Resumo

Livro 1: The Art of Software Testing

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. O software difundido 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 mais difícil de definir. Portanto, o software que escrevemos hoje afeta 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.

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 para desenvolvê-los.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 testes 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.

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.

A dificuldade em se fazer o teste se remete às linguagens orientadas a objetos, como Java e C++. 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.

Poderíamos 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.

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 e, em seguida, teste o programa para encontrar o maior número possível de erros.

Assim, uma definição mais apropriada a teste de software é:
Teste é o processo de executar um programa com a intenção de encontrar erros.

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.

Tendemos a selecionar dados de teste com baixa probabilidade de causar falhas no programa. Porém, se temos como objetivo demonstrar que um programa tem erros, nossos dados de teste terão maior probabilidade de encontrar erros.

O teste é um processo destrutivo. 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 em como os casos de teste (dados de teste) devem ser projetados e quem não deve testar um determinado programa.

Reforçar a definição adequada de teste é analisar o uso das palavras “bem-sucedido” e “fracasso” implicando o 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”.

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 mal sucedido é aquele que não examina adequadamente o software e, na maioria dos casos, um teste que não encontrou erros provavelmente seria considerado mal sucedido.

Um caso de teste que encontra um novo erro dificilmente pode ser considerado mal sucedido, provou ser um investimento valioso. Esse caso de teste mal sucedido é aquele que faz com que um programa produza o resultado correto sem encontrar nenhum erro.

Na segunda definição temos um problema:“teste é o processo de demonstrar que os erros não estão presentes''. É impossível de alcançar para praticamente todos os programas, mesmo programas triviais.

Já na terceira definição, o problema encontra-se: “teste é o processo de demonstrar que um programa faz o que deveria fazer”. É muito mais eficaz pensar que o teste de programa é como o processo de encontrar erros do que se o encararmos 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 assumida). Um caso de teste bem-sucedido é aquele que promove o progresso nessa direção, fazendo com que o programa falhe.

O próximo passo apropriado é determinar se é possível testar um programa para encontrar todos os seus erros. Em geral, é impraticável, muitas vezes impossível, encontrar todos os erros em um programa. Terá implicações para a economia do teste, suposições que o testador terá que fazer sobre o programa e a maneira pela qual 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,

Uma estratégia de teste importante é o teste de caixa preta (também conhecido como teste orientado a dados ou orientado a entrada/saída). Para entender melhor, você deve pensar em encontrar circunstâncias nas quais o programa não se comporte de acordo com suas especificações.

Aqui, os dados de teste são derivados apenas das especificações (ou seja, sem tirar vantagem do conhecimento da estrutura interna do programa). Pode usar o teste de caixa branca para encontrar todos os erros no programa, o critério é o teste de entrada exaustivo.

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 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. Esta discussão mostra que o teste de entrada exaustivo é impossível.

Assim, como testes exaustivos estão fora de questão, o objetivo deve ser maximizar o rendimento do investimento em testes, 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.

O teste de caixa branca (ou orientado por lógica), permite examinar a estrutura interna do programa. O objetivo neste ponto é estabelecer para esta estratégia o teste de entrada analógico para exaustivo na abordagem caixa preta.

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.

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. A ideia de que o teste de

caminho exaustivo significa um teste completo é falha, por três fatores:

- A primeira é que um teste de caminho exaustivo de forma alguma garante que um programa corresponda à sua especificação.

- Em segundo lugar, um programa pode estar incorreto devido à falta de caminhos. O teste exaustivo de caminhos não detectaria a ausência de caminhos necessários.

- Terceiro, um teste de caminho exaustivo pode não revelar erros de sensibilidade de dados.

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

Diretrizes de Teste do Programa Vital

1. Uma parte necessária de um caso de teste é uma definição da saída ou resultado esperado.
    - Em outras palavras, apesar da definição destrutiva adequada de teste, ainda há um desejo subconsciente de ver o resultado correto. Uma maneira de combater isso é encorajar um exame detalhado de todos os resultados especificando com precisão, com antecedência, a saída esperada do programa.
2. Um programador deve evitar tentar testar seu próprio programa.
    - Os programadores não podem testar efetivamente seus próprios programas porque 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. implica que o teste é mais eficaz e bem-sucedido se outra pessoa o fizer.
3. Uma organização de programação não deve testar seus próprios programas.
    - É difícil para uma organização de programação ser objetiva ao testar seus próprios programas, porque o processo de teste, se abordado com a definição adequada, pode ser visto como diminuindo a probabilidade de cumprir o cronograma e os objetivos de custo. Sendo mais econômico que os testes sejam realizados por uma parte objetiva e independente.

