Testes Automatizados

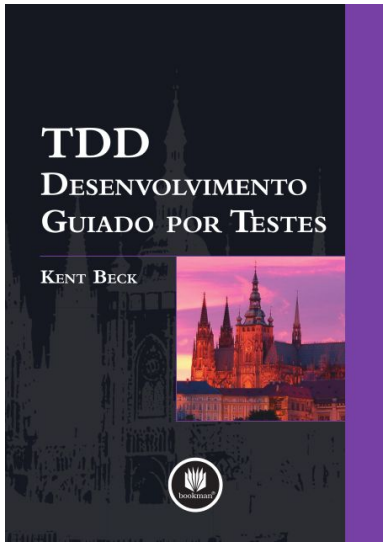
Maximizando a Qualidade e Eficiência.

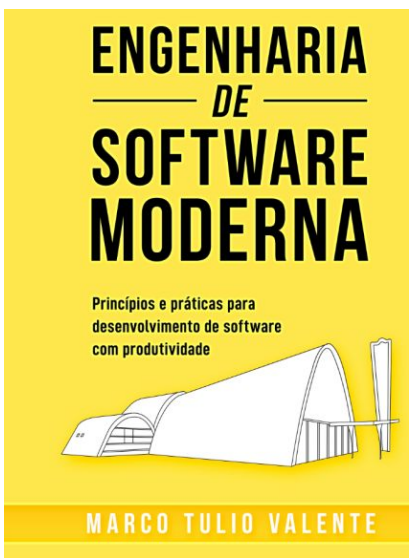
Olá

Eu sou Willian Brito

- Desenvolvedor FullStack na Msystem Software.
- Formado em Análise e Desenvolvimento de Sistemas.
- Pós Graduado em Segurança Cibernética.
- Certificação SYCP (Solyd Certified Pentester) v2018.

Este conteúdo foi baseado nestas obras





Tópicos Abordados

1 Introdução

Definição e importância dos testes automatizados

2 Pirâmide de Testes

Uma pirâmide de testes visa demonstrar níveis de quantidade, dificuldade e importância.

3 Princípios F.I.R.S.T

Propriedades que todos os de testes de unidades devem satisfazer.

4 Abordagens

Test-Driven Development (TDD), Behavior-Driven Development (BDD)

5 Cultura e Adoção

Educação e conscientização sobre a importância dos testes automatizados.

6 Outros Tipos de Testes

Testes de aceitação, regressão, carga e segurança.

“Antes de falarmos sobre **Testes Automatizados** vamos nos contextualizar sobre a **Engenharia de Software** e como construir software é uma tarefa complexa”

Engenharia de Software

No mundo moderno, tudo é software. Hoje em dia, por exemplo, empresas de qualquer tamanho dependem dos mais diversos sistemas de informação para automatizar seus processos. Governos também interagem com os cidadãos por meio de sistemas computacionais, por exemplo, para coletar impostos ou realizar eleições.

Empresas vendem, por meio de sistemas de comércio eletrônico, uma gama imensa de produtos, diretamente para os consumidores. Software está também embarcado em diferentes dispositivos e produtos de engenharia, incluindo automóveis, aviões, satélites, robôs, etc. Por fim, software está contribuindo para renovar indústrias e serviços tradicionais, como telecomunicações, transporte em grandes centros urbanos, hospedagem, lazer e publicidade.

Engenharia de Software

Portanto, devido a sua relevância no nosso mundo, não é surpresa que exista uma área da Computação destinada a investigar os desafios e propor soluções que permitam desenvolver sistemas de software principalmente aqueles mais complexos e de maior tamanho de forma produtiva e com qualidade. Essa área é chamada de **Engenharia de Software.**

Engenharia de Software trata da aplicação de abordagens sistemáticas, disciplinadas e quantificáveis para desenvolver, operar, manter e evoluir software. Ou seja, Engenharia de Software é a área da Computação que se preocupa em propor e aplicar princípios de engenharia na construção de software.

Engenharia de Software

Hoje, já se tem conhecimento de que software na maioria das vezes não deve ser construído em fases estritamente sequenciais, como ocorre com produtos tradicionais de engenharia, tais como Engenharia Civil, Engenharia Mecânica, Engenharia Eletrônica, etc. Já existem também padrões que podem ser usados por Engenheiros de Software em seus novos sistemas, de forma que eles não precisem reinventar a roda toda vez que enfrentarem um novo problema de projeto.

Bibliotecas e frameworks para os mais diversos fins estão largamente disponíveis, de forma que desenvolvedores de software podem reusar código sem se preocupar com detalhes inerentes a tarefas como implementar interfaces gráficas, criar estruturas de dados, acessar bancos de dados, criptografar mensagens, etc.

Engenharia de Software

Prosseguindo, as mais variadas técnicas de testes podem (e devem) ser usadas para garantir que os sistemas em construção tenham qualidade e que falhas não ocorram quando eles entrarem em produção e forem usados por clientes reais. Sabe-se também que sistemas envelhecem, como outros produtos de engenharia. Logo, software também precisa de manutenção, não apenas corretiva, para corrigir bugs reportados por usuários, mas também para garantir que os sistemas continuem fáceis de manter e entender, mesmo com o passar dos anos.

Desenvolvimento de software é diferente de qualquer outro produto de Engenharia, principalmente quando se compara software com hardware. **Frederick Brooks**, Prêmio Turing em Computação (1999) e um dos pioneiros da área de Engenharia de Software, foi um dos primeiros a chamar a atenção para esse fato.

Engenharia de Software

Em **1987**, em um ensaio intitulado **Não Existe Bala de Prata:** Essência e Acidentes em Engenharia de Software, ele discorreu sobre as particularidades da área de Engenharia de Software.

Segundo Brooks, existem dois tipos de dificuldades em desenvolvimento de software: **dificuldades essenciais** e **dificuldades acidentais**. As essenciais são da natureza da área e dificilmente serão superadas por qualquer nova tecnologia ou método que se invente. Daí a menção à bala de prata no título do ensaio. Diz a lenda que uma bala de prata é a única maneira de matar um lobisomem, desde que usada em uma noite de lua cheia. Ou seja, por causa das dificuldades essenciais, não podemos esperar soluções milagrosas em Engenharia de Software, na forma de balas de prata. O interessante é que, mesmo conhecendo o ensaio de Brooks, sempre surgem novas tecnologias que são vendidas como se fossem balas de prata.

Engenharia de Software

Segundo **Brooks**, as dificuldades essenciais são as seguintes:

1. Complexidade: dentre as construções que o homem se propõe a realizar, software é uma das mais desafiadoras e mais complexas que existe. Na verdade, como dissemos antes, mesmo construções de engenharia tradicional, como um satélite, uma usina nuclear ou um foguete, são cada vez mais dependentes de software.

2. Conformidade: pela sua natureza software tem que se adaptar ao seu ambiente, que muda a todo momento no mundo moderno. Por exemplo, se as leis para recolhimento de impostos mudam, normalmente espera-se que os sistemas sejam rapidamente adaptados à nova legislação. Brooks comenta que isso não ocorre, por exemplo, na Física, pois as leis da natureza não mudam de acordo com os caprichos dos homens.

Engenharia de Software

3. Facilidade de mudanças: que consiste na necessidade de evoluir sempre, incorporando novas funcionalidades. Na verdade, quanto mais bem sucedido for um sistema de software, mais demanda por mudanças ele recebe.

4. Invisibilidade: devido à sua natureza abstrata, é difícil visualizar o tamanho e consequentemente estimar o esforço de construir um sistema de software.

Engenharia de Software

As dificuldades (2), (3) e (4) são específicas de sistemas de software, isto é, elas não ocorrem em outros produtos de Engenharia, pelo menos na mesma intensidade. Por exemplo, quando a legislação ambiental muda, os fabricantes de automóveis têm anos para se conformar às novas leis. Adicionalmente, carros não são alterados, pelo menos de forma essencial, com novas funcionalidades, após serem vendidos.

Por fim, um carro é um produto físico, com peso, altura, largura, assentos, forma geométrica, etc., o que facilita sua avaliação e precificação por consumidores finais.

Ainda segundo Brooks, desenvolvimento de software enfrenta também dificuldades acidentais.

Engenharia de Software

Dificuldades acidentais, no entanto, elas estão associadas a problemas tecnológicos, que os Engenheiros de Software podem resolver, se devidamente treinados e caso tenham acesso às devidas tecnologias e recursos.

Como exemplo, podemos citar as seguintes dificuldades: um compilador que produz mensagens de erro obscuras, uma IDE que possui muitos bugs e frequentemente sofre travamentos, um framework que não possui documentação, uma aplicação Web com uma interface pouco intuitiva, etc. Todas essas dificuldades dizem respeito à solução adotada e, portanto, não são uma característica inerente dos sistemas mencionados.

Para ilustrar a complexidade envolvida na construção de sistemas de software reais, vamos dar alguns números sobre o tamanho desses sistemas, em linhas de código.

Engenharia de Software

Por exemplo, o sistema operacional Linux, em sua versão 4.1.3, de 2017, possui cerca de 25 milhões de linhas de código e contribuições de quase 1.700 engenheiros.

Para mencionar um segundo exemplo, os sistemas do Google somavam 2 bilhões de linhas de código, distribuídas por 9 milhões de arquivos, em janeiro de 2015.

Nesta época, cerca de 40 mil solicitações de mudanças de código (commits) eram realizadas, em média, por dia, pelos cerca de 25 mil Engenheiros de Software empregados pelo Google nessa época.

O que se Estuda em Engenharia de Software ?

Para responder a essa pergunta, vamos nos basear no **Guide to the Software Engineering Body of Knowledge**, também conhecido pela sigla **SWEBOK**.

Trata-se de um documento, organizado pela **IEEE Computer Society** (uma sociedade científica internacional), com o apoio de diversos pesquisadores e de profissionais da indústria.

O objetivo do SWEBOK é precisamente documentar o corpo de conhecimento que caracteriza a área que hoje chamamos de Engenharia de Software.

O que se Estuda em Engenharia de Software ?

O SWEBOK define 15 áreas de conhecimento em Engenharia de Software:

1: Requisitos de Software;
2: Projeto de Software;
3: Construção de Software;
4: Teste de Software;
5: Manutenção de Software;
6: Gerência de Configuração de Software;
7: Gerência da Engenharia de Software;
8: Processos de Engenharia de Software;
9: Ferramentas e Métodos da Engenharia de Software;
10: Qualidade de Software;
11: Práticas Profissionais em Engenharia de Software;
12: Economia da Engenharia de Software;
13: Fundamentos de Computação;
14: Fundamentos de Matemática;
15: Fundamentos de Engenharia;

Engenharia de Software

Este conteúdo será focado na área de conhecimento sobre testes automatizados, que é apenas uma, das 15 áreas de estudos da engenharia de software.

Os testes automatizados são uma parte fundamental do ciclo de desenvolvimento de software, proporcionando uma série de benefícios, que vamos ver a seguir.

1.

Introdução

Definição e importância dos testes automatizados

Introdução

Teste consiste na execução de um programa com um conjunto finito de casos, com o objetivo de **verificar** se ele possui o **comportamento esperado**.

A seguinte frase, bastante famosa, de **Edsger W. Dijkstra**, também prêmio Turing em Computação (1982), sintetiza não apenas os benefícios de testes, mas também suas limitações:

“Testes de software mostram a presença de bugs, mas não a sua ausência.”

Introdução

Pelo menos três pontos podem ser comentados sobre testes.

Primeiro, existem diversos tipos de testes. Por exemplo:

- **Testes de Unidade:** quando se testa uma pequena unidade o código, como uma classe.
- **Testes de Integração:** quando se testa uma unidade de maior granularidade, como um conjunto de classes e integração entre outros componentes como banco de dados e APIs.
- **Testes de End-To-End:** são uma maneira de realizar testes atravessando todas as camadas da arquitetura de um sistema. Os testes são realizados do início ao fim.

Introdução

Segundo, testes podem ser usados tanto para **verificação** como para **validação** de sistemas. Verificação tem como o objetivo garantir que um sistema atende à sua especificação. Já com validação, o objetivo é garantir que um sistema atende às necessidades de seus clientes.

A diferença entre os conceitos só faz sentido porque pode ocorrer de, a especificação de um sistema não expressar as necessidades de seus clientes. Por exemplo, essa diferença pode ser causada por um erro na fase de levantamento de requisitos; isto é, os desenvolvedores não entenderam os requisitos do sistema ou o cliente não foi capaz de explicá-los precisamente.

Introdução

Existem duas frases, muito usadas, que resumem as diferenças entre verificação e validação:

- **Verificação:** estamos implementando o sistema corretamente? Isto é, de acordo com seus requisitos.

- **Validação:** estamos implementando o sistema correto? Isto é, aquele que os clientes ou o mercado está querendo.

Assim, quando se realiza um teste de um método, para verificar se ele retorna o resultado especificado, estamos realizando uma atividade de verificação. Por outro lado, quando realizamos um teste funcional e de aceitação, ao lado do cliente, isto é, mostrando para ele os resultados e funcionalidades do sistema, estamos realizando uma atividade de validação.

Introdução

Terceiro, é importante definir e distinguir três conceitos relacionados a testes: **defeitos**, **bugs** e **falhas**. Para ilustrar a diferença entre eles, suponha o seguinte código para calcular a área de um círculo, dependendo de uma determinada condição:

```
if (condicao)
    area = pi * raio * raio * raio;
```

Esse código possui um defeito, pois a área de um círculo é PI vezes raio ao quadrado, e não ao cubo. Bug é um termo mais informal, usado com objetivos às vezes diversos. Mas o uso mais comum é como sinônimo de defeito.

