Se livre dos seus bugs: Testes unitários, de integração, E2E e boas práticas 🥰

Por Clara Beatriz Bonifácio, 08/01/2024

Introdução

O presente documento tem como finalidade explicar e exemplificar a utilização e importância de uma rotina bem feita e elaborada de testes - manuais e automatizados - dentro do ambiente corporativo. Sua finalidade vem de encontro ao aprimoramento e organização dos códigos elaborados bem como a supressão de bugs e retrabalho no futuro.

Tipos de Testes

Uma analogia que gosto bastante para os tipos de testes é uma que vi em um post do Kent Dodds. Basicamente testes e os tipos de testes são comparados com tinta e os pincéis usados para pintar uma parede. Imagine que você quer pintar a parede do seu quarto na cor da estação (Pantone me falou que é o "peach fuzz"). Para pintar toda a parede você pode usar um rolo grande, um pincel pequeno ou usar um jato de tinta. Usar apenas um provavelmente vai gerar um resultado ruim, ineficiente ou ambos. Se usar só o rolo grande, vai ter dificuldade de pintar os cantos e onde precisa de mais detalhes enquanto só usar o pincel fino demora demais, enquanto usar o jato de tinta relativamente seria mais custoso devido a materiais e maquinários próprios para realizar tal ação. Enfim, para dar cobertura de tinta adequada a sua parede, é necessário que use as ferramentas adequadas.

No mundo dos testes, temos os pincéis para cada nível de cobertura que queremos dar. Começando do mais fino ao mais grosso:

Testes Unitários: são testes que vão cobrir uma *unidade* de funcionamento da sua aplicação. O conceito de unidade é abstrato mas tipicamente vai ser um módulo com um propósito próprio, como um uma função ou um componente Button em React.

Nos testes unitários, normalmente tudo que podemos remover de conexões com o "mundo externo", vamos remover, e colocar valores de teste no lugar (*mocks* - https://www.alura.com.br/artigos/testes-com-mocks-e-stubs), logo a importância desse teste é justamente essa: poder verificar se a especificação que demos ao projeto, a uma função, método ou classe está sendo seguida e está produzindo aquele resultado que nós esperamos.

Testes de Integração: quando for necessário testar múltiplas *unidades* juntas para ver se integradas umas às outras funcionam corretamente podemos escrever testes de integração. Aqui já tentamos usar menos mocks (i.e. deixamos as conexões mais "reais" entre si) para assimilar-se a um uso mais real. Um exemplo de teste de integração é ver se um formulário de login funciona (e.g. se o componente Button junto com Input e Form vão chamar sua callback se os valores de entrada são válidos);