4. Qualquer processo de teste deve incluir uma inspeção completa dos resultados de cada teste.
    - Os erros encontrados em testes posteriores muitas vezes foram perdidos nos resultados de testes anteriores.
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 esperadas.
    - 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.
6. Examinar um programa para ver se ele não faz o que deveria fazer é apenas metade da batalha; a outra metade está vendo se o programa faz o que não deveria fazer.
    - Os programas devem ser examinados quanto a efeitos colaterais indesejados.
7. Evite casos de teste descartáveis, a menos que o programa seja realmente um programa descartável.
    - 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.
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 de uso da definição incorreta de teste - isto é, a suposição de que teste é o processo de mostrar que o programa funciona corretamente.
    - A definição de teste é o processo de execução de um programa com a intenção de encontrar erros.
9. A probabilidade da existência de mais erros em uma seção de um programa é proporcional ao número de erros já encontrados nessa seção.
    - Se uma seção específica de um programa parece ser muito mais propensa a erros do que outras seções, esse fenômeno nos diz que, em termos de rendimento do nosso investimento em testes, testes adicionais os esforços são mais bem concentrados nesta seção propensa a erros.

Livro 2: Teste de software eficaz e sistemático

Introdução

Todo desenvolvedor de software entende que falhas de software podem causar sérios danos a empresas, pessoas ou até mesmo à sociedade como um todo. E embora os desenvolvedores de software já tenham sido os principais responsáveis pela construção de sistemas de software, hoje eles também são responsáveis pela qualidade dos sistemas de software que produzem.

Aprendemos a usar o processo de escrever testes para refletir sobre o que os programas precisam fazer e obter feedback sobre o design do código (ou design de classe, se você estiver usando uma linguagem orientada a objetos). Também aprendemos que escrever código de teste é um desafio, e prestar atenção à qualidade do código de teste é fundamental para a evolução harmoniosa do conjunto de testes. E, finalmente, sabemos quais são os bugs comuns e como procurá-los.

Raramente aplicam técnicas de teste sistemáticas para explorar e encontrar bugs. Muitos profissionais argumentam que os testes são uma ferramenta de feedback e devem ser usados principalmente para ajudá-lo a se desenvolver. Embora isso seja verdade (e mostrarei ao longo deste livro como ouvir seu código de teste), os testes também podem ajudá-lo a encontrar bugs. Afinal, é disso que se trata o teste de software: *encontrar bugs* !

Meu objetivo com este livro é convencê-lo de que (1) como desenvolvedor, é *sua* responsabilidade garantir a qualidade do que você produz; (2) que os testes são as únicas ferramentas para ajudá-lo nessa responsabilidade; e (3) que, se você seguir uma coleção de técnicas, poderá testar seu código de maneira eficaz e sistemática.

Desenvolvedores que testam versus desenvolvedores que não testam

Testes de *bom tempo* : ou seja, testes que exercitam o comportamento válido do programa. Em testes baseados em propriedades, nosso objetivo é afirmar uma propriedade específica

E ter testes simples que expliquem rapidamente o comportamento do código de produção é sempre benéfico, mesmo que isso signifique ter um pouco de duplicação em seu conjunto de testes.

Teste de software eficaz para desenvolvedores

A menina do exemplo usou testes automatizados e casos de teste projetados de forma sistemática e eficaz. Ela dividiu os requisitos em pequenas partes e os usou para derivar casos de teste, aplicando uma técnica chamada *teste de domínio*. Quando ela foi feita com a especificação, ela se concentrou no código; e através *de testes estruturais* (ou código cobertura), ela avaliou se os casos de teste atuais eram suficientes.

Para alguns casos de teste, ela escreveu *testes baseados em exemplos* (ou seja, ela escolheu um único ponto de dados para um teste). Para um caso específico, ela usou *testes baseados em propriedades*, pois ajudou a explorar melhor possíveis bugs no código. Finalmente, ela refletiu frequentemente sobre os *contratos* e *pré* e *pós-condições* do método que estava desenvolvendo (embora no final ela tenha implementado um conjunto de verificações de validação e não pré-condições em si; Isso é o que chamo *de teste de software eficaz e sistemático para desenvolvedores* .