Introdução

Uma falha ocorre quando um código com defeito for executado por exemplo, a condição do if do programa acima for verdadeira e, com isso, levar o programa a apresentar um resultado incorreto. Portanto, nem todo defeito ou bug ocasiona falhas, pois pode ser que o código defeituoso nunca seja executado.

Resumindo: código defeituoso é aquele que não está de acordo com a sua especificação. Se esse código for executado e de fato levar o programa a apresentar um resultado incorreto, dizemos que ocorreu uma falha.

Introdução

Na literatura sobre testes, às vezes são mencionados os termos **erro** e **falta (fault)**.

Quando isso ocorre, o significado é o mesmo daquele que adotamos para defeito.

Por exemplo, o IEEE Standard Glossary of Software Engineering Terminology define que falta é um passo, processo ou definição de dados incorretos em um programa de computador, os termos erro e bug são [também] usados para expressar esse significado. Resumindo, **defeito**, **erro**, **falta** e **bug** são **sinônimos**.

Falhas Históricas

❑ Foguete Ariane 5 (1996)

Existe uma lista enorme de falhas de software, com consequências graves, tanto em termos financeiros como de vidas humanas. Um dos exemplos mais famosos é a explosão do foguete francês Ariane 5, lançado em 1996, de Kourou, na Guiana Francesa.

Cerca de 30 segundos após o lançamento, o **foguete explodiu** devido a um comportamento inesperado de um dos sistemas de bordo, causando um prejuízo de cerca de **meio bilhão de dólares**. Interessante, o defeito que causou a falha no sistema de bordo do Ariane 5 foi bem específico, relativamente simples e restrito a poucas linhas de código, implementadas na linguagem de programação ADA, até hoje muito usada no desenvolvimento de software militar e espacial.

Falhas Históricas

Essas linhas eram responsáveis pela conversão de um número real, em ponto flutuante, com 64 bits, para um número inteiro, com 16 bits. Durante os testes e, provavelmente, lançamentos anteriores do foguete, essa conversão sempre foi bem sucedida: o número real sempre cabia em um inteiro.

Porém, na data da explosão, alguma situação nunca testada previamente exigiu a conversão de um número maior do que o maior inteiro que pode ser representado em 16 bits, causando um **integer overflow**.

Com isso, gerou-se um resultado incorreto, que fez com que o sistema de controle do foguete funcionasse de forma inesperada, causando a explosão.

Falhas Históricas

❑ Máquina medicinal (1985)

Em 1985 quando um máquina de radiação canadense **Therac-25** irradiou **doses letais** em pacientes, causando três mortos e três seriamente feridos.

A causa foi um bug sutil chamado de race condition (condição de corrida), um técnico acidentalmente configurou o Therac-25 de modo que o feixe de elétrons seria como um fogo de alta potência.

Falhas Históricas

Bug do Milênio (1999)

O desastre de um homem é a fortuna de outro, como demonstra o Bug do Milênio. Empresas gastaram cerca de **500 bilhões** com programadores para corrigir uma falha no software legado. Embora nenhuma falha significativa ocorreu, a preparação para o Bug do Milênio teve um custo significativo e impacto no tempo em todas as indústrias que usam a tecnologia computacional.

A Causa desse problema foi que ao economizar espaço de armazenamento de computador, softwares legados muitas vezes armazenavam anos para datas com números de dois dígitos, como 99 para 1999. Esses softwares também interpretavam 00 para significar 1900, em vez de 2000, por isso, quando o ano de 2000 veio, bugs apareceriam.

Beneficios

Os **testes automatizados** são uma parte fundamental do ciclo de desenvolvimento de software, proporcionando uma série de benefícios, incluindo:

- Redução de erros humanos.
- Detecção precoce de bugs.
- Aceleração do processo de desenvolvimento.
- Garantia da estabilidade e funcionamento do software.

Automátizados X Manuais

Software é uma das construções humanas mais complexas, como discutimos anteriormente. Portanto, é compreensível que sistemas de software estejam sujeitos aos mais variados tipos de erros e inconsistências.

Para evitar que tais erros cheguem aos usuários finais e causem prejuízos de valor incalculável, é fundamental introduzir atividades de teste em projetos de desenvolvimento de software.

De fato, teste é uma das práticas de programação mais valorizadas hoje em dia, em qualquer tipo de software. É também uma das práticas que sofreram mais transformações nos anos recentes.

Automátizados X Manuais

Quando o desenvolvimento era em cascata, os testes ocorriam em uma fase separada, após as fases de levantamento de requisitos, análise, projeto e codificação.

Além disso, existia uma equipe separada de testes, responsável por verificar se a implementação atendia aos requisitos do sistema.

Para garantir isso, frequentemente os testes eram manuais, isto é, uma pessoa usava o sistema, informava dados de entrada e verificava se as saídas ou resultado eram aquelas esperadas.

Assim, o objetivo de tais testes era apenas detectar bugs, antes que o sistema entrasse em produção.

Automátizados X Manuais

Com métodos ágeis, a prática de testes de software foi profundamente reformulada, conforme explicamos a seguir:

- Grande parte dos testes passou a ser automatizada, isto é, além de implementar as classes de um sistema, os desenvolvedores passaram a implementar também código para testar tais classes. Assim, os programas tornaram-se auto-testáveis.

- Testes não são mais implementados após todas as classes de um sistema ficarem prontas. Muitas vezes, eles são implementados até mesmo antes dessas classes.

Automátizados X Manuais

- Não existem mais grandes equipes de testes ou elas são responsáveis por testes específicos. Em vez disso, o desenvolvedor que implementa uma classe também deve implementar os seus testes.

- Testes não são mais um instrumento exclusivo para detecção de bugs. Claro, isso continua sendo importante, mas testes ganharam novas funções, como garantir que uma classe continuará funcionando após um bug ser corrigido em uma outra parte do sistema. E testes são também usados como documentação para o código de produção.

Automátizados X Manuais

Essas transformações tornaram testes uma das práticas de programação mais **valorizadas** em desenvolvimento moderno de software.

É nesse contexto que devemos entender a frase de Michael Feathers que abre o nosso próximo assunto:

“Se um código não é acompanhado de testes, ele pode ser considerado de baixa qualidade ou até mesmo um código legado.”

Na próxima seção , vamos focar em **testes automatizados** e o conceito de **pirâmide de testes**, pois testes manuais dão muito trabalho, são demorados e caros. Pior ainda, eles devem ser repetidos toda vez que o sistema sofrer uma modificação.

2. Pirâmide de Testes

Uma pirâmide de testes visa demonstrar níveis de quantidade, dificuldade e importância.

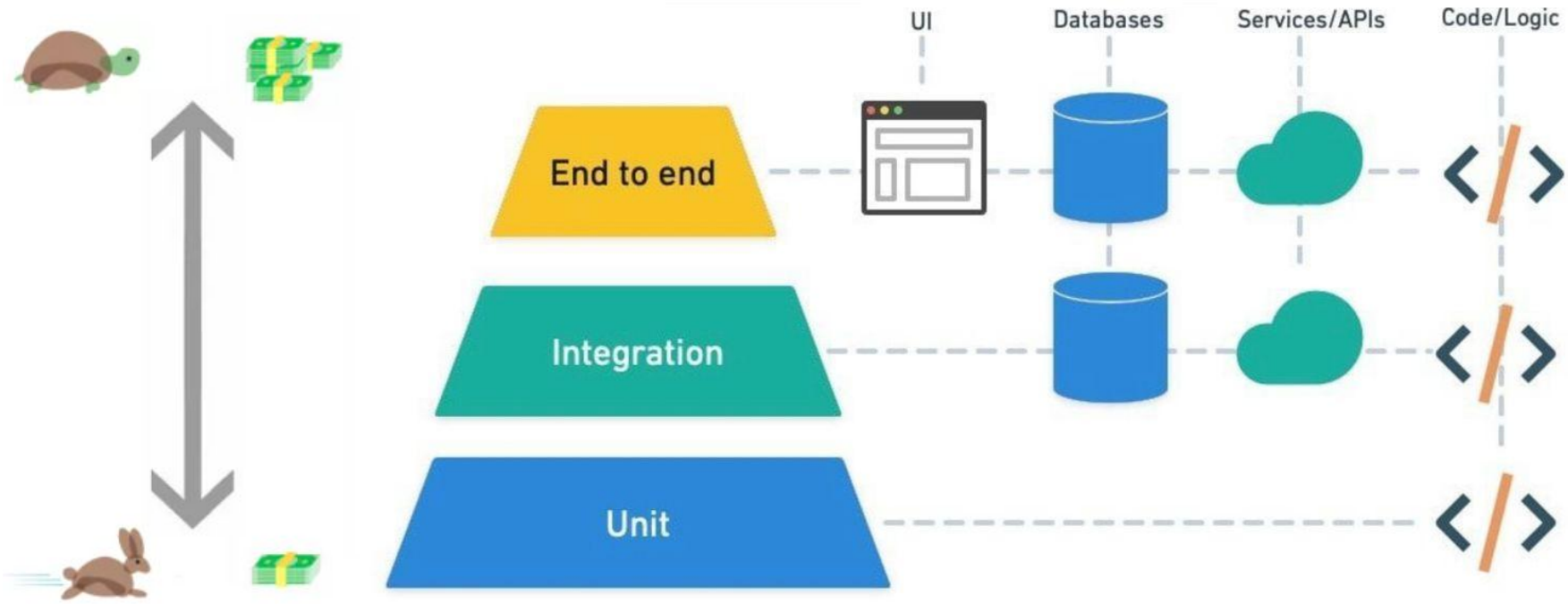
Pirâmide de Testes

Uma forma interessante de classificar testes automatizados é por meio de uma pirâmide de testes, originalmente proposta por Mike Cohn.

Os testes são divididos em três grupos, testes de unidade, testes de integração e testes de interface ou end-to-end.

Como mostra a próxima figura, essa pirâmide particiona os testes de acordo com sua granularidade.

Pirâmide de Testes



Pirâmide de Testes

Testes de Unidade verificam automaticamente pequenas partes de um código, normalmente uma classe apenas (acompanhe também pelas figuras da próxima página). Eles formam a base da pirâmide, ou seja, a maior parte dos testes estão nessa categoria. Testes de unidade são simples, mais fáceis de implementar e executam rapidamente.

No próximo nível, temos **Testes de Integração** ou **Testes de Serviços**, que verificam uma funcionalidade ou transação completa de um sistema. Logo, são testes que usam diversas classes, de pacotes distintos, e podem ainda testar componentes externos, como bancos de dados. Testes de integração demandam mais esforço para serem implementados e executam de forma mais lenta.

Pirâmide de Testes

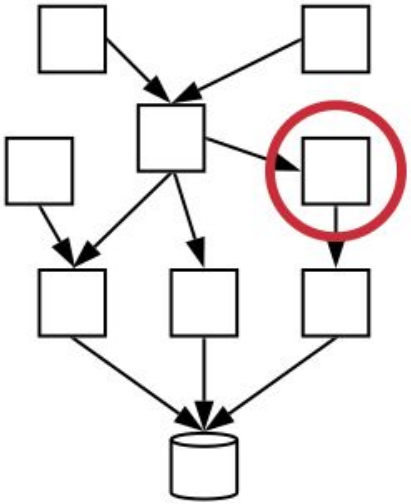
Por fim, no topo da pirâmide, temos os **testes de sistema**, também chamados de **testes de interface** com o usuário. Eles simulam, da forma mais fiel possível, uma sessão de uso do sistema por um usuário real.

Como são **testes de ponta a ponta (end-to-end)**, eles são mais caros, mais lentos e menos numerosos. Testes de interface costumam ser também frágeis, isto é, mínimas alterações nos componentes da interface podem demandar modificações nesses testes.

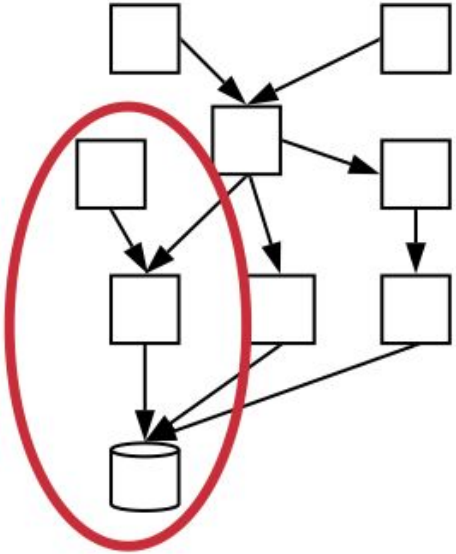
Pirâmide de Testes

- **Escopo de testes:**

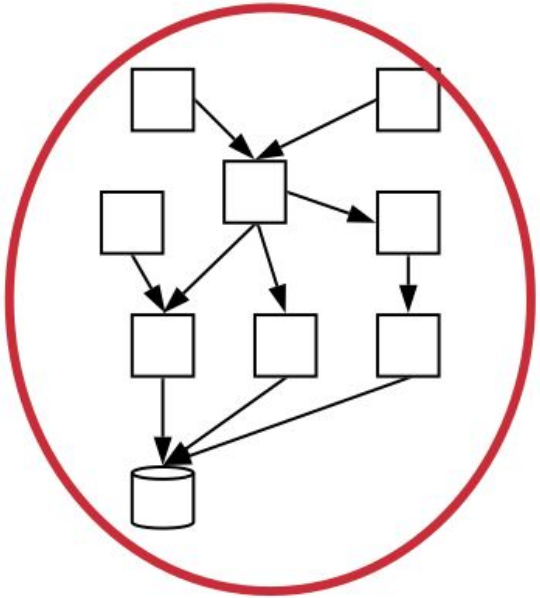
Unidade



Integração



End-to-End



Pirâmide de Testes

Uma recomendação genérica é que esses três testes sejam implementados na seguinte proporção: **70%** como **testes de unidades**, **20%** como **testes de integração** e **10%** como **testes de sistema**. Agora, vamos estudar os três tipos de testes da pirâmide de testes. O espaço que dedicaremos a cada teste também será compatível com seu espaço na pirâmide. Ou seja, falaremos mais de testes de unidade do que de testes de sistema, pois os primeiros são muito mais comuns.

