, o gerente de projeto pode雇用 uma pessoa de fora para analisar a

casos de teste para julgar se o problema é (1) casos de teste inadequados ou (2) casos de teste excelentes, mas falta de erros para detectar.

O terceiro tipo de critério de conclusão é fácil na superfície, mas envolve muito julgamento e intuição. Requer que você trace o número de erros encontrados por unidade de tempo durante a fase de teste. Ao examinar a forma da curva, muitas vezes você pode determinar se deve continuar a fase de teste ou encerrá-la e iniciar a próxima fase de teste.

Suponha que um programa esteja sendo testado em função e o número de erros encontrados por semana esteja sendo plotado. Se, na sétima semana, a curva for a de cima da Figura 6.5, seria imprudente interromper o teste de função, mesmo que tivéssemos atingido nosso critério de número de erros a serem encontrados.

Como na sétima semana ainda parecemos estar em alta velocidade (encontrando muitos erros), a decisão mais sábia (lembrando que nosso objetivo é encontrar erros) é continuar o teste de função, projetando casos de teste adicionais, se necessário.

Por outro lado, suponha que a curva seja a inferior da Figura 6.5.

A eficiência de detecção de erros caiu significativamente, o que implica que talvez tenhamos escolhido o teste de função de forma limpa e que talvez a melhor jogada seja encerrar o teste de função e iniciar um novo tipo de teste (um teste de sistema, talvez). É claro que também devemos considerar outros fatores, como se a queda na eficiência de detecção de erros foi devido à falta de tempo do computador ou ao esgotamento dos casos de teste disponíveis.

A Figura 6.6 é uma ilustração do que acontece quando você não consegue traçar o número de erros detectados. O gráfico representa três fases de teste de um sistema de software extremamente grande. Uma conclusão óbvia é que o projeto não deveria ter mudado para uma fase de teste diferente após o período 6. Durante o período 6, a taxa de detecção de erros foi boa (para um testador, quanto maior a taxa, melhor), mas mudar para uma segunda fase neste ponto fez com que a taxa de detecção de erros caísse significativamente.

O melhor critério de conclusão é provavelmente uma combinação dos três tipos que acabamos de discutir. Para o teste do módulo, principalmente porque a maioria dos projetos não rastreia formalmente os erros detectados durante esta fase, o melhor critério de conclusão é provavelmente o primeiro. Você deve solicitar que um conjunto específico de metodologias de projeto de caso de teste seja usado. Para as fases de teste de função e sistema, a regra de conclusão pode ser parar quando um número predefinido de erros for detectado ou quando o tempo programado tiver decorrido, o que ocorrer depois, mas desde que uma análise do gráfico de erros versus tempo indique que o teste se tornou improdutivo.

140 A Arte do Teste de Software

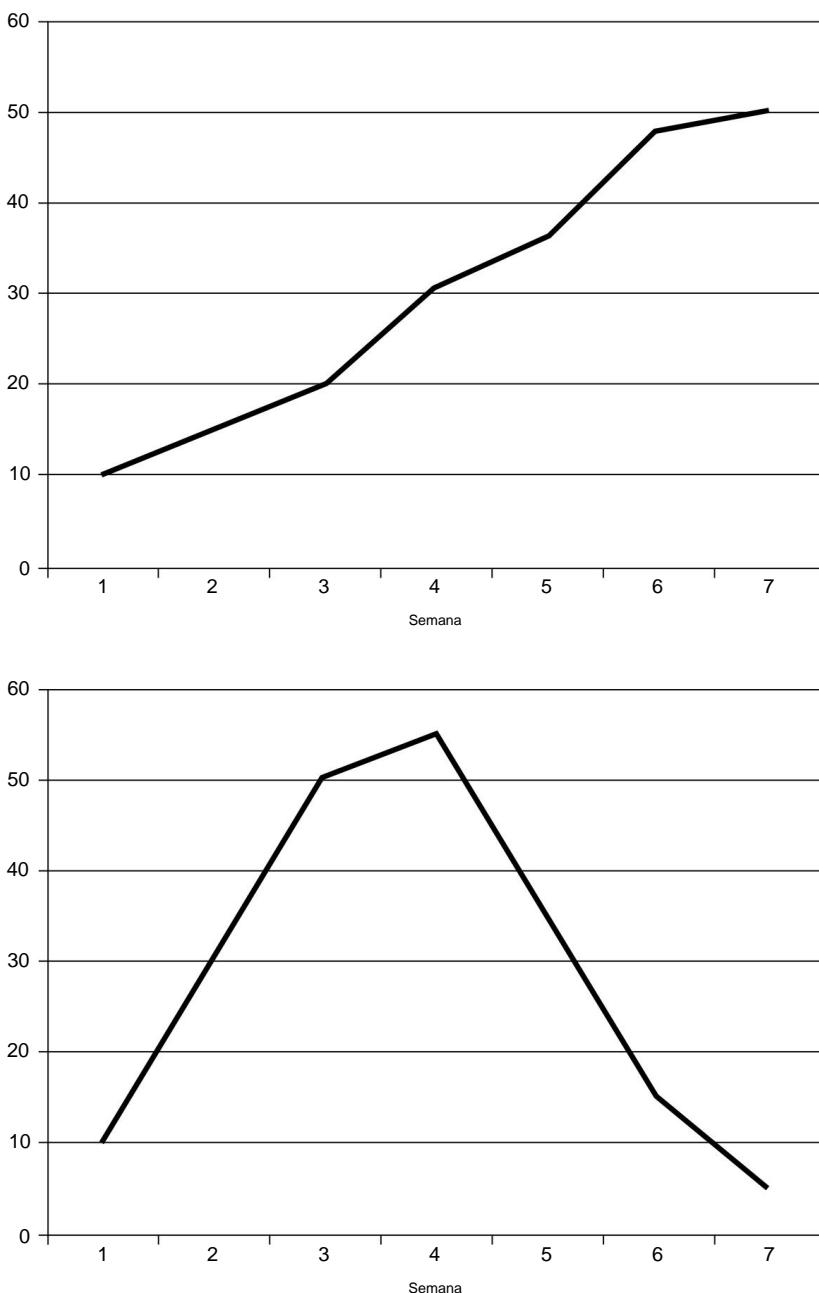


FIGURA 6.5 Estimativa de conclusão por plotagem de erros detectados por Unidade de tempo.

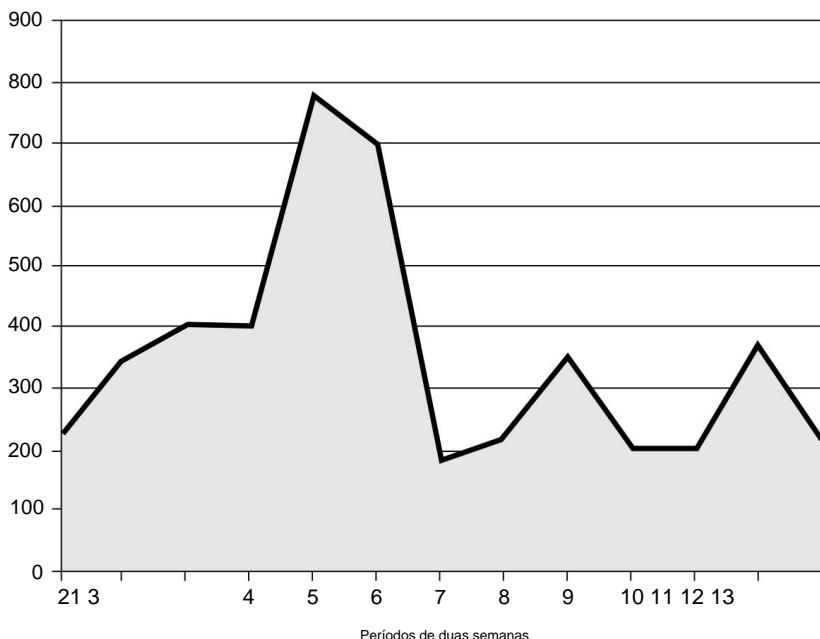


FIGURA 6.6 Estudo post mortem dos processos de teste de um grande Projeto.

A Agência de Testes Independente

Anteriormente neste capítulo e no Capítulo 2, enfatizamos que uma organização deve evitar tentar testar seus próprios programas. Nossa raciocínio é que a organização responsável pelo desenvolvimento de um programa tem dificuldade em testar objetivamente o mesmo programa. A organização do teste deve ser tão o mais distante possível, em termos da estrutura da empresa, da organização de desenvolvimento. Na verdade, é deseável que a organização do teste não faça parte da mesma empresa, pois se for, ainda é influenciada pela mesma pressões gerenciais que influenciam a organização de desenvolvimento.

Uma maneira de evitar esse conflito é contratar uma empresa separada para o software teste. Esta é uma boa idéia, se a empresa que projetou o sistema e irá usá-lo desenvolveu o sistema, ou se um desenvolvedor de terceiros produziu o sistema. As vantagens geralmente observadas são o aumento da motivação no processo de teste, uma competição saudável com o desenvolvimento organização, remoção do processo de teste sob a gestão

controle da organização de desenvolvimento, e as vantagens de conhecimento que a agência de teste independente traz para lidar com o problema.

Resumo

Testes de ordem superior podem ser considerados o próximo passo. Nós discutimos e defendeu o conceito de teste de módulo - usando várias técnicas para componentes de software de teste, os blocos de construção que se combinam para formar o produto final. Com componentes individuais testados e depurados, é tempo para ver o quanto bem eles trabalham juntos.

Testes de ordem superior são importantes para todos os produtos de software, mas tornam-se cada vez mais importantes à medida que o tamanho do projeto aumenta. Fica raciocinar que quanto mais módulos e mais linhas de código um projeto contém, mais oportunidades existem para erros de codificação ou mesmo de design.

O teste de função tenta descobrir erros de projeto, isto é, discrepâncias entre o programa finalizado e suas especificações externas — uma descrição precisa do comportamento do programa da perspectiva do usuário final.

O teste do sistema, por outro lado, testa a relação entre o software e seus objetivos originais. O teste do sistema é projetado para descobrir erros cometidos durante o processo de tradução dos objetivos do programa para o especificação externa e, finalmente, em linhas de código. É esta tradução etapa em que os erros têm os efeitos de maior alcance; da mesma forma, é o palco no processo de desenvolvimento que é mais propenso a erros. Talvez a parte mais difícil do teste do sistema seja projetar os casos de teste. Em geral você quer para se concentrar nas principais categorias de teste e, em seguida, seja realmente criativo nos testes essas categorias. A Tabela 6.1 resume 15 categorias que detalhamos neste capítulo que pode orientar seus esforços de teste do sistema.

Não se engane, testes de ordem superior certamente são uma parte importante do testes de software completos, mas também pode se tornar um processo assustador, especialmente para sistemas muito grandes, como um sistema operacional. A chave para o sucesso é um planejamento de teste consistente e bem planejado. Apresentamos este tópico neste capítulo, mas se você estiver gerenciando o teste de sistemas grandes, mais pensamento e planejamento serão necessários. Uma abordagem para lidar com isso é contratar uma empresa externa para testes ou gerenciamento de testes.

No Capítulo 7, expandimos um aspecto importante do teste de ordem superior: teste de usuário ou de usabilidade.

7

Usabilidade (Usuário)
Teste

Uma categoria importante de casos de teste de sistema é aquela que tenta encontrar problemas de fator humano ou usabilidade. Quando a primeira edição deste livro foi publicada, a indústria da computação ignorou principalmente os fatores humanos associados ao software de computador. Os desenvolvedores deram pouca atenção a como os humanos interagiam com seu software. Isso não quer dizer que não havia desenvolvedores testando aplicativos no nível do usuário. No início da década de 1980, alguns — incluindo desenvolvedores do Xerox Palo Alto Research Center (PARC), por exemplo — estavam realizando testes de software baseados no usuário.

Em 1987 ou 1988, nós três estávamos intimamente envolvidos em testes de usabilidade de hardware e software de computador pessoal inicial, quando contratamos fabricantes de computadores para testar e revisar seus novos computadores de mesa antes de serem lançados ao público. Ao longo de talvez dois anos, este teste de pré-lançamento evitou possíveis problemas de usabilidade com novos designs de hardware e software. Esses primeiros fabricantes de computadores obviamente estavam convencidos de que o tempo e os gastos necessários para esse nível de teste do usuário resultavam em vantagens reais de marketing e financeiras.

Noções básicas de teste de usabilidade

Os sistemas de software de hoje — particularmente aqueles projetados para um mercado comercial de massa — geralmente passaram por extensos estudos sobre o fator humano, e os programas modernos, é claro, se beneficiam dos milhares de programas e sistemas anteriores. No entanto, uma análise do ser humano

fatores ainda é uma questão altamente subjetiva. Aqui está nossa lista de perguntas que você pode fazer para derivar considerações de teste:

1. Cada interface de usuário foi adaptada à inteligência, formação educacional e pressões ambientais do usuário final?
2. As saídas do programa são significativas, não insultam o usuário e desprovidas de rabiscos de computador?

3. Os diagnósticos de erros, como mensagens de erro, são diretos ou o usuário precisa de um doutorado em ciência da computação para compreendê-los?

Por exemplo, o programa produz mensagens como IEK022A OPEN ERROR ON FILE 'SYSIN' ABEND CODE%102? Mensagens como essas não eram tão incomuns em sistemas de software das décadas de 1970 e 1980. Os sistemas do mercado de massa se saem melhor hoje a esse respeito, mas os usuários ainda encontrarão mensagens inúteis, como "Ocorreu um erro desconhecido" ou "Este programa encontrou um erro e deve ser reiniciado".

Os programas que você projeta estão sob seu controle e não devem ser atormentados por mensagens tão inúteis. Mesmo que você não tenha projetado o programa, se estiver na equipe de testes, você pode pressionar por melhorias nessa área da interface humana.

4. O conjunto total de interfaces de usuário exibe considerável integridade conceitual, consistência subjacente e uniformidade de sintaxe, convenções, semântica, formato, estilo e abreviações?

5. Onde a precisão é vital, como em um sistema bancário on-line, há redundância suficiente na entrada? Por exemplo, esse sistema deve solicitar um número de conta, um nome de cliente e um número de identificação pessoal (PIN) para verificar se a pessoa adequada está acessando as informações da conta.

6. O sistema contém um número excessivo de opções ou opções que provavelmente não serão usadas? Uma tendência no software moderno é apresentar aos usuários apenas as opções de menu que eles provavelmente usarão, com base em testes de software e considerações de design. Então, um programa bem projetado pode aprender com usuários individuais e começar a apresentar os itens de menu que eles acessam com frequência. Mesmo com um sistema de menu tão inteligente, os programas bem-sucedidos ainda devem ser projetados para que o acesso às várias opções seja lógico e intuitivo.

7. O sistema retorna algum tipo de reconhecimento imediato para todas as entradas?

Onde um clique do mouse é a entrada, por exemplo, o

item pode mudar de cor, ou um objeto de botão pode ser pressionado ou apresentado em um formato elevado. Se for esperado que o usuário escolha em uma lista, o número selecionado deve ser apresentado na tela quando a escolha for feita. Além disso, se a ação selecionada requer algum tempo de processamento – o que é frequentemente o caso quando o software está acessando um sistema remoto – então uma mensagem deve ser exibida informando o usuário sobre o que está acontecendo. Esse nível de teste às vezes é chamado de teste de componente, pelo qual os componentes de software interativos são testados para seleção razoável e feedback do usuário.

8. O programa é fácil de usar? Por exemplo, a entrada faz distinção entre maiúsculas e minúsculas sem deixar esse fato claro para o usuário? Além disso, se um programa requer navegação por uma série de menus ou opções, está claro como retornar ao menu principal? O usuário pode facilmente subir ou descer um nível?
9. O design é adequado para a precisão do usuário? Um teste seria uma análise de quantos erros cada usuário comete durante a entrada de dados ou ao escolher as opções do programa. Esses erros foram meramente por conveniência - erros que o usuário conseguiu corrigir - ou uma escolha ou ação correta causou algum tipo de falha no aplicativo?
10. As ações do usuário são facilmente repetidas em sessões posteriores? Em outras palavras, o projeto de software é propício para o usuário aprender a ser mais eficiente no uso do sistema?
11. O usuário se sentiu confiante ao navegar pelos diversos caminhos ou opções de menu? Uma avaliação subjetiva pode ser a resposta do usuário ao usar o aplicativo. Ao final da sessão o usuário se sentiu estressado ou satisfeito com o resultado? O usuário provavelmente escolheria este sistema para seu próprio uso ou o recomendaria a outra pessoa?
12. O software cumpriu sua promessa de design? Finalmente, o teste de usabilidade deve incluir uma avaliação das especificações do software versus a operação real. Da perspectiva do usuário - pessoas reais usando o software em um ambiente do mundo real - o software funcionou de acordo com suas especificações?

Usabilidade ou teste baseado no usuário basicamente é uma técnica de teste de caixa preta. Lembre-se de nossa discussão no Capítulo 2 que o teste caixa-preta se concentra em encontrar situações nas quais o programa não se comporta de acordo com as especificações. Em um cenário de caixa preta, você não está preocupado

com o funcionamento interno do software, ou mesmo com a compreensão da estrutura do programa. Apresentado desta forma, o teste de usabilidade obviamente é uma parte importante de qualquer processo de desenvolvimento. Se os usuários perceberem, devido a um design inadequado, uma interface de usuário complicada ou especificações perdidas ou ignoradas, que um determinado aplicativo não funciona de acordo com suas especificações, o processo de desenvolvimento falhou. O teste do usuário deve descobrir problemas de falhas de design a erros de ergonomia de software.

Processo de teste de usabilidade

Deve ser óbvio em nossa lista de itens para testar que o teste de usabilidade é mais do que simplesmente buscar opiniões de usuários ou reações de alto nível a um aplicativo de software. Quando os erros forem encontrados e corrigidos, e um aplicativo estiver pronto para lançamento ou venda, grupos focais podem ser usados para obter opiniões de usuários ou compradores em potencial. Isso é marketing e foco.

O teste de usabilidade ocorre mais cedo no processo e é muito mais envolvido.

Qualquer teste de usabilidade deve começar com um plano. (Reveja nossas diretrizes vitais de teste de software no Capítulo 2, Tabela 2.1.) Você deve estabelecer exercícios práticos, reais e repetíveis para cada usuário conduzir. Projete esses cenários de teste para apresentar ao usuário todos os aspectos do software, talvez em ordem variada ou aleatória. Por exemplo, entre os processos que você pode testar em um aplicativo de rastreamento de clientes estão:

Localize um registro de cliente individual e modifique-o.

Localize um registro de empresa e modifique-o.

Crie um novo registro de empresa.

Excluir um registro da empresa.

Gerar uma lista de todas as empresas de um determinado tipo.

Imprima esta lista.

Exporte uma lista selecionada de contatos para um arquivo de texto ou formato de planilha.

Importe um arquivo de texto ou arquivo de planilha de contatos de outro aplicativo.

Adicione uma fotografia a um ou mais registros.

Crie e salve um relatório personalizado.

Personalize a estrutura do menu.

Durante cada fase do teste, faça com que os observadores documentem a experiência do usuário à medida que executam cada tarefa. Quando o teste estiver concluído, faça uma

entrevistar o usuário ou fornecer um questionário escrito para documentar outros aspectos da experiência do usuário, como sua percepção de uso versus especificação.

Além disso, anote instruções detalhadas para testes de usuário, para garantir que cada usuário comece com as mesmas informações, apresentadas da mesma maneira. Caso contrário, você corre o risco de colorir alguns dos testes se alguns usuários receberem instruções diferentes.

Seleção de usuário de teste

Um protocolo de teste de usabilidade completo geralmente envolve vários testes dos mesmos usuários, bem como testes de vários usuários. Por que vários testes dos mesmos usuários? Uma área que queremos testar é a lembrança do usuário, ou seja, quanto do que um usuário aprende sobre a operação do software é retido de sessão para sessão. Qualquer novo sistema apresentado aos usuários pela primeira vez exigirá algum tempo para aprender, mas se o design de uma aplicação específica for consistente com a indústria ou tecnologia com a qual a comunidade de usuários-alvo está familiarizada, o processo de aprendizado deve ser bastante rápido.

Um usuário já familiarizado com design de engenharia baseado em computador, por exemplo, esperaria que qualquer novo software nessa mesma indústria seguisse certas convenções de terminologia, design de menu e talvez até cor, sombreamento e uso de fonte. Certamente, um desenvolvedor pode se desviar dessas convenções propositalmente para obter melhorias operacionais percebidas, mas se o design for muito longe dos padrões e expectativas da indústria, o software levará mais tempo para que os novos usuários aprendam; na verdade, a aceitação do usuário pode ser tão lenta a ponto de fazer com que o aplicativo seja um fracasso comercial. Se o aplicativo for desenvolvido para um único cliente, tais diferenças podem resultar na rejeição do projeto pelo cliente ou na necessidade de um reprojeto completo da interface do usuário. Qualquer um dos resultados é um erro caro do desenvolvedor.

Portanto, softwares direcionados a um tipo específico de usuário final ou indústria devem ser testados pelo que pode ser descrito como usuários especialistas, pessoas já familiarizadas com essa classe de aplicação em um ambiente do mundo real.

Em contraste, softwares com um mercado-alvo mais geral – software de dispositivos móveis, por exemplo, ou páginas da Web de uso geral – podem ser testados melhor por usuários selecionados aleatoriamente. (Essa seleção de usuário de teste às vezes é chamada de teste de corredor ou teste de interceptação de corredor, o que significa que os usuários escolhidos para teste de software são selecionados aleatoriamente entre as pessoas que passam no corredor.)

De quantos usuários você precisa?

Ao projetar um plano de teste de usabilidade, a pergunta "De quantos testadores eu preciso?" virá à tona. A contratação de testadores de usabilidade geralmente é negligenciada no processo de desenvolvimento e pode adicionar um custo inesperado e caro ao projeto. Você precisa encontrar o número certo de testadores que possam identificar o maior número de erros com o menor investimento de capital.

Intuitivamente, você pode pensar que quanto mais testadores você usar, melhor. Afinal, se você tiver avaliadores suficientes testando seu produto, todos os erros deverão ser encontrados. Primeiro, como mencionado, isso é caro. Em segundo lugar, pode se tornar um pesadelo logístico. Finalmente, é improvável que você consiga detectar 100% dos problemas de usabilidade do seu aplicativo.

Felizmente, pesquisas significativas sobre usabilidade foram realizadas nos últimos 15 anos. Com base no trabalho de Jakob Nielsen, especialista em testes de usabilidade, você pode precisar de menos testadores do que pensa. A pesquisa da Nielsen descobriu que o número de problemas de usabilidade encontrados nos testes é:

$$E \approx 100 \delta_1 \delta_1 L^{\frac{1}{n}}$$

onde: E ≈ 100 por cento dos erros encontrados
n ≈ número de testadores

L ≈ por cento dos problemas de usabilidade encontrados por um testador

Usando a equação com L ≈ 31 por cento, um valor razoável Nielsen também extraído de sua pesquisa, produz o gráfico mostrado na Figura 7.1.

O exame do gráfico revela alguns pontos interessantes. Primeiro, como intuitivamente sabemos, nunca será possível detectar todos os erros de usabilidade no aplicativo. Não é teoricamente possível, porque a curva só converge em 100%; ele nunca realmente o alcança. Em segundo lugar, você só precisa de um pequeno número de testadores. O gráfico mostra que aproximadamente 83% dos erros são detectados por apenas 5 testadores.

Do ponto de vista de um gerente de projeto, esta é uma notícia refrescante. Você não precisa mais arcar com o custo e a complexidade de trabalhar com um grande grupo de testadores para verificar seu aplicativo. Em vez disso, você pode se concentrar em projetar, executar e analisar seus testes, colocando seu esforço e dinheiro no que fará a maior diferença.

Além disso, com menos testadores, você tem menos análises para fazer, para que possa implementar rapidamente as alterações no aplicativo e na metodologia de teste; então

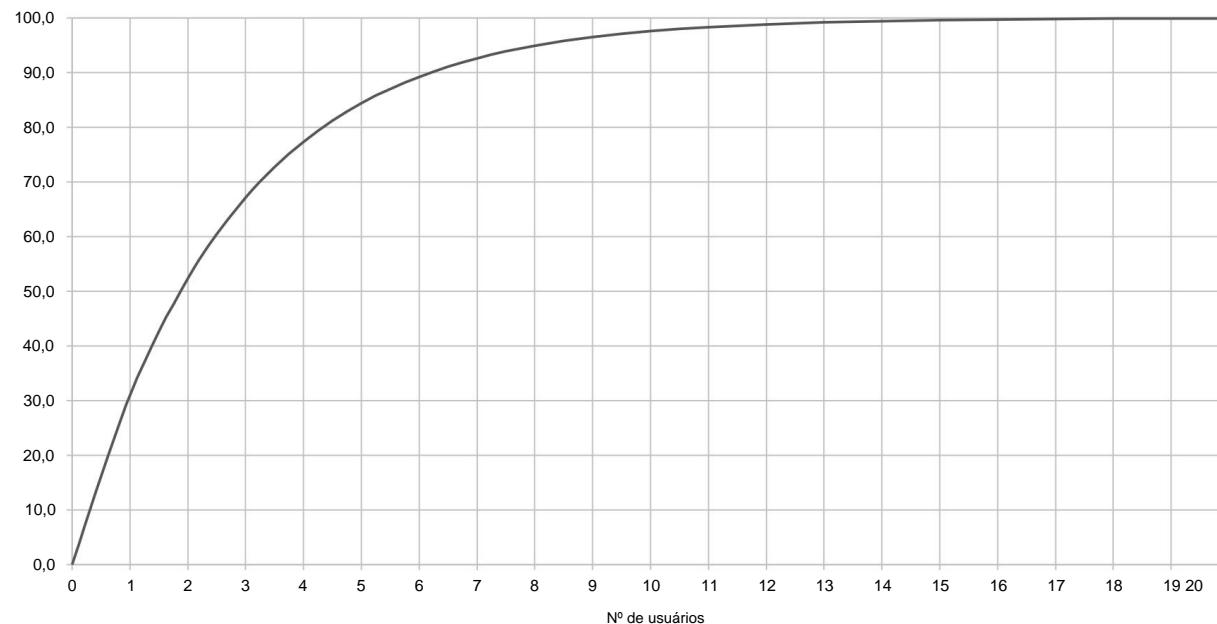


FIGURA 7.1 Porcentagem de Erros Encontrados versus Número de Usuários.

teste novamente com um novo grupo de testadores. Dessa forma iterativa, você pode garantir a resolução da maioria dos problemas com custo e tempo mínimos.