Testes eficazes como um processo iterativo

Um desenvolvedor pode estar testando rigorosamente uma classe e de repente perceber que uma decisão de codificação que tomou algumas horas atrás não era ideal. Eles então voltam e redesenham o código. Eles podem estar realizando ciclos TDD e perceber que o requisito não está claro sobre algo. O desenvolvedor então volta para a análise de requisitos para entender melhor as expectativas. Muito comumente, durante o teste, o desenvolvedor encontra um bug. Eles voltam ao código, corrigem e continuam testando. Ou o desenvolvedor pode ter implementado apenas metade do recurso, mas eles acham que seria mais produtivo testá-lo rigorosamente agora do que continuar a implementação.

Foco no desenvolvimento e depois no teste

Acho libertador focar separadamente no desenvolvimento e teste. Quando estou codificando um recurso, não quero me distrair com casos de canto obscuros.

O mito da “correção por design”

No entanto, a simplicidade está longe de ser suficiente. É ingênuo acreditar que o teste pode ser totalmente substituído pela simplicidade. O mesmo vale para “correção por design”: projetar bem seu código não significa evitar todos os possíveis bugs.

O custo do teste

Você pode estar pensando que forçar os desenvolvedores a aplicar testes rigorosos pode ser muito caro. É verdade: testar o software corretamente é mais trabalhoso do que não fazê-lo. Deixe-me convencê-lo por que vale a pena:

- O custo dos bugs que ocorrem na produção muitas vezes supera o custo da prevenção (como mostrado por Boehm e Papaccio, 1988).
- Equipes que produzem muitos bugs tendem a perder tempo em um loop eterno onde os desenvolvedores escrevem os bugs, os clientes (ou QAs dedicados) encontram os bugs, os desenvolvedores corrigem os bugs, os clientes encontram um conjunto diferente de bugs e assim por diante.
- A prática é fundamental.

O significado de efetivo e sistemático

Ser *eficaz* significa que nos concentramos em escrever os testes certos. Ser *sistemático* significa que, para um determinado trecho de código, qualquer desenvolvedor deve criar o mesmo conjunto de testes.

O papel da automação de teste

A automação é fundamental para um processo de teste eficaz. Cada caso de teste que criamos aqui é posteriormente automatizado por meio de uma estrutura de teste como o JUnit. Deixe-me distinguir claramente entre o projeto do caso de teste e a execução do caso de teste . Depois que um caso de teste é escrito, uma estrutura o executa e mostra relatórios, falhas e assim por diante. Isso é tudo o que esses frameworks fazem. Seu papel é muito importante, mas o verdadeiro desafio no teste de software não é escrever código JUnit, mas projetar casos de teste decentes que possam revelar bugs. Projetar casos de teste é principalmente uma atividade humana e é o foco principal deste livro.

Princípios de teste de software (ou, por que o teste é tão difícil)

Uma visão simplista do teste de software é que, se queremos que nossos sistemas sejam bem testados, devemos continuar adicionando testes até que tenhamos o suficiente. Eu gostaria que fosse assim tão simples. Garantir que os programas não tenham bugs é praticamente impossível, e os desenvolvedores devem entender por que isso acontece.

Testes exaustivos são impossíveis

Não temos recursos para testar completamente nossos programas. Testar todas as situações possíveis em um sistema de software pode ser impossível mesmo se tivéssemos recursos ilimitados. Sabendo que testar tudo não é possível, temos que escolher (ou priorizar) o que testar. É por isso que enfatizo a necessidade de *testes eficazes* .

Saber quando parar de testar

Priorizar quais testes projetar é difícil. Criar poucos testes pode nos deixar com um sistema de software que não se comporta como pretendido (ou seja, está cheio de bugs). Por outro lado, criar teste após teste sem a devida consideração pode levar a testes ineficazes (e custar tempo e dinheiro). O foco deve ser sempre maximizar o número de bugs encontrados enquanto minimizamos os recursos que gastamos para encontrar esses bugs.

A variabilidade é importante (o paradoxo dos pesticidas)

Não existe bala de prata em testes de software. Em outras palavras, não existe uma única técnica de teste que você possa sempre aplicar para encontrar todos os bugs possíveis. Diferentes técnicas de teste ajudam a revelar diferentes bugs. Se você usar apenas uma única técnica, poderá encontrar todos os bugs que puder com essa técnica é nada mais.

Isso é conhecido como o *paradoxo dos pesticidas* : todo método que você usa para prevenir ou encontrar bugs deixa um resíduo de bugs mais sutis contra os quais esses métodos são ineficazes. Os testadores devem usar diferentes estratégias de teste para minimizar o número de bugs deixados no software.