Antes de começar de fato, gostaríamos de relembrar alguns conceitos que apresentamos na Introdução. Onde um código possui um defeito ou um bug, de modo mais informal quando ele não está de acordo com a sua especificação. Se um código com defeito for executado e levar o programa a apresentar um resultado ou comportamento incorreto, dizemos que ocorreu uma falha .

Testes de Unidade

Testes de Unidade são testes automatizados de pequenas unidades de código, normalmente classes, as quais são testadas de forma isolada do restante do sistema.

Um teste de unidade é um programa que chama métodos de uma classe e verifica se eles retornam os resultados esperados.

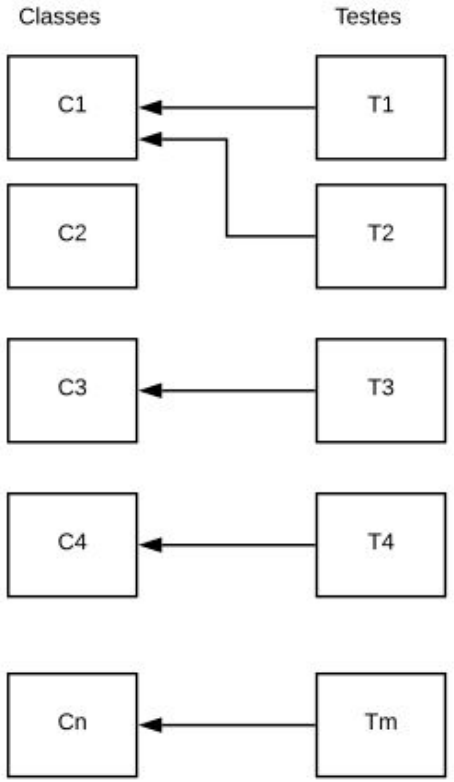
Assim, quando se usa testes de unidades, o código de um sistema pode ser dividido em dois grupos: um conjunto de classes que implementam os requisitos do sistema e um conjunto de testes, conforme ilustrado na próxima figura.

Testes de Unidade

Correspondência entre classes e testes.

A figura mostra um sistema com **n** classes e **m** testes. Como pode ser observado, não existe uma correspondência de 1 para 1 entre classes e testes. Por exemplo, uma classe pode ter mais de um teste.

É o caso da classe **C1**, que é testada por **T1** e **T2**. Provavelmente, isso ocorre porque C1 é uma classe importante, que precisa ser testada em diferentes contextos. Por outro lado, **C2** não possui testes, ou porque os desenvolvedores esqueceram de implementar ou porque ela é uma classe menos importante.



Testes de Unidade

Vamos para primeira implementação de testes de unidade. Imagine-se passeando em uma loja virtual qualquer na web.

Ao selecionar um produto, o sistema coloca-o no seu carrinho de compras. Ao finalizar a compra, o sistema fala com a operadora de cartão de crédito, retira o produto do estoque, dispara um evento para que a equipe de logística separe os produtos comprados e te envia um e-mail confirmando a compra.

O software que toma conta de tudo isso é complexo. Ele contém regras de negócio relacionadas ao carrinho de compras, ao pagamento, ao fechamento da compra.

Testes de Unidade

Mas, muito provavelmente, todo esse código não está implementado em apenas um único arquivo, esse sistema é composto por diversas pequenas classes, cada uma com sua tarefa específica. Desenvolvedores, quando pensam em teste de software, geralmente imaginam um teste que cobre o sistema como um todo. Um teste de unidade não se preocupa com todo o sistema, ele está interessado apenas em saber se uma pequena parte do sistema funciona.

Como disse anteriormente um teste de unidade testa uma única unidade do nosso sistema. Geralmente, em sistemas orientados a objetos, essa unidade é a classe. Em nosso sistema de exemplo, muito provavelmente existem classes como **“CarrinhoDeCompras”**, **“Pedido”**, e assim por diante. A ideia é termos baterias de testes de unidade separadas para cada uma dessas classes, cada bateria preocupada apenas com a sua classe.

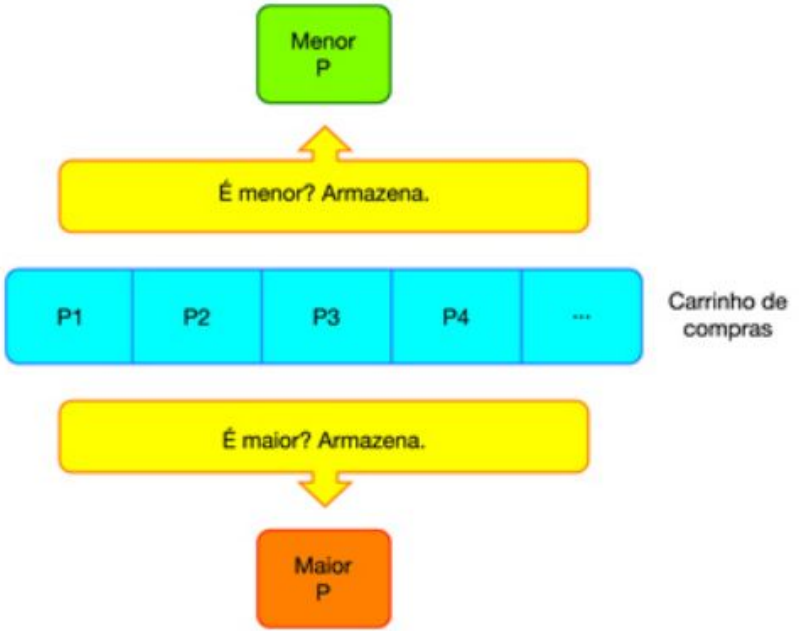
Testes de Unidade

Essa mesma loja virtual precisa encontrar, dentro do seu carrinho de compras, os produtos de maior e menor valor. Um possível algoritmo para esse problema seria percorrer a lista de produtos no carrinho, comparar um a um, e guardar sempre a referência para o menor e o maior produto encontrado até então.

```
public class MaiorEMenor
{
    4 references
    public Produto Menor { get; private set; }
    4 references
    public Produto Maior { get; private set; }

    1 reference
    public void Encontra(CarrinhoDeCompras carrinho)
    {
        foreach(Produto produto in carrinho.Produtos)
        {
            if(Menor == null || produto.Valor < Menor.Valor)
            {
                Menor = produto;
            }
            else if (Maior == null || produto.Valor > Maior.Valor)
            {
                Maior = produto;
            }
        }
    }
}
```

Testes de Unidade

Veja o método **Encontra()**. Ele recebe um **CarrinhoDeCompras** e percorre a lista de produtos, comparando sempre o produto corrente com o "menor e maior de todos". Ao final, temos nas propriedades Maior e Menor os produtos desejados. A figura mostra como o algoritmo funciona:



Testes de Unidade

Para exemplificar o uso dessa classe, veja o código abaixo:

```
var carrinho = new CarrinhoDeCompras();

carrinho.Adiciona(new Produto("Jogo de pratos", 70.0M));
carrinho.Adiciona(new Produto("Liquidificador", 250.0M));
carrinho.Adiciona(new Produto("Geladeira", 450.0M));

var algoritmo = new MaiorEMenor();

algoritmo.Encontra(carrinho);

Console.WriteLine($"\nO menor produto: {algoritmo.Menor.Nome}");
Console.WriteLine($"O maior produto: {algoritmo.Maior.Nome}\n");
```

Testes de Unidade

O carrinho contém três produtos: liquidificador, geladeira e jogo de pratos. É fácil perceber que o jogo de pratos é o produto mais barato (R$ 70,00), enquanto que a geladeira é o mais caro (R$ 450,00).

A saída do programa é exatamente igual à esperada:

```
O menor produto: Jogo de pratos
O maior produto: Geladeira
```

Apesar de aparentemente funcionar, se esse código for para produção, a loja virtual terá problemas.

Testes de Unidade

A classe **MaiorEMenor** respondeu corretamente ao teste feito acima, mas ainda não é possível dizer se ela realmente funciona para outros cenários. Observe o código abaixo:

```
var carrinho = new CarrinhoDeCompras();

carrinho.Adiciona(new Produto("Geladeira", 450.0M));
carrinho.Adiciona(new Produto("Liquidificador", 250.0M));
carrinho.Adiciona(new Produto("Jogo de pratos", 70.0M));

var algoritmo = new MaiorEMenor();

algoritmo.Encontra(carrinho);

Console.WriteLine($"\nO menor produto: {algoritmo.Menor.Nome}");
Console.WriteLine($"O maior produto: {algoritmo.Maior.Nome}\n");
```

Testes de Unidade

O código acima não é tão diferente do anterior. Os produtos, bem como os valores, são os mesmos, apenas a ordem em que eles são inseridos no carrinho foi trocada. Espera-se então que o programa produza a mesma saída. Mas, ao executá-lo, temos uma **exceção**.

```
O menor produto: Jogo de pratos
Unhandled exception. System.NullReferenceException: Object reference not set to an instance of an object.
```

Problema! Essa não era a saída esperada! Agora está claro que a classe **MaiorEMenor** não funciona bem para todos os cenários. Se os produtos, por algum motivo, forem adicionados no carrinho em ordem decrescente, a classe não consegue calcular corretamente.

Testes de Unidade

Uma pergunta importante para o momento é: será que o desenvolvedor, ao escrever a classe, perceberia esse bug? Será que ele faria todos os testes necessários para garantir que a classe realmente funciona?

Como discutido no capítulo anterior, equipes de software tendem a não testar software, pois o teste leva tempo e, por consequência, dinheiro. Na prática, equipes acabam por executar testes básicos, que garantem apenas o cenário feliz e mais comum.

Todos os problemas da falta de testes, que já foram discutidos anteriormente, agora fazem sentido. Imagine se a loja virtual colocasse esse código em produção. Quantas compras seriam perdidas por causa desse problema?

Testes de Unidade

Para diminuir a quantidade de bugs levados para o ambiente de produção, é necessário testar o código constantemente. Idealmente, a cada alteração feita, todo o sistema deve ser testado por inteiro novamente. Mas isso deve ser feito de maneira sustentável: é impraticável pedir para que seres humanos testem o sistema inteiro a cada alteração feita por um desenvolvedor.

A solução para o problema é fazer com que a máquina teste o software. A máquina executará o teste rapidamente e sem custo, e o desenvolvedor não gastaria mais tempo executando testes manuais, mas sim apenas em evoluir o sistema.

Testes de Unidade

Escrever um teste automatizado não é uma tarefa tão custosa. Ele, na verdade, se parece muito com um teste manual.

Imagine um desenvolvedor que deva testar o comportamento do carrinho de compras da loja virtual quando existem dois produtos lá cadastrados: primeiramente, ele “clicaria em comprar em dois produtos”, em seguida “iria para o carrinho de compras”, e por fim, verificaria “a quantidade de itens no carrinho (deve ser 2)” e o “o valor total do carrinho (que deve ser a soma dos dois produtos adicionados anteriormente)”.

Ou seja, de forma generalizada, o desenvolvedor primeiro pensa em um **cenário** (dois produtos comprados), depois executa uma **ação** (vai ao carrinho de compras), e por fim, **valida** a saída (vê a quantidade de itens e o valor total do carrinho).

Testes de Unidade

A figura abaixo mostra os passos que um testador geralmente faz quando deseja testar uma funcionalidade.

Um teste automatizado é similar. Ele descreve um cenário, executa uma ação e valida uma saída. A diferença é que quem fará tudo isso será a máquina, sem qualquer intervenção humana.

Testes de Unidade

De certa forma, um teste automatizado já foi escrito neste capítulo.
Veja o código abaixo, escrito para testar superficialmente a classe **MaiorEMenor**:

```
var carrinho = new CarrinhoDeCompras();

carrinho.Adiciona(new Produto("Jogo de pratos", 70.0M));
carrinho.Adiciona(new Produto("Liquidificador", 250.0M));
carrinho.Adiciona(new Produto("Geladeira", 450.0M));

var algoritmo = new MaiorEMenor();

algoritmo.Encontra(carrinho);

Console.WriteLine($"\nO menor produto: {algoritmo.Menor.Nome}");
Console.WriteLine($"O maior produto: {algoritmo.Maior.Nome}\n");
```

Testes de Unidade

Veja que esse código contém, de forma automatizada, boa parte do que foi descrito em relação ao teste manual: ele monta um cenário (um carrinho de compras com 3 produtos), executa uma ação (invoca o método **Encontra()**), e valida a saída (imprime o maior e o menor produto).