A pesquisa de Nielsen foi realizada no início da década de 1990, enquanto ele era analista de sistemas da Sun Microsystems. Por um lado, seus dados e abordagem para testes de usabilidade fornecem orientações concretas para aqueles envolvidos no design de software. Por outro lado, como os testes de usabilidade se tornaram mais importantes e comuns, e mais evidências foram coletadas a partir de testes práticos e melhores análises de fórmulas, alguns pesquisadores passaram a questionar as afirmações firmes de Nielsen de que três a cinco usuários deveriam ser suficientes.

O próprio Nielsen adverte que o número preciso de testadores depende de considerações econômicas (quantos testadores seu orçamento suportará) e do tipo de sistema que você está testando. Sistemas críticos, como aplicativos de navegação, softwares bancários ou outros softwares financeiros, ou programas relacionados à segurança, exigirão, por força, um exame mais minucioso do usuário do que softwares menos críticos.

Entre as considerações importantes para os desenvolvedores que estão projetando um programa de teste de usabilidade estão se o número de usuários e suas orientações individuais representam suficientemente a população total de usuários potenciais. Além disso, como observa Nielsen, alguns programas são mais complexos do que outros, o que significa que detectar uma porcentagem significativamente grande de erros será mais difícil. E, como usuários diferentes, por causa de suas origens e experiências, provavelmente detectarão diferentes tipos de erros, uma situação de teste individual pode exigir um número maior de testadores.

Como acontece com qualquer metodologia de teste, cabe aos desenvolvedores e administradores de projeto projetar os testes, apresentar um orçamento razoável, avaliar resultados provisórios e conduzir testes regressivos conforme apropriado ao sistema de software, ao projeto geral e ao cliente.

Métodos de coleta de dados

Os administradores de teste ou observadores podem coletar os resultados do teste de várias maneiras.

Gravar um teste de usuário e usar um protocolo de pensamento em voz alta pode fornecer dados excelentes sobre a usabilidade do software e as percepções do usuário sobre o aplicativo.

Um protocolo de pensar em voz alta envolve os usuários falando em voz alta seus pensamentos e observações enquanto executam as tarefas de teste de software atribuídas.

Usando esse processo, os participantes do teste descrevem em voz alta sua tarefa, o que estão pensando sobre a tarefa e/ou qualquer outra coisa que lhes venha à mente à medida que avançam no cenário de teste. Mesmo ao usar o pensamento em voz alta

No teste de protocolo, os desenvolvedores podem querer acompanhar os participantes após o teste para obter comentários, sentimentos e observações pós-teste. Juntos, esses dois níveis de pensamentos e comentários do usuário podem fornecer feedback valioso aos desenvolvedores para correções ou melhorias de software.

Uma desvantagem do processo de pensar em voz alta, onde estão envolvidos filmagens ou servidores de observação, é a possibilidade de que a experiência do usuário seja obscurecida ou modificada pelo ambiente não natural do usuário. Os desenvolvedores também podem querer realizar testes de usuários remotos, onde o aplicativo é instalado na empresa do usuário de teste, onde o software pode ser aplicado. O teste remoto tem a vantagem de colocar o usuário em um ambiente familiar, no qual o aplicativo final provavelmente seria usado, eliminando assim a possibilidade de influências externas modificarem os resultados do teste. Obviamente, a desvantagem é que os desenvolvedores podem não receber feedback tão detalhado quanto seria possível com um protocolo de pensamento em voz alta.

No entanto, em um ambiente de teste remoto, dados precisos do usuário ainda podem ser coletados. Software adicional pode ser instalado com o aplicativo a ser testado para coletar pressionamentos de tecla do usuário e capturar o tempo necessário para que o usuário conclua cada tarefa atribuída. Isso requer tempo de desenvolvimento adicional (e mais software), mas os resultados de tais testes podem ser esclarecedores e muito detalhados.

Na ausência de software de captura de tempo ou tecla, os usuários de teste podem ser encarregados de anotar os horários de início e término de cada tarefa atribuída, juntamente com breves comentários de uma palavra ou frase curta durante o processo. Questionários ou entrevistas pós-teste podem ajudar os usuários a recordar seus pensamentos e opiniões sobre o software.

Um protocolo de coleta de dados sofisticado, mas potencialmente útil, é o rastreamento ocular. Quando lemos uma página impressa, vemos uma apresentação gráfica ou interagimos com uma tela de computador, nossos olhos se movem sobre o material digitalizado em padrões específicos. Dados de pesquisa coletados sobre o movimento dos olhos ao longo de mais de 100 anos mostram que o movimento dos olhos – particularmente quanto tempo um observador pausa em certos elementos visuais – reflete pelo menos até certo ponto os processos de pensamento do observador. O rastreamento desse movimento ocular, que pode ser feito com sistemas de vídeo e outras tecnologias, mostra aos pesquisadores quais elementos visuais atraem a atenção do observador, em que ordem e por quanto tempo. Esses dados são potencialmente úteis para determinar a eficiência das telas de software apresentadas aos usuários.

Apesar de extensa pesquisa durante a última metade do século XX, no entanto, alguma controvérsia permanece sobre o valor final do olho.

pesquisa de movimento em aplicações específicas. Ainda assim, juntamente com outras técnicas de teste do usuário, onde os desenvolvedores precisam dos dados de entrada do usuário mais profundos possíveis para garantir o mais alto nível de eficiência do software (sistemas de orientação de armas, sistemas de controle robótico, controles de veículos ou outros sistemas que exigem respostas rápidas e precisas), rastreamento pode ser uma ferramenta útil.

Questionário de usabilidade

Assim como no próprio procedimento de teste de software, um questionário de usabilidade deve ser cuidadosamente planejado para retornar as informações necessárias do procedimento de teste associado. Embora você possa querer incluir algumas perguntas que estimulem comentários de forma livre do usuário, em geral você deseja desenvolver questionários que gerem respostas que possam ser contadas e analisadas em todo o espectro de testadores. Estes caem em três tipos gerais:

Sim/não respostas

Respostas verdadeiro/falso

Concordo/discordo em uma escala

Por exemplo, em vez de perguntar "Qual é a sua opinião sobre o sistema de menu principal", você pode fazer uma série de perguntas que exigem uma resposta de 1 a 5, onde 5 é concordo totalmente e 1 discordo totalmente:

1. O menu principal era fácil de navegar.
2. Foi fácil encontrar a operação adequada do software na página principal cardápio.
3. O design da tela me levou rapidamente ao software operacional correto escolhas.
4. Depois de operar o sistema, foi fácil lembrar como repetir minhas ações.
5. As operações do menu não forneceram feedback suficiente para verificar meu escolhas.
6. O menu principal era mais difícil de navegar do que outros profissionais semelhantes gramas que eu uso.
7. Tive dificuldade em repetir operações realizadas anteriormente.

Observe que pode ser uma boa prática fazer a mesma pergunta mais de uma vez, mas apresentá-la da perspectiva oposta para que se elicia

uma resposta negativa e outra positiva. Tal prática pode garantir que o usuário tenha entendido a questão e que as percepções permaneçam constantes. Além disso, você deseja separar o questionário do usuário em seções que correspondem às áreas de software testadas ou às tarefas de teste atribuídas.

A experiência lhe ensinará rapidamente quais tipos de perguntas são propícias à análise de dados e quais não são muito úteis. O software de análise estatística está disponível para ajudar a capturar e interpretar os dados. Com um pequeno número de usuários de teste, os resultados do teste de usabilidade podem ser óbvios; ou você pode desenvolver uma rotina de análise ad hoc em um aplicativo de planilha para documentar melhor os resultados. Para grandes sistemas de software que passam por testes extensivos com uma grande base de usuários, o software estatístico pode ajudar a descobrir tendências que não são óbvias com métodos de interpretação manual.

Quando é o suficiente, o suficiente?

Como você planeja o teste de usabilidade para que todos os aspectos do software sejam razoavelmente testados, mantendo-se dentro de um orçamento aceitável? A resposta a essa pergunta, é claro, depende em parte da complexidade do sistema ou unidade que está sendo testada. Se o orçamento e o tempo permitirem, é aconselhável testar o software em etapas, à medida que cada segmento é concluído. Se componentes individuais foram testados ao longo do processo de desenvolvimento, a série final de testes precisa apenas testar a operação integrada das peças.

Além disso, você pode projetar testes de componentes, que se destinam a testar a usabilidade de um componente interativo, algo que requer entrada do usuário e que responde a essa entrada de maneira perceptível pelo usuário. Esse tipo de teste de feedback pode ajudar a melhorar a experiência do usuário, reduzir erros operacionais e melhorar a consistência do software. Novamente, se você testou um sistema de software neste nível enquanto a interface do usuário estava sendo projetada, você terá coletado um conjunto significativo de testes importantes e conhecimento operacional antes do início do teste total do sistema.

Quantos usuários individuais devem testar seu software? Novamente, a complexidade do sistema e os resultados dos testes iniciais devem ditar o número de testadores individuais. Por exemplo, se três ou cinco (ou um número razoável) de usuários tiverem dificuldade em navegar da tela de abertura para as telas que suportam as tarefas atribuídas e se esses usuários forem suficientemente representativos do mercado-alvo, você provavelmente terá informações suficientes para informar você que a interface do usuário precisa de mais trabalho de design.

Um corolário razoável para isso pode ser que, se nenhum dos testadores iniciais tiver problemas para navegar pelas tarefas atribuídas e nenhum descobrir erros ou mau funcionamento, talvez o conjunto de testes seja muito pequeno.

Afinal, é razoável supor que os testes de usabilidade de um sistema de software razoavelmente complexo não descobrirão erros ou alterações necessárias? Lembre-se do princípio 6, da Tabela 2.1: Examinar um programa para ver se ele não faz o que deveria fazer é apenas metade da batalha; a outra metade é ver se o programa faz o que não deveria fazer. Há uma diferença sutil nessa comparação. Você pode descobrir que uma série de usuários determina que um programa, de fato, parece fazer o que deveria fazer. Eles não encontram erros ou problemas ao trabalhar com o software. Mas eles também provaram que o programa não está fazendo nada que não deveria fazer? Se as coisas parecem estar funcionando muito bem durante o teste inicial, provavelmente é hora de mais testes.

Não acreditamos que exista uma fórmula que diga quantos testes cada usuário deve realizar ou quantas iterações de cada teste devem ser necessárias. Acreditamos, no entanto, que a análise cuidadosa e a compreensão dos resultados que você coleta de um número razoável de testadores e testes podem orientá-lo para a resposta de quando testes suficientes são suficientes.

Resumo

Softwares modernos, aliados à pressão da competição intensa e prazos apertados, tornam o teste do usuário de qualquer produto de software crucial para o desenvolvimento bem-sucedido. É lógico que o usuário de software visado pode ser um ativo valioso durante o teste. O usuário experiente pode determinar se o produto atende ao objetivo de seu projeto e, ao realizar tarefas do mundo real, pode encontrar erros de comissão e omissão.

Dependendo do mercado-alvo do software, os desenvolvedores também podem se beneficiar da seleção de usuários aleatórios – pessoas que não estão familiarizadas com a especificação do programa, ou talvez até mesmo a indústria ou mercado para o qual ele se destina – que podem descobrir erros ou problemas de interface do usuário. Pela mesma razão que os desenvolvedores não são bons testadores de erros, usuários experientes podem evitar áreas operacionais que possam produzir problemas porque sabem como o software deve funcionar. Ao longo de muitos anos de desenvolvimento de software, descobrimos uma verdade de teste inevitável: o software que o desenvolvedor testou por muitas horas pode ser quebrado facilmente e em um

curto espaço de tempo, por um usuário não sofisticado que tenta uma tarefa para a qual o interface do usuário ou o software não foi projetado.

Lembre-se, também, que a chave para o teste de usuário (ou usabilidade) bem-sucedido é a coleta e análise de dados precisos e detalhados. O processo de coleta de dados realmente começa com o desenvolvimento de instruções detalhadas para o usuário e uma lista de tarefas. Ele termina compilando os resultados da observação do usuário ou questionários pós-teste.

Finalmente, os resultados dos testes devem ser interpretados e, em seguida, os desenvolvedores deve efetuar alterações de software identificadas a partir dos dados. Este pode ser um processo iterativo em que os mesmos usuários de teste são solicitados a concluir tarefas após a conclusão das alterações de software identificadas.

8

Depuração

Em resumo, depuração é o que você faz depois de executar um caso de teste. Lembre-se de que um caso de teste bem-sucedido é aquele que mostra que um programa não faz o que foi projetado para fazer. A depuração é um processo de duas etapas que começa quando você encontra um erro como resultado de um caso de teste bem-sucedido. A etapa 1 é a determinação da natureza e localização exatas do erro suspeito dentro do programa. O passo 2 consiste em corrigir o erro.

Por mais necessário e integral que a depuração seja para testar o programa, parece ser o aspecto do processo de produção de software que os programadores menos apreciam, principalmente por estas razões:

Seu ego pode atrapalhar. Goste ou não, a depuração confirma que os programadores não são perfeitos; eles cometem erros no design ou na codificação do programa.

Você pode ficar sem vapor. De todas as atividades de desenvolvimento de software, a depuração é a atividade mais desgastante mentalmente. Além disso, a depuração geralmente é realizada sob uma tremenda pressão organizacional ou auto-induzida para corrigir o problema o mais rápido possível.

Você pode perder o seu caminho. A depuração é mentalmente desgastante porque o erro que você encontrou pode ocorrer em praticamente qualquer instrução dentro do programa. Sem examinar o programa primeiro, você não pode ter certeza absoluta, por exemplo, que a origem de um erro numérico em um cheque de pagamento produzido por um programa de folha de pagamento não é uma sub-rotina que pede ao operador para carregar um formulário específico na impressora . Compare isso com

a depuração de um sistema físico, como um automóvel. Se um carro parar ao subir uma inclinação (o sintoma), você pode eliminar imediatamente certas partes do sistema como a causa do problema – o rádio AM/FM, por exemplo, ou o velocímetro ou a trava do porta-malas. O problema deve estar no motor; e, com base em nosso conhecimento geral de motores automotivos, podemos até descartar certos componentes do motor, como a bomba d'água e o filtro de óleo.

Você pode estar por conta própria. Em comparação com outras atividades de desenvolvimento de software, existem comparativamente poucas pesquisas, literatura e instruções formais sobre o processo de depuração.

Embora este seja um livro sobre teste de software, não sobre depuração, os dois processos estão obviamente relacionados. Dos dois aspectos da depuração, localizar o erro e corrigi-lo, localizar o erro representa talvez 95% do problema. Portanto, este capítulo se concentra no processo de encontrar a localização de um erro, dado que um caso de teste bem-sucedido encontrou um.

Depuração por força bruta

O esquema mais comum para depurar um programa é o chamado método de força bruta. É popular porque requer pouca reflexão e é o menos mentalmente desgastante dos métodos; infelizmente, é ineficiente e geralmente mal sucedido.

Os métodos de força bruta podem ser divididos em pelo menos três categorias:

Depuração com um dump de armazenamento.

Depuração de acordo com a sugestão comum de "distribuir instruções de impressão por todo o seu programa". Depuração com ferramentas de depuração automatizadas.

O primeiro, a depuração com um dump de armazenamento (geralmente uma exibição grosseira de todos os locais de armazenamento em formato hexadecimal ou octal) é o mais ineficiente dos métodos de força bruta. Aqui está o porquê:

É difícil estabelecer uma correspondência entre os locais de memória e as variáveis em um programa fonte.

Com qualquer programa de complexidade razoável, tal despejo de memória produzirá uma enorme quantidade de dados, a maioria dos quais é irrelevante.

Um despejo de memória é uma imagem estática do programa, mostrando o estado do programa em apenas um instante no tempo; para encontrar erros, você tem que estudar a dinâmica de um programa (o estado muda ao longo do tempo).

Um despejo de memória raramente é produzido no ponto exato do erro, portanto, não mostra o estado do programa no ponto do erro. As ações do programa entre a hora do dump e a hora do erro podem mascarar as pistas que você precisa para encontrar o erro.

Não existem metodologias adequadas para encontrar erros analisando um dump de memória (muitos programadores ficam olhando, com olhos vidrados, esperando ansiosamente que o erro se exponha magicamente a partir do dump do programa).

Espalhar instruções ao longo de um programa com falha para exibir valores de variáveis não é muito melhor. Pode ser melhor do que um despejo de memória porque mostra a dinâmica de um programa e permite examinar informações mais fáceis de relacionar com o programa de origem, mas esse método também tem muitas deficiências:

Em vez de encorajá-lo a pensar sobre o problema, é em grande parte um método de acertar ou errar.

Produz uma enorme quantidade de dados a serem analisados.

Requer que você altere o programa; tais mudanças podem mascarar o erro, alterar relacionamentos críticos de temporização ou introduzir novos erros.

Pode funcionar em programas pequenos, mas o custo de usá-lo em programas grandes é bastante alto. Além disso, muitas vezes nem é viável em certos tipos de programas, como sistemas operacionais ou programas de controle de processos.

As ferramentas de depuração automatizadas funcionam de maneira semelhante à inserção de instruções de impressão no programa, mas, em vez de fazer alterações no programa, você analisa a dinâmica do programa com os recursos de depuração da linguagem de programação ou ferramentas de depuração interativas especiais. Recursos típicos de linguagem que podem ser usados são recursos que produzem rastreamentos impressos de execuções de instruções, chamadas de sub-rotinas e/ou alterações de variáveis especificadas. Uma capacidade e função comum das ferramentas de depuração é definir pontos de interrupção que fazem com que o programa seja suspenso quando uma instrução específica é executada ou quando uma variável específica é alterada, permitindo que o programador examine o estado atual do programa. Este método,

também, é em grande parte um acerto ou erro, e muitas vezes resulta em uma quantidade excessiva de dados irrelevantes.

O problema geral com esses métodos de força bruta é que eles ignoram o processo de pensamento. Você pode fazer uma analogia entre a depuração do programa e a solução de um homicídio. Em praticamente todos os romances de mistério de assassinato, o crime é resolvido por uma análise cuidadosa das pistas e juntando detalhes aparentemente insignificantes. Este não é um método de força bruta; configurar bloqueios de estradas ou realizar buscas de propriedades seria.

Há também algumas evidências que indicam que se as equipes de depuração são compostas por programadores experientes ou estudantes, pessoas que usam seus cérebros em vez de um conjunto de ajudas trabalham mais rápido e com mais precisão para encontrar erros de programa. Portanto, só podemos recomendar métodos de força bruta: (1) quando todos os outros métodos falharem, ou (2) como um complemento, não um substituto, para os processos de pensamento que descreveremos a seguir.

Depuração por indução

Deve ser óbvio que um pensamento cuidadoso encontrará a maioria dos erros sem que o depurador chegue perto do computador. Um processo de pensamento em particular é a indução, onde você se move das particularidades de uma situação para o todo.

Ou seja, comece com as pistas (os sintomas do erro e possivelmente os resultados de um ou mais casos de teste) e procure as relações entre as pistas. O processo de indução é ilustrado na Figura 8.1.

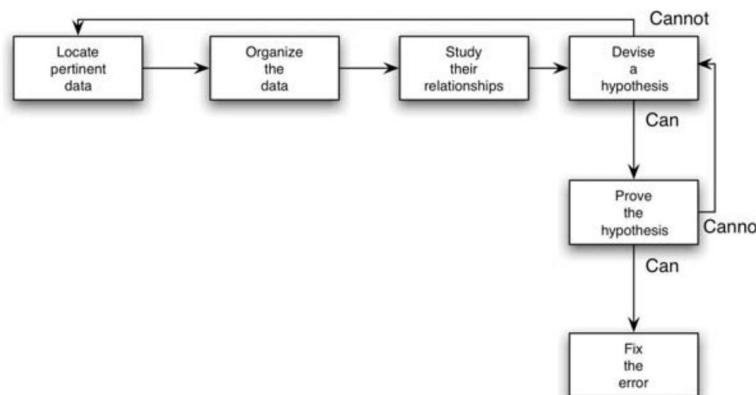


FIGURA 8.1 O Processo de Depuração Indutiva.

Os passos são os seguintes:

1. Localize os dados pertinentes. Um grande erro cometido pelos depuradores é não levar em conta todos os dados ou sintomas disponíveis sobre o problema. Portanto, o primeiro passo é a enumeração de tudo o que você sabe sobre o que o programa fez corretamente e o que fez incorretamente – os sintomas que levaram a acreditar que houve um erro. Pistas valiosas adicionais são fornecidas por casos de teste semelhantes, mas diferentes, que não causam o aparecimento dos sintomas.
2. Organize os dados. Lembre-se de que a indução implica que você está processando do particular ao geral, então o segundo passo é estruturar os dados pertinentes para permitir que você observe os padrões. De particular importância é a busca de contradições, eventos como o erro ocorrem apenas quando o cliente não possui saldo devedor em sua conta margem.

Você pode usar um formulário como o mostrado na Figura 8.2 para estruturar os dados disponíveis. Nas caixas "o que" liste os sintomas gerais; nas caixas "onde" descreva onde os sintomas foram observados; nas caixas "quando", liste tudo o que você sabe sobre os momentos em que os sintomas ocorreram; e nas caixas "até que ponto" descrevem o escopo e a magnitude dos sintomas. Observe o "é" e "não é"

?	Is	Is not
What		
Where		
When		
To what extent		

FIGURA 8.2 Um Método para Estruturar as Pistas.

colunas: Nelas descrevem as contradições que podem eventualmente levar a uma hipótese sobre o erro.

3. Elabore uma hipótese. Em seguida, estude as relações entre as pistas e elabore, usando os padrões que possam ser visíveis na estrutura das pistas, uma ou mais hipóteses sobre a causa do erro. Se você não pode conceber uma teoria, mais dados são necessários, talvez de novos casos de teste. Se várias teorias parecerem possíveis, selecione primeiro a mais provável.
4. Prove a hipótese. Um grande erro neste ponto, dadas as pressões sob as quais a depuração geralmente é executada, é pular esta etapa e tirar conclusões precipitadas para corrigir o problema. Resista a esse impulso, pois é vital provar a razoabilidade da hipótese antes de prosseguir. Se você pular esta etapa, provavelmente conseguirá corrigir apenas o sintoma do problema, não o problema em si. Prove a hipótese comparando-a com as pistas ou dados originais, certificando-se de que esta hipótese explica completamente a existência das pistas. Caso contrário, a hipótese é inválida, a hipótese está incompleta ou vários erros estão presentes.
5. Corrija o problema. Você pode prosseguir com a correção do problema depois de concluir as etapas anteriores. Ao dedicar um tempo para trabalhar completamente em cada etapa, você pode se sentir confiante de que sua correção corrigirá o bug. Lembre-se, porém, que você ainda precisa realizar algum tipo de teste de regressão para garantir que sua correção de bug não criou problemas em outras áreas do programa. À medida que o aplicativo cresce, aumenta também a probabilidade de sua correção causar problemas em outros lugares.

Como um exemplo simples, suponha que um erro aparente foi relatado no programa de classificação do exame descrito no Capítulo 4. O erro aparente é que a nota mediana parece incorreta em alguns casos, mas não em todos. Em um caso de teste específico, 51 alunos foram avaliados. A pontuação média foi impressa corretamente como 73,2, mas a mediana impressa foi 26 em vez do valor esperado de 82. Ao examinar os resultados deste caso de teste e de alguns outros casos de teste, as pistas são organizadas conforme mostrado na Figura 8.3.

O próximo passo é derivar uma hipótese sobre o erro procurando padrões e contradições. Uma contradição que vemos é que o erro parece ocorrer apenas em casos de teste que usam um número ímpar de alunos. Isso pode ser uma coincidência, mas parece significativo, já que você calcula uma mediana de maneira diferente para conjuntos de números pares e ímpares. Há outro padrão estranho: em alguns casos de teste, a mediana calculada sempre é menor

?	Is	Is not
What	The median printed in report 3 is incorrect.	The calculation of the mean or standard deviation.
Where	Only on report 3.	On the other reports. The students' grades seem to be calculated correctly.
When	Occurred in a test run using 51 students.	Did not occur in the test runs for 2 and 200 students.
To what extent	The median printed was 26. It also occurred in the test run using one student; the median printed in this case was 1!	