Bugs acontecem em alguns lugares mais do que em outros

Como eu disse anteriormente, dado que testes exaustivos são impossíveis, os testadores de software precisam priorizar os testes que realizam. Ao priorizar os casos de teste, observe que os bugs não são distribuídos uniformemente. Empiricamente, nossa comunidade observou que alguns componentes apresentam mais bugs que outros.

Não importa qual teste você faça, nunca será perfeito ou suficiente. Como Dijkstra costumava dizer, “o teste de programa pode ser usado para mostrar a presença de bugs, mas nunca para mostrar sua ausência”. Em outras palavras, embora possamos encontrar mais bugs simplesmente testando mais, nossos conjuntos de testes, por maiores que sejam, nunca garantirão que o sistema de software esteja 100% livre de bugs.

O contexto é rei

O contexto desempenha um papel importante na forma como elaboramos os casos de teste. Por exemplo, criar casos de teste para um aplicativo móvel é muito diferente de criar casos de teste para um aplicativo da Web ou software usado em um foguete. Em outras palavras, o teste é dependente do contexto.

Verificação não é validação

Finalmente, observe que um sistema de software que funciona perfeitamente, mas não tem utilidade para seus usuários, não é um bom sistema de software. Como um revisor deste livro me disse: "A cobertura de código é fácil de medir; cobertura de requisitos é outra questão." Os testadores de software enfrentam essa falácia de ausência de erros quando se concentra apenas na verificação e não na validação.

A pirâmide de testes e onde devemos nos concentrar

Sempre que falamos sobre testes pragmáticos, uma das primeiras decisões que precisamos tomar é o nível em que testar o código. Para um nível de teste , quero dizer o nível de unidade , integração ou sistema .

Teste de unidade

Em algumas situações, o objetivo do testador é testar um único recurso do software, ignorando propositalmente as outras unidades do sistema. Este nível de teste oferece as seguintes vantagens:

- Os testes de unidade são rápidos.
    - Um teste de unidade geralmente leva apenas alguns milissegundos para ser executado. Testes rápidos nos permitem testar grandes porções do sistema em um pequeno período de tempo. Conjuntos de testes rápidos e automatizados nos dão feedback constante. Essa rede de segurança rápida nos faz sentir mais confortáveis e confiantes em realizar mudanças evolutivas no sistema de software em que estamos trabalhando.
- Os testes de unidade são fáceis de controlar.
    - Um teste de unidade testa o software fornecendo determinados parâmetros a um método e, em seguida, comparando o valor de retorno desse método com o resultado esperado. Esses valores de entrada e o valor do resultado esperado são fáceis de adaptar ou modificar no teste. Novamente, observe o exemplo identifyExtremes() e veja como foi fácil fornecer diferentes entradas e afirmar sua saída.
- Os testes de unidade são fáceis de escrever.
    - Eles não requerem uma configuração complicada ou trabalho adicional. Uma única unidade também costuma ser coesa e

pequena, facilitando o trabalho do testador. Os testes se tornam muito mais complicados quando temos bancos de dados, front-ends e serviços da Web juntos.

Quanto às desvantagens, deve-se considerar o seguinte:

- Os testes unitários carecem de realidade.
    - Um sistema de software raramente é composto por uma única classe. O grande número de classes em um sistema e sua interação podem fazer com que o sistema se comporte de forma diferente em sua aplicação real do que nos testes unitários. Portanto, testes unitários não representam perfeitamente a execução real de um sistema de software.
- Alguns tipos de bugs não são capturados.
    - Alguns tipos de bugs não podem ser detectados no nível de teste de unidade; eles só acontecem na integração dos diferentes componentes (que não são exercitados em um teste unitário puro). Pense em um aplicativo da Web que tenha uma interface do usuário complexa: você pode ter testado o back-end e o front-end completamente, mas um bug pode se revelar apenas quando o back-end e o front-end estiverem juntos. Ou imagine um código multithread: tudo pode funcionar no nível da unidade, mas os bugs podem aparecer quando os threads estiverem sendo executados juntos.

Curiosamente, um dos desafios mais difíceis no teste de unidade é definir o que constitui uma unidade. Uma unidade pode ser um método ou várias classes. Aqui está uma definição para teste de unidade que eu gosto, dada por RoyOsherove (2009): “Um teste de unidade é um pedaço de código automatizado que invoca uma unidade de trabalho no sistema. E uma unidade de trabalho pode abranger um único método, uma classe inteira ou várias classes trabalhando juntas para alcançar um único propósito lógico que pode ser verificado.”