E o melhor: uma máquina faz (quase) tudo isso. Ao rodar o código acima, a máquina monta o cenário e executa a ação sem qualquer intervenção humana. Mas uma ação humana ainda é requerida para descobrir se o comportamento executou de acordo. Um humano precisa ver a saída e conferir com o resultado esperado.

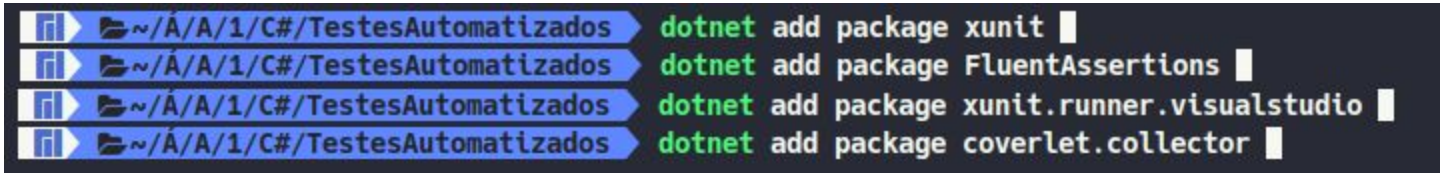
Testes de Unidade

É preciso que o próprio teste faça a validação e informe o desenvolvedor caso o resultado não seja o esperado.

Para melhorar o código acima, agora só introduzindo um framework de teste automatizado. Esse framework nos daria um relatório mais detalhado dos testes que foram executados com sucesso e, mais importante, dos testes que falharam, trazendo o nome do teste e a linha que apresentaram problemas.

Testes de Unidade

Neste conteúdo, faremos uso do **XUnit**, um framework de testes de unidade popular do mundo **.NET**. Para instalar execute os comandos abaixo no terminal integrado do seu VS Code.



Testes de Unidade

Para converter o código acima em um teste que o XUnit entenda, não é necessário muito trabalho.

A única parte que ainda não é feita 100% pela máquina é a terceira parte do teste: a validação. É justamente ali que invocaremos os métodos do XUnit.

Eles que farão a comparação do resultado esperado com o calculado. No XUnit, o método para isso é o Assert.Equal():

Testes de Unidade

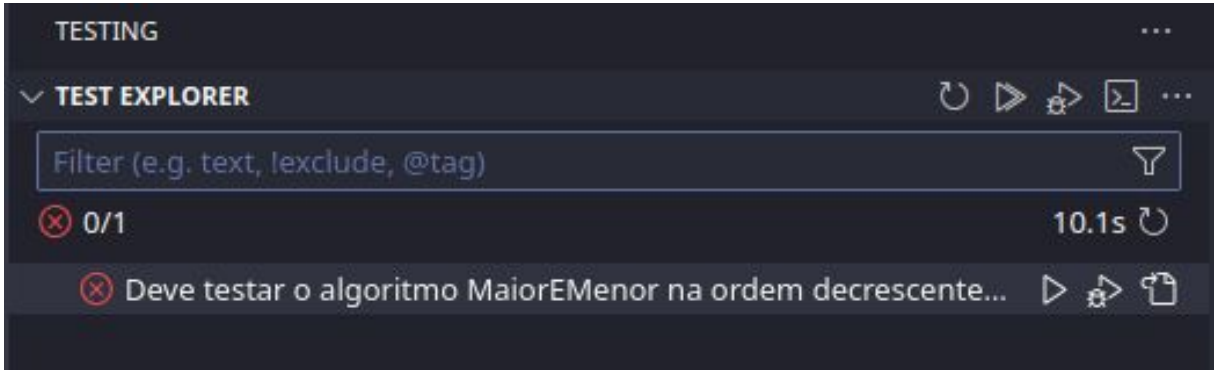
Pronto. Esse código ao lado é executado pelo **XUnit**.

Veja que criamos uma classe com o nome **MaiorEMenorTest** e colocamos o atributo **[Fact(DisplayName)]**, para adicionar uma descrição para aparecer no TestExplorer do VS Code.

```
using Xunit;

namespace TestesAutomatizados.Tests;

0 references
public class MaiorEMenorTest
{

    [Fact(DisplayName = "Deve testar o algoritmo MaiorEMenor na ordem decrescente")]
    0 references
    public void OrdemDecrescente()
    {
        var carrinho = new CarrinhoDeCompras();

        carrinho.Adiciona(new Produto("Geladeira", 450.0M));
        carrinho.Adiciona(new Produto("Liquidificador", 250.0M));
        carrinho.Adiciona(new Produto("Jogo de pratos", 70.0M));

        var algoritmo = new MaiorEMenor();
        algoritmo.Encontra(carrinho);

        Assert.Equal("Jogo de pratos", algoritmo.Menor.Nome);
        Assert.Equal("Geladeira", algoritmo.Maior.Nome);
    }
}
```

Testes de Unidade

Esse teste, como bem sabemos, falhará:



Testes de Unidade

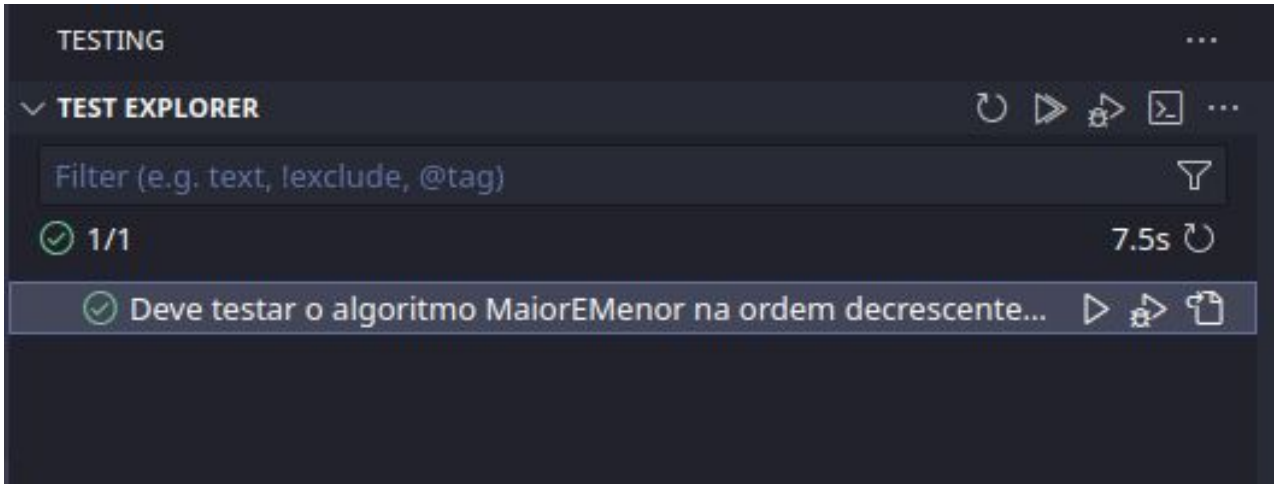
Para fazer o teste passar, é necessário corrigir o bug na classe de produção.

O que faz o código falhar é justamente a presença do else (ele fazia com que o código nunca passasse pelo segundo if, que verificava justamente o maior elemento).

```
public class MaiorEMenor
{
    5 references
    public Produto Menor { get; private set; }
    5 references
    public Produto Maior { get; private set; }

    2 references
    public void Encontra(CarrinhoDeCompras carrinho)
    {
        foreach(Produto produto in carrinho.Produtos)
        {
            if(Menor == null || produto.Valor < Menor.Valor)
            {
                Menor = produto;
            }

            if (Maior == null || produto.Valor > Maior.Valor)
            {
                Maior = produto;
            }
        }
    }
}
```

Testes de Unidade

Ao removê-lo, o teste passa:



Testes de Unidade

Repare também no tempo que a máquina levou para executar o teste: 7.5 segundos. Esse teste ainda foi demorado, pois minha máquina é um Manjaro Linux 23.1.2, Intel Core i3 de 4 geração lançado em 2014 e 8 GB de memória RAM, porém é mais rápido do que qualquer ser humano. Isso possibilita ao desenvolvedor executar esse teste diversas vezes ao longo do seu dia de trabalho.

E o melhor: se algum dia um outro desenvolvedor alterar esse código, ele poderá executar a bateria de testes automatizados existente e descobrir se a sua alteração fez alguma funcionalidade que já funcionava anteriormente parar de funcionar. Isso é conhecido como **testes de regressão**. Eles garantem que o sistema não regrediu.

Testes de Unidade

Ainda são necessários muitos testes para garantir que a classe **MaiorEMenor** funcione corretamente. Nesse momento, o único cenário testado são produtos em ordem decrescente. Muitos outros cenários precisam ser testados. Dentre eles:

- Produtos com valores em ordem crescente.
- Produtos com valores em ordem variada.
- Um único produto no carrinho.

Conhecendo o código da classe MaiorEMenor, é possível prever que os cenários acima funcionarão corretamente. Escrever testes automatizados para os cenários acima pode parecer inútil nesse momento.

Testes de Unidade

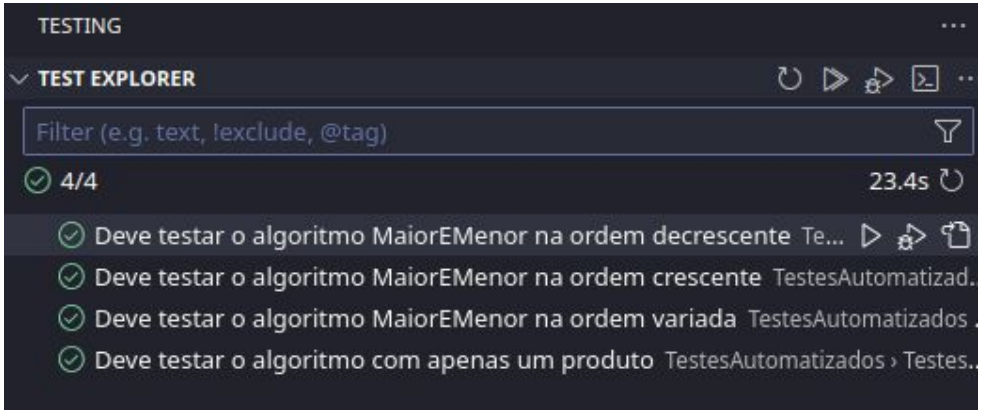
Entretanto, é importante lembrar que em códigos reais, muitos desenvolvedores fazem alterações nele. Para o desenvolvedor que implementa a classe, o código dela é bem natural e simples de ser entendido.

Mas para os futuros desenvolvedores que a alterarão, esse código pode não parecer tão simples.

É necessário então prover segurança para eles nesse momento. Portanto, apesar de alguns testes parecerem desnecessários nesse momento, eles garantirão que a evolução dessa classe será feita com qualidade.

Testes de Unidade

Veja que todos os testes passaram e demorou 23.4 segundos.



Esse tempo ainda é considerado um tempo muito longo, pois temos apenas 4 testes, vou utilizar apenas linha de comando para melhorar o tempo de execução.

Testes de Unidade

Agora repare que melhorou muito o tempo de execução dos testes, apenas 15 milissegundos, isso é bem mais rápido do que um exército de programadores testando o código.

Testes de Unidade

Até agora, escrever testes de unidade está bem fácil. E todos eles são bem parecidos.

O teste é basicamente um código que montava um cenário, instanciava um objeto, invocava um comportamento e verificava sua saída.

Mas será que é sempre fácil assim? Imagine o cenário em que nosso sistema deve gerar uma nota fiscal, porém esse processo geralmente é muito complicado, envolve uma série de cálculos de acordo com as diversas características do produto vendido.

Testes de Unidade

Mas, mais complicado ainda, pode ser o fluxo de negócio após a geração da nota, enviá-la para um sistema maior, como um SAP, enviar um e-mail para o cliente com a nota fiscal gerada, persistir as informações na base de dados, e assim por diante.

Testes de Unidade

De maneira simplificada, uma possível representação da **Nota Fiscal** e **Pedido** pode ser semelhante ao código abaixo:

```
1 reference
public class Pedido
{
    1 reference
    public string Cliente { get; private set; }
    1 reference
    public decimal ValorTotal { get; private set; }
    1 reference
    public int QuantidadeItens { get; private set; }

    0 references
    public Pedido(string cliente, decimal valorTotal, int quantidadeItens)
    {
        Cliente = cliente;
        ValorTotal = valorTotal;
        QuantidadeItens = quantidadeItens;
    }
}
```

Testes de Unidade

```
5 references
public class NotaFiscal
{
    1 reference
    public string Cliente { get; set; }
    2 references
    public decimal Valor { get; set; }
    1 reference
    public DateTime Data { get; set; }

    1 reference
    public NotaFiscal(string cliente, decimal valor, DateTime data)
    {
        Cliente = cliente;
        Valor = valor;
        Data = data;
    }
}
```

Testes de Unidade

Imagine que hoje esse processo consiste em gerar a nota, persistir no banco de dados e enviá-la por e-mail. Para o problema proposto, precisamos de 3 diferentes classes: uma classe responsável por enviar o e-mail, outra responsável por persistir as informações na base de dados, e por fim, uma classe responsável pelo cálculo do valor da nota fiscal.