FIGURA 8.3 Um Exemplo de Estruturação de Pistas.

maior ou igual ao número de alunos (26 51 e 1 1). Um caminho possível neste momento é executar o caso de teste de 51 alunos novamente, dando aos alunos notas diferentes de antes para ver como isso afeta o cálculo da mediana. Se fizermos isso, a mediana ainda será 26, então o "até que ponto! a caixa is not" poderia ser preenchida com, "A mediana parece ser independente das notas reais." Embora este resultado forneça uma pista valiosa, poderíamos ter sido capazes de supor o erro sem ele. A partir dos dados disponíveis, a mediana calculada parece ser igual à metade do número de alunos, arredondado para o próximo número inteiro. Em outras palavras, se você pensar nas notas como sendo armazenadas em uma tabela ordenada, o programa está imprimindo o número de entrada do aluno do meio ao invés de sua nota. Portanto, temos uma hipótese firme sobre a natureza precisa do erro. Em seguida, provamos a hipótese examinando o código ou executando alguns casos de teste extras.

Depuração por Dedução

O processo de dedução parte de algumas teorias ou premissas gerais, usando os processos de eliminação e refinamento, para chegar a uma conclusão (a localização do erro), conforme mostrado na Figura 8.4.

Ao contrário do processo de indução em um caso de assassinato, por exemplo, onde você induz um suspeito a partir das pistas, usando a dedução, você começa com um conjunto de suspeitos e, pelo processo de eliminação (o jardineiro tem

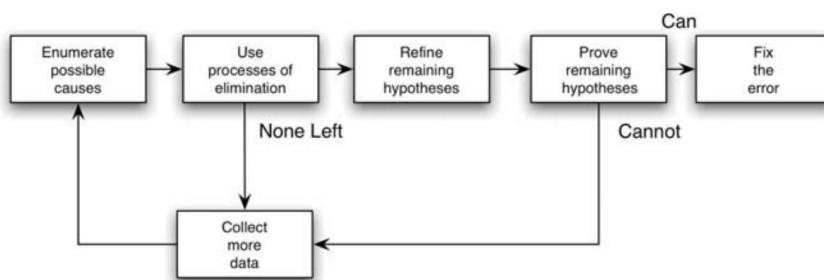


FIGURA 8.4 O processo de depuração dedutiva.

um álibi válido) e refinamento (deve ser alguém ruivo), decidem que o mordomo deve ter feito isso. Os passos são os seguintes:

1. Enumere as possíveis causas ou hipóteses. O primeiro passo é desenvolver uma lista de todas as causas concebíveis do erro. Eles não precisam ser explicações completas; são apenas teorias para ajudá-lo a estruturar e analisar os dados disponíveis.
2. Use os dados para eliminar possíveis causas. Examine cuidadosamente todos os dados, principalmente procurando por contradições (você pode usar a Figura 8.2 aqui), e tente eliminar todas as causas possíveis, exceto uma. Se todos forem eliminados, você precisará de mais dados obtidos de casos de teste adicionais para elaborar novas teorias. Se mais de uma causa possível permanecer, selecione primeiro a causa mais provável – a hipótese principal.
3. Refinar a hipótese restante. A possível causa neste ponto pode estar correta, mas é improvável que seja específica o suficiente para identificar o erro. Assim, o próximo passo é usar as pistas disponíveis para refinar a teoria. Por exemplo, você pode começar com a ideia de que "há um erro no tratamento da última transação no arquivo" e refinar para "a última transação no buffer é sobreposta com o indicador de fim de arquivo".
4. Prove a hipótese restante. Este passo vital é idêntico ao passo 4 em o método de indução.
5. Corrija o erro. Novamente este passo é idêntico ao passo 5 no método de indução. Para enfatizar novamente, você deve testar completamente sua correção para garantir que ela não crie problemas em outras partes do aplicativo.

Como exemplo, suponha que estamos iniciando o teste de função do comando DISPLAY discutido no Capítulo 4. Dos 38 casos de teste

Test case input	Expected output	Actual Output
DISPLAY.E	000000 = 0000 4444 8888 CCCC	M1 INVALID COMMAND SYNTAX
DISPLAY 21 v- 29	0000020 = 0000 4444 8888 CCCC	000020 = 4444 8888 CCCC 0000
DISPLAY .11	000000 = 0000 4444 8888 CCCC 000010 = 0000 4444 8888 CCCC	000000 = 0000 4444 8888 CCCC
DISPLAY 8000 - END	M2 STORAGE REQUESTED IS BEYOND ACTUAL MEMORY LIMITS	008000 = 0000 4444 8888 CCCC

FIGURA 8.5 Resultados do caso de teste do comando DISPLAY.

identificado pelo processo de representação gráfica de causa-efeito, começamos executando quatro casos de teste. Como parte do processo de estabelecimento de condições de entrada, vamos inicializar a memória que o primeiro, quinto, nono, . palavras têm o valor 000; o segundo, sexto, . o terceiro, . . . , palavras têm o valor 4444; sétimo, . oitavo, . palavras têm o valor 8888; e a quarta, CCCC. Ou seja, cada palavra de memória é inicializado com o dígito hexadecimal de ordem inferior no endereço do primeiro byte da palavra (os valores das localizações 23FC, 23FD, 23FE e 23FF são C).

Os casos de teste, sua saída esperada e a saída real após o teste são mostrados na Figura 8.5.

Obviamente, temos alguns problemas, já que aparentemente nenhum dos testes casos produziram os resultados esperados (todos foram bem sucedidos). Mas vamos começar por depurar o erro associado ao primeiro caso de teste. O comando indica que, começando no local 0 (o padrão), os locais E (14 em decimal) devem ser exibidos. (Lembre-se de que a especificação afirmava que todas as saídas conter quatro palavras, ou 16 bytes por linha.)

Enumerando as possíveis causas para a mensagem de erro inesperada, pode obter:

1. O programa não aceita a palavra DISPLAY.
2. O programa não aceita o período.
3. O programa não permite um default como primeiro operando; ele espera um endereço de armazenamento para preceder o período.
4. O programa não permite um E como uma contagem de bytes válida.

166 A Arte do Teste de Software

O próximo passo é tentar eliminar as causas. Se todos forem eliminados, devemos recuar e expandir a lista. Se houver mais de um, podemos examinar casos de teste adicionais para chegar a uma única hipótese de erro ou prosseguir com a causa mais provável. Como temos outros casos de teste em mãos, vemos que o segundo caso de teste na Figura 8.5 parece eliminar a primeira hipótese; e o terceiro caso de teste, embora tenha produzido um resultado incorreto, parece eliminar a segunda e a terceira hipóteses.

O próximo passo é refinar a quarta hipótese. Parece bastante específico, mas a intuição pode nos dizer que há mais do que aparenta – soa como um exemplo de um erro mais geral. Podemos afirmar, então, que o programa não reconhece os caracteres hexadecimais especiais A–F. Essa ausência de tais caracteres nos outros casos de teste faz com que isso pareça uma explicação viável.

Em vez de tirar uma conclusão precipitada, no entanto, devemos primeiro considerar todas as informações disponíveis. O quarto caso de teste pode representar um erro totalmente diferente ou pode fornecer uma pista sobre o erro atual. Dado que o endereço válido mais alto em nosso sistema é 7FFF, como o quarto caso de teste poderia exibir uma área que parece não existir? O fato de que os valores exibidos são nossos valores inicializados, e não lixo, pode levar à suposição de que este comando está de alguma forma exibindo algo no intervalo 0–7FFF. Uma ideia que pode surgir é que isso poderia ocorrer se o programa estivesse tratando os operandos no comando como valores decimais em vez de hexadecimais, conforme indicado na especificação. Isso é confirmado pelo terceiro caso de teste: em vez de exibir 32 bytes de memória, o próximo incremento acima de 11 em hexadecimal (17 na base 10), ele exibe 16 bytes de memória, o que é consistente com nossa hipótese de que os 11 estão sendo tratado como um valor de base 10. Portanto, a hipótese refinada é que o programa está tratando a contagem de bytes como operandos de endereço de armazenamento e os endereços de armazenamento na listagem de saída como valores decimais.

O último passo é provar esta hipótese. Observando o quarto caso de teste, se 8000 for interpretado como um número decimal, o valor de base 16 correspondente será 1F40, o que levaria à saída mostrada. Como prova adicional, examine o segundo caso de teste. A saída está incorreta, mas se 21 e 29 forem tratados como números decimais, os locais dos endereços de armazenamento 15–1D serão exibidos; isso é consistente com o resultado errônneo do caso de teste. Portanto, quase certamente localizamos o erro: O programa está assumindo que os operandos são valores decimais e está imprimindo os endereços de memória como valores decimais, o que é inconsistente com a especificação.

Além disso, esse erro parece ser a causa dos resultados errôneos de todos os quatro casos de teste. Um pouco de reflexão levou ao erro e também resolveu três outros problemas que, à primeira vista, parecem não estar relacionados.

Observe que o erro provavelmente se manifesta em dois locais no programa: a parte que interpreta o comando de entrada e a parte que imprime os endereços de memória na listagem de saída.

Além disso, esse erro, provavelmente causado por um mal-entendido na especificação, reforça a sugestão de que um programador não deve tentar testar seu próprio programa. Se o programador que criou este erro também estiver projetando os casos de teste, ele provavelmente cometerá o mesmo erro ao escrever os casos de teste. Em outras palavras, as saídas esperadas do programador não seriam as da Figura 8.5; seriam as saídas calculadas sob a suposição de que os operandos são valores decimais. Portanto, esse erro fundamental provavelmente passaria despercebido.

Depuração por Backtracking

Um método eficaz para localizar erros em pequenos programas é retroceder os resultados incorretos através da lógica do programa até encontrar o ponto em que a lógica se desviou. Em outras palavras, comece no ponto em que o programa fornece o resultado incorreto, como onde os dados incorretos foram impressos. Aqui, você deduz da saída observada quais devem ter sido os valores das variáveis do programa. Realizando uma execução reversa mental do programa a partir deste ponto e aplicando repetidamente a lógica if-then que afirma "se este era o estado do programa neste ponto, então este deve ter sido o estado do programa aqui em cima", você pode identificar rapidamente o erro. Você está procurando o local no programa entre o ponto em que o estado do programa era o esperado e o primeiro ponto em que o estado do programa não era o esperado.

Depuração por teste

O último método de depuração do tipo "pensamento" é o uso de casos de teste. Isso provavelmente soa um pouco peculiar, já que, no início deste capítulo, distinguimos depuração de teste. No entanto, considere dois tipos de casos de teste: casos de teste para teste, cuja finalidade é expor um erro não detectado anteriormente, e casos de teste para depuração, cuja finalidade é fornecer

informações úteis para localizar um erro suspeito. A diferença entre os dois é que os casos de teste para testes tendem a ser "gordos", pois você está tentando cobrir muitas condições em um pequeno número de casos de teste. Os casos de teste para depuração, por outro lado, são "simples", porque você deseja cobrir apenas uma única condição ou algumas condições em cada caso de teste.

Em outras palavras, depois de descobrir um sintoma de um erro suspeito, você escreve variantes do caso de teste original para tentar identificar o erro. Na verdade, este não é um método totalmente separado; muitas vezes é usado em conjunto com o método de indução para obter informações necessárias para gerar uma hipótese e/ou provar uma hipótese. Também é usado com o método de dedução para eliminar causas suspeitas, refinar a hipótese restante e/ou provar uma hipótese.

Princípios de depuração

Nesta seção, queremos discutir um conjunto de princípios de depuração que são de natureza psicológica. Assim como os princípios de teste no Capítulo 2, muitos desses princípios de depuração são intuitivamente óbvios, mas muitas vezes são esquecidos ou ignorados.

Como a depuração é um processo de duas partes – localizar um erro e depois repará-lo – discutimos dois conjuntos de princípios aqui.

Princípios de localização de erros

Pense Conforme implícito na seção anterior, a depuração é um processo de solução de problemas. O método mais eficaz de depuração envolve uma análise mental das informações associadas aos sintomas do erro. Um depurador de programa eficiente deve ser capaz de identificar a maioria dos erros sem se aproximar de um computador. Veja como:

1. Posicione-se em um lugar calmo, onde estímulos externos — vozes de colegas de trabalho, telefones, rádio ou outras possíveis interrupções — não interfiram em sua concentração.
2. Sem olhar para o código do programa, reveja em sua mente como o programa foi projetado, como o software deve funcionar dentro da área que está funcionando incorretamente.
3. Concentre-se no processo para um desempenho correto e, em seguida, imagine maneiras pelas quais o código pode ser projetado incorretamente.

Esse tipo de pré-pensar o processo de depuração física, em muitos casos, o levará diretamente à área do programa que está causando problemas e o ajudará a obter uma correção rapidamente.

Se você chegar a um impasse, durma nele O subconsciente humano é um poderoso solucionador de problemas. O que muitas vezes chamamos de inspiração é simplesmente a mente subconsciente trabalhando em um problema quando a mente consciente está focada em outra coisa, como comer, caminhar ou assistir a um filme.

Se você não conseguir localizar um erro em um período de tempo razoável (talvez 30 minutos para um programa pequeno, várias horas para um maior), abandone-o e volte sua atenção para outra coisa, pois sua eficiência de pensamento está prestes a entrar em colapso de qualquer maneira. Depois de deixar o problema de lado por um tempo, sua mente subconsciente terá resolvido o problema, ou sua mente consciente estará limpa para um novo exame de seus sintomas.

Temos usado esta técnica regularmente ao longo dos anos, tanto como processo de desenvolvimento quanto como processo de depuração. Pode exigir alguma prática para aceitar esse funcionamento extraordinário do cérebro humano e fazer uso eficiente dele, mas funciona. Na verdade, acordamos durante a noite para perceber que resolvemos um problema de software enquanto dormimos. Por isso, recomendamos que você tenha ao lado da cama um pequeno gravador, um telefone com capacidade de gravação de voz, um PDA ou um bloco de notas para capturar a solução encontrada durante o sono. Resista à tentação de voltar a dormir acreditando que poderá regenerar a solução pela manhã. Você provavelmente não vai — pelo menos não em nossa experiência.

Se você chegar a um impasse, descreva o problema para outra pessoa Falar sobre o problema com outra pessoa pode ajudá-lo a descobrir algo novo. Na verdade, muitas vezes, simplesmente descrevendo o problema para um bom ouvinte, de repente você verá a solução sem qualquer ajuda da pessoa.

Use as ferramentas de depuração apenas como um segundo recurso Recorra às ferramentas de depuração somente depois de tentar outros métodos e apenas como um complemento, não um substituto para o pensamento. Conforme observado anteriormente neste capítulo, as ferramentas de depuração, como dumps e rastreamentos, representam uma abordagem aleatória para depuração. Experimentos mostram que as pessoas que evitam essas ferramentas, mesmo quando estão depurando programas que não conhecem, são mais bem-sucedidas do que as pessoas que usam as ferramentas.

170 A Arte do Teste de Software

Por que deveria ser assim? Depender de uma ferramenta para resolver um problema pode causar um curto-circuito no processo de diagnóstico. Se você acredita que a ferramenta pode resolver o problema, provavelmente estará menos atento às pistas que já pegou, informações que poderiam ajudá-lo a resolver o problema diretamente, sem o auxílio de uma ferramenta genérica de diagnóstico.

Evite a experimentação — use-a apenas como último recurso O erro mais comum cometido por depuradores novatos é tentar resolver um problema fazendo alterações experimentais no programa. Você pode pensar, "Eu sei o que está errado, então vou mudar esta instrução DO e ver o que acontece." Essa abordagem totalmente aleatória não pode nem ser considerada depuração; representa um ato de esperança cega. Não só tem uma chance minúscula de sucesso, mas muitas vezes você vai agravar o problema adicionando novos erros ao programa.

Técnicas de reparo de erros onde há

um bug, é provável que haja outro Esta é uma reafirmação do princípio 9 no Capítulo 2, que afirma que quando você encontra um erro em uma seção de um programa, a probabilidade da existência de outro erro nessa mesma seção é maior do que se você ainda não tivesse encontrado um erro. Em outras palavras, os erros tendem a se agrupar. Ao reparar um erro, examine sua vizinhança imediata em busca de qualquer outra coisa que pareça suspeita.

Corrija o erro, não apenas um sintoma dele Outra falha comum é reparar os sintomas do erro, ou apenas uma instância do erro, em vez do erro em si. Se a correção proposta não corresponder a todas as pistas sobre o erro, você pode estar corrigindo apenas uma parte do erro.

A probabilidade de a correção ser correta não é 100 por cento Diga isso a alguém em uma conversa geral e é claro que ela concordaria; mas conte para alguém no processo de correção de um erro e você pode obter uma resposta diferente—"Sim, na maioria dos casos, mas esta correção é tão pequena que tem que funcionar." Nunca assuma que o código adicionado a um programa para corrigir um erro está correto. Declaração por declaração, as correções são muito mais propensas a erros do que o código original no programa. Uma implicação é que as correções de erros devem ser testadas, talvez com mais rigor do que o original.

programa. Um plano de teste de regressão sólido pode ajudar a garantir que a correção de um erro não introduza outro erro em outro lugar do aplicativo.

A probabilidade de a correção ser correta diminui à medida que o tamanho do programa aumenta. Afirmando de forma diferente, em nossa experiência, a proporção de erros causados por correções incorretas versus erros originais aumenta em programas grandes. Em um grande programa amplamente utilizado, um em cada seis novos erros descobertos é um erro em uma correção anterior do programa.

Se você aceita isso como um fato, como pode evitar causar problemas tentando consertá-los? Leia as três primeiras técnicas nesta seção, para começar. Um erro encontrado não significa que todos os erros foram encontrados e você deve ter certeza de que está corrigindo o erro real, não apenas seu sintoma.

Cuidado com a possibilidade de que uma correção de erro crie um novo erro. Não apenas você precisa se preocupar com correções incorretas, mas também com uma correção aparentemente válida tendo um efeito colateral indesejado, introduzindo assim um novo erro. Não só há uma probabilidade de que uma correção seja inválida, mas também há uma probabilidade de que uma correção introduza um novo erro. Uma implicação é que você não apenas precisa testar a situação de erro após a correção, mas também deve realizar testes de regressão para determinar se um novo erro foi introduzido.

O processo de reparo de erros deve colocá-lo de volta temporariamente na fase de projeto. Perceba que a correção de erros é uma forma de projeto de programa. Dada a natureza propensa a erros das correções, o senso comum diz que quaisquer procedimentos, metodologias e formalismos usados no processo de design também devem ser aplicados ao processo de correção de erros. Por exemplo, se o projeto racionalizou que as inspeções de código eram desejáveis, então deve ser duplamente importante que elas sejam implementadas após a correção de um erro.

Alterar o código-fonte, não o código-objeto. Ao depurar sistemas grandes, principalmente aqueles escritos em linguagem assembly, ocasionalmente há a tendência de corrigir um erro fazendo uma alteração imediata no código-objeto, com a intenção de alterar o programa-fonte posteriormente. Dois problemas estão associados a esta abordagem: (1) Geralmente é um sinal de que a “depuração por experimentação” está sendo praticada; e (2) o código-objeto e o programa-fonte estão agora fora de sincronia, o que significa que o erro pode facilmente ressurgir quando o programa for recompilado ou

remontado. Essa prática é uma indicação de uma abordagem desleixada e pouco profissional à depuração.

Erro de análise

O último ponto a ser percebido sobre a depuração de programa é que, além de seu valor em remover um erro do programa, ela pode ter outro efeito valioso: ela pode nos dizer algo sobre a natureza dos erros de software, algo sobre o qual ainda sabemos muito pouco. . As informações sobre a natureza dos erros de software podem fornecer feedback valioso em termos de melhoria dos processos futuros de projeto, codificação e teste.

Cada programador e organização de programação poderia melhorar imensamente realizando uma análise detalhada dos erros detectados, ou pelo menos um subconjunto deles. Reconhecidamente, é uma tarefa difícil e demorada, pois implica muito mais do que um agrupamento superficial como "x por cento dos erros são erros de projeto lógico" ou "x por cento dos erros ocorrem em instruções IF ". " Uma análise cuidadosa pode incluir os seguintes estudos:

Onde foi cometido o erro? Essa pergunta é a mais difícil de responder, pois requer uma busca retroativa na documentação e no histórico do projeto; ao mesmo tempo, é também a pergunta mais valiosa. Requer que você identifique a fonte original e a hora do erro. Por exemplo, a fonte original do erro pode ser uma declaração ambígua em uma especificação, uma correção de um erro anterior ou um mal-entendido de um requisito do usuário final.

Quem cometeu o erro? Não seria útil descobrir que 60% dos erros de projeto foram criados por um dos 10 analistas, ou que o programador X comete três vezes mais erros que os outros programadores? (Não para fins de punição, mas para fins de educação.)

O que foi feito de forma incorreta? Não é suficiente determinar quando e por quem cada erro foi cometido; o elo perdido é uma determinação de exatamente por que o erro ocorreu. Foi causado pela incapacidade de alguém de escrever com clareza? A falta de educação de alguém na linguagem de programação? Um erro de digitação? Uma suposição inválida? Uma falha em considerar uma entrada válida?

Como o erro poderia ter sido evitado? O que pode ser feito de diferente no próximo projeto para evitar esse tipo de erro? A resposta para isso

pergunta constitui muito do valioso feedback ou aprendizado que estamos procurando.

Por que o erro não foi detectado antes? Se o erro foi detectado durante uma fase de teste, você deve estudar por que o erro não foi descoberto durante as fases de teste anteriores, inspeções de código e revisões de projeto.

Como o erro pode ter sido detectado antes? A resposta para isso

oferece outro feedback valioso. Como os processos de revisão e teste podem ser melhorados para encontrar esse tipo de erro mais cedo em projetos futuros? Desde que não estejamos analisando um erro encontrado por um usuário final (ou seja, o erro foi encontrado por um caso de teste), devemos perceber que algo valioso aconteceu: escrevemos um caso de teste bem-sucedido. Por que este caso de teste foi bem-sucedido? Podemos aprender algo com isso que resultará em casos de teste adicionais bem-sucedidos, seja para este programa ou para programas futuros?

Repetimos, esse processo de análise é difícil e caro, mas as respostas que você pode descobrir ao passar por ele podem ser inestimáveis para melhorar os esforços de programação subsequentes. A qualidade dos produtos futuros aumentará enquanto o investimento de capital diminuirá. É alarmante que a grande maioria dos programadores e organizações de programação não o empregue.

Resumo

O foco principal deste livro é o teste de software: como você descobre o maior número possível de erros de software? Portanto, não queremos gastar muito tempo na próxima etapa - depuração - mas o fato simples é que os erros encontrados por casos de teste bem-sucedidos levam diretamente a ela.

Neste capítulo, abordamos alguns dos aspectos mais importantes da depuração de software. O método menos desejável, depuração por força bruta, envolve técnicas como despejar locais de memória, colocar instruções de impressão em todo o programa ou usar ferramentas automatizadas. As técnicas de força bruta podem apontar para a solução de alguns erros descobertos durante o teste, mas não são uma maneira eficiente de depurar.

Demonstramos que você pode começar a depurar estudando os sintomas de erro, ou pistas, e passando deles para a imagem maior (depuração induutiva). Outra técnica inicia o processo de depuração considerando teorias gerais, então, através do processo de eliminação, identifica

174 A Arte do Teste de Software

os locais de erro (depuração dedutiva). Também abordamos o retrocesso do programa – começando com o erro e retrocedendo no programa para determinar onde as informações incorretas se originaram. Por fim, discutimos a depuração por meio de testes.

Se, no entanto, oferecêssemos uma única diretriz para aqueles encarregados de depurar um sistema de software, diríamos: "Pense!" Revise os inúmeros princípios de depuração descritos neste capítulo. Acreditamos que eles podem guiá-lo na direção certa, em direção à depuração precisa e eficiente. Mas a linha inferior é, depende de sua experiência e conhecimento do próprio programa. Abra sua mente para soluções criativas, revise o que você sabe e deixe seu conhecimento e subconsciente levá-lo aos locais de erro.

No próximo capítulo, abordaremos o assunto de testes extremos, técnicas adequadas para ajudar a descobrir erros em ambientes de programação extremos, como o desenvolvimento ágil.

9

Testes no Agile Meio Ambiente

O aumento da concorrência e a interconexão em todos os mercados **forçaram** empresas a encurtar seu tempo de colocação no mercado, continuando a fornecer produtos de alta qualidade para seus clientes. Isso é particularmente verdadeiro na indústria de desenvolvimento de software, onde a Internet possibilita entrega de aplicativos e serviços de software. Seja criando um produto para as massas ou para o departamento de recursos humanos, um fato permanece imutável: o cliente do século XXI exige uma aplicação de qualidade entregue quase imediatamente. Infelizmente, os processos tradicionais de desenvolvimento de software não conseguem acompanhar esse ambiente competitivo.

No início dos anos 2000, um grupo de desenvolvedores se reuniu para discutir o estado das metodologias leves e de desenvolvimento rápido. Na reunião, eles compararam notas para identificar como são os projetos de software bem-sucedidos; o que fez com que alguns projetos tivessem sucesso enquanto outros se arrastavam. No final, eles criou o "Manifesto para Desenvolvimento Ágil de Software", um documento que tornou-se a pedra angular do movimento ágil. Menos uma metodologia discreta, o Manifesto Ágil (Figura 9.1) é uma filosofia única que se concentra em clientes e funcionários, em vez de abordagens e hierarquias rígidas.

Características do desenvolvimento ágil

O desenvolvimento ágil promove o desenvolvimento iterativo e incremental, com testes significativos, que são centrados no cliente e aceitam mudanças durante o processo. Todos os atributos das abordagens tradicionais de desenvolvimento de software

Estamos descobrindo melhores maneiras de desenvolver software fazendo-o e ajudando outros a fazê-lo.