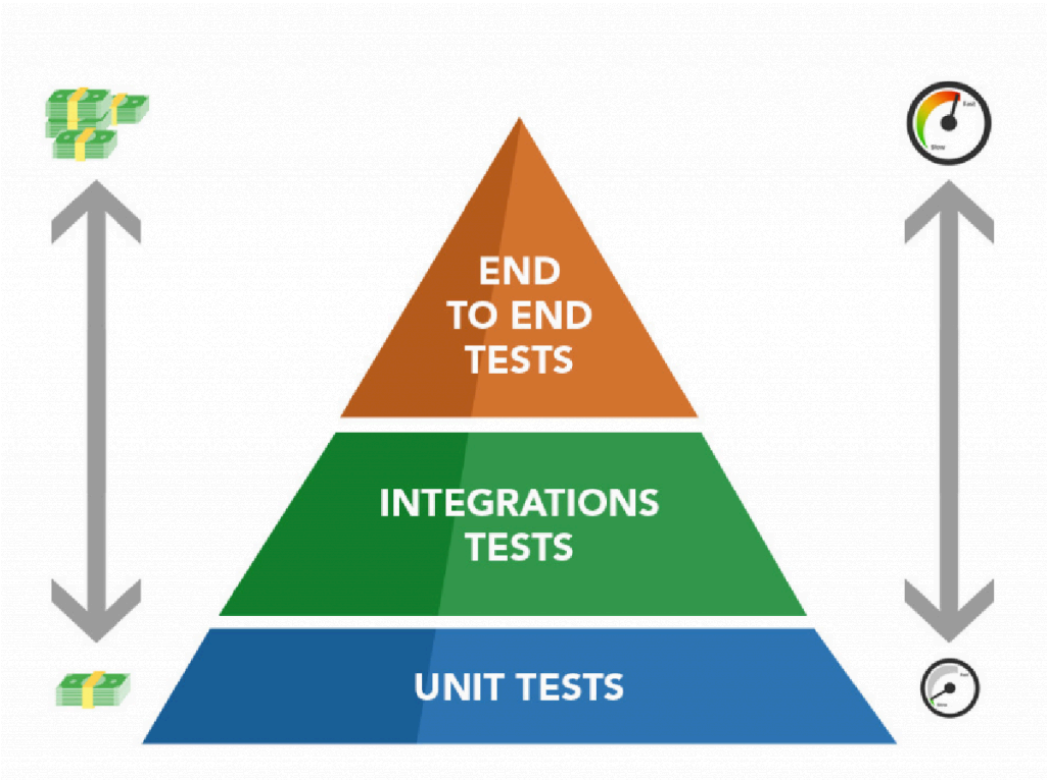
Testes Funcionais ou End-to-End (E2E): os rolos grandes, serão conhecidos como "End-to-End" porque testam de ponta a ponta (no pt-BR informal, "de cabo a rabo"). Esses testes vão testar se uma funcionalidade completa da sua aplicação está de acordo, idealmente sem nenhum mock utilizado. A ideia é replicar ao máximo a experiência do seu usuário, como se fosse um robô no lugar dele fazendo as mesmas tarefas e vendo se é possível ou não completar de forma correta. Um exemplo pode ser verificar se um usuário consegue entrar na página do perfil dele, detectar se aparece o formulário de login, entrar com os dados dele, fazer o login e ver se ele é roteado para a página privada /my-profile após isso tudo caso os dados estiverem corretos, ou se voltamos para /login caso contrário. Por consequência, o teste *End-to-End* é um teste que vai analisar o fluxo completo da nossa aplicação, então ele passará por todos os módulos e *stacks*, mas não direta e detalhadamente. Por

exemplo, no caso de um sistema web, chamaremos esse teste de alto nível, ou seja, vamos utilizar algum programa que clique na interface, que faça cadastros, que navegue entre as páginas que existem e consiga observar as respostas daquela página e do uso, como se fosse uma pessoa mesmo, e dizer se aquilo está se comportando da forma que esperamos, porque afinal estamos testando e queremos que ele siga um padrão, uma especificação determinada. Assim sendo, ele não vai observar muito o mérito da implementação em si do projeto. Ele verá se como um todo está funcionando e vai testar as funcionalidades que utilizamos e propomos ao usuário final. Particularmente, essa etapa acaba sendo uma das mais custosas para utilizar, porque para chegar ao teste *End-to-End* você já deve ter boa parte da aplicação funcionando, ou pelo menos deve ter um pedaço de cada área da aplicação funcionando para que você possa testá-la como um todo.

Repare que existe bastante mistura do que é um teste ou outro. É normal que se escreva um teste de integração que faz um pouco o papel de um teste unitário ou um teste E2E que fará uma pequena parte como um teste de integração, e não tem problema. Uma recomendação que comumente é sugerida é evitar que testes menores (i.e. unitário) façam o papel de testes maiores, justamente para manterem o escopo definido e para que mantenham seu ***custo*** baixo. Sim, é isso mesmo, cada teste tem custo.

O Custo de Testes

Cada teste possui um custo próprio. O custo de um teste não é tão somente monetário sendo exigido, esforço, o tempo necessário para elaborar, escrever e manter o teste e o custo do *tempo de execução* desse teste. O custo do tempo de criação e manutenção do teste é evidente — quanto mais testes forem criados, mais tempo irá investir neles.



Que tipo de teste tem maior custo: um teste unitário, um teste de integração ou teste E2E?