Para facilitar o entendimento do exemplo, a implementação das classes que se comunicam com o banco de dados e com o SAP serão simplificadas. Veja o código ao lado:

```
5 references
public class SAP
{
    3 references
    public virtual void Envia(NotaFiscal nf)
    {
        // envia NF para o SAP
    }
}
```

```
5 references
public class NFDao
{
    3 references
    public virtual void Persiste(NotaFiscal nf)
    {
        // persiste NF
    }
}
```

Testes de Unidade

Agora é possível iniciar a escrita da classe **GeradorDeNotaFiscal**, responsável por calcular o valor final da nota e, em seguida, disparar a sequência do processo de negócio. Imagine que a regra para cálculo da nota fiscal seja simples: o valor da nota deve ser o valor do produto subtraído de 6%. Ou seja, 94% do valor total. Esse é um problema parecido com os anteriores. Começando pelo teste:

```
0 references
public class GeradorDeNotaFiscalTest
{
    [Fact(DisplayName = "Deve gerar nota fiscal com valor de imposto descontado")]
    0 references
    public void DeveGerarNFComValorDeImpostoDescontado()
    {
        var gerador = new GeradorDeNotaFiscal();
        var pedido = new Pedido("Willian", 1000, 1);
        var nf = gerador.Gera(pedido);

        Assert.Equal(1000 * 0.94M, nf.Valor);
    }
}
```

Testes de Unidade

Fazer o teste passar é trivial. Basta instanciarmos uma nota fiscal com 6% do valor a menos do valor do pedido. Além disso, a data da nota fiscal será a data de hoje:

```
1 reference
public class GeradorDeNotaFiscal
{
    1 reference
    public NotaFiscal Gera(Pedido pedido)
    {
        var nf = new NotaFiscal(pedido.Cliente, pedido.ValorTotal * 0.94M, DateTime.Now);

        return nf;
    }
}
```

Testes de Unidade

O teste passa. O próximo passo é persistir essa nota fiscal.

A classe **NFDao** já existe. Basta fazermos uso dela. Dessa vez, sem escrever o teste antes, vamos ver como ficaria a implementação final:

```
1 reference
public class GeradorDeNotaFiscal
{
    1 reference
    public NotaFiscal Gera(Pedido pedido)
    {
        var nf = new NotaFiscal(pedido.Cliente, pedido.ValorTotal * 0.94M, DateTime.Now);

        new NFDao().Persiste(nf);

        return nf;
    }
}
```

Testes de Unidade

A grande pergunta agora é: como testar esse comportamento? Acessar o banco de dados e garantir que o dado foi persistido? Não parece uma boa ideia.

❑ Mock Objects

A ideia de um teste de unidade é realmente testar a classe de maneira isolada, sem qualquer interferência das classes que a rodeiam. A vantagem desse isolamento é conseguir um maior **feedback** do teste em relação a classe sob teste, em um teste onde as classes estão integradas, se o teste falha, qual classe gerou o problema?

Testes de Unidade

Além disso, testar classes integradas pode ser difícil para o desenvolvedor. Em nosso exemplo, como garantir que o elemento foi realmente persistido? Seria necessário fazer uma consulta posterior ao banco de dados, garantir que a linha está lá, limpar a tabela para que na próxima vez que o teste for executado, os dados já existentes na tabela não atrapalhem o teste, e assim por diante.

É muito trabalho. E na verdade, razoavelmente desnecessário. Persistir o dado no banco de dados é tarefa da classe **NFDao**. A tarefa da classe **GeradorDeNotaFiscal** é somente invocar o DAO.

Não há porquê os testes da **GeradorDeNotaFiscalTest** garantirem que o dado foi persistido com sucesso, isso seria tarefa da classe **NFDaoTest**.

Testes de Unidade

Nesse caso, uma alternativa seria simular o comportamento do NFDao no momento do teste do gerador de nota fiscal. Ou seja, queremos criar um clone de NFDao ao banco de dados.

Classes que simulam o comportamento de outras são chamadas de **mock objects**, ou **objetos dublês.**

Um **framework de mock** facilita a vida do desenvolvedor que deseja criar classes falsas para seus testes.

Um mock pode ser bem inteligente. É possível ensinar o mock a reagir da maneira que queremos quando um método for invocado, descobrir a quantidade de vezes que o método foi invocado pela classe de produção, e assim por diante.

Testes de Unidade

No caso de nosso exemplo, precisamos garantir que o método **dao.Persiste()** foi invocado pela classe **GeradorDeNotaFiscal**.

Nosso teste deve então criar o mock e garantir que o método esperado foi invocado. Aqui utilizaremos o framework conhecido como **Moq**.

Com ele, criar mocks e validar o esperado é fácil, para instalar realize o comando abaixo no terminal integrado do VS Code.

Testes de Unidade

Sua API é bem simples e fluente. Veja o teste abaixo, onde criamos o mock e depois validamos que o método foi invocado:

```
[Fact(DisplayName = "Deve persistir nota fiscal gerada")]
0 references
public void DevePersistirNFGerada()
{
    // criando o mock
    var dao = new Mock<NFDao>();

    var gerador = new GeradorDeNotaFiscal();
    var pedido = new Pedido("Willian", 1000, 1);
    var nf = gerador.Gera(pedido);

    /// Configurar o comportamento do método Persiste
    dao.Setup(t => t.Persiste(It.IsAny<NotaFiscal>()));

    // verificando que o método foi invocado
    dao.Verify(t => t.Persiste(nf));
}
```

Testes de Unidade

O problema é que se rodarmos o teste, ele falha. Isso acontece porque, apesar de termos criado o mock, o GeradorDeNotaFiscal não faz uso dele.

Repare que na implementação, ele instancia um NFDao. É necessário que o gerador seja mais esperto: ele deve usar o mock no momento do teste, e a classe NFDao quando o código estiver em produção.

Uma solução para isso é receber a classe pelo construtor. Dessa forma, é possível passar o mock ou a classe concreta. E em seguida, fazer uso do DAO recebido:

Testes de Unidade

```
3 references
public class GeradorDeNotaFiscal
{
    2 references
    private NFDao dao;

    2 references
    public GeradorDeNotaFiscal(NFDao dao)
    {
        this.dao = dao;
    }

    2 references
    public NotaFiscal Gera(Pedido pedido)
    {
        var nf = new NotaFiscal(pedido.Cliente, pedido.ValorTotal * 0.94M, DateTime.Now);

        dao.Persiste(nf);

        return nf;
    }
}
```

Testes de Unidade

Agora, basta passar o mock para a classe sob teste e o comportamento do DAO será simulado:

```
[Fact(DisplayName = "Deve persistir nota fiscal gerada")]
0 references
public void DevePersistirNFGerada()
{
    // criando o mock
    var dao = new Mock<NFDao>();

    var gerador = new GeradorDeNotaFiscal(dao.Object);
    var pedido = new Pedido("Willian", 1000, 1);
    var nf = gerador.Gera(pedido);

    /// Configurar o comportamento do método Persiste
    dao.Setup(t => t.Persiste(It.IsAny<NotaFiscal>()));

    // verificando que o método foi invocado
    dao.Verify(t => t.Persiste(nf));
}
```

Testes de Unidade

Podemos fazer a mesma coisa com os próximos passos da geração da nota fiscal, como enviar para o SAP, enviar por e-mail, e etc. Basta passar mais uma dependência pelo construtor, criar outro mock e garantir que o método foi enviado. Com o SAP, por exemplo:

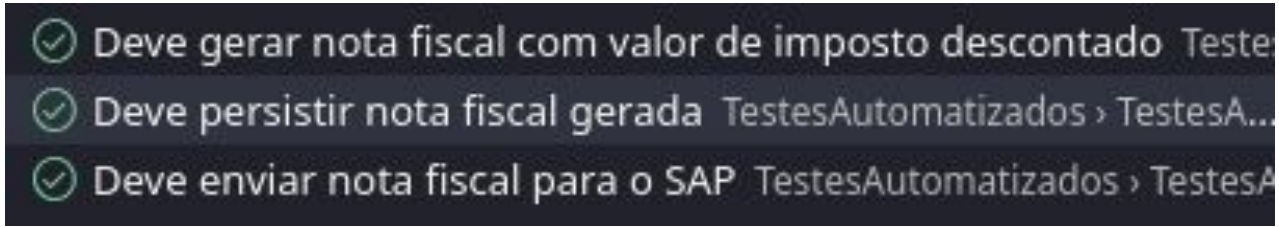
```
[Fact(DisplayName = "Deve enviar nota fiscal para o SAP")]
0 references
public void DeveEnviarNFGeradaParaSAP()
{
    var dao = new Mock<NFDao>();
    var sap = new Mock<SAP>();

    var gerador = new GeradorDeNotaFiscal(dao.Object, sap.Object);
    var pedido = new Pedido("Willian", 1000, 1);
    var nf = gerador.Gera(pedido);

    // Configurar o comportamento do método Envia
    sap.Setup(t => t.Envia(It.IsAny<NotaFiscal>()));

    // verificando que o método foi invocado
    sap.Verify(t => t.Envia(nf));
}
```

Testes de Unidade

```
4 references
public class GeradorDeNotaFiscal
{
    2 references
    private NFDao dao;
    2 references
    private SAP sap;

    3 references
    public GeradorDeNotaFiscal(NFDao dao, SAP sap)
    {
        this.dao = dao;
        this.sap = sap;
    }

    3 references
    public NotaFiscal Gera(Pedido pedido)
    {
        var nf = new NotaFiscal(pedido.Cliente, pedido.ValorTotal * 0.94M, DateTime.Now);

        dao.Persiste(nf);
        sap.Envia(nf);

        return nf;
    }
}
```

Testes de Unidade

Veja na imagem abaixo que todos os testes passaram.



Repare que por uma necessidade do teste, optamos por receber as dependências da classe pelo construtor. Isso é, na verdade, uma boa prática quando se pensa em orientação a objetos. Em primeiro lugar o desenvolvedor deixa as dependências da classe explícitas. Basta olhar seu construtor, e ver quais classes são necessárias para que ela faça seu trabalho por completo. Em segundo lugar, ao receber a dependência pelo construtor, a classe facilita sua extensão.

Testes de Unidade

Por meio de polimorfismo, o desenvolvedor pode, a qualquer momento, optar por passar alguma classe filho da classe que é recebida (ou, no caso de uma interface, outra classe a implementa), mudando/evoluindo seu comportamento.

Novamente retomo a frase de que um código fácil de ser testado possui características interessantes do ponto de vista de design.

Explicitar as dependências é uma necessidade quando se pensar em testes de unidade, afinal essa é a única maneira de se passar um mock para a classe.

Testes de Unidade

Desenvolvedores gastam toda sua vida automatizando processos de outras áreas de negócio, criando sistemas para RHs, controle financeiro, entre outros, com o intuito de facilitar a vida daqueles profissionais. Por que não criar software que automatize o seu próprio ciclo de trabalho?

Testes automatizados são fundamentais para um desenvolvimento de qualidade, e é obrigação de todo desenvolvedor escrevê-los. Sua existência traz diversos benefícios para o software, como o aumento da qualidade e a diminuição de bugs em produção.

Nas próximas seções, discutiremos sobre como aumentar ainda mais o feedback que os testes nos dão sobre a qualidade do software. Mas antes de falar sobre essa abordagem vamos dar continuidade aos próximos testes da pirâmide de testes.

Testes de Integração

Podemos escrever um teste de diferentes maneiras, de acordo com o que esperamos obter dele. Todos os testes que escrevemos até agora são conhecidos por testes de unidade.

Um teste de unidade é aquele que garante que uma classe funciona, de maneira isolada ao resto do sistema. Ou seja, testamos o comportamento dela sem se preocupar com o comportamento das outras classes. Uma vantagem dos testes de unidade é que eles são fáceis de serem escritos e rodam muito rápido.

A desvantagem deles é que eles não simulam bem a aplicação no mundo real. No mundo real, temos as mais diversas classes trabalhando juntas para produzir o comportamento maior esperado.

Testes de Integração

Se quisermos um teste que se pareça com o mundo real, ou seja, que realmente teste a aplicação do ponto de vista do usuário, é necessário escrever o que chamamos de teste de sistema.

Um teste de sistema é aquele que é idêntico ao executado pelo usuário da aplicação. Se sua aplicação é uma aplicação web, esse teste deve então subir o browser, clicar em links, submeter formulários, e etc.

A vantagem desse tipo de teste é que ele consegue encontrar problemas que só ocorrem no mundo real, como problemas de integração entre a aplicação e banco de dados, e etc. O problema é que eles geralmente são mais difíceis de serem escritos e levam muito mais tempo para serem executados.

Testes de Integração

No entanto, muitas vezes queremos testar não só uma classe, mas também não o sistema todo, queremos testar a integração entre uma classe e um sistema externo.