Através deste trabalho passamos a valorizar:

Indivíduos e interações sobre processos e ferramentas

Software que trabalha sobre uma documentação completa

Colaboração do cliente sobre a negociação do contrato

Responder à mudança ao invés de seguir um plano

Ou seja, enquanto houver valor nos itens

à direita, valorizamos mais os itens à esquerda.

Kent Beck

Mike Beedle

Arie van Bennekum

Alistair Cockburn

Ward Cunningham

Martin Fowler

James Grenning

Jim Highsmith

Andrew Hunt

Ron Jeffries

Jon Kern

Brian Marick

Robert C. Martin

Steve Mellor

Ken Schwaber

Jeff Sutherland

Dave Thomas

2001, os autores acima

esta declaração pode ser copiada livremente em qualquer forma,

mas apenas na sua totalidade através deste aviso.

FIGURA 9.1 Manifesto de Desenvolvimento Ágil de Software.

negligenciar ou minimizar a importância do cliente. Embora as metodologias ágeis incorporem flexibilidade em seus processos, a ênfase principal é na satisfação do cliente. O cliente é um componente chave do processo; Simplificando, sem o envolvimento do cliente, o método Agile falha. E saber que sua interação é bem-vinda ajuda os clientes a aumentar a satisfação e confiança no produto final e na equipe de desenvolvimento. Se o cliente não está comprometido, então processos mais tradicionais podem ser uma melhor escolha de desenvolvimento.

Ironicamente, o desenvolvimento ágil não tem uma metodologia de desenvolvimento única ou processo; muitas abordagens de desenvolvimento rápido podem ser consideradas Agile. Essas abordagens, no entanto, compartilham três tópicos comuns: elas dependem de envolvimento do cliente, exigem testes significativos e têm ciclos de desenvolvimento iterativos curtos. Está além do escopo deste livro cobrir cada metodologia em detalhes, mas na Tabela 9.1 identificamos as metodologias consideradas Ágeis e damos uma breve descrição de cada uma. (Pedimos que você aprenda

TABELA 9.1 Metodologias de Desenvolvimento Ágil

Metodologia	Descrição
Modelagem Ágil	Não tanto uma única metodologia de modelagem, mas uma coleção de princípios e práticas para modelagem e documentação de sistemas de software. Usado para dar suporte a outros métodos, como Extreme Programming e Scrum.
Processo Unificado Ágil	Versão simplificada do Rational Unified Process (RUP) adaptada para desenvolvimento Agile.
Sistemas Dinâmicos	Com base em abordagens de desenvolvimento de aplicativos rápidos, esta metodologia depende do envolvimento contínuo do cliente e usa uma abordagem iterativa e incremental
Método de Desenvolvimento	abordagem, com o objetivo de entregar software no prazo e dentro do orçamento.
Processo Unificado Essencial (EssUP)	Uma adaptação do RUP em que você escolhe as práticas (por exemplo, casos de uso ou programação em equipe) que se adequam ao seu projeto. O RUP geralmente usa todas as práticas, necessárias ou não.
Programação extrema	Outra abordagem iterativa e incremental que depende muito de testes unitários e de aceitação. Provavelmente a mais conhecida das metodologias ágeis.
Guiado por recursos	Uma metodologia que usa as melhores práticas do setor, como compilações regulares, modelagem de objetos de domínio e equipes de recursos, que são orientadas pelo conjunto de recursos do cliente.
Desenvolvimento	
Processo Unificado Aberto	Uma abordagem ágil para implementar práticas unificadas padrão que permite que uma equipe de software desenvolva rapidamente seu produto.
Scrum	Uma abordagem de gerenciamento de projetos iterativa e incremental que suporta muitas metodologias ágeis.
Rastreamento de velocidade	Aplica-se a todas as metodologias de desenvolvimento Agile. Ele tenta medir a taxa, ou "velocidade", na qual o processo de desenvolvimento está se movendo.

mais sobre eles porque representam a essência da filosofia Ágil.) Além disso, abordamos a Programação Extrema, uma das metodologias Ágeis mais populares, com mais detalhes posteriormente neste capítulo, e oferecemos um exemplo prático.

Vale a pena notar que algumas metodologias ágeis são coleções, ou adaptações, de processos tradicionais de desenvolvimento de software. O Processo Unificado Essencial (EssUP) é um exemplo. O EssUP utiliza processos do Rational Unified Process (RUP) e outros modelos de processos de desenvolvimento de software conhecidos que suportam a filosofia de desenvolvimento Agile.

Não se engane, adotar uma metodologia de desenvolvimento ágil é um desafio. É preciso a combinação certa de desenvolvedores, gerentes e clientes para que funcione. Mas no final, o produto se beneficiará de testes constantes e forte envolvimento do cliente.

Testes ágeis

Em essência, o teste ágil é uma forma de teste colaborativo, em que todos estão envolvidos no processo por meio do design, implementação e execução do plano de teste. Os clientes estão envolvidos na definição de testes de aceitação definindo casos de uso e atributos do programa. Os desenvolvedores colaboram com os testadores para criar equipamentos de teste que podem testar a funcionalidade automaticamente.

O teste ágil exige que todos estejam envolvidos no processo de teste, o que exige muita comunicação e colaboração.

Como acontece com a maioria dos aspectos do desenvolvimento ágil, o teste ágil exige envolver o cliente o mais cedo possível e durante todo o ciclo de desenvolvimento. Por exemplo, uma vez que os desenvolvedores produzam uma base de código estável, os clientes devem começar o teste de aceitação e fornecer feedback à equipe de desenvolvimento. Isso também significa que o teste não é uma fase; em vez disso, é integrado aos esforços de desenvolvimento para impulsionar o progresso contínuo.

Para garantir que o cliente receba um produto estável com o qual realizar testes de aceitação, os desenvolvedores geralmente começam escrevendo testes de unidade primeiro e depois passam para unidades de software de codificação. Os testes de unidade são testes de falha, em que os desenvolvedores os projetam para fazer com que seu software falhe em algum requisito. Paradoxalmente, os desenvolvedores devem escrever software com falha para, de fato, testar o teste. Uma vez que os equipamentos de teste estão em vigor, os desenvolvedores continuam a escrever o software que passa nos testes de unidade.

Para facilitar o feedback oportuno necessário para o desenvolvimento rápido, o teste ágil depende de testes automatizados. Os ciclos de desenvolvimento são curtos, então o tempo é valioso e o teste automatizado é mais confiável do que as abordagens de teste manual. O teste manual não só é demorado, como também pode introduzir bugs. Existem vários conjuntos de testes comerciais e de código aberto. Realmente não importa qual desses conjuntos de testes disponíveis é usado, apenas

que desenvolvedores e testadores usam um. Embora alguns problemas possam exigir testes manuais exploratórios, os testes automatizados são preferidos.

Os ambientes de desenvolvimento ágeis geralmente compreendem apenas pequenas equipes de desenvolvedores, que também atuam como testadores. Projetos maiores com mais recursos podem incluir um testador individual ou um grupo de teste. Em ambos os casos, os testadores não devem ser considerados indicadores. Seu trabalho é levar o projeto adiante, fornecendo feedback sobre a qualidade do software para que os desenvolvedores possam implementar correções de bugs e fazer alterações de requisitos e melhorias gerais.

O teste ágil se encaixa bem na metodologia Extreme Programming, na qual os desenvolvedores criam primeiro os testes de unidade, depois o software. No restante deste capítulo, abordamos Programação Extrema e Teste Extremo com mais detalhes.

Programação e testes extremos

Na década de 1990 nasceu uma metodologia inovadora de desenvolvimento de software chamada Extreme Programming (XP). Um gerente de projeto chamado Kent Beck é creditado por conceber este processo de desenvolvimento ágil e leve, testando-o pela primeira vez enquanto trabalhava em um projeto na Daimler-Chrysler em 1996. Embora vários outros processos de desenvolvimento de software Agile tenham sido criados, o XP ainda é o mais popular. Na verdade, existem inúmeras ferramentas de código aberto para apoiá-lo, o que comprova a popularidade do XP entre desenvolvedores e gerentes de projeto.

O XP provavelmente foi desenvolvido para suportar a adoção de linguagens de programação como Java, Visual Basic e C#.

Essas linguagens baseadas em objetos permitem que os desenvolvedores criem aplicativos grandes e complexos muito mais rapidamente do que com linguagens tradicionais, como C, Fortran ou COBOL. O desenvolvimento com essas linguagens geralmente requer a construção de bibliotecas de uso geral para dar suporte aos esforços de codificação do aplicativo. Métodos para tarefas comuns como impressão, classificação, rede e análise estatística não são componentes padrão. Linguagens como C# e Java são fornecidas com interfaces de programação de aplicativos (APIs) com recursos completos que eliminam ou reduzem a necessidade de criar bibliotecas personalizadas.

No entanto, juntamente com os benefícios das linguagens de desenvolvimento rápido de aplicativos, vieram as responsabilidades. Embora os desenvolvedores estivessem criando aplicativos muito mais rapidamente, sua qualidade não era garantida. Se um aplicativo compilado, muitas vezes não atende às especificações do cliente ou

expectativas. A metodologia de desenvolvimento XP facilita a criação de programas de qualidade em prazos curtos. Embora os processos de software clássicos ainda funcionem, eles geralmente levam muito tempo, o que equivale a perda de receita na arena altamente competitiva do desenvolvimento de software.

Além do envolvimento do cliente, o modelo XP depende muito de testes unitários e de aceitação. Em geral, os desenvolvedores executam testes de unidade para cada alteração de código incremental, não importa quão pequena, para garantir que a base de código ainda atenda às suas especificações. Na verdade, o teste é tão importante no XP que o processo exige que você crie primeiro a unidade (módulo) e os testes de aceitação, depois sua base de código. Essa forma de teste é chamada, apropriadamente, Extreme Testing (XT).

Noções básicas de programação extrema

Como mencionado, o XP é um processo de software que ajuda os desenvolvedores a criar códigos de alta qualidade rapidamente. Aqui, definimos "qualidade" como uma base de código que atende às especificações do projeto e às expectativas do cliente.

XP se concentra em:

- Implementação de projetos simples.
- Comunicação entre desenvolvedores e clientes.
- Testando continuamente a base de código.
- Refatoração, para acomodar mudanças de especificação.
- Buscando feedback do cliente.

O XP tende a funcionar bem para esforços de desenvolvimento de pequeno a médio porte em ambientes que têm mudanças frequentes de especificação e onde a comunicação quase instantânea é possível.

XP difere dos processos de desenvolvimento tradicionais de várias maneiras. Primeiro, evita a síndrome do projeto em grande escala em que o cliente e a equipe de programação se reúnem para projetar cada detalhe do aplicativo antes do início da codificação. Os gerentes de projeto sabem que essa abordagem tem suas desvantagens, entre as quais as especificações e os requisitos do cliente mudam constantemente para refletir novas regras de negócios ou condições de mercado.

Por exemplo, o departamento financeiro pode querer que os relatórios de folha de pagamento sejam classificados por data de processamento em vez de números de cheque; ou o departamento de marketing pode determinar que os consumidores não comprarão o produto XYZ se ele não enviar um e-mail após o registro no site. Em contraste, as sessões de planejamento XP

concentre-se na coleta de requisitos gerais de aplicativos, não se limitando a todos os detalhes.

Outra diferença com a metodologia XP é que ela evita codificar funcionalidades desnecessárias. Se o seu cliente achar que o recurso é necessário, mas não obrigatório, ele geralmente é deixado de fora da versão. Assim, você pode se concentrar na tarefa em mãos, agregando valor a um produto de software. Concentrar-se apenas na funcionalidade necessária ajuda a produzir software de qualidade em curtos períodos de tempo.

Mas a principal diferença do XP em comparação com as metodologias tradicionais é sua abordagem de teste. Após uma fase de design com tudo incluído, os modelos tradicionais de desenvolvimento de software sugerem que você codifique primeiro e crie interfaces de teste posteriormente. No XP, você deve primeiro criar os testes de unidade e depois escrever o código para passar nos testes. Você projeta testes de unidade em um ambiente XP seguindo os conceitos discutidos no Capítulo 5.

O modelo de desenvolvimento XP tem 12 práticas principais que orientam o processo, resumidas na Tabela 9.2. Em poucas palavras, você pode agrupar as 12 práticas principais do XP em quatro conceitos:

1. Ouvindo o cliente e outros programadores.
2. Colaborar com o cliente para desenvolver a especificação da aplicação e os casos de teste.
3. Codificação com um parceiro de programação.
4. Testando e retestando a base de código.

A maioria dos comentários para cada prática listada na Tabela 9.2 são autoexplicativos. No entanto, alguns dos princípios mais importantes, ou seja, planejamento e teste, merecem uma discussão mais aprofundada.

Planejamento XP Uma fase de planejamento bem-sucedida estabelece a base do processo XP. A fase de planejamento no XP difere daquela nos modelos de desenvolvimento tradicionais, que geralmente combinam a coleta de requisitos e o projeto de aplicativos. O planejamento no XP concentra-se na identificação dos requisitos do aplicativo de seu cliente e na criação de histórias de usuários (ou histórias de caso) que os atendam. Você obtém insights significativos sobre a finalidade e os requisitos do aplicativo criando histórias de usuários. Além disso, o cliente emprega as histórias do usuário ao realizar testes de aceitação no final de um ciclo de lançamento. Finalmente, um benefício intangível da fase de planejamento é que o cliente ganha propriedade e confiança no aplicativo participando intimamente dele.

182 A Arte do Teste de Software

TABELA 9.2 As 12 práticas de programação extrema

Prática	Comente
1. Planejamento e requisitos	O pessoal de marketing e desenvolvimento de negócios trabalha em conjunto para identificar o valor máximo de negócios de cada recurso de software. Cada recurso principal do software é escrito como uma história de usuário. Os programadores fornecem estimativas de tempo para completar cada história de usuário. O cliente escolhe os recursos do software com base em estimativas de tempo e valor comercial.
2. Lançamentos pequenos e incrementais	Esforce-se para adicionar recursos pequenos, tangíveis e de valor agregado e lance uma nova base de código com frequência.
3. Metáforas do sistema	Sua equipe de programação identifica uma metáfora de organização para ajudar nas convenções de nomenclatura e no fluxo do programa.
4. Projetos simples	Implemente o design mais simples que permita que seu código passe em seus testes de unidade. Assuma que a mudança virá, então não gaste muito tempo projetando; basta implementar.
5. Testes contínuos	Escreva testes de unidade antes de escrever o módulo de código. Cada unidade não está completa até que passe em seu teste de unidade. Além disso, o programa não está completo até que passe em todos os testes de unidade e os testes de aceitação estejam completos.
6. Refatoração	Limpe e simplifique sua base de código. Os testes de unidade ajudam a garantir que você não destrua a funcionalidade no processo. Você deve executar novamente todos os testes de unidade após qualquer refatoração.
7. Programação em pares	Você e outro programador trabalham juntos, na mesma máquina, para criar a base de código. Isso permite a revisão de código em tempo real, o que facilita drasticamente a detecção e resolução de bugs.
8. Coletivo propriedade do código	Todo o código é de propriedade de todos os programadores. Nenhum programador é dedicado a uma base de código específica.
9. Contínuo integração 10. Semana de trabalho de quarenta horas	Todos os dias, integre todas as mudanças; depois que o código passar nos testes de unidade, adicione-o novamente à base de código. Nenhuma hora extra é permitida. Se você trabalha com dedicação por 40 horas semanais, as horas extras não serão necessárias. A exceção é a semana antes de um grande lançamento.

Tabela 9.2 (continuação)

Prática	Comente
11. Cliente no local presença	Você e sua equipe de programação têm acesso ilimitado ao cliente, para permitir que você resolva questões de forma rápida e decisiva, o que evita que o processo de desenvolvimento fique paralizado.
12. Padrões de codificação	Todo o código deve ter a mesma aparência. Desenvolver uma metáfora de sistema ajuda a atender a esse princípio.

Testes XP Testes contínuos são fundamentais para o sucesso de um esforço baseado em XP. Embora o teste de aceitação se enquadre nesse princípio, o teste de unidade ocupa a maior parte do esforço. Os testes de unidade são planejados para fazer o software falhar. Somente garantindo que seus testes detectem erros, você pode começar a corrigir o código para que ele passe nos testes. Garantir que seus testes de unidade detectem falhas é fundamental para o processo de teste — e para a confiança de um desenvolvedor. Neste ponto, o desenvolvedor pode experimentar diferentes implementações, sabendo que os testes de unidade detectarão quaisquer erros.

Você deseja garantir que qualquer alteração de código melhore o aplicativo e não introduza bugs. O princípio de teste contínuo também suporta esforços de refatoração usados para otimizar e simplificar a base de código. Testes constantes também levam a esse benefício intangível já mencionado: confiança.

A equipe de programação ganha confiança na base de código porque você a valida constantemente com testes de unidade. Além disso, a confiança de seus clientes em seus investimentos aumenta porque eles sabem que a base de código passa nos testes de unidade todos os dias.

Exemplo de Fluxo de Projeto XP Agora que apresentamos as 12 práticas do processo XP, você pode estar se perguntando, como um projeto XP típico flui? Aqui está um exemplo rápido do que você pode experimentar se trabalhou em um projeto baseado em XP:

1. Os programadores se reúnem com o cliente para determinar os requisitos do produto e construir histórias de usuários.
2. Os programadores se reúnem sem o cliente para dividir os requisitos em tarefas independentes e estimar o tempo para concluir cada tarefa.

3. Os programadores apresentam ao cliente a lista de tarefas e as estimativas de tempo e pedem que ele gere uma lista prioritária de recursos.
4. A equipe de programação atribui tarefas a pares de programadores, com base em seus conjuntos de habilidades.
5. Cada par cria testes de unidade para sua tarefa de programação usando o especificação do aplicativo.
6. Cada par trabalha em sua tarefa com o objetivo de criar uma base de código que passa nos testes de unidade.
7. Cada par corrige e testa novamente seu código até que todos os testes de unidade sejam aprovados.
8. Todos os pares se reúnem todos os dias para integrar suas bases de código.
9. A equipe lança uma versão de pré-produção do aplicativo.
10. Os clientes executam testes de aceitação e aprovam o aplicativo ou produzem um relatório identificando os bugs/deficiências.
11. Após testes de aceitação bem-sucedidos, os programadores lançam uma versão em Produção.
12. Os programadores atualizam as estimativas de tempo com base na experiência mais recente.

Embora atraente, o XP não é para todos os projetos ou organizações.

Os proponentes do XP concluem que, se uma equipe de programação implementa totalmente as 12 práticas, as chances de desenvolvimento de aplicativos bem-sucedidos aumentam drasticamente. Os detratores dizem que, como XP é um processo, você deve fazer tudo ou nada; se você pular uma prática, então você não está implementando o XP corretamente e a qualidade do seu programa pode ser prejudicada. Os detratores também afirmam que o custo de alterar um programa no futuro para adicionar mais recursos é maior do que o custo de antecipar e codificar inicialmente o requisito. Finalmente, alguns programadores acham que trabalhar em pares é muito complicado e invasivo; portanto, eles não adotam a filosofia XP.

Quaisquer que sejam suas opiniões, recomendamos que você considere o XP como uma metodologia de software para seu projeto. Pese cuidadosamente seus prós e contras em relação aos atributos do seu projeto e tome a melhor decisão com base nessa avaliação.

Testes extremos: os conceitos

Para acompanhar o ritmo e a filosofia do XP, os desenvolvedores usam o Extreme Testing, que se concentra em testes constantes. Como mencionado anteriormente, duas formas de teste compõem a maior parte do XT: teste de unidade e teste de aceitação. A teoria usada na redação dos testes não difere significativamente da teoria apresentada no Capítulo 5; no entanto, o estágio no processo de desenvolvimento em

que você cria os testes é diferente. O XT exige a criação de testes antes do início da codificação, não depois. No entanto, XT e testes tradicionais compartilham o mesmo objetivo: identificar erros em um programa.

No restante desta seção, fornecemos mais informações sobre unidade e ac testes de aceitação, de uma perspectiva de programação extrema.

Extreme Unit Testing Teste unitário, a principal abordagem de teste usada no Extreme Testing, e tem duas regras simples: Todos os módulos de código devem ter testes de unidade antes do início da codificação e todos os módulos de código devem passar por testes de unidade antes de serem liberados para testes de aceitação. À primeira vista, isso pode não parecer tão extremo. Uma inspeção mais detalhada revela a grande diferença entre o teste unitário, conforme descrito anteriormente, e o teste XTunit: Os testes unitários devem ser definidos e criados antes da codificação do módulo.

Inicialmente, você pode se perguntar por que deveria ou como pode criar drivers de teste para código que ainda não escreveu. Você também pode achar que não tem tempo para criar os testes e ainda cumprir o prazo do projeto. Essas são preocupações válidas, mas podemos resolver facilmente listando vários benefícios importantes associados à escrita de testes de unidade antes de começar a codificar o aplicativo:

Você ganha confiança de que seu código atenderá às especificações e requisitos.

Você expressa o resultado final antes de começar a codificar.

Você entende melhor a especificação e os requisitos do aplicativo.

Você pode implementar projetos simples inicialmente e refatorar o código com confiança posteriormente para melhorar o desempenho, sem se preocupar em quebrar a especificação.

Desses benefícios, o insight e a compreensão que você obtém da especificação e dos requisitos do aplicativo não podem ser subestimados. Por exemplo, se você começar a codificar primeiro, talvez não entenda completamente os tipos de dados e limites aceitáveis para os valores de entrada de um aplicativo. Como você pode escrever um teste de unidade para realizar a análise de limites sem entender as entradas aceitáveis? O aplicativo pode aceitar apenas números, apenas caracteres ou ambos? Se você criar os testes de unidade primeiro, deverá entender a especificação. A prática de criar testes de unidade primeiro é a estrela brilhante na metodologia XP, pois força você a entender a especificação para resolver ambiguidades antes de começar a codificar.

Conforme mencionado no Capítulo 5, você determina o escopo da unidade. Dado que as linguagens de programação populares de hoje, como Java, C# e Visual Basic, são principalmente orientadas a objetos, os módulos geralmente são classes ou até mesmo métodos de classe individuais. Às vezes, você pode definir um módulo como um grupo de classes ou métodos que representam alguma funcionalidade. Somente você, como programador, conhece a arquitetura do aplicativo e a melhor forma de construir os testes unitários para ele.

A execução manual de testes de unidade, mesmo para o menor aplicativo, pode ser uma tarefa assustadora. À medida que o aplicativo cresce, você pode gerar centenas ou milhares de testes de unidade. Portanto, você normalmente usa um conjunto de testes de software automatizado para aliviar a carga de executar esses testes de unidade. Com esses conjuntos, você cria o script dos testes e, em seguida, executa todos ou parte deles. Além disso, os conjuntos de testes normalmente permitem gerar relatórios e classificar os bugs que ocorrem com frequência em seu aplicativo. Essas informações podem ajudá-lo a eliminar bugs de forma proativa no futuro.

Curiosamente, uma vez que você cria e valida seus testes de unidade, a base de código de "teste" se torna tão valiosa quanto o aplicativo de software que você está tentando criar. Como resultado, você deve manter os testes em um repositório de código, para proteção. Da mesma forma, você deve instituir backups adequados do código de teste e garantir que a segurança necessária esteja em vigor.

Teste Extremo de Aceitação O teste de aceitação representa o segundo e igualmente importante tipo de XT que ocorre na metodologia XP. O teste de aceitação determina se o aplicativo atende a outros requisitos, como funcionalidade e usabilidade. Você e o cliente criam os testes de aceitação durante as fases de projeto/planejamento.

Ao contrário das outras formas de teste discutidas até agora, os clientes, não você ou seus parceiros de programação, realizam os testes de aceitação. Dessa forma, os clientes fornecem a verificação imparcial de que o aplicativo atende às suas necessidades. Os clientes criam os testes de aceitação a partir de histórias de usuários. A proporção de histórias de usuários para testes de aceitação geralmente é demais; ou seja, mais de um teste de aceitação pode ser necessário para cada história de usuário.

Os testes de aceitação em XT podem ou não ser automatizados. Por exemplo, um teste não automatizado é necessário quando o cliente precisa validar se uma tela de entrada do usuário atende às especificações em relação à cor e ao layout da tela. Um exemplo de teste automatizado é quando o aplicativo deve calcular os valores da folha de pagamento usando a entrada de dados por meio de alguma fonte de dados, como um arquivo simples, para simular os valores de produção.

Por meio de testes de aceitação, o cliente valida um resultado esperado da aplicação. Um desvio do resultado esperado é considerado um bug e é reportado à equipe de desenvolvimento. Se o cliente descobrir vários bugs, ele deverá priorizá-los antes de passar a lista para seu grupo de desenvolvimento. Após corrigir os bugs, ou após qualquer alteração, o cliente reexecuta os testes de aceitação. Dessa forma, os testes de aceitação também se tornam uma forma de teste de regressão.

Uma observação importante é que um programa pode passar em todos os testes de unidade, mas falhar nos testes de aceitação. Como isso é possível? Porque um teste de unidade valida se uma unidade de programa atende a alguma especificação, como o cálculo de deduções da folha de pagamento, corretamente, não a alguma funcionalidade ou estética definida. Para uma aplicação comercial, a aparência é um componente muito importante. Compreender a especificação, mas não a funcionalidade, geralmente resulta neste cenário.

Teste Extremo Aplicado

Nesta seção, criamos um pequeno aplicativo Java e empregamos o JUnit, um conjunto de testes unitários de código aberto baseado em Java, para ilustrar os conceitos de Extreme Testing (veja a Figura 9.2). O exemplo em si é trivial; os conceitos, no entanto, se aplicam à maioria das situações de programação.