Para mim, teste de unidade significa testar um (pequeno) conjunto de classes que não dependem de sistemas externos (como bancos de dados ou serviços da Web) ou qualquer outra coisa que eu não controle totalmente.

Teste de integração

Os testes de unidade se concentram nas menores partes do sistema. No entanto, testar componentes isoladamente às vezes não é suficiente. Isso é especialmente verdadeiro quando o código em teste ultrapassa as fronteiras do sistema e usa outros componentes (geralmente externos). O teste de integração é o nível de teste que usamos para testar a integração entre nosso código e partes externas.

O teste de integração visa testar vários componentes de um sistema juntos, focando nas interações entre eles em vez de testar o sistema como um todo. O teste de integração se concentra em duas partes: nosso componente e o componente externo. Escrever um teste desse tipo é menos complicado do que escrever um teste que passe por todo o sistema e inclua componentes com os quais não nos importamos.

Teste do sistema

Para obter uma visão mais realista do software e, assim, realizar testes mais realistas, devemos executar todo o sistema de software com todos os seus bancos de dados, aplicativos front-end e outros componentes. Quando testamos o sistema em sua totalidade, em vez de testar pequenas partes do sistema isoladamente, estamos testando o sistema

O teste do sistema, no entanto, tem suas desvantagens:

- Os testes de sistema geralmente são lentos em comparação com os testes de unidade. Imagine tudo o que um teste de sistema precisa fazer, incluindo iniciar e executar todo o sistema com todos os seus componentes. O teste também precisa interagir com o aplicativo real, e as ações podem levar alguns segundos. Imagine um teste que inicia um container com uma aplicação web e outro container com um banco de dados. Em seguida, ele envia uma solicitação HTTP para um serviço Web exposto por este aplicativo Web. Este serviço da web recupera dados do banco de dados e grava uma resposta JSON no teste. Obviamente, isso leva mais tempo do que executar um teste de unidade simples, que praticamente não possui dependências.
- Testes de sistema também são mais difíceis de escrever . Alguns dos componentes (como bancos de dados) podem exigir uma configuração complexa antes de serem usados em um cenário de teste. Pense em conectar, autenticar e garantir que o banco de dados tenha todos os dados exigidos por esse caso de teste. O código adicional é necessário apenas para automatizar os testes.
- Os testes do sistema são mais propensos a falhas . Um teste flaky apresenta comportamento errático: se você executá-lo, pode ser aprovado ou reprovado para a mesma configuração. Testes irregulares são um problema importante para equipes de desenvolvimento de software, e discutimos esse assunto no capítulo 10. Imagine um teste de sistema que exerça um aplicativo da web. Depois que o testador clica em um botão, a solicitação HTTP POST para o aplicativo da Web leva meio segundo a mais do que o normal (devido a pequenas variações que geralmente não controlamos em cenários da vida real). O teste não espera isso e, portanto, falha. O teste é executado novamente, o aplicativo da Web leva o tempo normal para responder e o teste é aprovado. Muitas incertezas em um teste de sistema podem levar a um comportamento inesperado.

Quando usar cada nível de teste

Com uma compreensão clara dos diferentes níveis de teste e seus benefícios, temos que decidir se investir mais em testes de unidade ou teste de sistema e determinar quais componentes devem ser testados por meio de teste de unidade e quais componentes devem ser testados por meio de teste de sistema.

O teste unitário está na base da pirâmide e tem a maior área. Isso significa que os desenvolvedores que seguem esse esquema favorecem o teste de unidade (ou seja, escrevem mais testes de unidade). Subindo no diagrama, o próximo nível é o teste de integração. A área é menor, indicando que, na prática, esses desenvolvedores escrevem menos testes de integração do que testes unitários. Dado o esforço extra que os testes de integração exigem, os desenvolvedores escrevem testes apenas para as integrações de que precisam. O diagrama mostra que esses desenvolvedores preferem testes de sistema menos do que testes de integração e têm ainda menos testes manuais.

Por que sou a favor de testes unitários?

Como eu disse, tende a favorecer o teste de unidade. Eu aprecio as vantagens que os testes de unidade me dão. Eles são fáceis de escrever, são rápidos, posso escrevê-los entrelaçados com o código de produção e assim por diante. Também acredito que o teste de unidade se encaixa muito bem com a maneira como os desenvolvedores de software trabalham