Os testes mais custosos de tempo de execução são os testes **E2E**. A razão por trás disso é simples: como um teste E2E vai testar um pedaço grande da aplicação, é necessário que a aplicação esteja em execução, significando que uma instância dela deverá "estar rodando", possivelmente um backend/API vai precisar ser iniciado também para ter as conexões funcionais, e se esse teste for ser executado numa pipeline de CI/CD*****, haverá a necessidade de instalar todas as dependências do seu projeto e provavelmente fazer um build da versão final para poder executá-lo naquele ambiente. Por isso que além do custo de tempo mencionei a possibilidade de custo monetário, uma vez que executar um teste em CI requer algum tipo de computação adicional, incluindo algum sistema de integração contínua — e isso vai te custar (tempo e alguns centavos de dólar aqui e ali).

*****CI/CD é a abreviação de Continuous Integration/Continuous Delivery, traduzindo para o português: integração e entrega contínuas. Trata-se de uma prática de desenvolvimento de software que visa tornar a integração de

código mais eficiente por meio de builds e testes automatizados. Com a abordagem CI/CD é possível entregar aplicações com mais frequência aos clientes. Para tanto, regras de automação são aplicadas nas etapas de desenvolvimento de apps.

Os principais conceitos atribuídos ao método são: integração, entrega e implantação contínuas. Com a prática CI/CD é possível solucionar os problemas que a integração de novos códigos pode causar às equipes de operações e desenvolvimento — o famoso "inferno de integração".

Especificamente, CI/CD aplica monitoramento e automação contínuos a todo o ciclo de vida das aplicações, incluindo as etapas de teste e integração, além da entrega e implantação. Juntas, essas práticas relacionadas são muitas vezes chamadas de "pipeline de CI/CD". Elas são compatíveis com o trabalho conjunto das equipes de operação e desenvolvimento que usam métodos ágeis, com uma abordagem de DevOps ou de engenharia de confiabilidade de sites (SRE).

Cobertura de Testes

Ouve-se muito o termo cobertura quando estamos falando de testes. Mas o que exatamente queremos dizer com isso?

Cobertura é uma métrica usada para saber quantos porcento (%) do nosso código está sendo testado. Normalmente usamos os testes unitários como base para esses cálculos. De forma genérica, a cobertura é a porcentagem do número de linhas que são testadas dividido pelo número total de linhas que existem no seu projeto, que resulta em um número. Esse cálculo pode ser menos ou mais preciso se a ferramenta de cobertura analisar individualmente as funções e ramificações do seu código e pode variar um tanto dependendo do método que escolher.

Numa escala de 0 a 100%, qual a porcentagem do seu código deve estar coberta por testes?

Apesar dessa ser uma resposta simples no primeiro momento, ela é um pouco mais complicada de responder pois depende de alguns fatores.

Vejamos, lembra que nós conversamos sobre o **custo** de testes agora pouco e concordamos que um teste tem um custo de criação, manutenção e execução. As leis da economia ditam que se algo possui um custo é natural que se tenha uma quantidade limitada desse item. Inclusive, para testes, existe um declínio do retorno de valor de adicionar testes após um dado ponto.

A cobertura de testes necessária e o ponto de declínio de retorno depende da especificação do projeto. Projetos críticos (hospitalares, aeroespaciais, etc.) normalmente terão coberturas maiores justamente para garantir que todos os "pedaços" estejam devidamente testados, inclusive com possíveis redundâncias e testes de casos extremamente improváveis mas que seriam fatais caso acontecessem.

No entanto, a maioria dos projetos não cai dentro dessas categorias e portanto o custo de testar excessivamente tem um valor agregado final bem alto. Afinal, de quantas formas diferentes podemos testar um componente Button no seu projeto React, ou qualquer outro framework front-end tal qual o OpenUI5, para garantir que ele funcione corretamente? Precisamos fazer um teste para garantir que ele continua funcionando num browser IE6 rodando no Windows 98 sem javascript numa madrugada de maio?

Apenas lembre-se que quanto mais teste seu projeto tiver, em geral mais *qualidade* terá, ao passo de do custo de maior tempo de criação e execução.

Teste manual vs. automatizado

Agora que entendemos alguns dos principais tipos de testes, vêm algumas questões que são muito comuns. O que são testes automatizados? Quais as vantagens e quando devemos utilizá-los?

Bem...a maioria dos testes podem ser executados de duas formas: manual ou automatizada.