Nosso exemplo é um aplicativo de linha de comando que simplesmente determina se um valor de entrada é um número primo. Para resumir, o código-fonte,

JUnit é uma ferramenta de código aberto disponível gratuitamente usada para automatizar testes de unidade de aplicativos Java em ambientes Extreme Programming. Os criadores, Kent Beck e Erich Gamma, desenvolveram o JUnit para dar suporte aos testes unitários significativos que ocorrem no ambiente Extreme Programming. JUnit é muito pequeno, mas muito flexível e rico em recursos. Você pode criar testes individuais ou um conjunto de testes. Você pode gerar automaticamente relatórios detalhando os erros.

Antes de usar o JUnit, ou qualquer suíte de testes, você deve entender completamente como usá-lo. JUnit é poderoso, mas somente depois de dominar sua API. No entanto, independentemente de você adotar ou não uma metodologia XP, o JUnit é uma ferramenta útil para fornecer verificações de integridade para seu próprio código.

Acesse www.junit.org para obter mais informações e fazer o download do conjunto de testes. Além disso, há uma riqueza de informações sobre XP e XT neste site.

FIGURA 9.2 Descrição e Background da JUnit.

check4Prime.java e seu equipamento de teste, check4PrimeTest.java, estão listados no Apêndice. Nesta seção, fornecemos trechos do aplicativo para ilustrar os pontos principais.

A especificação deste programa é a seguinte:

Desenvolva um aplicativo de linha de comando que aceite qualquer inteiro positivo, n, onde $0 < n < 1.000$, e determine se é um número primo. Se n for um número primo, o aplicativo deverá retornar uma mensagem informando que é um número primo. Se n não for um número primo, então o aplicativo deve retornar uma mensagem informando que não é um número primo. Se n não for uma entrada válida, o aplicativo deverá exibir uma mensagem de ajuda.

Seguindo a metodologia XP e os princípios listados no Capítulo 5, iniciamos a aplicação projetando testes unitários. Com esta aplicação, podemos identificar duas tarefas discretas: validar entradas e determinar números primos. Poderíamos usar abordagens de teste de caixa preta e caixa branca, análise de valor de limite e critério de cobertura de decisão, respectivamente.

No entanto, a prática do XT exige uma abordagem de caixa preta sem intervenção, para eliminar qualquer viés.

Projeto de casos de teste Começamos a projetar casos de teste identificando uma abordagem de teste. Nesse caso, usaremos a análise de limites para validar as entradas, pois esse aplicativo só pode aceitar números inteiros positivos dentro de um determinado intervalo. Todos os outros valores de entrada, incluindo tipos de dados de caracteres e números negativos, devem gerar um erro e não devem ser usados. É claro que você certamente poderia argumentar que a validação de entrada pode se enquadrar no critério de cobertura de decisão, pois o aplicativo deve decidir se a entrada é válida. O conceito importante é identificar e se comprometer com uma abordagem de teste ao projetar seus testes.

Com a abordagem de teste identificada, o próximo passo é desenvolver uma lista de casos de teste com base em possíveis entradas e resultados esperados. A Tabela 9.3 mostra os oito casos de teste que identificamos para este exemplo. (Observação: como dito, estamos usando um exemplo muito simples aqui para ilustrar os fundamentos do Extreme Testing. Na prática, você teria uma especificação de programa muito mais detalhada, que pode incluir itens como requisitos de interface do usuário e palavreado de saída. Como resultado, a lista de casos de teste aumentaria substancialmente.)

TABELA 9.3 Descrições do Caso de Teste para check4Prime.java

Caso		Esperado	
Entrada de número		Resultado	Comentários
1	$n \neq 3$	Afirme que n é um número primo.	Testes para um número primo válido. Testa a entrada dentro dos limites.
2	$n = 1.000$	Afirme que n não é um número primo.	Testa a entrada igual aos limites superiores. Testa se n é inválido melhor.
3	$n = 0$	Afirme que n não é um número primo.	Testa a entrada igual aos limites inferiores.
4	$n = -1$	Imprimir ajuda mensagem.	Testa a entrada abaixo dos limites inferiores.
5	$n = 1.001$	Imprimir ajuda mensagem.	Testa entrada maior que a superior limites.
6	$n = "a"$	Imprimir ajuda mensagem.	A entrada de testes é um número inteiro e não um tipo de dados de caractere.
7	Dois ou mais entradas	Imprimir ajuda mensagem.	Testes para o número correto de entrada valores.
8	n está vazio (em branco)	Imprimir ajuda mensagem.	Testa se um valor de entrada é fornecido.

O caso de teste 1 da Tabela 9.3 combina dois cenários de teste. Ele verifica se a entrada é um primo válido e como o aplicativo se comporta com um valor de entrada válido. Você pode usar qualquer primo válido neste teste.

Também testamos dois cenários com o caso de teste 2: O que acontece quando o valor de entrada é igual aos limites superiores e quando a entrada não é um primo número? Este caso poderia ter sido dividido em dois testes de unidade, mas um. O objetivo do teste de software em geral é minimizar o número de casos de teste enquanto ainda verifica adequadamente as condições de erro.

O caso de teste 3 verifica o limite inferior de entradas válidas, bem como o teste para primos inválidos. A segunda parte da verificação não é necessária porque o teste o caso 2 trata desse cenário. No entanto, ele é incluído por padrão porque 0 é não um número primo. Os casos de teste 4 e 5 garantem que as entradas estejam dentro o intervalo definido, que é maior ou igual a 0 e menor ou igual para 1.000.

O caso 6 testa se o aplicativo lida adequadamente com a entrada de caracteres valores. Como estamos fazendo um cálculo, é óbvio que o

TABELA 9.4 Métodos do driver de teste

Métodos	Casos de teste) Examinado
testCheckPrime_true()	1
testCheckPrime_false()	2, 3
testCheck4Prime_checkArgs_char_input()	6
testCheck4Prime_checkArgs_above_upper_bound()	5
testCheck4Prime_checkArgs_neg_input()	4
testCheck4Prime_checkArgs_2_inputs()	7
testCheck4Prime_checkArgs_0_inputs()	8

o aplicativo deve rejeitar tipos de dados de caracteres. A suposição com este caso de teste é que o Java manipulará a verificação do tipo de dados. Este aplicativo deve tratar a exceção gerada quando um tipo de dados inválido é fornecido.

Esse teste garantirá que a exceção seja lançada. Por último, os testes 7 e 8 verificam o número correto de valores de entrada; qualquer número de entradas diferente de 1 deve falhar.

Driver de teste e aplicativo Agora que projetamos os dois casos de teste, podemos criar a classe de driver de teste, check4PrimeTest. A Tabela 9.4 mapeia os métodos JUnit em check4PrimeTest para os casos de teste cobertos.

Observe que o método testCheckPrime_false() testa duas condições, porque os valores de limite não são números primos. Portanto, podemos verificar erros de valor de limite e primos inválidos com um método de teste. Examinar esse método em detalhes revela que os dois testes realmente ocorrem dentro dele. Aqui está o método JUnit completo da classe check4JavaTest listada no Apêndice.

```
public void testCheckPrime_false(){
    assertFalse(check4prime.primeCheck(0));
    assertFalse(check4prime.primeCheck(10000));
}
```

Observe que o método JUnit, assertFalse(), verifica se o parâmetro fornecido faz com que o método retorne um valor booleano falso.

Se false for retornado, o teste é considerado um sucesso.

O snippet também demonstra um dos benefícios de criar casos de teste e equipamentos de teste primeiro. Você pode notar que o parâmetro no

`assertFalse()` é outro método, `check4prime`. `PrimeCheck(n)`. Esse método residirá em uma classe do aplicativo.

A criação do equipamento de teste primeiro nos forçou a pensar na estrutura do aplicativo. Em alguns aspectos, o aplicativo foi projetado para oferecer suporte ao equipamento de teste. Aqui precisamos de um método para verificar se a entrada é um número primo, então o incluímos no aplicativo.

Com o equipamento de teste completo, a codificação do aplicativo pode começar. Com base na especificação do programa, nos casos de teste e no equipamento de teste, o aplicativo Java resultante consistirá em uma única classe, `check4Prime`, com a seguinte definição:

```
classe pública check4Prime {  
    public static void main (String [] args) public void  
    checkArgs(String [] args) throws  
        Exceção  
    pública booleana primeCheck (int num)  
}
```

Resumidamente, de acordo com os requisitos do Java, o procedimento `main()` fornece o ponto de entrada no aplicativo. O método `checkArgs()` afirma que o valor de entrada é um inteiro positivo, `n`, onde $0 < n < 1.000$. O procedimento `primeCheck()` verifica o valor de entrada em relação a uma lista calculada de números primos. Implementamos a peneira de Eratóstenes para calcular rapidamente os números primos. Essa abordagem é aceitável devido ao pequeno número de números primos envolvidos.

Resumo

Com o aumento da competitividade do desenvolvimento de software hoje, há uma necessidade crescente de introduzir produtos muito rapidamente no mercado.

O processo de desenvolvimento ágil, quando estritamente adotado, fornece uma maneira para os desenvolvedores criarem software de qualidade para seus clientes em um ritmo mais rápido do que usando modelos tradicionais de desenvolvimento de software. O resultado final é um cliente satisfeito, seja consumidor interno ou comercial.

O modelo Extreme Programming é uma das metodologias Agile mais populares. Esse processo de desenvolvimento leve se concentra na comunicação, planejamento e teste. O aspecto de teste da Extreme Programming, denominado Extreme Testing, concentra-se em testes unitários e de aceitação. Você executa testes de unidade durante o desenvolvimento e sempre que ocorrer uma alteração na base de código. O cliente executa os testes de aceitação nos principais pontos de lançamento.

192 A Arte do Teste de Software

O Extreme Testing também exige que você crie o equipamento de teste, com base na especificação do programa, antes de começar a codificar seu aplicativo. Dessa forma, você projeta seu aplicativo para passar nos testes de unidade, aumentando assim a probabilidade de ele atender à especificação.

10

Testando a Internet Formulários

Apenas alguns anos atrás, os aplicativos baseados na Internet pareciam ser a onda do futuro; hoje, a onda chegou em terra e clientes, funcionários e parceiros de negócios esperam que as empresas tenham uma presença na Web. Essa expectativa não se limita apenas aos negócios. A maioria das igrejas, grupos cívicos, escolas e governos têm presenças na Internet para servir seus clientes.

Geralmente, as pequenas e médias empresas têm páginas da Web simples que usam para divulgar seus produtos e serviços. Empresas maiores geralmente criam aplicativos de comércio eletrônico completos para vender seus produtos, de cookies a carros e de serviços de consultoria a empresas virtuais inteiras que existem apenas na Internet.

Os aplicativos da Internet são essencialmente aplicativos cliente-servidor nos quais o cliente é um navegador da Web e o servidor é um servidor da Web ou de aplicativos. Embora conceitualmente simples, a complexidade desses aplicativos varia muito. Algumas empresas têm aplicativos criados para uso entre empresas, como serviços bancários e lojas de varejo, enquanto outras têm aplicativos empresa a empresa, como cadeia de suprimentos ou gerenciamento de força de vendas. As estratégias de desenvolvimento e apresentação do usuário/interface do usuário variam para esses diferentes tipos de sites e, como você pode imaginar, a abordagem de teste também varia.

O objetivo de testar aplicativos baseados na Internet não é diferente daquele dos aplicativos tradicionais. Você precisa descobrir erros no aplicativo antes de implantá-lo na Internet e no usuário final. E, dado o

complexidade desses aplicativos e a interdependência dos componentes, você provavelmente conseguirá encontrar muitos erros.

A importância de erradicar os erros em um aplicativo da Internet não pode ser exagerada. Como resultado da abertura e acessibilidade de na Internet, a concorrência na arena business-to-consumer e business-to-business é intensa. Assim, a Internet criou um comprador mercado de bens e serviços. Os consumidores desenvolveram alta expectativas, e se o seu site não carregar rapidamente, responder imediatamente e fornecer recursos de navegação intuitivos, é provável que o usuário encontrará outra empresa com a qual realizar negócios. Este questão não se limita estritamente ao comércio eletrônico ou à promoção de produtos locais. Sites que são desenvolvidos como recursos de pesquisa ou informação freqüentemente são mantidos por publicidade ou doações de usuários. De qualquer jeito, existe ampla concorrência para atrair usuários, reduzindo assim a atividade e receitas concomitantes.

Parece que os consumidores têm expectativas de maior qualidade para aplicações de Internet do que para aqueles que vêm embrulhados. Quando as pessoas compram software em caixa de uma loja ou varejista on-line, contanto que como a qualidade é "média", eles continuarão a usá-los. Uma razão para esse comportamento é que eles pagaram pelo aplicativo, então deve ser um produto percebido como útil ou desejável. E mesmo um programa menos do que satisfatório não pode ser corrigido facilmente, então se pelo menos satisfizer o necessidades básicas dos usuários, eles provavelmente manterão o programa. Em contrapartida, um pobre, ou até mesmo um aplicativo de qualidade média na Internet, provavelmente fará com que seu cliente mudar para o site de um concorrente. Não só o cliente sairá se seu site apresentar baixa qualidade, sua imagem corporativa também ficará manchada. Afinal, quem se sente à vontade para comprar um carro de uma empresa que não consegue construir um site adequado? Goste ou não, os sites têm tornar-se a nova primeira impressão para os negócios. Em geral, os consumidores não pagar para acessar a maioria dos sites, então há pouco incentivo para permanecer fiel a face do design ou desempenho mediocre.

Este capítulo aborda alguns dos fundamentos do teste de aplicativos da Internet. Este assunto é amplo e complexo, e existem muitas referências que exploram seus detalhes. No entanto, você descobrirá que as técnicas explicadas no os primeiros capítulos também se aplicam aos testes da Internet. Mesmo assim, porque há são, de fato, diferenças funcionais e de design entre aplicativos da Web e convencionais, queremos destacar algumas das particularidades do teste de aplicativos baseados na Web.

Arquitetura básica de comércio eletrônico

Antes de mergulhar nos testes de aplicativos baseados na Internet, forneceremos uma visão geral da arquitetura cliente-servidor (C/S) de três camadas usada em um aplicativo típico de comércio eletrônico baseado na Internet. Conceitualmente, cada camada é tratada como uma caixa preta com interfaces bem definidas. Este modelo permite que você altere as partes internas de cada camada sem se preocupar em quebrar outra camada. A Figura 10.1 ilustra cada camada e os componentes associados usados pela maioria dos sites de comércio eletrônico.

Embora não seja uma camada oficial na arquitetura, vale a pena explicar o lado do cliente e sua relevância. A maior parte do acesso aos seus aplicativos ocorre a partir de um navegador da Web executado em um computador, embora muitos dispositivos, como telefones celulares, PDAs, consoles de jogos, tocadores de música, pagers e até geladeiras e automóveis, estejam cada vez mais sendo desenvolvidos com conectividade à Internet em mente. Os navegadores variam drasticamente em como renderizam o conteúdo de um site. Como discutiremos mais adiante neste capítulo, testar a compatibilidade do navegador é um desafio associado ao teste de aplicativos da Internet. Os fornecedores seguem livremente os padrões publicados para ajudar a fazer com que os navegadores se comportem de forma consistente, mas também incorporaram melhorias proprietárias que causam um comportamento inconsistente. O restante dos clientes emprega aplicativos personalizados que usam a Internet como um pipeline para um determinado site. Nesse cenário, o aplicativo imita um aplicativo cliente-servidor padrão que você pode encontrar na rede local de uma empresa.

O servidor Web representa a primeira camada na arquitetura de três camadas e abriga o site. A aparência de um aplicativo de Internet

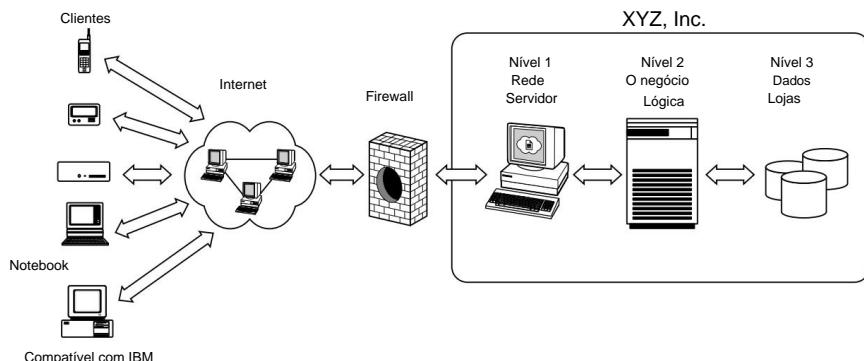


FIGURA 10.1 Arquitetura típica de um site de comércio eletrônico.

vem do primeiro nível. Assim, outro termo para essa camada é a camada ou camada de apresentação, assim apelidada porque fornece o conteúdo visual ao usuário final. O servidor Web pode usar páginas HTML (HyperText Markup Language) estáticas ou scripts CGI (Common Gateway Interface) para criar HTML dinâmico, mas provavelmente usa uma combinação de páginas estáticas e dinâmicas.

A camada 2, ou camada de negócios, abriga o servidor de aplicativos. Aqui, você executa o software que modela seus processos de negócios. A seguir, listamos algumas das funcionalidades associadas à camada de negócios:

Processamento de transações

Autenticação de usuário

Data de validade

Registro de aplicativos

A terceira camada concentra-se em armazenar e recuperar dados de uma fonte de dados, normalmente um sistema de gerenciamento de banco de dados relacional (RDBMS). Outro termo para a camada 3 é a camada de dados. Esta camada consiste em uma infra-estrutura de banco de dados para se comunicar com a segunda camada. A interface na camada de dados é definida pelo modelo de dados, que descreve como você deseja armazenar os dados. Às vezes, vários servidores de banco de dados compõem essa camada. Você normalmente ajusta os sistemas de banco de dados nessa camada para lidar com as altas taxas de transação encontradas em um site de comércio eletrônico. Além de um servidor de banco de dados, alguns sites de comércio eletrônico podem colocar um servidor de autenticação nessa camada. Na maioria das vezes, você usa um servidor LDAP (Lightweight Directory Application Protocol) para essa função.

Desafios de teste

Você enfrentará muitos desafios ao projetar e testar aplicativos baseados na Internet devido ao grande número de elementos que você não pode controlar e ao número de componentes interdependentes. Testar adequadamente seu aplicativo requer que você faça algumas suposições sobre seus clientes e como eles usam o site.

Um aplicativo baseado na Internet tem muitos pontos de falha que você deve considerar ao projetar uma abordagem de teste. A lista a seguir fornece alguns exemplos dos desafios associados ao teste de aplicativos baseados na Internet:

Base de usuários grande e variada. Os usuários do seu site possuem diferentes conjuntos de habilidades, empregam uma variedade de navegadores e usam diferentes sistemas operacionais ou dispositivos. Você também pode esperar que seus clientes acessem seu site usando uma ampla variedade de velocidades de conexão. Há dez anos, nem todos tinham acesso à Internet de banda larga. Hoje, a maioria faz.

No entanto, você ainda precisa considerar a largura de banda à medida que o conteúdo da Internet se torna "mais rico" e mais interativo.

Ambiente de negócios. Se você opera um site de comércio eletrônico, deve considerar questões como calcular impostos, determinar custos de envio, concluir transações financeiras e rastrear arquivos de perfis de clientes. Eses requisitos podem exigir vários links externos para servidores ou bancos de dados de terceiros para gerenciar essas tarefas de cobrança e envio de ping, por exemplo. O desenvolvedor deve entender completamente a estrutura do sistema remoto e trabalhar em estreita colaboração com seus proprietários e desenvolvedores para garantir segurança e precisão.

Locais. Os usuários podem residir em outros países e, nesse caso, você terá problemas de internacionalização, como tradução de idiomas, diferenças de fuso horário e conversão de moeda.

Segurança. Como seu site está aberto ao mundo, você deve protegê-lo contra hackers. Eles podem paralisar seu site com ataques de negação de serviço (DoS) ou roubar as informações do cartão de crédito de seus clientes.

Ambientes de teste. Para testar adequadamente seu aplicativo, você precisará duplicar o ambiente de produção. Isso significa que você deve usar servidores Web, servidores de aplicativos e servidores de banco de dados idênticos ao equipamento de produção. Para obter resultados de teste mais precisos, a infraestrutura de rede também deverá ser duplicada, o que inclui roteadores, switches e firewalls.

Mesmo nessa lista, que pode ser expandida consideravelmente com a inclusão de pontos de vista de uma ampla variedade de desenvolvedores e empresas, você pode ver que configurar um ambiente de teste é um dos aspectos mais desafiadores do desenvolvimento de e-commerce. Testar aplicativos que processam transações financeiras requer mais esforço e despesas. Você deve replicar todos os componentes, tanto de hardware quanto de software, usados para o aplicativo para produzir resultados de teste válidos. Configurar tal ambiente é um esforço caro. Você incorrerá não apenas em custos de equipamentos, mas também em custos de mão de obra. A maioria das empresas não leva em conta essas despesas ao criar um

orçamento para suas aplicações, e aqueles que geralmente subestimam os requisitos de tempo e dinheiro. Além disso, o ambiente de teste precisa de um plano de manutenção para dar suporte aos esforços de atualização do aplicativo.

Outro desafio de teste significativo que você enfrenta é testar a compatibilidade do navegador. Existem vários navegadores diferentes no mercado hoje, e cada um se comporta de maneira diferente. Embora existam padrões para a operação do navegador, a maioria dos fornecedores aprimora seus produtos em um esforço para atrair uma base de usuários leais. Infelizmente, isso faz com que os navegadores operem de maneira não padronizada. Abordaremos esse tópico com mais detalhes posteriormente neste capítulo.

Conforme observado, você enfrentará muitos desafios ao testar aplicativos baseados na Internet; portanto, a melhor maneira de proceder é restringir seus esforços de teste a áreas específicas. A Tabela 10.1 identifica algumas das áreas mais importantes a serem testadas, para ajudar a garantir que os usuários tenham uma experiência positiva em seu site.

TABELA 10.1 Exemplos de apresentação, negócios e teste de camada de dados

Nível de apresentação	Nível de negócios	Camada de dados
Certifique-se de que as fontes sejam as mesmas em todos os navegadores.	Verifique o cálculo adequado do imposto sobre vendas e das despesas de envio.	Garanta que as operações do banco de dados atendam às metas de desempenho.
Confirme se todos os links apontam para arquivos ou sites válidos.	Assegurar documentado as taxas de desempenho são atendidas	Verifique se os dados estão armazenados corretamente e com precisão.
Verifique se os gráficos estão na resolução correta e tamanho.	para tempos de resposta e taxas de transferência.	Verifique se você pode recuperar usando os backups atuais.
Verifique a ortografia de cada página.	Verifique se as transações foram concluídas corretamente.	Failover de teste ou operações de redundância.
Peça a um editor de texto que verifique a gramática e o estilo.	Confirme que falhou as transações são revertidas corretamente.	Teste para criptografia e segurança de dados adequadas (cartão de crédito e informações pessoais do usuário, em particular).
Verifique o posicionamento do cursor quando a página for carregada para garantir que esteja na caixa de texto correta.	Certifique-se de que os dados sejam coletados corretamente.	

Tabela 10.1 (continuação)

Nível de apresentação	Nível de negócios	Camada de dados
Confirme esse padrão botão é selecionado quando a página carrega.		Teste a entrada de dados de back-end e rotinas de gerenciamento para usabilidade e precisão.
Verifique se há feedback consistente e fácil de usar sobre operações interativas.		
Verifique se há termos e estilos específicos de negócios ou do setor.		

Como a primeira impressão é a mais importante, alguns de seus testes se concentrarão na usabilidade e nas preocupações com o fator humano. Esta área concentra-se na aparência do seu aplicativo. Itens como fontes, cores e gráficos desempenham um papel importante para que os usuários aceitem ou rejeitem seu aplicativo. Lembre-se de que o desenvolvedor tem pouco ou nenhum controle sobre quem acessará um determinado aplicativo, quanto conhecimento de informática ele possui, se está ou não motivado a permanecer com um aplicativo diante de problemas de navegação ou o que os usuários podem esperar. em termos de informação ou desempenho.

O desempenho do sistema influencia muito a primeira impressão do cliente. Como mencionado anteriormente, os usuários da Internet querem gratificação instantânea. Eles não vão esperar muito para que as páginas sejam carregadas ou as transações sejam concluídas. Literalmente, alguns segundos de atraso podem fazer com que um cliente tente outro site. O baixo desempenho também pode levar os clientes a duvidar da confiabilidade do seu site. Portanto, você deve definir metas de desempenho e, em seguida, projetar testes que revelem problemas que fazem com que seu site perca as metas.

Os usuários também exigem que suas transações sejam concluídas com rapidez e precisão ao comprar produtos ou serviços em seu site. Eles não toleram e não devem tolerar cobranças imprecisas ou erros de envio. Provavelmente pior do que perder um cliente é ser responsável por mais do que o valor da transação se seu aplicativo não processar as transações financeiras corretamente.

Seu aplicativo provavelmente coletará dados para concluir tarefas como compras ou registros de e-mail. Portanto, você deve garantir que os dados coletados sejam válidos. Por exemplo, certifique-se de que os números de telefone, ID

200 A Arte do Teste de Software

números, moedas, endereços de e-mail e números de cartão de crédito são os comprimento correto e estão formatados corretamente. Além disso, verifique a integridade dos seus dados. Problemas de localização podem facilmente causar corrupção de dados por meio de truncamento devido a problemas de conjunto de caracteres.