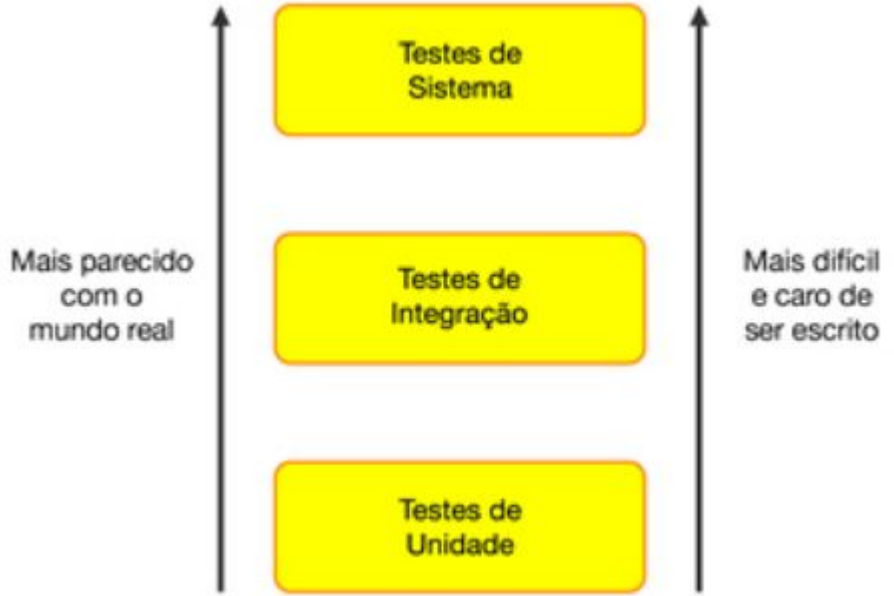
Por exemplo, classes DAO (responsáveis por fazer toda a comunicação com o banco de dados) devem ser testadas para garantir que as consultas SQL estão escritas corretamente, mas de maneira isolada as outras classes do sistema.

Esse tipo de teste, que garante a integração entre 2 pontos da aplicação, é conhecido por **teste de integração**.

Testes de Integração

O desenvolvedor deve fazer uso dos diferentes níveis de teste para garantir qualidade do seu sistema.

Mas deve sempre ter em mente que, quanto mais parecido com o mundo real, mais difícil e caro o teste será.



Testes de Integração

O primeiro passo para que um teste deixe de ser exclusivamente de unidade e passe a ser de integração é **não usar mocks** e passar dependências concretas para a classe sob teste.

Muitos desenvolvedores, inclusive, defendem que um bom teste nunca faz uso de mocks. O argumento deles é de que mocks acabam por "esconder" possíveis problemas que só seriam pegos na integração.

O argumento de sempre em relação a testes de unidade e testes de integração. O ponto não é descobrir se devemos ou não usar mocks, mas sim quando ou não usá-los.

Testes de Integração

Veja, esse de teste implementado para uma calculadora de salário:

```
[Test]
public void DeveCalcularSalarioParaDesenvolvedoresComSalarioAbaixoDoLimite()
{
    CalculadoraDeSalario calculadora = new CalculadoraDeSalario();
    Funcionario desenvolvedor = new Funcionario("Willian", 1500.0, Cargo.DESENVOLVEDOR);

    double salario = calculadora.CalculaSalario(desenvolvedor);
    Assert.AreEqual(1500.0 * 0.9, salario, 0.00001);
}
```

Esse teste garante o comportamento da classe **CalculadoraDeSalario**. Idealmente, gostaríamos de testá-la independente do comportamento das outras classes.

Testes de Integração

Mas veja que no teste, instanciamos um **Funcionario**. Usamos a classe concreta e não fizemos uso de mock. Por quê?

Geralmente classes que representam entidades, serviços, utilitários, ou qualquer outra coisa que encosta em infra estrutura, não são mockadas. Elas são classes puras e simples e mocká-las só irá dar mais trabalho ao desenvolvedor.

Ao tomar essa decisão, diminuímos a qualidade do retorno desse teste, afinal ele pode falhar não por culpa da calculadora de salário, mas sim por culpa do funcionário. Apesar de não ser o melhor dos mundos, é uma troca justa entre produtividade e feedback. Opte por mockar classes que lidam com **infra estrutura** e que tornariam seu teste muito complicado.

Testes de Integração

Por exemplo, relembre nosso **gerador de nota fiscal**. A nota era persistida em um banco de dados e depois enviada para o SAP.

Preparar tanto o banco quanto o SAP para receber o teste não é fácil. Portanto, simular a interação de ambas as classes é uma boa ideia.

Testes de Integração

Vamos começar testar as classes DAOs de nosso sistema, toda classe que lida com infra estrutura, testá-la é um pouco mais complicado.

É razoavelmente óbvio que para se testar um DAO, é necessário comunicar-se de verdade com um banco de dados real. Nada de simulações ou mocks, afinal essa é a única maneira de garantir que a consulta foi escrita e executada com sucesso pelo banco de dados.

Imagine um DAO qualquer, por exemplo, o abaixo, que contém um método que salva o produto, outro que busca um produto pelo seu id, e outro que devolve somente os produtos ativos.

Veja que esse DAO faz uso da **ISession** do **NHibernate** (uma maneira elegante de acessar dados no mundo **.NET**, semelhante ao Entity Framework):

Testes de Integração

```
public class ProdutoDAO
{
    4 references
    private ISession Session;

    0 references
    public ProdutoDAO(ISession session)
    {
        Session = session;
    }

    0 references
    public void Adiciona(Produto produto)
    {
        Session.Save(produto);
    }

    0 references
    public Produto PorId(int id)
    {
        return Session.Load<Produto>(id);
    }

    0 references
    public List<Produto> Ativos()
    {
        var produtosAtivos = (List<Produto>)Session.CreateQuery("from Produto p where p.Ativo = true");
        return produtosAtivos;
    }
}
```

Testes de Integração

Precisamos testar cada um desses métodos. Vamos começar pelo método **Adiciona()**. Como todo teste, ele conterá as mesmas 3 partes: **cenário**, **ação** e **validação**.

O cenário e ação são similares aos dos nossos testes anteriores. Basta criarmos um produto e invocar o método Adiciona:

```
[Fact(DisplayName = "Deve adicionar um produto no banco de dados")]
0 references
public void DeveAdicionarUmProduto()
{
    var dao = new ProdutoDAO(Session);
    var produto = new Produto("Geladeira", 150.0M);

    dao.Adiciona(produto);

    // como validar?
}
```

Testes de Integração

O problema é justamente como validar que o dado foi inserido com sucesso. Precisamos garantir que o elemento foi salvo no banco de dados. A única maneira é justamente fazendo uma consulta de seleção no banco de dados e verificando o produto lá. Para isso, podemos usar o próprio método **porId()** do DAO, e verificar se o objeto salvo é igual ao produto criado:

```
[Fact(DisplayName = "Deve adicionar um produto no banco de dados")]
0 references
public void DeveAdicionarUmProduto()
{
    var dao = new ProdutoDAO(Session);
    var produto = new Produto("Geladeira", 150.0M);

    dao.Adiciona(produto);

    // buscando no banco pelo id
    // para ver se está igual ao produto do cenário
    var salvo = dao.PorId(produto.IdProduto);
    Assert.Equal(salvo, produto);
}
```

Testes de Integração

Ainda temos um problema. Nosso **ProdutoDao** faz uso de uma **Session** do **Hibernate**. Todo DAO tem sua forma de conectar com o banco de dados. É necessário passar uma sessão concreta, que bata em um banco de dados de teste, para que o teste consiga executar.

Não há uma maneira ideal para fazer isso. Crie a sessão da maneira que achar necessário, apontando para um banco de dados de teste. Geralmente isso é feito em um método de inicialização do próprio teste:

```
private ISession Session;
1 reference
private ITransaction Transaction;

0 references
public void Inicializa()
{
    // pegando uma conexão com banco de testes
    Session = new CriadorDeSessoes().BancoDeTestes();
    Transaction = Session.BeginTransaction();
}
```

Testes de Integração

Outro problema é limpar esse banco de dados. Imagine por exemplo um teste que conte a quantidade de produtos cadastrados. Se já tivermos algum produto cadastrado antes da execução do teste, provavelmente o teste falhará, pois ele não contava com esse dado a mais. Portanto, cada teste precisa ter o banco de dados limpo para que dados antigos não influenciem no resultado.

É comum fazermos uso de métodos de finalização para isso. Esses métodos geralmente dão um simples rollback na transação. Veja um exemplo:

```
public void LimpaTudo()
{
    Transaction.Rollback();
    Session.Close();
}
```

Testes de Integração

Testar o método **Ativos()** também não é difícil. Precisamos montar um cenário onde o banco de dados contenha produtos ativos e não ativos.

O retorno do método deve conter apenas os não ativos. Repare ali que, após instanciarmos os objetos, os salvamos explicitamente no banco de dados:

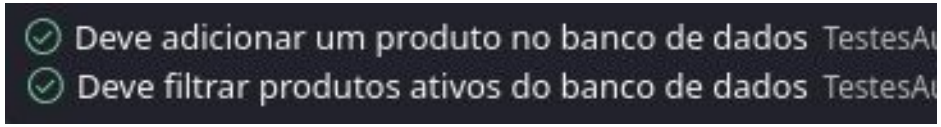
```
[Fact(DisplayName = "Deve filtrar produtos ativos do banco de dados")]
0 references
public void DeveFiltrarAtivos()
{
    var dao = new ProdutoDAO(Session);
    var ativo = new Produto("Geladeira", 150.0M);
    var inativo = new Produto("Geladeira", 150.0M);

    inativo.Inativa();

    // salvando ambos os produtos no banco
    Session.Save(ativo);
    Session.Save(inativo);

    var produtos = dao.Ativos();

    Assert.Equal(1, produtos.Count);
    Assert.Equal(inativo, produtos[0]);
}
```

Testes de Integração

Portanto, lembre-se de testar seus DAOs batendo em um banco de dados real. Não use mocks. Garanta realmente que suas consultas SQLs funcionam da maneira como você espera.

Use também sua criatividade para escrever o teste, basta ter em mente que é um teste como qualquer outro.

Você precisa montar um cenário, invocar um método no seu DAO, e encontrar uma maneira de garantir que seu banco respondeu corretamente.



Testes de Integração

Não há uma receita mágica para escrever testes de integração. Mas lembre-se que, caso você tenha uma classe cuja responsabilidade é justamente se integrar com outro sistema, você deve fazer um teste de integração.

Consiga uma réplica do sistema ou alguma maneira de consumi-lo para testes, seja ele um banco de dados, uma API, ou qualquer outra coisa.

O grande desafio, no fim, é conseguir montar o cenário e validar a saída nesse serviço externo. Talvez você escreva um “teste feio”, com muitas linhas de código. Mas não se preocupe, seu sistema estará bem testado e você dormirá em paz.

Testes End-to-End

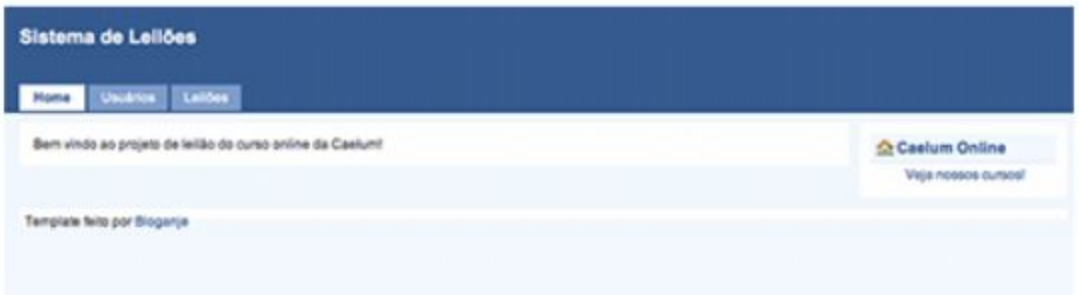
Empresas que não dão valor a boas práticas de engenharia de software e qualidade de software, estão muito acostumados a testar nossas aplicações manualmente.

Essas geralmente possuem imensos roteiros de script, e fazem com que seus analistas de teste executem esses scripts incansavelmente. Mas quais os problemas dessa abordagem?

Imagine agora um sistema web que toma conta de leilões. Nele, o usuário pode adicionar novos usuários, cadastrar leilões e efetuar lances.

Testes End-to-End

Essa aplicação foi desenvolvida em Java e está pronta para ser executada. A imagem a seguir mostra a tela inicial do sistema:



Testes End-to-End

A aplicação não é muito grande, possui cadastro de usuários, leilões e lances. Mas, apesar da pequena quantidade de funcionalidades, pense na quantidade de cenários que você precisa testar para garantir seu funcionamento:

- Cadastrar um usuário com sucesso;
- Editar um usuário;
- Validar o cadastro de usuário com nome ou e-mail inválido;
- Excluir um usuário;
- Exibir ficha completa de um usuário;
- ...

Testes End-to-End

Se pensarmos em todos os cenários que devemos testar, percebemos que teremos uma quantidade enorme! Quanto tempo uma pessoa leva para executar todos esses cenários? Agora imagine o mesmo problema em uma aplicação grande. Testar aplicações grandes de maneira manual **leva muito tempo**! E, por consequência, **custa muito caro**.

Na prática, o que acontece é que, como testar sai caro, as empresas optam por não testar! No fim, entregamos software com defeito para nosso cliente. Precisamos mudar isso.

Se removermos a parte humana do processo e fizermos com que a máquina execute o teste, resolvemos o problema: **a máquina vai executar o teste rápido, repetidas vezes, e de graça**!

Testes End-to-End

A grande questão é: como ensinar a máquina a fazer o trabalho de um ser humano? Como fazê-la abrir o browser, digitar valores nos campos, preencher formulários, clicar em links etc.?