O teste manual, como o próprio nome nos indica, é feito manualmente por um analista, desenvolvedor ou especialista em qualidade. Nessa situação, a pessoa responsável pelos testes irá executar cada passo necessário para que

o teste seja realizado com sucesso, sempre atento para as condições que o teste precisa para ser realizado da forma correta.

O teste manual costuma ter baixo valor de investimento e também permite que a pessoa que os realiza experimente condições semelhantes às do ambiente de produção, já que pode definir os parâmetros do teste manualmente.

Em compensação, testes manuais são mais lentos e como dependem totalmente da interação humana, sempre existe uma alta possibilidade de um problema passar despercebido por quem testa.

Já os testes automatizados nos trazem a praticidade de ter scripts, ferramentas como os mocks, citados neste artigo e técnicas que agilizam o processo. Eles nos ajudam a descobrir rapidamente se o sistema está com o desempenho esperado, e por serem automatizados, podem ser executados sem a necessidade de uma pessoa em todas as etapas de testes.

São mais confiáveis, já que são definidos por uma ferramenta ou scripts específicos; Assim o teste será executado automaticamente, sem interferência humana direta, diminuindo a possibilidade de erros passarem despercebidos.

Costumam ser mais caros, pois dependem de ferramentas específicas e o nível de automação que escolhemos influencia no tipo de ferramenta a ser utilizada, o que pode trazer mais custos. Além disso, existem problemas que apenas um testador humano poderá detectar, como os de usabilidade. Nesses casos não conseguimos utilizar a automatização de forma eficiente.

Por fim, fica a dúvida: Usar testes automatizados ou manuais? Essa é uma daquelas perguntas que têm como resposta um: depende...

O mais comum é que os dois tipos sejam utilizados em simultâneo, pois como vimos, temos vantagens e desvantagens em ambos e existem tipos de testes que preferencialmente serão automatizados, enquanto outros tendem a permanecer manuais, pois se faz necessária uma interação humana real, ou ainda por apresentarem um custo muito elevado.

Haveria, portanto, alguma porcentagem aproximada de testes a serem realizados para uma melhor cobertura de bugs a serem descobertos?

A partir das pesquisas feitas para este artigo em questão, um número genérico que tenta-se seguir quando não temos uma especificação melhor é **70%**. O raciocínio por trás disso é que 70% cobre a maioria do projeto, especialmente as partes mais críticas e importantes, sem excessivamente demandar que todo e qualquer pedaço do projeto possua cobertura.

É importante destacar que isso não significa que tem várias partes do seu projeto sem nenhum teste — isso significa que pode ter casos de uso dele sem teste. Por exemplo, se uma dada função se comporta de algumas maneiras mas pode gerar um valor diferente em alguns casos bem específicos e esses casos não são muito importantes ou são difíceis de testar mas são meio evidentes que vão funcionar, talvez o investimento em testar essa parte não vale o tempo. Mas, novamente, é necessário ter bastante zelo em analisar cada caso.

Outra recomendação *genérica* é limitar a quantidade de testes E2E que foram feitos. Como comentado antes, é possível que um teste E2E seja praticamente um teste unitário, dada suas capacidades. Mas lembre também que testes E2E são os mais custosos para executar, uma vez que o ambiente de execução inteiro estará de pé para isso. Consequentemente é recomendado procurar tentar cobrir as "rotas" ou casos de uso mais essenciais ou importantes da sua aplicação e ir adicionando esses testes gradativamente em outras partes, caso sentir que elas estejam pouco protegidas de erros ou mesmo para garantir que um bug que apareceu esteja consertado e não venha a ter regressão.

Distribuição de Testes

Já que estamos falando de cobertura e tipos de testes, uma pergunta que se pode fazer é em quais testes devemos dar mais importância.

Unitários porque validam as partes menores? Integração porque garantem bom comportamento entre as partes? Ou E2E que garantem que sua aplicação funcione da forma que você espera que seus usuários a usem?