No ambiente da Internet, é fundamental manter o site disponível para uso do cliente. Isso requer que você desenvolva e implemente diretrizes de manutenção para todos os aplicativos e servidores de suporte. Uma Web servidor e RDBMS requerem um alto nível de gerenciamento. Você deve monitorar logs, recursos do sistema e backups, e responda a quaisquer anomalias imediatamente. Conforme descrito no Capítulo 6, você deseja maximizar o tempo médio entre falhas (MTBF) e minimizar o tempo médio de recuperação (MTTR) para esses sistemas.

Finalmente, a conectividade de rede é outra área em que é importante concentrar seus esforços de teste. Em algum momento, você pode contar com a perda de rede conectividade. A fonte da falha pode ser a própria Internet, seu provedor de serviços ou sua rede interna. Portanto, você precisa criar planos de contingência para seu aplicativo e infraestrutura para que seus sistemas respondam graciosamente quando ocorre uma interrupção. Mantendo o tema dos testes, projete seus testes para quebrar seus planos de contingência.

Estratégias de teste

Desenvolver uma estratégia de teste para aplicativos baseados na Internet requer uma sólida compreensão dos componentes de hardware e software que fazem o aplicativo. Como é fundamental para o teste bem-sucedido de aplicativos padrão, você precisará de um documento de especificação para descrever o funcionalidade e desempenho do seu site. Sem este documento, você não será capaz de projetar os testes apropriados.

Você precisa testar componentes desenvolvidos internamente e aqueles adquiridos de terceiros. Para os componentes desenvolvidos internamente você deve empregar as táticas apresentadas nos capítulos anteriores. Isso inclui criar testes de unidade/módulo e realizar revisões de código. Integre os componentes em seu sistema somente após verificar se eles atender às especificações de projeto e funcionalidade descritas no documento de especificação.

Se você compra componentes, precisa desenvolver uma série de testes do sistema para validar que eles executam corretamente, independentemente de sua inscrição. Não confie no programa de controle de qualidade do fornecedor para detectar

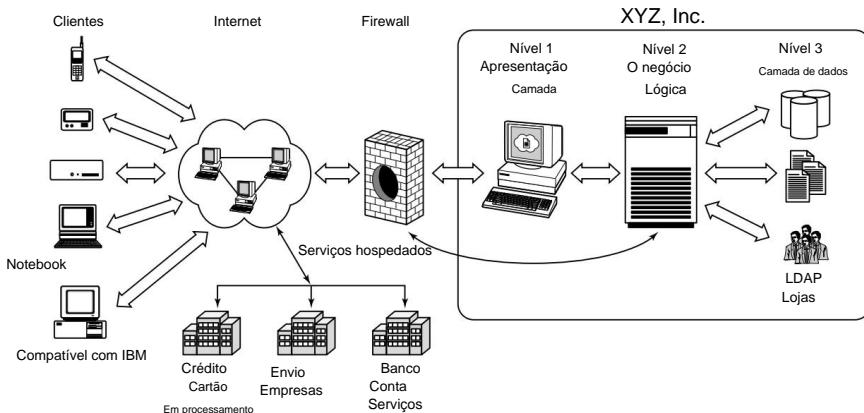


FIGURA 10.2 Visão detalhada da arquitetura de aplicativos da Internet.

erros em seus componentes. Idealmente, você deve concluir essa tarefa independentemente do teste de seu aplicativo. Integre esses componentes somente depois de determinar que eles têm um desempenho aceitável. A inclusão de um componente de terceiros não funcional em sua arquitetura dificulta a interpretação dos resultados dos testes e a identificação da origem dos erros. Geralmente, você usará abordagens de caixa preta para componentes de terceiros porque raramente terá acesso aos componentes internos.

O teste de aplicativos baseados na Internet é melhor abordado com uma abordagem de divisão e conquista. Felizmente, a arquitetura dos aplicativos da Internet permite identificar áreas discretas para testes de destino. A Figura 10.1 apresentou a arquitetura básica dos aplicativos da Internet. A Figura 10.2 fornece uma visão mais detalhada de cada camada.

Conforme mencionado anteriormente neste capítulo, os aplicativos da Internet são considerados aplicativos cliente-servidor de três camadas. Cada camada, ou camada, da Figura 10.2 é definida da seguinte forma:

Camada de apresentação. A camada de um aplicativo da Internet que fornece a interface do usuário (UI; ou GUI, interface gráfica do usuário).

Camada de negócios. A camada que modela seus processos de negócios, como autenticação de usuário e transações.

Camada de dados. A camada que abriga os dados usados pelo aplicativo ou que são coletados do usuário final.

Cada camada tem suas próprias características que incentivam a segmentação de teste. Testar cada camada de forma independente permite identificar bugs com mais facilidade

TABELA 10.2 Itens a serem testados em cada camada

Área de teste	Comentários
Usabilidade/fatores humanos	Revise a aparência geral. Fontes, cores e gráficos desempenham um papel importante na estética do aplicativo.
atuação	Certifique-se de que todas as entradas do usuário sejam reconhecidas para que fique claro para o usuário que a entrada foi aceita. Verifique se há páginas de carregamento rápido. Verifique se há transações rápidas. O mau desempenho muitas vezes cria uma má impressão.
Regras do negócio	Verifique a representação precisa do processo de negócios. Considere o ambiente de negócios para grupos de usuários de destino. Certifique-se de que as convenções de terminologia e estilo do negócio ou da indústria sejam seguidas.
Transação	Verifique se as transações são concluídas com precisão.
precisão	Confirme se as transações canceladas são revertidas corretamente. A verificação de entrada é suficientemente forte para suportar os requisitos de segurança e precisão?
Validade e integridade dos dados	Verifique se há formatos válidos de número de telefone, endereços de e-mail e valores monetários. Garanta conjuntos de caracteres adequados.
Confiabilidade do sistema	Teste os recursos de failover de seus servidores Web, de aplicativos e de banco de dados. Maximize o MTBF e minimize o MTTR.
Arquitetura de rede	Teste a redundância de conectividade. Teste o comportamento do aplicativo durante interrupções de rede.

e erros antes do início do teste completo do sistema. Se você confiar apenas no teste do sistema, poderá ter dificuldade em localizar os componentes específicos que estão criando o problema.

A Tabela 10.2 lista os itens que você deve testar em cada camada. A lista não é abrangente, mas fornece um ponto de partida para desenvolver seus próprios critérios de teste. No restante deste capítulo, fornecemos mais detalhes sobre como testar cada camada.

Teste da camada de apresentação

Testar a camada de apresentação consiste em encontrar erros na GUI, ou front-end, do seu aplicativo. Essa importante camada serve como o "recurso" do seu site, portanto, detectar e corrigir erros aqui é fundamental para apresentar um site robusto e de qualidade. Se seus clientes encontrarem erros nessa camada, eles podem não retornar. Eles podem concluir, por exemplo, que se sua empresa publicar páginas da Web com palavras incorretas, não poderá ser confiável para executar com sucesso uma transação com cartão de crédito.

Em poucas palavras, o teste da camada de apresentação é muito trabalhoso. No entanto, assim como você pode segmentar o teste de um aplicativo da Internet em entidades discretas, você pode fazer o mesmo ao testar a camada de apresentação. Aqui estão as três principais áreas de teste da camada de apresentação:

1. Teste de conteúdo. Estética geral, fontes, cores, ortografia, precisão do conteúdo, valores padrão.
2. Arquitetura do site. Links ou gráficos quebrados.
3. Ambiente do usuário. Versões do navegador da Web e configuração do sistema operacional.

O teste de conteúdo envolve a verificação do elemento de interface humana de um site. Você precisa procurar erros no tipo de fonte, layout de tela, cores, resoluções gráficas e outros recursos que afetam diretamente a experiência do usuário final. Além disso, você deve verificar a exatidão das informações em seu site. Fornecer informações gramaticalmente corretas, mas imprecisas, prejudica a credibilidade de sua empresa tanto quanto qualquer outro bug da GUI.

Informações imprecisas também podem causar problemas legais para sua empresa.

Teste a arquitetura do site tentando encontrar erros de navegação e estruturais. Procure links quebrados, páginas ausentes, arquivos errados ou qualquer coisa que envie o usuário para a área errada do site. Esses erros podem ocorrer com muita facilidade, especialmente para sites dinâmicos e durante as fases de desenvolvimento ou atualização. Tudo o que um membro da equipe do projeto precisa fazer é renomear um arquivo e seu hiperlink se tornará inválido. Da mesma forma, se um elemento gráfico for renomeado ou movido, haverá um buraco em sua página da Web porque o arquivo não pode ser encontrado. Você pode validar a arquitetura do seu site criando um teste de unidade que verifica cada página quanto a problemas de arquitetura.

Como prática recomendada, você deve migrar o teste de arquitetura para o

processo de teste de regressão também. Existem inúmeras ferramentas que podem automatizar o processo de verificação de links e de arquivos ausentes.

As técnicas de teste de caixa branca são úteis ao testar a arquitetura do site. Assim como as unidades de programa têm pontos de decisão e caminhos de execução, as páginas da Web também têm. Os usuários podem clicar em links e botões em qualquer ordem, que irão navegar para outra página. Para sites grandes, existem muitas combinações de eventos de navegação que podem ocorrer. Revise o Capítulo 4 para obter mais informações sobre testes de caixa branca e teoria de cobertura lógica.

Conforme mencionado anteriormente, testar o ambiente do usuário final – também conhecido como teste de compatibilidade do navegador – geralmente é o aspecto mais desafiador do teste de aplicativos baseados na Internet. A combinação de navegadores e um sistema operacional (SO) é muito grande. Não apenas você deve testar a configuração de cada navegador, mas também as diferentes versões do mesmo navegador. Os fornecedores geralmente melhoraram alguns recursos de seus navegadores a cada versão, que podem ou não ser compatíveis com versões mais antigas. É interessante (e frustrante) ver que, mesmo nesta era de desenvolvimento e funcionalidade avançados da Internet, você ainda pode encontrar páginas da Web que exibem uma mensagem informando que o site não é compatível com o navegador da Web que você está usando. Não deve ser responsabilidade do usuário escolher o navegador certo para acessar seu site. Para garantir uma visita de usuário bem-sucedida, gaste mais tempo no design, desenvolvimento e teste de aplicativos com uma ampla variedade de navegadores e sistemas operacionais.

O teste do ambiente do usuário se torna mais complicado quando seu aplicativo depende muito do processamento de script do lado do cliente. Cada navegador tem um mecanismo de script ou máquina virtual diferente para executar scripts e códigos no computador do cliente. Preste atenção especial aos problemas de compatibilidade do navegador se você usar qualquer um dos seguintes:

Controles ActiveX

JavaScript

VBScript

miniacativos Java

HTML 5

Adobe Flash

PHP

Você pode superar a maioria dos desafios associados ao teste de compatibilidade do navegador gerando requisitos funcionais bem definidos. Por

Por exemplo, durante a fase de coleta de requisitos, seu departamento de marketing pode decidir que o aplicativo deve ser certificado para funcionar apenas com determinados navegadores. Por um lado, esse requisito elimina uma quantidade significativa de testes porque você terá uma plataforma de destino bem definida para testar. Por outro lado, embora esta possa ser uma decisão de economia de custo e tempo, pode não ser uma decisão de negócios inteligente.

Os dias em que um único (ou mesmo alguns) aplicativos de navegador da Web dominavam a comunidade de usuários já se foram. Uma boa prática de negócios seria projetar e testar uma ampla variedade de aplicativos de navegador da Web de usuário possíveis.

Teste da camada de negócios

O teste da camada de negócios se concentra em encontrar erros na lógica de negócios do seu aplicativo de Internet. Você descobrirá que testar essa camada é muito semelhante ao de aplicativos independentes, pois pode empregar técnicas de caixa branca e preta. Você desejará criar planos e procedimentos de teste que detectem erros na especificação de desempenho do aplicativo, aquisição de dados e processamento de transações.

Você deve empregar abordagens de caixa branca para componentes desenvolvidos internamente, porque você tem acesso à lógica do programa. No entanto, para componentes de terceiros, as técnicas de teste de caixa preta devem incluir sua abordagem de teste principal. Você começará desenvolvendo drivers de teste para testar os componentes individuais. Em seguida, você pode realizar um teste de sistema para determinar se todos os componentes funcionam juntos corretamente.

Ao realizar um teste de sistema para essa camada, você precisa imitar as etapas que um usuário executa ao comprar um produto ou serviço. Por exemplo, para um site de comércio eletrônico, talvez seja necessário criar um driver de teste que pesquise o inventário, preencha um carrinho de compras e faça o check-out. A modelagem pragmática dessas etapas pode ser um desafio.

As tecnologias que você usa para construir a lógica de negócios ditam como você cria e conduz seus testes. Existem inúmeras tecnologias e técnicas que você pode usar para construir essa camada, o que torna impossível sugerir um método de teste simples. Por exemplo, você pode arquitetar sua solução usando um servidor de aplicativos dedicado, como o JBoss. Ou você pode ter módulos CGI independentes escritos em C, Python ou Perl.

Independentemente de sua abordagem, existem certas características de seu aplicativo que você deve sempre testar. Essas áreas incluem o seguinte:

Atuação. Teste para ver se o aplicativo atende às especificações de desempenho documentadas (geralmente especificadas em tempos de resposta e taxas de transferência).

Validade dos dados. Teste para detectar erros nos dados coletados dos clientes.

Transações. Teste para descobrir erros no processamento de transações, que podem incluir processamento de cartão de crédito, verificações de e-mail e cálculo de impostos sobre vendas.

Teste de desempenho Um aplicativo de Internet com desempenho insatisfatório levanta dúvidas na mente do usuário sobre sua robustez e muitas vezes afasta a pessoa.

Carregamentos de página demorados e transações lentas são exemplos típicos. Para ajudar a alcançar níveis de desempenho adequados, você precisa garantir que as especificações operacionais sejam escritas durante a fase de coleta de requisitos.

Sem especificações ou objetivos escritos, você não pode saber se seu aplicativo tem um desempenho aceitável. As especificações operacionais geralmente são declaradas em termos de tempos de resposta ou taxas de transferência. Por exemplo, uma página deve ser carregada em x segundos ou o servidor de aplicativos concluirá y transações de cartão de crédito por minuto.

Uma abordagem comum que você pode usar ao avaliar o desempenho é o teste de estresse. Frequentemente, o desempenho diminui a ponto de ficar inutilizável quando o sistema fica sobrecarregado com solicitações. Isso pode fazer com que os componentes transacionais sensíveis ao tempo falhem. Se você realizar transações financeiras, falhas de componentes podem fazer com que você ou seu cliente percam dinheiro. Os conceitos sobre teste de estresse apresentados no Capítulo 6 se aplicam ao teste de desempenho da camada de negócios.

Como uma revisão rápida, o teste de estresse envolve explodir o aplicativo com vários logins e simular transações até o ponto de falha para que você possa determinar se seu aplicativo atende aos objetivos de desempenho.

Claro, você precisa modelar uma visita de usuário típica para obter resultados válidos. Apenas carregar a página inicial não equivale à sobrecarga de encher um carrinho de compras e processar uma transação. Você deve taxar totalmente o sistema para descobrir erros de processamento.

O teste de estresse do aplicativo também permite que você investigue a robustez e a escalabilidade de sua infraestrutura de rede. Você pode pensar que

sua aplicação tem gargalos que permitem apenas x transações por segundo. Mas uma investigação mais aprofundada mostra que um roteador, servidor ou firewall mal configurado está limitando a largura de banda. Portanto, você deve garantir que seus componentes de infraestrutura de suporte estejam em ordem antes de iniciar o teste de estresse. Não fazer isso pode levar a resultados errôneos.

Validação de dados Uma função importante da camada de negócios é garantir que os dados coletados dos usuários sejam válidos. Se o seu sistema operar com informações válidas, como números de cartão de crédito incorretos ou endereços incorretos, poderão ocorrer erros graves. Se você não tiver sorte, os erros podem ter implicações financeiras para você e seus clientes. Você deve testar os erros de coleta de dados da mesma forma que procura erros de entrada do usuário ou de parâmetro ao testar aplicativos independentes. Consulte o Capítulo 5 para obter mais informações sobre como projetar testes dessa natureza.

Testes transacionais Seu site de comércio eletrônico deve processar as transações corretamente 100% do tempo. Sem exceções. Os clientes não tolerarão transações com falha. Além de uma reputação manchada e clientes perdidos, você também pode incorrer em responsabilidades legais associadas a transações com falha.

Você pode considerar o teste transacional como um teste de sistema da camada de negócios. Em outras palavras, você testa a camada de negócios do início ao fim, tentando descobrir erros. Mais uma vez, você deve ter um documento especificando exatamente o que constitui uma transação. Inclui um usuário pesquisando um site e preenchendo um carrinho de compras ou consiste apenas em processar a compra?

Para um aplicativo típico da Internet, o componente de transação é mais do que concluir uma transação financeira (como o processamento de cartões de crédito).

Eventos típicos relacionados a transações de clientes incluem:

- Pesquisando inventário.
- Coletando itens que o usuário deseja comprar.
- Apresentar ao usuário itens relacionados que possam ser de seu interesse.
- Apresentar aos usuários avaliações de produtos ou empresas de outros usuários.
- Solicitar e capturar avaliações de produtos ou empresas do usuário atual.
- Criando ou acessando uma conta de usuário.

Compra de itens, que pode envolver o cálculo de impostos sobre vendas e custos de envio, bem como o processamento de transações financeiras.

Notificar o usuário da transação concluída, geralmente por e-mail.

Além de testar os processos de transações internas, você deve testar os serviços externos, como validação de cartão de crédito, serviços bancários e verificação de endereço. Você normalmente usará componentes de terceiros e interfaces bem definidas para se comunicar com instituições financeiras ao realizar transações financeiras. Não presuma que esses itens funcionam corretamente. Você deve testar e validar se pode se comunicar com os serviços externos e se recebe os dados corretos deles.

Teste de camada de dados

Uma vez que seu site está funcionando, os dados que você coleta se tornam muito valiosos. Números de cartão de crédito, informações de pagamento e perfis de usuário são exemplos dos tipos de dados que você pode coletar durante a execução de seu site de comércio eletrônico. Perder essas informações pode ser desastroso e incapacitante para o seu negócio. Portanto, você deve desenvolver um conjunto de procedimentos para proteger seus sistemas de armazenamento de dados.

Testar a camada de dados consiste principalmente em testar o sistema de gerenciamento de banco de dados que seu aplicativo usa para armazenar e recuperar informações.

Sites menores podem armazenar dados em arquivos de texto ou bancos de dados de código aberto. Sites maiores e mais complexos usam bancos de dados de nível empresarial completos. Dependendo de suas necessidades, você pode usar ambas as abordagens.

Um dos maiores desafios associados ao teste dessa camada é duplicar o ambiente de produção. Você deve usar plataformas de hardware e versões de software equivalentes para realizar testes válidos. Além disso, depois de obter os recursos, tanto financeiros quanto de mão de obra, você deve desenvolver uma metodologia para manter os ambientes de produção e teste sincronizados.

Assim como nas outras camadas, você deve procurar erros em determinadas áreas ao testar a camada de dados. Estes incluem o seguinte:

Tempo de resposta. Quantificando os tempos de conclusão para operações de linguagem de consulta estruturada (SQL).

Integridade de dados. Verificar se os dados estão armazenados corretamente e com precisão.

Tolerância a falhas e capacidade de recuperação. Maximizando o MTBF e minimizando o MTTR.

Teste de tempo de resposta Aplicativos de comércio eletrônico lentos causam e clientes desconfiados. Assim, é do seu interesse garantir que o seu site responde em tempo hábil às solicitações e ações do usuário. O teste de tempo de resposta nesta camada não inclui o tempo de carregamento da página; em vez de, concentra-se na identificação de operações de banco de dados que não atendem ao desempenho Objetivos. Ao testar o tempo de resposta da camada de dados, você deseja garantir que as operações individuais do banco de dados ocorram rapidamente para não causar gargalos outras operações.

Dito isso, antes que você possa medir as operações do banco de dados, você deve entender o que constitui um. Para esta discussão, uma operação de banco de dados envolve inserir, excluir, atualizar ou consultar dados do RDBMS.

Medir o tempo de resposta consiste simplesmente em determinar quanto tempo cada operação leva. Você não está interessado em medir transações vezes, pois isso pode envolver várias operações de banco de dados. A criação de perfil das velocidades de transação ocorre durante o teste da camada de negócios.

Como você deseja isolar as operações do banco de dados com problemas, você não deseja medir a velocidade de uma transação completa ao testar dados tempos de resposta da camada. Muitos fatores podem distorcer os resultados do teste se você testar toda a transação. Por exemplo, se os usuários demoram muito para recuperar seus perfis, você precisará determinar onde está o gargalo para isso. operação reside. É a instrução SQL, servidor Web ou firewall? Teste a operação do banco de dados de forma independente permite identificar o problema. Neste exemplo, se a instrução SQL for mal escrita, ela se revelará quando você testa o tempo de resposta.

O teste de tempo de resposta da camada de dados está repleto de desafios. Você deve ter um ambiente de teste que corresponda ao que você usa na produção; caso contrário, você pode obter resultados de teste inválidos. Além disso, você deve ter uma compreensão completa do seu sistema de banco de dados para garantir que está configurado corretamente e operando de forma eficiente. Você pode achar que um a operação do banco de dados está funcionando mal porque o RDBMS está configurado incorretamente.

De um modo geral, porém, você realiza a maioria dos testes de tempo de resposta usando métodos de caixa preta. Tudo o que lhe interessa é o tempo decorrido para transações de banco de dados. Existem muitas ferramentas para ajudar nesses esforços, ou você pode escrever o seu próprio.

210 A Arte do Teste de Software

Teste de integridade de dados O teste de integridade de dados é o processo de encontrar dados precisos em seus armazenamentos de dados. Esse teste difere da validação de dados, que você realiza ao testar a camada de negócios. O teste de validação de dados tenta encontrar erros na coleta de dados. O teste de integridade de dados se esforça para encontrar erros na forma como você armazena os dados.

Muitos fatores podem afetar como o banco de dados armazena dados. O tipo de dados e o comprimento podem causar truncamento de dados ou perda de precisão. Para campos de data e hora, problemas de fuso horário entram em jogo. Por exemplo, você armazena o tempo com base na localização do cliente, do servidor Web, do servidor de aplicativos ou do RDBMS? A internacionalização e os conjuntos de caracteres também podem afetar a integridade dos dados. Por exemplo, conjuntos de caracteres multibyte podem dobrar a quantidade de armazenamento necessária, além de fazer com que as consultas retornem dados preenchidos.

Você também deve investigar a precisão das tabelas de referência usadas pelo seu aplicativo, como imposto sobre vendas, CEPs e informações de fuso horário. Você não apenas deve garantir que essas informações sejam precisas, mas também mantê-las atualizadas.

Teste de tolerância a falhas e capacidade de recuperação Se o seu site de comércio eletrônico depende de um RDBMS, o sistema deve permanecer em funcionamento. Há muito pouca ou nenhuma disponibilidade de tempo de inatividade neste cenário. Assim, você deve testar a tolerância a falhas e a capacidade de recuperação de seu sistema de banco de dados.

Um objetivo das operações de banco de dados, em geral, é maximizar o MTBF e minimizar o MTTR. Você deve encontrar esses valores especificados na documentação de requisitos do sistema para seu site de comércio eletrônico. Seu objetivo ao testar a robustez do sistema de banco de dados é tentar superar esses números.

Maximizar o MTBF depende do nível de tolerância a falhas do seu sistema de banco de dados. Você pode ter uma arquitetura de failover que permite que transações ativas mudem para um novo banco de dados quando o sistema primário falhar. Nesse caso, seus clientes podem sofrer uma breve interrupção do serviço, mas o sistema deve permanecer utilizável. Outro cenário é que você crie tolerância a falhas em seu aplicativo para que um banco de dados inativo afete muito pouco o sistema. Os tipos de testes executados dependem da arquitetura.

Você deve considerar a recuperação do banco de dados como igualmente importante. O objetivo do teste de recuperabilidade é criar um cenário no qual você não possa recuperar esse banco de dados. Em algum momento, seu banco de dados irá travar, então você precisa ter procedimentos para recuperá-lo muito rapidamente. O planejamento da recuperação começa na obtenção de backups válidos. Se você não puder recuperar o banco de dados durante o teste de capacidade de recuperação, precisará modificar seu

Plano B. Um sistema de banco de dados tolerante a falhas pode residir em vários locais conectados em uma rede privada ou compartilhada. Esse aspecto do gerenciamento de banco de dados também deve ser testado. Se o servidor local falhar, os sistemas remotos estão atualizados e seu software pode se conectar a um sistema remoto rapidamente? O que acontece se uma ou mais conexões de rede falharem? O que acontece se ocorrer uma falha do sistema enquanto os dados estão sendo gravados?

Em geral, esforce-se para testar todos os aspectos do sistema, tudo o que for necessário para suportar todos os níveis de atividade e integridade de dados para os quais seu aplicativo foi projetado.

Resumo

A Internet pública não existia quando a primeira edição deste livro foi escrita. De fato, os sistemas acessados remotamente e os aplicativos em geral eram infantis em comparação com os da Internet de hoje. Os usuários naqueles primeiros dias eram, em sua maioria, pessoas sofisticadas e condecoradas de computadores que podiam tolerar um nível bastante alto de dificuldade em acessar e usar aplicativos remotos.