Para isso, faremos uso do Selenium. O Selenium é um framework que facilita e muito a vida do desenvolvedor que quer escrever um teste automatizado. A primeira coisa que uma pessoa faria para testar a aplicação seria abrir o browser. Com Selenium, precisamos apenas da linha a seguir:

```
WebDriver driver = new FirefoxDriver();
```

Testes End-to-End

Nesse caso, estamos abrindo o Firefox! Em seguida, entraríamos em algum site. Vamos entrar no Google, por exemplo, usando o método **get()**:

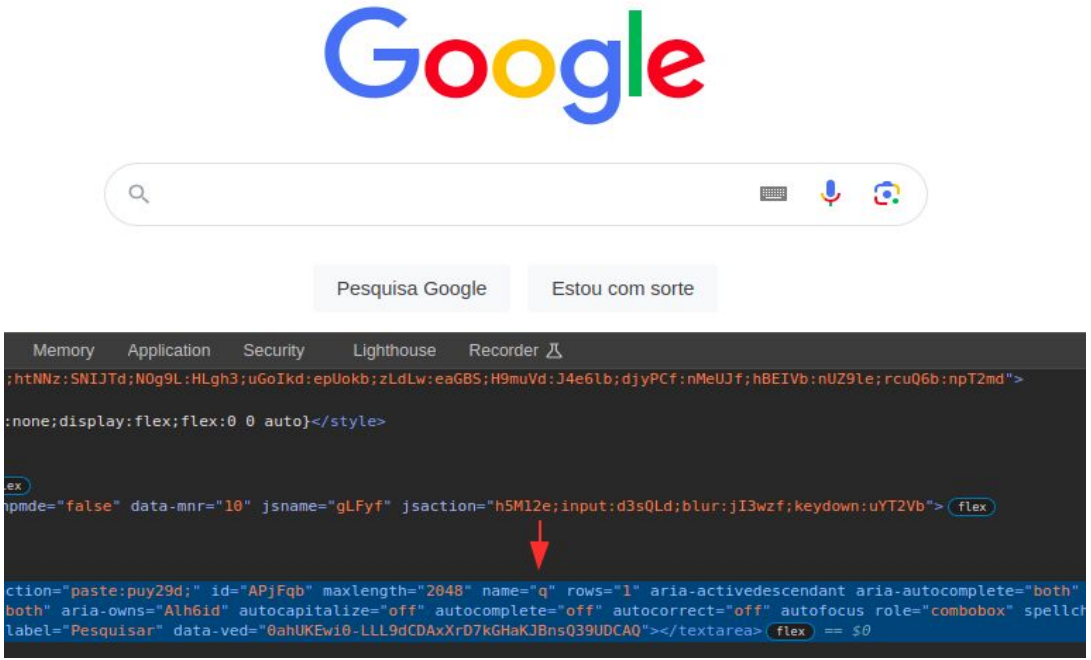
```
driver.get("http://www.google.com.br/");
```

Para buscar o termo **“Teste”**, precisamos digitar no campo de texto. No caso do Google, o nome do campo é **“q”**.

Testes End-to-End

Para descobrir, podemos fazer uso do Inspector, no Chrome ou Firefox. Basta apertar F12, e a janela abrirá.

Nela, selecionamos a lupa e clicamos no campo cujo nome queremos descobrir. Ele nos levará para o HTML, onde podemos ver **name="q"**.



Testes End-to-End

Com Selenium, basta dizermos o nome do campo de texto, e enviarmos o texto:

```
WebElement campoDeTexto = driver.findElement(By.name("q"));
campoDeTexto.sendKeys("Teste");
```

Agora, basta submetermos o form! Podemos fazer isso pelo próprio **campoDeTexto**:

```
campoDeTexto.submit();
```

Testes End-to-End

Pronto! Juntando todo esse código em uma classe Java simples, temos:

```java
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class TesteAutomatizado
{
    // abre firefox
    WebDriver driver = new FirefoxDriver();

    // acessa o site do google
    driver.get("http://www.google.com.br/");

    // digita no campo com nome "q" do google
    WebElement campoDeTexto = driver.findElement(By.name("q"));
    campoDeTexto.sendKeys("Teste");

    // submete o form
    campoDeTexto.submit();
}
```

Testes End-to-End

Se você não entendeu algum método invocado, não se preocupe. Vamos estudá-los com mais detalhes nos próximos capítulos. Nesse momento, rode a classe! Acabamos de fazer uma busca no Google de maneira automatizada!

Execute o teste novamente. Veja agora como é fácil, rápido e barato! Qual a vantagem? Podemos executá-los a tempo! Ou seja, a cada mudança que fazemos em nosso software, podemos testá-lo por completo, clicando apenas em um botão. Saberemos em poucos minutos se nossa aplicação continua funcionando!

Quantas vezes não entregamos software sem tê-lo testado por completo?

Testes End-to-End

Imagine uma tela de cadastro padrão, onde o usuário deve preencher um formulário qualquer. E que, ao clicar em “Salvar”, o sistema devolve o usuário para a listagem com o novo usuário cadastrado:

Testes End-to-End

A funcionalidade funciona atualmente, mas nossa experiência nos diz que futuras alterações no sistema podem fazer com que a funcionalidade pare. Vamos automatizar um teste para o cadastro de um novo usuário. O cenário é o mesmo que acabamos de testar de maneira manual.

A primeira parte do teste manual é entrar na página de cadastro de usuários. Vamos fazer o Selenium abrir o **Firefox** nessa página. Suponha que o endereço seja **http://localhost:8080/usuarios/new**:

```
public static void main(String[] args) {

    WebDriver driver = new FirefoxDriver();
    driver.get("http://localhost:8080/usuarios/new");
}
```

Testes End-to-End

Nessa página, precisamos cadastrar algum usuário, por exemplo, “Ronaldo Luiz de Albuquerque” com o e-mail “ronaldo2009@terra.com.br”.

Para preencher esses valores de maneira automatizada, precisamos saber o nome dos campos de texto para que o Selenium saiba aonde colocar essa informação!

Aperte CTRL + U (no Firefox e Chrome) ou Ctrl + F12 (no Internet Explorer) para exibir o código-fonte da página. Veja que o nome dos campos de texto são “usuario.nome” e “usuario.email”.

Testes End-to-End

Com essa informação em mãos, precisamos encontrar esses elementos na página e preencher com os valores que queremos:

```
// encontrando ambos elementos na pagina
WebElement nome = driver.findElement(By.name("usuario.nome"));
WebElement email = driver.findElement(By.name("usuario.email"));

// digitando em cada um deles
nome.sendKeys("Ronaldo Luiz de Albuquerque");
email.sendKeys("ronaldo2009@terra.com.br");
```

Testes End-to-End

Veja o código. Para encontrarmos um elemento, utilizamos o método **driver.findElement**. Como existem muitas maneiras para encontrar um elemento na página (pelo id, nome, classe CSS etc.), o **Selenium** nos provê uma classe chamada **By** que tem um conjunto de métodos que nos ajudam a achar o elemento. Nesse caso, como queremos encontrar o elemento pelo seu nome, usamos **By.name("nome-aqui")**.

Tudo preenchido! Precisamos submeter o formulário! Podemos fazer isso de duas maneiras. A primeira delas é clicando no botão que temos na página. Ao olhar o código-fonte da página novamente, é possível perceber que o **id** do botão de Salvar é **btnSalvar**. Basta pegarmos esse elemento e clicar nele:

```
WebElement botaoSalvar = driver.findElement(By.id("btnSalvar"));
botaoSalvar.click();
```

Testes End-to-End

Se tudo der certo, voltamos à listagem de usuários. Mas, desta vez, esperamos que o usuário Ronaldo esteja lá. Para terminar nosso teste, precisamos garantir de maneira automática que o usuário adicionado está lá. Para fazer esses tipos de verificação, utilizaremos um framework muito conhecido do mundo **Java**, que é o **JUnit**.

O JUnit nos provê um conjunto de instruções para fazer essas comparações e ainda conta com um plugin que nos diz se os testes estão passando ou, caso contrário, quais testes estão falhando!

Para garantir o usuário na listagem, precisamos procurar pelos textos **“Ronaldo Luiz de Albuquerque”** e **“ronaldo2009@terra.com.br”** na página atual.

Testes End-to-End

O Selenium nos dá o código-fonte HTML inteiro da página atual, através do método **driver.getPageSource()**. Basta verificarmos se existem o nome e e-mail do usuário lá:

```
boolean achouNome = driver.getPageSource().contains("Ronaldo Luiz de Albuquerque");
boolean achouEmail = driver.getPageSource().contains("ronaldo2009@terra.com.br");
```

Sabemos que essas duas variáveis devem ser iguais a true. Vamos avisar isso ao **JUnit** através do método **assertTrue()** e, dessa forma, caso essas variáveis fiquem com **false**, o JUnit nos avisará:

```
assertTrue(achouNome);
assertTrue(achouEmail);
```

Testes End-to-End

Lembre-se que, para o método assertTrue funcionar, precisamos fazer o import estático do método **import static org.junit.Assert.assertTrue**.

Devemos agora encerrar o **Selenium**:

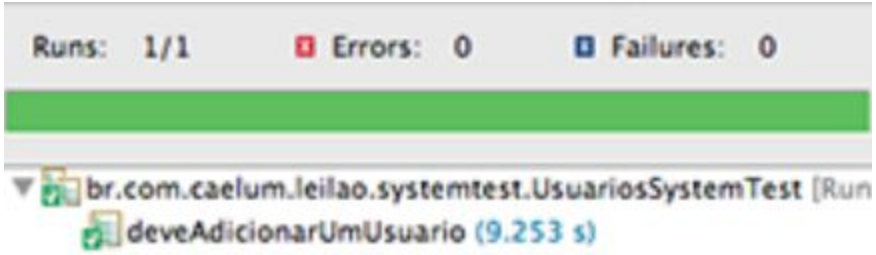
```
driver.close();
```

Por fim, para que o **JUnit** entenda que isso é um método de teste, precisamos mudar sua assinatura. Todo método do JUnit deve ser **público**, não retornar nada, e deve ser anotado com **@Test**. Veja:

```java
import static org.junit.Assert.assertTrue;

import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class UsuariosSystemTest
{
    @Test
    public void deveAdicionarUmUsuario()
    {
        WebDriver driver = new FirefoxDriver();
        driver.get("http://localhost:8080/usuarios/new");

        // encontrando ambos elementos na pagina
        WebElement nome = driver.findElement(By.name("usuario.nome"));
        WebElement email = driver.findElement(By.name("usuario.email"));

        // digitando em cada um deles
        nome.sendKeys("Ronaldo Luiz de Albuquerque");
        email.sendKeys("ronaldo2009@terra.com.br");

        WebElement botaoSalvar = driver.findElement(By.id("btnSalvar"));
        botaoSalvar.click();

        boolean achouNome = driver.getPageSource().contains("Ronaldo Luiz de Albuquerque");
        boolean achouEmail = driver.getPageSource().contains("ronaldo2009@terra.com.br");

        assertTrue(achouNome);
        assertTrue(achouEmail);

        driver.close();
    }
}
```

Testes End-to-End

Pronto! Nosso primeiro teste para a aplicação de leilão está escrito.



O que acabamos de fazer é o que chamamos de **Teste End-to-End**. Ou seja, é um teste que exercita a aplicação do ponto de vista do usuário final, como um todo, sem conhecer seus detalhes internos.

Testes End-to-End

Se sua aplicação é web, então o teste de end-to-end é aquele que navegará pela interface web, interagir com ela, os dados serão persistidos no banco de dados, os serviços web serão consumidos etc.

A vantagem desse tipo de teste é que ele é muito parecido com o teste que o usuário faz. No teste de unidade, garantíamos que uma classe funcionava muito bem. Mas nosso software em produção faz uso de diversas classes juntas. É aí que o teste de sistema entra; ele garante que o sistema funciona quando “tudo está ligado”.

O **Selenium** é uma excelente ferramenta para automatizar os testes. Sua **API** é bem clara e fácil de usar, além da grande quantidade de documentação que pode ser encontrada na internet.

Testes End-to-End

Novamente, as vantagens do teste automatizado

- O teste automatizado é muito mais rápido do que um ser humano.
- A partir do momento em que você escreveu o teste, você poderá executá-lo infinitas vezes a um custo baixíssimo.
- Mais produtividade, afinal, você gastará menos tempo testando (escrever um teste automatizado gasta menos tempo do que testar manualmente diversas vezes a mesma funcionalidade) e mais tempo desenvolvendo.
- Bugs encontrados mais rápido pois, já que sua bateria de testes roda rápido, você a executará a todo instante, encontrando possíveis partes do sistema que deixaram de funcionar devido a novas implementações.

Testes End-to-End

Testes de sistema estão posicionados no topo da pirâmide de testes. Trata-se de testes que simulam o uso de um sistema por um usuário real. Testes de sistema são também chamados de testes ponta-a-ponta (end-to-end) ou então testes de interfaces.

São os testes mais caros, que demandam maior esforço para implementação e que executam em mais tempo. Testes de interface são mais difíceis de escrever, pelo menos do que testes de unidade e mesmo do que testes de integração.

Por exemplo, a API do Selenium é mais complexa do que aquela do JUnit. Além disso, o teste deve tratar eventos de interfaces, como timeouts que ocorrem quando uma página demora mais tempo do que o usual para ser carregada.

Testes End-to-End

Testes de interface também são mais frágeis, isto é, eles podem quebrar devido a pequenas mudanças na interface.

Por exemplo, se o nome do campo de um formulário mudar, o teste terá que ser atualizado.

Mas, se compararmos com a alternativa de realizar o teste manualmente eles ainda são muito melhores, pois, a partir do momento em que você escreveu o teste, você poderá executá-lo infinitas vezes a um custo baixíssimo.

3. Princípios F.I.R.S.T

Propriedades que todos os de testes de unidades devem satisfazer.