Semelhante quando falamos sobre cobertura anteriormente, responder isso aqui de uma forma exata é difícil. Mas já que estamos apelando para genéricos, eu diria que uma distribuição saudável seria:

Testes Unitários: garanta que as unidades principais funcionem corretamente, teste todas as outras unidades com testes básicos, pode pular testes difíceis de realizar nas não-essenciais → **~30% dos esforços**

Testes de Integração: garanta que as combinações críticas estejam conversando adequadamente (i.e. sem elas, o usuário não conseguiria usar nada), evite *mockear* o que puder → **~50% dos esforços**

Testes E2E: garanta que os casos de uso principais estejam cobertos, adicione testes de funcionamento necessário ("smoke tests", ou seja, consiste em um **conjunto mínimo de testes para validar as principais funcionalidades**) → **~20% dos esforços**

Boas práticas ao executar testes nas aplicações

O TDD (Test Driven Development / Desenvolvimento orientado a teste) é parte da metodologia XP e também utilizado em diversas outras metodologias, além de poder ser utilizado livremente.

O que você tem a perder utilizando o TDD?

Segundo teóricos, como pessoas desenvolvedoras não temos nada a perder, a não ser os bugs que porventura ocorrem com certa frequência! Sendo assim, a criação de testes unitários ou de componentes é parte crucial para o TDD.

O TDD transforma o desenvolvimento, pois deve-se primeiro escrever os testes, antes de implementar o sistema. Os testes são utilizados para facilitar no entendimento do projeto, pois clareiam a ideia em relação ao que se deseja em relação ao código. Logo, os componentes individuais são testados para garantir que operem corretamente. Cada componente é testado independentemente, sem os outros componentes do sistema. Os componentes podem ser entidades simples, tais como funções ou classes de objetos, ou podem ser grupos coerentes dessas entidades.

Mas não é só o teste unitário que vai trazer o sucesso à aplicação, é necessário testar o sistema como um todo, no qual os componentes são

integrados para compor o sistema. Esse processo está relacionado com a busca de erros que resultam das interações não previstas entre os componentes.

Para tanto, um sistema é um conjunto de unidades integradas, por este motivo é importante os testes unitários para ver se no micromundo tudo funciona, mas também temos de testar a integração, ou seja, ao integrar dois ou mais componentes, devemos realizar testes para verificar se a integração funciona.

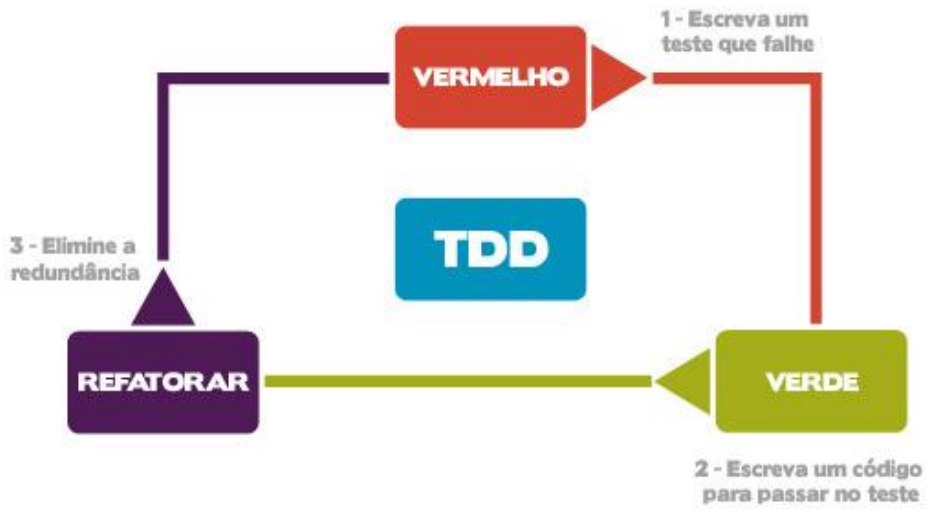
E qual o benefício em utilizar o TDD?

Em primeira instância, torna o processo mais confiável, reduz-se custos, pois desenvolvemos e já sabemos o erro, pois como os testes são criados antes do processo de desenvolvimento, conseguimos testar constantemente. Outro ponto é que se os testes foram criados, isso quer dizer que foram entendidas as regras de negócio durante a fase de desenvolvimento dos testes unitários.

Além disso, evita retrabalho da equipe, que ao final reduz custo e tem maior chance de sucesso.