Hoje, os usuários da Internet podem saber muito pouco sobre a operação real de computadores e software de computador, mas têm uma escolha virtualmente infinita de sites comerciais para escolher. Conseqüentemente, eles têm pouca ou nenhuma paciência para um aplicativo baseado na Web que não seja atraente, difícil de usar ou disfuncional. Portanto, testes aprofundados de qualquer aplicativo da Internet são extremamente importantes.

O teste de software no ambiente da Internet apresenta muitos desafios, principalmente a grande e variada base de usuários e a necessidade de extrema precisão e segurança para aplicativos de comércio eletrônico. Em geral, queremos testar três áreas principais de aplicação da Internet: apresentação (ou interface do usuário), lógica de negócios e gerenciamento de dados. Como seria de se esperar, grandes aplicativos baseados em usuários exigem testes extensivos de usuários (consulte o Capítulo 7 para obter mais informações sobre esse processo) para garantir que o software atenda às especificações de projeto e aos critérios de aceitação do usuário. É importante que qualquer aplicativo de software seja atraente e fácil de usar, mas os aplicativos para a Internet são julgados com mais severidade. Nesse ambiente, o sucesso do software geralmente equivale ao sucesso do negócio, e esse fator por si só deve levar os desenvolvedores a testes agressivos e completos.

11

Aplicativo móvel Teste

A tecnologia dos computadores muda rapidamente. Enquanto buscamos em um laptop ou servidor, agora pensamos em dispositivos móveis portátil. Essa migração mudou a maneira como conduzimos nossas vidas, negócios e governos. Também afetou significativamente a maneira como os desenvolvedores e testadores de software fazem seu trabalho.

A maioria dos profissionais de teste de software considera o teste de aplicativos móveis muito desafiador – mais do que quase qualquer outro tipo ou plataforma de software. Na verdade, são os dispositivos e o ambiente móvel mais do que o "aplicativo" que impõem o desafio. Esses dois componentes adicionam muitas variáveis e complexidades que podem distorcer ou mascarar problemas em seu aplicativo, o que dificulta o projeto de um plano de teste robusto. Resumidamente, você precisa considerar o desempenho e a confiabilidade da rede, interfaces de usuário consistentes, influências do transcodificador, diversidade de dispositivos e plataformas de recursos limitados.

Neste capítulo, apresentamos uma área relativamente nova de teste de software: testar aplicativos para dispositivos móveis e smartphones. Começamos descrevendo o ambiente de aplicativo móvel, que difere daquele de um aplicativo autônomo em desktops, laptops e servidores. Em seguida, enumeramos os desafios de testar aplicativos móveis – alguns dos quais abordamos anteriormente neste livro. Por fim, abordamos algumas abordagens de teste e considerações de casos de teste para ajudar a diminuir sua curva de aprendizado neste novo território.

Depois de ler este capítulo, você deve entender melhor os desafios e obstáculos de testar aplicativos móveis.

Ambiente móvel

Com a ampla implantação de hotspots sem fio, a linha entre a computação móvel e as atividades baseadas em rede sem fio "tradicional" ficou turva.

Assim, para começar aqui precisamos definir os termos dispositivo móvel e aplicativos móveis, com relação ao conteúdo deste capítulo. Nesse sentido, nos referimos a um dispositivo móvel como aquele que tem a capacidade de executar aplicativos baseados em rede por meio de um link de dados de celular ou satélite. Isso abrange a maioria dos smartphones, tablets e PDAs. Dito isso, não cometa o erro de identificar dispositivos móveis apenas pela aparência. Os laptops modernos podem aceitar placas de celular ou satélite plug-in, e alguns laptops têm esse acesso integrado. Com base nessa definição de dispositivo móvel, um aplicativo móvel é um programa baseado em rede que é executado em um dispositivo móvel.

Essa distinção é importante. Sim, é verdade que a maioria dos dispositivos móveis pode usar hotspots e pontos de acesso sem fio sem problemas. No entanto, essas conexões oferecem maior confiabilidade e velocidades mais altas do que as redes celulares, mesmo com a adoção das tecnologias 3G e 4G. Assim, você projeta seu aplicativo móvel com a expectativa de que ele usará links de dados relativamente lentos e comparativamente não confiáveis. Você também pode desenvolver aplicativos autônomos, como jogos, que rodam em um dispositivo móvel sem a necessidade de usar a rede da operadora. Mas, para os propósitos deste capítulo, não consideraremos aplicativos independentes como aplicativos móveis. Nosso foco está nos desafios associados a aplicativos executados em redes de dados celulares.

A chave para criar planos de teste bem-sucedidos para seus aplicativos móveis é entender o ambiente de computação móvel. A Tabela 11.1 identifica várias áreas cruciais que você deve investigar ao projetar planos de teste.

Primeiro, você deve entender os problemas de conectividade do dispositivo e as velocidades de rede, disponibilidade regional e latência. Tenha em mente a filosofia subjacente deste livro: Seus testes não devem provar que seu aplicativo funciona, mas que seu aplicativo não funciona para os casos de uso. Por exemplo, se você tiver um serviço baseado em localização ou aplicativo de e-mail, seus testes deverão identificar problemas de software quando a rede da operadora estiver lenta ou indisponível.

A seguir, há três áreas relacionadas a dispositivos — diversidade, restrições e métodos de entrada — que abordaremos em detalhes mais adiante neste capítulo. Para criar planos de teste bem-sucedidos, você e sua equipe de teste devem considerar os vários dispositivos no mercado, os recursos variados de cada um e como o usuário interage com os dispositivos.

TABELA 11.1 Considerações de projeto de teste de ambiente móvel

Área	Comente
Conectividade	Provisionamento de dispositivos Velocidade da rede Latência da rede Disponibilidade de rede em áreas remotas Confiabilidade do serviço
Dispositivos de diversidade	Vários navegadores da web para testar Várias versões de ambientes de execução para Java ou outras linguagens
Restrições do dispositivo	Memória ou processador limitado Tamanho de tela pequeno Vários sistemas operacionais Recursos multitarefa Tamanhos de cache de dados
Dispositivos de entrada	Telas sensíveis ao toque Caneta Rato Botões Rolos
Instalação e Manutenção	Instalando e desinstalando Aplicação de patches Atualizando

Por último, você precisa determinar como instalar e manter seu aplicativo. Alguns fornecedores, como a Apple, mantêm lojas online onde o usuário compra o aplicativo, mas somente após a Apple certificar seu aplicativo para sua plataforma. Isso facilita um pouco a instalação e a manutenção, pois você tem um sistema de distribuição único e certificado.

Desafios de teste

Como afirmado, o teste de aplicativos móveis está repleto de desafios. Para ajudar a alcançá-los, podemos categorizar a maioria em quatro categorias: diversidade de dispositivos, infraestrutura de rede da operadora, scripts e usabilidade. Você precisa pensar cuidadosamente em cada um ao projetar casos de teste. A combinação de cisalhamento de tipos de dispositivos, sistemas operacionais, métodos de entrada do usuário e preocupações de rede significam que as compensações devem ser equilibradas com tempo, finanças e

recursos para chegar a um plano de teste econômico que detecte a maioria dos bugs em um prazo razoável. A construção de uma estratégia de teste que combine os métodos discutidos nos capítulos anteriores ajudará.

No restante desta seção, discutimos essas categorias e oferecemos conselhos sobre como lidar com cada um.

Diversidade de dispositivos móveis

A diversidade cada vez maior de dispositivos apresenta um desafio de teste significativo e muitas vezes subestimado para alguém novo no teste de aplicativos móveis. Às vezes parece que os fabricantes introduzem novos dispositivos diariamente, tornando quase impossível acompanhar os ciclos de lançamento.

Pior, mais dispositivos significam mais itens a serem considerados em seus testes. Aqui está um exemplo simples para ilustrar apenas alguns itens que você precisa avaliar quando um novo dispositivo é lançado:

Suponha que a Motorola desenvolva um novo método de entrada de texto via tela sensível ao toque para seus telefones baseados em Android. Você pode projetar um teste para determinar se o novo método de entrada do dispositivo interrompe seu aplicativo? Em caso afirmativo, você pode corrigir seu aplicativo sem interromper o suporte para outros dispositivos baseados em Android, como tablets? Você pode até obter um dispositivo para testar? Você tem acesso a uma rede de operadoras suportadas?

Quase por definição, juntamente com a diversidade dos dispositivos, vem a diversidade de sistemas operacionais, navegadores, ambientes de tempo de execução de aplicativos, resoluções de tela, interfaces de usuário, ergonomia, tamanho de tela e muito mais. Você deve estar ciente de todos esses fatores ao criar testes. A diversidade de dispositivos também força o teste de usabilidade na frente e no centro, o que em algum momento exige que os testadores avaliem seu aplicativo nos dispositivos de destino. Usar emuladores é uma ótima maneira de começar, mas no final você precisará testar dispositivos reais em redes de operadoras reais.

Isso levanta outra faceta do teste de aplicativos móveis: testar em dispositivos reais versus emuladores. Do ponto de vista econômico, você deve fazer o máximo de testes possível com emuladores. Pode ser inviável financeiramente, mesmo que você consiga obter um dispositivo e acessar a rede sem fio, para testar na plataforma real. Dito isso, os emuladores apenas emulam; eles não são os dispositivos reais. Portanto, é provável que você observe diferenças entre os testes.

com um emulador e o dispositivo real. Por exemplo, as cores e formas de botões e caixas de entrada podem passar nos testes de aceitação em um emulador, mas falham no dispositivo de destino devido às diferenças de resolução de tela e profundidade de cor entre o dispositivo e um emulador baseado em PC.

Em suma, você precisa perceber que pode haver centenas de dispositivos móveis com potencial para acessar seu aplicativo. Portanto, durante as fases de coleta de requisitos e redação de especificações, você será solicitado a tomar algumas decisões difíceis e escolher um subconjunto razoável de dispositivos para suporte e teste. Esteja ciente de que todos os dispositivos que você não testar podem não funcionar com seu aplicativo; portanto, você pode perder não apenas um cliente, mas também uma base de clientes.

Infraestrutura de rede da operadora

Testar seu aplicativo em uma rede de operadora configura outro desafio.

Isso é especialmente verdadeiro se você deseja oferecer suporte a várias operadoras. Dois dos maiores obstáculos a serem superados são: entender e adaptar-se à infraestrutura da operadora e superar obstáculos baseados em localização.

Compreender a infraestrutura de uma operadora é fundamental para desenvolver um bom plano de teste. Inicialmente, você pensaria que seu aplicativo móvel usa a rede de uma operadora como um ponto de acesso sem fio IP. Não tão. A Figura 11.1 ilustra a infraestrutura “típica” da maioria das operadoras sem fio. A primeira diferença é que o protocolo não é baseado em IP; geralmente é um protocolo baseado em RF

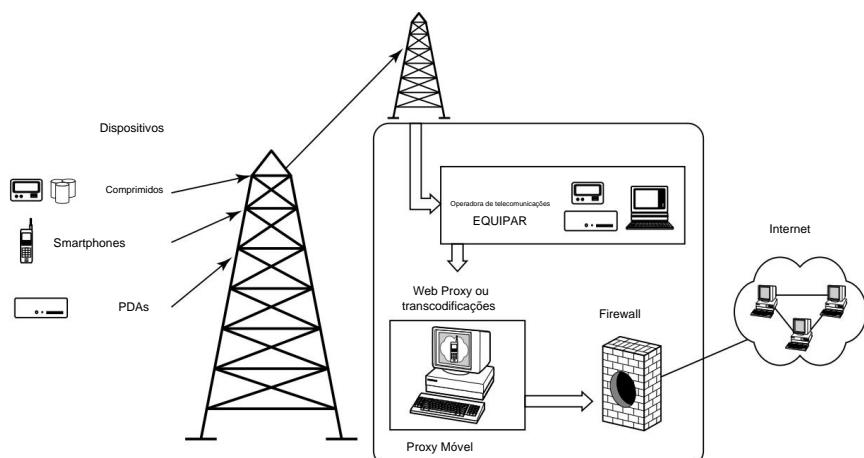


FIGURA 11.1 Rede de dados de operadora sem fio genérica.

como acesso múltiplo por divisão de código (CDMA), acesso múltiplo por divisão de tempo (TDMA) ou sistema global para celular (GSM). Os protocolos baseados em RF tratam os protocolos baseados em IP como uma "carga útil" e os entregam ao dispositivo móvel, que então decodifica a carga útil e a apresenta ao aplicativo.

Além disso, a maioria das operadoras usa algum tipo de transcodificador ou proxy da Web entre a Internet e o dispositivo. Esses dispositivos podem executar uma variedade de funções. E às vezes é difícil determinar exatamente o que ocorre, a menos que você trabalhe diretamente com as operadoras. Muitas vezes, eles não revelam essas informações para fins competitivos. A seguir, uma pequena lista do que pode ocorrer no proxy ou transcodificador da Web de uma operadora:

Transforme ou transcodifique conteúdo em WAP ou HTTP.

Compacte os dados para um melhor rendimento.

Criptografe o tráfego para privacidade e segurança.

Bloqueie o acesso a determinados sites de alta largura de banda.

Retire os cabeçalhos HTML e outros metadados das páginas da Web que seu aplicativo pode usar.

A transcodificação pode causar inconsistências na interface do usuário em vários dispositivos.

Alguns dispositivos suportam WAP (Wireless Application Protocol), enquanto outros suportam HTTP. WAP usa Wireless Markup Language (WML) para entrega de conteúdo. WAP e WML foram planejados para serem o "padrão" para entrega de conteúdo sem fio, mas nunca ganharam uma base sólida. No entanto, vários dispositivos o implementam, então você pode encontrá-lo durante seus testes. No entanto, a maioria dos smartphones e tablets suportam HTML e, portanto, dependem de HTTP para fornecer conteúdo. Se você tiver problemas de interface do usuário entre dispositivos e operadoras, verifique com cada um para determinar se WAP/WML ou HTTP/HTML está sendo usado.

Embora a compactação de dados se destine a melhorar a taxa de transferência, muitas vezes durante os períodos de alta atividade a taxa de transferência pode diminuir devido à sobrecarga da compactação. O mesmo vale para a segurança: firewalls e camadas semelhantes podem diminuir a taxa de transferência durante horas de alto volume.

Finalmente, você deve superar os obstáculos baseados em localização. Obviamente, para testar na rede de uma operadora, você precisa ter acesso a ela. Por exemplo, e se você tiver um aplicativo de viagem para um smartphone: como você testa redes de operadoras em outras partes do país ou em outros países?

Resposta: Você deve viajar para lá ou contratar alguém para testá-lo para você. Ambos aumentam o custo do teste.

Script

Uma área frequentemente negligenciada de teste de aplicativos móveis é a criação e execução de scripts de teste. Dispositivos reais não permitem que você carregue scripts automatizados e repetíveis no dispositivo; a equipe de teste executa manualmente todos os scripts. Ou seja, alguém percorre um script de teste escrito projetado para encontrar erros em um caso de teste no dispositivo de destino. Observe que dissemos "dispositivo de destino". Existem muitos destinos no ambiente móvel.

Como apontamos nos capítulos anteriores, o teste manual é propenso a erros. Infelizmente, isso é inevitável ao testar aplicativos móveis em dispositivos reais. Como mencionado, a maioria dos emuladores possui uma rica funcionalidade de script e pode realizar a maior parte dos testes de regressão e do sistema. No entanto, no final, você ainda precisa ter alguém para trabalhar com o dispositivo. (Mais adiante neste capítulo, explicaremos como criar um script de teste manual genérico para suportar vários dispositivos.)

A notícia refrescante é que os dispositivos móveis estão se tornando muito mais sofisticados e poderosos. Dada a competitividade do mercado, é razoável esperar que um produto de script automatizado apareça. O iOS da Apple, o sistema operacional Windows Mobile e o sistema operacional Android estão amadurecendo rapidamente, portanto, é provável que esse problema não seja um problema em versões futuras.

Usabilidade

O teste de usabilidade apresenta desafios semelhantes aos dos scripts de teste. Lembre-se dos capítulos anteriores de que o teste de usabilidade é principalmente uma abordagem de caixa branca. Assim como testar aplicativos de desktop independentes, uma equipe de teste deve tentar manualmente encontrar bugs na interface do usuário e nas camadas de interação do usuário do seu aplicativo.

Ao contrário do teste de aplicativos de desktop independentes, o teste de dispositivos móveis envolve mais de uma plataforma para testar. Por exemplo, você desejará pesquisar problemas de consistência da interface do usuário entre os produtos da Apple e as plataformas baseadas em Android. Embora você esteja testando aplicativos móveis, muitas das discussões do Capítulo 7 se aplicam.

Abordagens de teste

Algumas áreas de teste de dispositivos móveis são semelhantes ao teste de aplicativos da Internet, especialmente ao avaliar as infraestruturas de back-end. O prefeito

a diferença está em como você aborda o teste do próprio dispositivo. Com Internet testando, você tem apenas um punhado de navegadores para avaliar; com dispositivos móveis, você tem exponencialmente mais.

Naturalmente, ao testar componentes de back-end, você deve雇用 técnicas semelhantes e avaliar considerações semelhantes às discutidas em Capítulo 10, "Teste de aplicativos da Internet". Voltando à Figura 10.1, os níveis 2 e 3 devem ter aproximadamente a mesma configuração que um aplicativo de Internet normal. Como uma revisão rápida, você deve testar as especificações de desempenho, rotinas de validação de dados e componentes de processamento de transações da camada 2. Testar a camada 3 também é o mesmo que com a Internet formulários; testar tempos de resposta, integridade de dados, tolerância a falhas e capacidade de recuperação nesta camada. Se possível, teste os componentes de nível 2 e 3 separadamente do dispositivo para garantir que eles atendam às especificações do seu projeto usando testes de função.

A camada de teste 1, o ambiente do usuário, difere da Internet tradicional teste. Os conceitos apresentados ao testar seu conteúdo e site arquitetura ainda se aplicam. No entanto, o teste do ambiente do usuário equivale a teste do dispositivo.

Devemos observar a importância dos casos de uso ao desenvolver planos de teste para seus dispositivos. Saber quem usará seu aplicativo e como e quando, é imperativo, pois os aplicativos móveis têm vários pontos de falha. A Tabela 11.2 lista os itens que você geralmente não considera ao projetar casos de teste para aplicativos padrão, sejam autônomos ou baseados na Web. Por exemplo, testar seu aplicativo na rede da operadora é extremamente importante. Você deseja encontrar problemas relacionados à cobertura irregular ou perda de conectividade. Se sua aplicação envolve transferências de dados, procure problemas com cache de dados e sincronização incompleta com back-end armazenamentos de dados. O que acontece quando a cobertura é restaurada repentinamente após uma interrupção durante o download de um aplicativo? A compra ocorre duas vezes? Verifique se há bugs relacionados ao tratamento de reinicialização de sessão e corrupção de dados. Alguns desses problemas se aplicam a aplicativos baseados na Web executados em um Navegador baseado em PC. No entanto, as LANs/WANs são muito mais estáveis. Quando lidar com redes celulares, você deve esperar perder a conectividade.

Um caso de teste específico para testes móveis é como seu aplicativo lida chamadas de voz e mensagens de texto recebidas. As chances são de que os usuários finais vão querer para suspender seu aplicativo ou executá-lo em segundo plano, enquanto atendem o telefone ou lêem a mensagem de texto. Tente construir casos de teste em que chamadas e mensagens futuras causem problemas em seu aplicativo.

TABELA 11.2 Categorias de teste para teste de aplicativos móveis

Categoria de teste	Descrição
Instalar/Desinstalar	Certifique-se de que o usuário possa instalar corretamente seu aplicativo. Certifique-se de que o usuário possa desinstalar completamente seu aplicativo.
Rede	Verifique se o aplicativo responde adequadamente à perda de rede.
A infraestrutura	Verifique se o aplicativo responde adequadamente à restauração da rede. Verifique se o aplicativo responde adequadamente a sinais fracos.
Chamada recebida/ Mensagem	Teste se o usuário pode aceitar chamadas/mensagens de texto enquanto o aplicativo está em execução.
Manipulação	Teste se o usuário pode retomar o aplicativo ao finalizar chamadas/mensagens de texto. Teste se o usuário pode rejeitar chamadas/mensagens de texto sem interromper o aplicativo. Teste se o usuário pode iniciar uma chamada/mensagem de texto sem interromper o aplicativo.
Memória baixa	Certifique-se de que o aplicativo permaneça estável quando o dispositivo encontrar uma situação de pouca memória.
Mapeamentos de teclas	Teste se todos os mapeamentos de teclas funcionam conforme especificado.
Comentários	Certifique-se de que o feedback do usuário ao pressionar a tecla ocorra dentro das especificações de design do aplicativo.
Saindo	Verifique se o aplicativo sai normalmente quando iniciado pressionando as teclas, fechando a tampa ou usando o controle deslizante. Confirme se o aplicativo atende às especificações de design quando o usuário inicia o desligamento do dispositivo.
Carregamento	Certifique-se de que o aplicativo funcione conforme projetado ao entrar no modo de carregamento. Certifique-se de que o aplicativo funcione conforme projetado no modo de carregamento. Certifique-se de que o aplicativo funcione conforme projetado ao sair do modo de carregamento.
Bateria Condições	Teste como o aplicativo se comporta com bateria fraca. Meça a rapidez com que o aplicativo drena a bateria. Certifique-se de que o aplicativo responda de acordo com a especificação quando a bateria for removida enquanto o dispositivo estiver ligado.
Interação do dispositivo	Certifique-se de que o aplicativo não sobrecarregue a CPU. Certifique-se de que o aplicativo não consuma muita memória.

TABELA 11.3 Dispositivos versus Emuladores

Teste		
	Desvantagens da abordagem	Vantagens
Real Dispositivos	Caro, especialmente se você segmentar uma ampla base de dispositivos móveis	Capacidade de testar a capacidade de resposta do aplicativo
	Incapacidade de instalar ferramentas de desenvolvimento de medição ou diagnóstico	Inspeção visual do aplicativo no dispositivo real para verificar a consistência da interface do usuário
	Não é possível instalar em scripts de teste de execução	Rede de operadoras de teste
	Disponibilidade de rede	capacidade de resposta
		Identifique bugs específicos do dispositivo
Emuladores	Incapacidade de identificar bugs relacionados ao dispositivo	Custo-benefício
	O hardware subjacente pode distorcer o desempenho no dispositivo real	Fácil de gerenciar; suporte a vários dispositivos com um único emulador

No restante do capítulo, abordaremos algumas abordagens para teste de dispositivos nas quais você basicamente tem duas opções: testar em dispositivos reais ou usar emuladores de dispositivos. A Tabela 11.3 oferece algumas vantagens e desvantagens de cada abordagem.

Testes com dispositivos reais

O teste manual com dispositivos reais é inevitável. Apesar de caro, tem algumas vantagens. Somente testando com o dispositivo você pode experimentar suas nuances e ter uma noção real da experiência do usuário. Além disso, você só pode testar determinados casos com dispositivos reais. Testar a confiabilidade da rede de uma operadora e determinar o efeito de uma chamada ou mensagem de texto recebidas são exemplos óbvios. Em um dispositivo real você também pode avaliar como seu aplicativo se comporta. Ele carrega rápido e roda a uma velocidade aceitável? Parece bom?

A interface do usuário é consistente em seus dispositivos de destino? Por último, mas não menos importante, você pode determinar bugs específicos do dispositivo. Isso é quase impossível com um emulador. Se você encontrar um bug específico do dispositivo, o desafio é corrigi-lo sem quebrar a compatibilidade com outros dispositivos.

Apesar das vantagens, testar com dispositivos reais também apresenta algumas sérias desvantagens. Por exemplo, é caro porque você deve comprar o dispositivo,

bem como pagar pelo tempo de antena da transportadora. Nenhum dos dois é barato e, se você estiver testando vários dispositivos de várias operadoras em várias regiões, as despesas aumentam de acordo. Alguns fabricantes de dispositivos e provedores de serviços têm dispositivos que você pode alugar ou acessar remotamente, o que pode reduzir alguns custos. Se você segmentar uma plataforma individual, como a família Apple iPhone, poderá ser poupadão de grande parte dessa despesa. Ainda assim, você precisará de uma quantidade suficiente de cada tipo (iPad, iPhone, iTouch) para testar.

Além disso, testar com dispositivos reais é um processo manual de caixa branca. Alguém deve apertar os botões, tocar nas telas e inserir dados. Como você sabe, o teste manual é propenso a erros, mesmo com as melhores instruções e testadores treinados. Além disso, adiciona outra despesa ao processo. Você deve manter anotações precisas sobre cada script de teste bem documentado e seus resultados. Em seguida, avalie a eficácia dos scripts e elimine aqueles com pouco ou nenhum valor (ou seja, não encontre bugs).

Como observamos anteriormente, o uso de dispositivos reais elimina uma arma importante no arsenal do testador de software: scripts de teste automatizados. Portanto, você deve usar scripts manuais escritos que especificam ações genéricas, não detalhes sobre como executar a ação em um dispositivo. Scripts de teste detalhados para cada dispositivo seriam um desafio para criar e manter. Em pouco tempo, você teria uma biblioteca de scripts, que pode ficar obsoleta quando o dispositivo for atualizado. Os scripts genéricos permitem testar as especificações do sistema em vários dispositivos.