O Ciclo do TDD é simples:

Criamos um teste -> Fazemos a codificação para passar no teste -> Refatorar seu código, conforme a figura abaixo:



Notamos aqui que o teste visa auxiliar a codificação, reduzindo consideravelmente os problemas na fase de desenvolvimento. No TDD é indicado que o projeto de teste unitário ocorra antes da fase de codificação/implementação.

O Teste antes da codificação, ou test-first, primeiro define implicitamente tanto uma interface como uma especificação do comportamento para a funcionalidade que está sendo desenvolvida".

Note que ao criar o teste antes de implementar a unidade, são reduzidos problemas como mal entendimento de requisitos ou interfaces, pois como criar um teste se eu não sei o que devo testar?

Neste caso a pessoa desenvolvedora para implementar os testes iniciais, deve compreender com detalhes a especificação do sistema e as regras de negócio, só assim, será possível escrever testes para o sistema. Imagine o caso de querer testar um pneu criado para o carro, se não entendi que o pneu é redondo, por exemplo, criarei um teste para um pneu quadrado, não podendo ser realizado o teste. Desta forma, é de extrema importância, para o desenvolvedor, o entendimento dos requisitos do cliente. Além disso, não adianta criar testes que não validem o código como um todo para reduzir o tempo, é necessário criar testes para o conjunto completo de unidades, só assim o TDD vai funcionar como deve, devendo fornecer uma cobertura completa aos testes.

Além disso, os testes devem seguir o modelo F.I.R.S.T.

F (Fast) - Rápidos: devem ser rápidos, pois testam apenas uma unidade;

I (Isolated) - Testes unitários são isolados, testando individualmente as unidades e não sua integração;

R (Repeatable) - Repetição nos testes, com resultados de comportamento constante;

S (Self-verifying) - A auto verificação deve verificar se passou ou se deu como falha o teste;

T (Timely) - O teste deve ser oportuno, sendo um teste por unidade.

Atualmente, existem diversas ferramentas que analisam as coberturas de teste, podendo ser baixadas gratuitamente através da Internet.

Outra vantagem de possuir testes é a chamada regressão. Imagine que criamos os testes, fizemos o sistema e tudo foi entregue ao cliente, mas posteriormente o cliente pediu pequenas modificações no sistema, mas não nas regras de negócio. Os testes já prontos servirão para validar se as modificações não criaram problemas nas regras de negócio que já estavam em funcionamento. Este procedimento exige que testes unitários estejam prontos, aguardando serem reutilizados. Um teste de unidade deve garantir que a execução daquele trecho mínimo do código esteja correta.

Como implementar o processo de TDD ao desenvolvimento?

Para começar a desenvolver os primeiros testes, pode ser mais fácil a utilização de bibliotecas XUnit's que se aplicam tanto para front quanto back-end.

Existem diversas ferramentas que possibilitam esta prática, vamos a algumas:

JUnit: O JUnit é um framework de teste para Java, que permite a criação de testes unitários. Além disso, está disponível como plugin para os mais diversos IDES como Eclipse, NetBeans etc.

TesteNG: Outra ferramenta de teste unitária, disponível para Java;

PHPUnit: Framework XUnit para teste unitário em PHP, também é possível integrar aos IDE's assim como o JUnit;

SimpleTest: Outra ferramenta para realização de teste para PHP. Além de possibilitar os testes unitários, é possível realizar MOCKS e outros testes;

NUnit: Framework de teste no molde XUnit para a plataforma .NET;

Jasmine e jestJs: Framework para teste unitário de JavaScript;

CUnit: Ferramenta para os testes unitários disponível para Linguagem C;

PyUnit: Framework Xunit para testes na linguagem Python.

Depois, temos de iniciar a fase de desenvolvimento, criando primeiramente os testes. Nos sites das ferramentas existe documentação para saber como utilizá-las, mas é possível ver nestes tutoriais como utilizar algumas delas:

Introdução ao desenvolvimento guiado por teste: TDD com JUnit

Qualidade em desenvolvimento web: PHP com teste unitário

Utilizando o NUnit

Testes com Jasmine - melhore a qualidade do JavaScript

Como vimos, existem ferramentas para as mais diversas linguagens e plataformas, com integração a IDEs. E lembre-se: primeiro crie os testes para posteriormente efetuar a sua implementação e assim viver feliz sem os bugs! 😀