Por exemplo, iPhones, iPads e dispositivos baseados em Android dependem muito de telas sensíveis ao toque para entrada do usuário. Outros dispositivos, como BlackBerrys ou telefones "padrão", possuem teclados ou teclados numéricos para permitir a entrada do usuário. A Tabela 11.4 fornece um script de exemplo para verificar se seu aplicativo, um e-reader, aborta se você receber uma mensagem de texto enquanto lê um e-book. Observe que o script não especifica exatamente como executar nenhuma etapa, apenas executar a etapa usando os recursos de entrada do usuário do dispositivo. Podem ser botões, telas sensíveis ao toque ou comandos de voz. Em nenhum momento você especifica "Pressione OK" ou "Pressione Enviar". Essa abordagem genérica permitirá que você avalie casos de teste em vários dispositivos.

Por último, os fabricantes geralmente "bloqueiam" dispositivos reais, o que significa que você não pode carregar ferramentas para monitorar ou depurar seu aplicativo. Então, quando você atinge um bug, é mais difícil isolá-lo. Por exemplo, se seu aplicativo estiver lento, você não saberá se é a rede da operadora, problemas de transcodificação, seu aplicativo ou uma combinação deles. Somente por tentativa e erro você pode identificar problemas.

TABELA 11.4 Script de teste de dispositivo genérico

-
1. Inicie o aplicativo e-reader.
 2. Abra o e-book.
 3. Inicie a mensagem SMS para o dispositivo a partir de outro dispositivo.
 4. Verifique se o alerta de mensagem SMS é exibido.
 5. Abra a mensagem SMS.
 6. Escolha Responder à mensagem SMS.
 7. Compor mensagem SMS.
 8. Envie uma mensagem SMS.
 9. Verifique a notificação de envio de mensagem SMS.
 10. Retorne ao e-book.
 11. Verifique se o aplicativo de e-book está em execução.
 12. Verifique o retorno à mesma página ou marcador.
 13. Saia do aplicativo e-reader.
-

Testando com emuladores

Testar com emuladores pode não ser a abordagem preferida, mas geralmente é a mais prática e econômica, e ainda tem algumas vantagens.

Primeiro, os emuladores permitem testes funcionais rápidos e baratos de seu aplicativo. Você pode percorrer o aplicativo para encontrar eventos e circunstâncias que não atendem aos requisitos do programa. Identifique esses bugs usando emuladores antes de gastar com testes de dispositivos.

Em segundo lugar, os emuladores são fáceis de gerenciar e, como são executados em PCs, todo testador ou desenvolvedor pode ter um emulador. Os desenvolvedores podem gerenciar o software por conta própria, eliminando a necessidade de administradores de sistema.

Terceiro, a maioria dos pacotes de emuladores suporta vários dispositivos. Para testar um dispositivo diferente, basta carregar um perfil de dispositivo diferente. O melhor de tudo é que você não incorre em custos caros de tempo de antena da transportadora. Quarto, os emuladores são executados em computadores com mais recursos, como CPUs mais rápidas e mais memória. Tempos de resposta rápidos durante os testes permitem que você conclua os testes mais rapidamente.

A última e provavelmente mais significativa vantagem é que a maioria dos emuladores emprega linguagens de script de alto nível, para que você possa criar testes consistentes e automatizados, que são menos propensos a erros e mais rápidos do que os testes manuais.

O script automatizado também permite testes de regressão mais fáceis e rápidos, o que é especialmente importante ao verificar se as alterações feitas no seu

aplicativo para suportar um dispositivo não interrompa o suporte para outro. As linguagens de script em emuladores geralmente são independentes de dispositivo. Referindo-se à Tabela 11.4, quando você executa o script da Etapa 8, "Enviar mensagem SMS", o emulador executará essa função independentemente do dispositivo. Isso permite que scripts sejam usados entre dispositivos.

A desvantagem de usar emuladores para teste é que você não pode identificar as nuances e bugs de cada dispositivo. Como dissemos antes, em algum momento, você deve testar seu aplicativo nos dispositivos de destino. Sem testar em dispositivos reais, você nunca pode ter 100% de certeza de que atende às especificações de compatibilidade e desempenho. No entanto, não descarte o uso de emuladores para a maior parte de seus testes. É uma maneira econômica e eficiente de eliminar a maioria de seus bugs.

Resumo

O teste de aplicativos móveis representa uma nova fronteira no teste de software. O ambiente móvel adiciona maior complexidade e mais interações não experimentadas ao testar aplicativos autônomos padrão. Dito isso, com uma compreensão dos desafios, você pode melhorar muito suas chances de testar seu aplicativo com sucesso.

Comece tentando obter um controle sobre o universo de dispositivos que você deseja oferecer suporte. Você quer oferecer suporte apenas a smartphones e tablets baseados em Android ou quer dar suporte à maioria dos principais fornecedores de tablets e smartphones? Em seguida, entenda a infraestrutura de rede da operadora. Ele transcodifica, criptografa, compacta ou de alguma forma modifica os dados antes de enviá-los ao dispositivo?

Você também precisa encontrar um equilíbrio entre o emulador e o teste de dispositivo real. Ambos têm seus prós e contras. Devido aos custos, você provavelmente usará mais emuladores e economizará o teste do dispositivo para as fases finais. Use as categorias de teste na Tabela 11.2 como ponto de partida para desenvolver o seu próprio. Consulte as categorias com frequência ao definir seus casos de teste. Além disso, trate qualquer script de teste escrito e resultado como código-fonte; certifique-se de ter backups adequados e alguma forma de controle de alterações nos documentos de teste. Para economizar tempo e dinheiro, revise a eficácia de cada script e elimine aqueles que não agregam valor.

Depois de entender os fundamentos do teste de aplicativos móveis, você não deverá ter problemas para criar planos de teste e casos de uso. Uma coisa é certa, os aplicativos móveis estão aqui e, mais cedo ou mais tarde, você precisará aprender a testar esses aplicativos exclusivos. Por que não começar agora?

Apêndice

Teste Extremo de Amostra

Inscrição

1. check4Prime.java

Para compilar:

```
&> javac check4Prime.java
```

Para correr:

```
$> java -cp check4Prime 5
```

Certo . . . 5 é um número primo!

```
$> java -cp check4Prime 10
```

Desculpe . . . 10 NÃO é um número primo!

```
$> java -cp check4Prime A
```

Uso: check4Prime x –
onde 0<=x<=1000

Código fonte:

```
//check4Prime.java //
Imports import
java.lang.*;

classe pública check4Prime {

    estático final int max = 1000; // Definir limites superiores.
    estático final int min = 0; // Define os limites inferior e superior
    a variável de entrada

    public static void main (String [] args) {

        //Iniciar objeto de classe para trabalhar com
        check4Prime check = new check4Prime();

        try{ //
            Verifica argumentos e atribui valor à variável de entrada
            check.checkArgs(args);
        }
```

228 Apêndice

```

//Verifica a Exceção e exibe a ajuda }catch (Exceção e)
{
    System.out.println("Uso: check4Prime x"); System.out.println(" –
onde 0<=x<=1000"); System.exit(1); }

//Verifica se a entrada é um número primo if
(check.primeCheck(input))
    System.out.println("Direito... " + input + " é um número primo!"); senão
        System.out.println("Desculpe... " + input + " NÃO é um número primo!");

} // Finaliza principal

//Calcula números primos e compara com a entrada public boolean primeCheck
(int num){

    raiz quadrada dupla = Math.sqrt(max); // Encontra a raiz quadrada de n

    //Iniciar array para armazenar números primos boolean
    primeBucket [] = new boolean [max+1];

    //Inicializa todos os elementos como true, depois define os não primos como false for (int
    i=2; i<=max; i++){ primeBucket[i]=true;

}

//Faça todos os múltiplos de 2 primeiro
int j=2; for (int i=j+j; i<=max; i=i+j)
{ primeBucket[i]=false;                                //começa com 2j como 2 é primo //
define todos os múltiplos para false

    for (j=3; j<=raiz quadrada; j=j+2){                  // faz até sqrt de n if
        (primeBucket[j]==true) // só faz se j for primo for (int i=j-1; i>j; i--) // j é primo //
        define todos os múltiplos para false
            primeBucket[i]=false;
        }
    }
}

//Verifica a entrada em relação ao array principal
if (primeBucket[num] == true) {
    retorno
    verdadeiro; }
    else{ return false; }

}//fim do PrimeCheck()

```

```

//Método para validar entrada
public void checkArgs(String [] args) throws Exception{ //Verifica
    argumentos para o número correto de parâmetros if (args.length != 1) {
        lançar nova Exceção();
    } else{ //Obter inteiro do caractere
        Integer num = Integer.valueOf(args[0]);
        entrada = num.intValue();

        //Se for menor que zero
        if (input < 0) throw new Exception(); //Se menor que os limites inferiores
        senão se (entrada > max) //Se maior que os limites superiores
            lançar nova Exceção(); } }

}//Fim check4Prime

```

2. check4PrimeTest.java

Requer a API JUnit, junit.jar

Compilar:

```
$> javac -classpath ..junit.jar check4PrimeTest.java
```

Para correr:

```
$> java -cp ..junit.jar check4PrimeTest
```

Exemplos:

Iniciando teste. . .

.....

Tempo: 0,01

OK (7 testes)

Teste finalizado. . .

Código fonte:

```

//check4PrimeTest.java //
Imports import junit.framework.*;

public class check4PrimeTest estende TestCase{

    // Inicializa uma classe para trabalhar.
    private check4Prime check4prime = new check4Prime();

    //construtor public
    check4PrimeTest (nome da string){
        super(nome); }

```

230 Apêndice

```

//Ponto de entrada
principal public static void main(String[] args) {
    System.out.println("Iniciando teste... ");
    junit.textui.TestRunner.run(suite());
    System.out.println("Teste finalizado..."); } // fim de
main()
// Caso de teste
1 public void testCheckPrime_true()
    { assertTrue(check4prime.primeCheck(3));
}

// Casos de teste
2,3 public void testCheckPrime_false(){
    assertFalse(check4prime.primeCheck(0));
    assertFalse(check4prime.primeCheck(1000));
}

// Caso de teste
7 public void testCheck4Prime_checkArgs_char_input(){
    tente
    { String [] args= new String[1];
    args[0]="r"; check4prime.checkArgs(args);
    fail("Deve gerar uma exceção."); }
    catch (Exception success){ //teste bem
    sucedido }

} //fim de testCheck4Prime_checkArgs_char_input()

// Caso de teste
5 public void testCheck4Prime_checkArgs_above_upper_bound(){
    tente
    { String [] args= new String[1];
    args[0]="10001";
    check4prime.checkArgs(args);
    fail("Deve gerar uma exceção."); } catch
    (Exception success){ //teste bem sucedido }

} // fim de testCheck4Prime_checkArgs_upper_bound()

// Caso de teste
4 public void testCheck4Prime_checkArgs_neg_input(){
    tente
    { String [] args= new String[1];
    args[0="-1";
    check4prime.checkArgs(args);
    fail("Deve gerar uma exceção."); } catch
    (Exception success){ //teste bem sucedido }

}// fim testCheck4Prime_checkArgs_neg_input()

// Caso de teste
6 public void testCheck4Prime_checkArgs_2_inputs(){
    tentar {
        String [] args= new String[2];
        argumentos[0]="5"; argumentos[1]="99";
}

```

```
check4prime.checkArgs(args);
fail("Deve gerar uma exceção."); } catch
(Exception success){ //teste bem sucedido }

} // fim de testCheck4Prime_checkArgs_2_inputs
// Caso de teste
8 public void testCheck4Prime_checkArgs_0_inputs(){
    tente
    { String [] args= new String[0];
    check4prime.checkArgs(args);
    fail("Deve gerar uma exceção."); } catch
    (Exception success){ //teste bem sucedido }

} // fim testCheck4Prime_checkArgs_0_inputs

//Método obrigatório de JUnit.
public static Test suite() {
    Suite TestSuite = new TestSuite(check4PrimeTest.class); suíte de
    retorno; }//fim da suíte()

} //fim do check4PrimeTest
```


Índice

UMA

Teste de aceitação, 131 extremo,
184, 186

Desenvolvimento ágil, 175
manifesto, 176 metodologias
de tabela, 177

Testes ágeis, 175, 178

Servidor de aplicativos, 205

Ferramentas de depuração automatizadas, 159

B

Depuração de retrocesso, 167

Beck, Kent, 176

Beedle, Mike, 176

Teste de big-bang, 98

Testes de caixa preta, 8

particionamentos de equivalência, 49

testes de usabilidade, 145

Comparação caixa preta-caixa branca,
42

Teste de baixo para cima, 107

em comparação com teste de cima para baixo,
108

Análise de valor limite, 55 diretrizes
para, 56

programa MTEST para, 57

Teste de cobertura de filiais, 44

Teste de compatibilidade do navegador, 204

Depuração de força bruta, 158

Camada de negócios, 196, 201

Teste de camada de negócios, 205

Nível de negócios

critérios de teste de mesa, 198

C

Arquitetura C/S, 195

C++

teste de caixa preta de, 9

Infraestrutura de rede da operadora, 217

Símbolos de restrição de

gráfico de efeito de causa para, 65

diagramas lógicos para, 64

amostras, 64 amostras sem

restrições, 71 símbolos para, 63 com

restrição exclusiva, 66

Gráficos de causa e efeito, 61

casos de teste para, 62

CDMA, 218

CGI, 196, 205

Arquitetura cliente-servidor, 195

COBOL

história de, 26

Cockburn, Alistair, 176

Acesso múltiplo por divisão de código, 218

Inspeções de código, 22

Orientado a Negócios Comuns

Linguagem. Veja COBOL

- Interface de gateway comum, 196
Erros de comparação, 29
Teste de compatibilidade/conversão, 127
Testes de componentes, 153
Erros de computação, 28
Definição de computador, 1
Teste de cobertura de condição, 45
Mascaramento de condição, 46
Teste de configuração, 126
Gráfico de fluxo de controle, 11
Erros de fluxo de controle, 31
Cunningham, Ward, 176
- D
- Erros de declaração de dados, 28
Teste de usabilidade de métodos de coleta de dados, 150
Teste de integridade de dados, 210
Camada de dados, 201
Teste de camada de dados, 208
Erros de referência de dados, 25
Camada de dados critérios de teste de tabela, 198
Validação de dados, 207 Teste orientado a dados. Veja Depuração de teste de caixa preta, 157 ferramentas automatizadas para, 159 por retrocesso, 167 por força bruta, 158 por dedução, 163 por indução, 160 exemplo de estruturação de pistas, 163 análise de erros, 172 fluxograma indutivo, 160 princípios de, 168
- E
- Comércio eletrônico arquitetura básica de, 195 Economia de testes, 8 Formulário de classe de equivalência, 51 Identificação de classes de equivalência, lista de 51 classes de tabela, 54 casos de teste para, 52 Particionamento de equivalência, 49, 50
- Análise de erros com depuração, 172
- Lista de verificação de erros, 25
- Erro ao adivinhar, 80
- Erros comparação, 29 computação, 28
- resistência do programador, 157 com casos de teste, 167
- Depuração de erros de localização de principais, 168 reparo de erros, 170
- Teste de cobertura de decisão, 44
Teste de cobertura de decisão/condição, 46
- Depuração deductiva, 163 o fluxograma do processo, 164 as etapas, 164
- Verificação de mesa, 21, 37
- DISPLAY comando causa-efeito gráfico para, 72 gráfico para, 70
- Dispositivos móveis de diversidade, 216
- Documentação fluxograma de software, 116
- Módulo de driver, 98

- fluxo de controle, 31
declaração de dados, 28
estimativas por plotagem, 140
estimativas de número, 136 entradas/
saídas, 33 interfaces, 32
arredondamentos, 29 tabelas-
quando erros encontrados, 138
- Processo unificado essencial, 178
EssUP. Veja Processo unificado essencial
Testes de entrada exaustivos, 9 Testes de
aceitação extremos, 186 Programação extrema,
179 práticas de tabela-12, 182 Noções básicas
de programação extrema, 180 Testes
extremos, 179, 180
teste de aceitação com, 184 aplicados,
187 conceitos de, 184
Driver de teste JUnit, design
de 190 casos de teste, teste
de 188 unidades com, 184
Teste de unidade extrema, 185
Rastreamento ocular, 151
- F
Teste de instalações, 123
Teste tolerante a falhas, 210
Forma
classe de equivalência, 51
Formula Translation System. Veja Fortran
- Fortran
história de, 26
Fowler, Martin, 176
Teste de funcionamento
propósito de, 116
Teste de função, 119
- G
Sistema global para celular, 218
Interface Gráfica de Usuário, 2 Gráficos
de causa efeito, 61 Grenning, James,
176 GSM, 218 GUI. Veja a interface
gráfica do usuário
- H
Teste de corredor, 147
Testes de ordem superior, 113
realizando o teste, 130
componentes do plano de teste,
133 planejamento e controle de teste, 132
Highsmith, Jim, 176
HTML, 196
Testes humanos, 19
Hunt, André, 176
Linguagem de marcação de hipertexto, 196
- EU
Teste incremental, 96 agência
de teste independente, 141 depuração
de indução, 160 etapas de depuração
indutiva para, 161 estruturando as
pistas, 161 fluxograma indutivo
para depuração de programa, 160
erros de entrada/saída, 33 testes de
entrada/saída. Consulte a tabela de
resumo da lista de verificação de erros
de inspeção de teste de caixa preta, 35 agenda
de inspeções para, 23 eficácia de, 21

- Lista de verificação de erros de inspeções (continuação), 25
- benefícios colaterais de, 24
- descrição da equipe, 22
- tempo necessário, 24
- Inspeções e orientações, 20
- Teste de instalação, 127, 132
- Erros de interface, 32
- Testes de integridade de dados de aplicativos da Internet, 210
- testes de camada de dados, 208
- validação de dados, 207 testes de tolerância a falhas, 210 arquitetura de ilustração, 201 testes de desempenho, 206 testes de recuperação, 210 testes de tempo de resposta, 209 critérios de teste de tabela, 202 testes de, 193 estratégias de teste, 200 testes transacionais, 207
- Desafios de teste da Internet de, 196
- J
- JBoss, 205
- Jeffries, Ron, 176
- JUnit, 187
- piloto de testes, 190
- K
- Kern, Jon, 176
- eu
- LDAP, 196
- Protocolo de aplicativo de diretório leve, 196
- Teste de cobertura lógica, 43
- Testes baseados em lógica. Consulte Teste de caixa branca
- M
- Marick, Brian, 176
- Martin, Robert C., 176 Mean
- Time Between Failures, 128, 200
- Tempo médio de reparo, 129, 200
- Mellor, Steve, 176
- Definição de aplicativo móvel, 214
- Testes de aplicativos móveis, 213
- abordagens, 219 desafios, 215
- scripts, 219 categorias de tabela, 221
- dispositivos de tabela versus emuladores,
- 222
- script de teste genérico de tabela, 224 testes de usabilidade, 219 com emuladores, 224 com dispositivos reais, 222
- Dispositivo móvel
- definição, 214
- Diversidade de dispositivos móveis, 216
- Ambiente móvel, 214
- considerações de design de teste de mesa, 215
- Driver de módulo, 98
- tabelas de entrada para, 87 stub, 98
- Objetivo do teste do módulo de, 116
- Teste de módulo, 85
- realizando o teste, 109 projeto de caso de teste, 86

- MTBF, 200, 210, consulte o tempo médio
 - Entre falhas
- MTEST
 - gráfico de entrada do programa,
 - 58 especificações do programa, 57
- MTTR, 200, 210, consulte o tempo médio para
 - Reparar
- Teste de cobertura de várias condições, 47, 48
- N
 - Nielsen, Jakob, 148
 - Teste não incremental, 98
- P
 - Centro de Pesquisa de Palo Alto, 143
 - PARC. Veja Palo Alto Research Centro
 - Sensibilização de caminho, 73
 - Classificações de pares, 38
 - Teste de desempenho, 126, 206
 - Executando o teste teste
 - de ordem superior, 130
 - PL/1
 - fundo, 88
 - Camada de apresentação, 196, 201
 - Teste de camada de apresentação, 203
 - Nível de apresentação
 - critérios de teste de mesa, 198
 - Princípios de localização de
 - erros de depuração,
 - reparação de erros 168, 170
 - Teste de procedimento, 130
 - Exemplo
 - de módulo do programa 12, 102
 - Desenvolvimento ágil, 175
 - pontos de interrupção em, 159
- gráfico de fluxo de controle, 11 listas de verificação de erros, 25 inspeções, orientações e revisões, 19
- Amostra Java, 43
- tabelas de entrada de módulo, 87 testes de regressão, 16
- fluxogramas de amostra, 43
- diagrama de seis módulos, 98
- principais de teste, 13
- Definição de teste do programa, 17
- critérios de sucesso, 18
- Diretrizes de teste do programa, 13
- Psicologia do teste, 5
- Q
 - Teste de usabilidade do questionário, 152
- R
 - Teste de entrada aleatório, 41
 - Desenvolvimento rápido de aplicativos, 179
- Processo unificado Rational, 178
- RDBMS, 196, 209, 210 Teste de recuperação, 210 Teste de recuperação, 129 Teste de regressão, 16 Sistema de gerenciamento de banco de dados relacional, 196 Teste de confiabilidade, 127
 - Teste de usuário remoto, 151 Teste de tempo de resposta, 209 Tabela de decisão resultante, 76 Arredondamento erro amostra de código Java, 29 RUP. Veja o processo unificado do Rational

S

Schwaber, Ken, 176
 Scripts em
 testes de aplicativos móveis, 219
 Teste de segurança, 125
 Teste de manutenção/manutenção, 129
 Programas
 fluxograma de documentação, 116
 documentação de, 115
 especificações externas, 114 testes
 versus desenvolvimento, 117
 Desenvolvimento de software
 fluxograma de processo, 114
 Engenharia de confiabilidade de software
 (SRE), 128 definição correta de teste de software, 6 economia de, 8 definição errada, 5 princípios de teste de software, 12 SQL, 209 SRE. Consulte Teste de armazenamento de engenharia de confiabilidade de software, 126 Teste de estresse, 123, 206 Módulo Stub, 98
 Sutherland, Jeff, 176 Fluxograma de teste do sistema para, 121 propósito de, 116
 Teste do sistema, 119 instalação, 123 estresse, 123 volume, 123

T

TDMA, 218
 Caso de teste
 para testes extremos, 188
 Depuração de casos de teste, 167
 Projeto de caso de teste,
 teste de 41 módulos, teste de 86 unidades, 86
 Exame de caso de teste, 2
 Estratégia de caso de teste, 82
 Casos de teste
 categorias de tabela de, 122 tipos de, 167
 Critérios de conclusão do teste, 135
 Planejamento e controle de teste, 132
 Testar testes de usabilidade
 de seleção de usuários, 147
 Testes, 13, 44 de
 aceitação, 131 ágil, 178 ambiente ágil, 175
 big-bang, 98 cobertura de filiais, 44 compatibilidade de navegador, 204 camada de negócios, 205 orientados a código, 20 compatibilidade/conversão, 127 critérios de conclusão, 135 cobertura de condição, 45
 mascaramento de condição, 46 configuração, 126 depuração, 157 cobertura de decisão, 44 cobertura de decisão/condição, 46 testes de mesa, 21 erros de estimativa, 136
 humanos, 19 instalação, 127, 132

- Aplicativos de Internet, 193
aplicativos móveis, 213 de
cobertura de condições múltiplas, 47 não
incrementais, 98 de desempenho, 126 de
camada de apresentação, 203 de teste de
procedimento, 130 de recuperação, 129 de
confiabilidade, 127 de segurança, 125 de
facilidade de manutenção/manutenção, 129
de armazenamento, 126 de cima para
baixo, 101 usabilidade, 125, 143 questionário
de usabilidade, 152
- aplicativos da Web, 194
- O teste aborda aplicativos
móveis, 219
- Diretores de teste, 13
- Estratégias de teste
aplicativos da Internet, 200
- Pense em voz alta protocolo, 150
- Thomas, Dave, 176
- Acesso múltiplo por divisão de tempo, 218
- Design de cima para baixo, 101
- Desenvolvimento de cima para baixo, 101
- Teste de cima para baixo, 101
comparado com teste de baixo
para cima, 108
- Testes transacionais, 207
- Triângulo
Definição, 2
- você
- IU, 218
- Testes unitários, 85
extremos, 185
casos de teste, 86
- com testes extremos, 184
- Tabela de requisitos de
tempo de atividade-horas por ano, 129
- Usabilidade
em testes de aplicativos móveis, 219
- Testes de usabilidade, 125, 143
testes de componentes, 153
realizando testes suficientes, 153 métodos
de coleta de dados, 150 determinando o
número de testadores,
148
- rastreamento ocular,
151 erros de gráfico versus testadores,
149 testes de corredor, 147 questionários,
152 testes de usuários remotos, 151
seleção de usuários de teste, 147
considerações de teste, 144 o processo,
146 protocolo de pensamento em voz alta,
150
- Interface do usuário, 218
- Teste de usuário, 143
- V
- Van Bennekum, Áries, 176
- Teste de volume, 123
- C
- Passo a passo, 34
eficácia de, 21 WAP, 218
- compatibilidade de navegador
de aplicativos da Web, 195
testes de, 194 estratégias de teste,
200 testes de caixa branca, 10, 42
comparação de caixa branca-caixa
preta, 42 Wide Area Network. Ver

240 Índice

- Protocolo de aplicativo sem fio, 218
- Linguagem de marcação sem fio, 218
- WML, 218
- X
 - planejamento,
181 exemplo de fluxo de projeto,
183 testes, 183
 - Planejamento XP, 181
 - Fluxo do projeto XP, 183
 - Teste de XP, 183
 - XT, 180
- Xerox, 143
- EXP, 179, 180