Princípios F.I.R.S.T

A sigla **F.I.R.S.T** significa, **F**ast (Rápido), **I**solated (Isolado), **R**epeatable (Repetivel), **S**elf-Validating (Auto Validavel) e **T**imely (A Tempo).

A ideia é que comecemos a pensar nessas características ao criar um **teste de unidade**.

Princípios F.I.R.S.T

Rápidos (Fast): desenvolvedores devem executar testes de unidades frequentemente, para obter feedback rápido sobre bugs e regressões no código. Por isso, é importante que eles sejam executados rapidamente, em questões de milisegundos.

Se isso não for possível, pode-se dividir uma suíte de testes em dois grupos: testes que executam rapidamente e que, portanto, serão frequentemente chamados; e testes mais demorados, que serão, por exemplo, executados uma vez por dia.

Princípios F.I.R.S.T

Independentes (Independent): a ordem de execução dos testes de unidade não é importante. Para quaisquer testes T1 e T2, a execução de T1 seguida de T2 deve ter o mesmo resultado da execução de T2 e depois T1.

Pode acontecer ainda de T1 e T2 serem executados de forma paralela (ao mesmo tempo). Para que os testes sejam independentes, T1 não deve alterar alguma parte do estado global do sistema que depois será usada para computar o resultado de T2 e vice-versa.

Princípios F.I.R.S.T

Determinísticos (Repeatable): testes de unidade devem ter sempre o mesmo resultado. Ou seja, se um teste T é chamado n vezes, o resultado deve ser o mesmo nas n execuções. Isto é, ou T passa em todas as execuções, ou ele sempre falha. Testes com resultados não-determinísticos são chamados de Testes Flaky (ou Testes Erráticos). Concorrência é uma das principais responsáveis por comportamento flaky. Um exemplo é mostrado a seguir:

```
@Test
public void exemploTesteFlaky {
    TaskResult resultado;
    MyMath m = new MyMath();
    m.asyncPI(10,resultado);
    Thread.sleep(1000);
    assertEquals(3.1415926535, resultado.get());
}
```

Princípios F.I.R.S.T

Esse teste chama uma função que calcula o valor de PI, com uma certa precisão, e de forma assíncrona isto é, a função realiza o seu cálculo em uma nova thread, que ela mesmo cria internamente. No exemplo, a precisão requerida são 10 casas decimais. O teste faz uso de um sleep para esperar que a função assíncrona termine.

No entanto, isso torna o seu comportamento não-determinístico: se a função terminar antes de 1000 milissegundos, o teste irá passar, mas se a execução, por alguma circunstância particular, demorar mais, o teste irá falhar.

Uma possível alternativa seria testar apenas a versão síncrona da função. Se essa versão não existir, um refactoring poderia ser realizado para extraí-la do código da versão assíncrona.

Princípios F.I.R.S.T

Auto-verificáveis (Self-checking): O resultado de um teste de unidade deve ser facilmente verificável. Para interpretar o resultado do teste, o desenvolvedor não deve, por exemplo, ter que abrir e analisar um arquivo de saída ou fornecer dados manualmente.

Em vez disso, o resultado dos testes deve ser binário e mostrado na IDE, normalmente por meio de componentes que ficam com a cor verde (para indicar que todos os testes passaram) ou com a cor vermelha (para indicar que algum teste falhou).

Adicionalmente, quando um teste falha, deve ser possível identificar essa falha de forma rápida, incluindo a localização do comando assert que falhou.

Princípios F.I.R.S.T

Escritos o quanto antes (Timely): Uma boa prática é escrever os testes o mais cedo possível, de preferência antes mesmo do código de produção e executar regularmente durante o desenvolvimento. Isso ajuda a detectar problemas mais cedo no ciclo de desenvolvimento, reduzindo o custo e a complexidade de correções futuras.

Vamos discutir com mais profundidade na próxima seção sobre **TDD - Desenvolvimento Dirigido por Testes** que é uma abordagem de escrever código de testes antes mesmo do código de produção.

Princípios F.I.R.S.T

Os **Princípios F.I.R.S.T**, podem ajudar na construção de testes automatizados de qualidade e que nos ajudem a tirar o máximo proveito das vantagens da automação de testes.

Seguir esses princípios ao escrever testes unitários ajuda a garantir que os testes sejam eficazes, fáceis de manter e proporcionem um feedback rápido e confiável sobre a qualidade do código.

Eles são uma orientação valiosa para os desenvolvedores no processo de criação de testes unitários robustos e úteis.

4.

Abordagens

Test-Driven Development (TDD), Behavior-Driven Development (BDD)

TDD - Test Driven Development

O **TDD** do inglês **(Test Driven Development)** em português **(Desenvolvimento Guiado por Testes)** é uma técnica para construção de software que guia seu desenvolvimento por meio da escrita de testes, é uma das práticas de programação propostas por **Extreme Programming (XP)**.

A ideia é fazer com que o desenvolvedor escreva **testes automatizados** de maneira **constante** ao longo do desenvolvimento. Mas, diferentemente do que estamos acostumados, TDD sugere que o desenvolvedor **escreva o teste antes mesmo do código de produção**.

Sua essência está em seu mantra: **testar**, **codificar** e **refatorar**. Essa técnica surgiu a partir da refatoração, dando base para guiar a codificação.

TDD - Test Driven Development

Essa simples inversão no ciclo traz diversos benefícios para o projeto. Baterias de testes tendem a ser maiores, cobrindo mais casos, e garantindo uma maior qualidade externa.

Além disso, escrever testes de unidade forçará o desenvolvedor a escrever um código de maior qualidade, pois para escrever bons testes de unidade, o desenvolvedor é obrigado a fazer bom uso de orientação a objetos.

A prática nos ajuda a escrever um software melhor, com mais qualidade, e um código melhor, mais fácil de ser mantido e evoluído. Esses dois pontos são importantíssimos em qualquer software, e TDD nos ajuda a alcançá-los. Toda prática que ajuda a aumentar a qualidade do software produzido deve ser estudada.

TDD - Test Driven Development

Na contabilidade. Os contadores produzem uma quantidade gigantesca de documentos com teor profundamente técnico, cheio de símbolos. Cada símbolo registrado em seus documentos deve estar correto, para que não se percam fortunas e, possivelmente, até vidas. Como os contadores asseguram que todos os símbolos estejam corretos?

Os contadores têm um método que foi inventado há mil anos. Chama-se **método das partidas dobradas**. Cada transação que eles registram em seus livros é inserida duas vezes: uma como crédito em um conjunto de contas e outra como débito complementar em outro conjunto de contas.

Essas contas acabam sendo disponibilizadas em um único documento chamado balanço patrimonial, que subtrai a soma dos passivos e ações a partir da soma dos ativos. Essa diferença deve ser zero. Se não for zero, algum erro foi cometido.

TDD - Test Driven Development

Logo no início de sua formação, contadores são ensinados a registrar as transações uma por vez e sempre calcular o saldo. Isso lhes possibilita detectar rapidamente os erros. Eles aprendem a evitar um registro de um lote de transações entre as verificações de saldo, pois seria difícil identificar os erros. Essa prática é tão essencial para a contabilidade propriamente dita do dinheiro que se tornou um cânone em basicamente todas as partes do mundo.

O **Desenvolvimento Orientado a Testes** é o **Método das Partidas Dobradas** para os programadores. Todo comportamento exigido é inserido duas vezes: uma vez como teste e depois como código de produção, para que o teste seja aprovado. As duas entradas são complementares, assim como os ativos são complementares aos passivos e ações. Quando executadas juntas, as duas entradas geram zero como resultado: zero falhas nos testes.

TDD - Test Driven Development

Os programadores que aprendem sobre TDD são ensinados a registrar todos os comportamentos, um de cada vez uma vez como um teste que falha e, depois, como um código de produção que passa nos testes. Isso lhes permite detectar rapidamente os erros. Eles aprendem a evitar escrever muito código de produção e a acrescentar um lote de testes, pois seria difícil identificar os erros.

Esses dois métodos, o método das partidas dobradas e o TDD, são equivalentes. Ambos têm o mesmo propósito: evitar erros em documentos de importância crucial, nos quais todos os símbolos devem ser registrados de forma correta.

TDD - Test Driven Development

As 3 regras do TDD

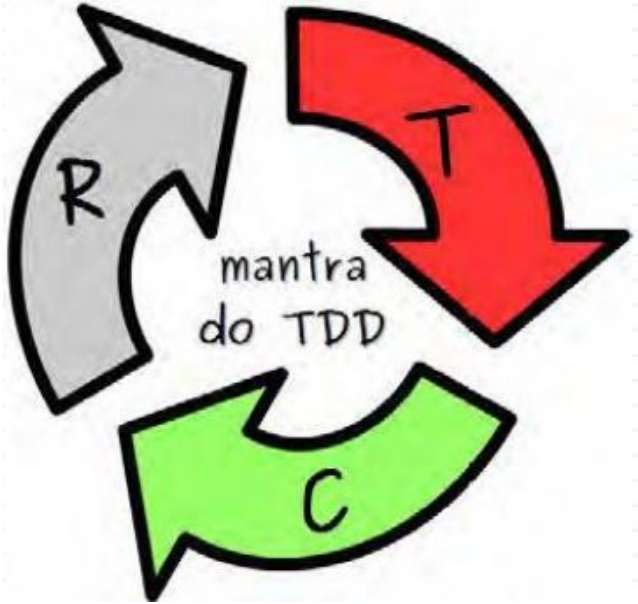
O TDD pode ser descrito em três regras simples:

- Não escreva nenhum código de produção antes de elaborar um teste que falhou devido à falta desse mesmo código.
- Não escreva mais testes do que o suficiente para identificação da falha e falhas na compilação ainda contam como falhas.
- Não escreva mais códigos de produção do que o suficiente para passar nos testes.

TDD - Test Driven Development

O processo de refatoração está intimamente relacionado com as 3 Regras do TDD e é conhecido como ciclo Vermelho/Verde/Refatore:

1. Primeiro, escreva um teste que falhe.
2. Depois, faça com que o teste passe.
3. Em seguida, limpe o código.
4. Volte para a etapa 1.



TDD - Test Driven Development

A ideia aqui é que escrever um código que funcione e escrever um código que seja limpo são duas dimensões separadas da programação. Tentar controlar ambas as dimensões ao mesmo tempo é, na melhor das hipóteses, complicado, então, as separamos em duas atividades distintas.

Em outras palavras, já é custoso o bastante fazer o código funcionar, imagine fazer sua limpeza. Desse modo, primeiro nos concentramos em fazer com que o código funcione por meio de quaisquer ideias loucas vindas de nossas cabeças.

Em seguida, depois que estiver funcionando, com a aprovação dos testes, limpamos a bagunça que fizemos. Isso deixa claro que a refatoração é um processo contínuo, e não um processo executado de forma planejada.

TDD - Test Driven Development

Não ficamos bagunçando as coisas por dias e depois tentamos limpar a bagunça. Ao contrário, fazemos o mínimo de bagunça, durante um minuto ou dois, e depois a limpamos.

A palavra Refatoração nunca deve aparecer em um cronograma. Esse não é o tipo de atividade que aparece em um planejamento. Não reservamos tempo para refatorar. A refatoração simplesmente faz parte da nossa abordagem diária para escrever o software.

Agora chega de teoria e vamos a prática, vamos desenvolver um algoritmo matemático, mas dessa vez utilizando TDD, o teste será escrito antes da implementação. Ao final, discutiremos as vantagens e desvantagens dessa abordagem.

TDD - Test Driven Development

Numerais romanos foram criados na Roma Antiga e eles foram utilizados em todo o seu império. Os números eram representados por sete diferentes símbolos, listados na tabela a seguir.

- **I,** unus, 1, (um)
- **V,** quinque, 5 (cinco)
- **X,** decem, 10 (dez)
- **L,** quinquaginta, 50 (cinquenta)
- **C,** centum, 100 (cem)
- **D,** quingenti, 500 (quinhentos)
- **M,** mille, 1.000 (mil)

TDD - Test Driven Development

Para representar outros números, os romanos combinavam estes símbolos, começando do algarismo de maior valor e seguindo a regra:

- Algarismos de menor ou igual valor à direita são somados ao algarismo de maior valor.

- Algarismos de menor valor à esquerda são subtraídos do algarismo de maior valor.

Por exemplo, XV representa 15 (10 + 5) e o número XXVIII representa 28 (10 + 10 + 5 + 1 + 1 + 1). Há ainda uma outra regra: nenhum símbolo pode ser repetido lado a lado por mais de 3 vezes. Por exemplo, o número 4 é representado pelo número IV (5 - 1) e não pelo número IIII.

TDD - Test Driven Development

Existem outras regras, mas em linhas gerais, este é o problema a ser resolvido. Dado um numeral romano, o programa deve convertê-lo para o número inteiro correspondente.

Conhecendo o problema dos numerais romanos, é possível levantar os diferentes cenários que precisam ser aceitos pelo algoritmo: um símbolo, dois símbolos iguais, três símbolos iguais, dois símbolos diferentes do maior para o menor, quatro símbolos dois a dois, e assim por diante.

Dado todos estes cenários, uns mais simples que os outros, começaremos pelo mais simples: um único símbolo.

TDD - Test Driven Development

Começando pelo teste **“deve entender o símbolo I”**. A classe responsável pela conversão pode ser chamada, por exemplo, de **ConversorDeNumeroRomano**, e o método **Converte()**, recebendo uma String com o numeral romano e devolvendo o valor inteiro representado por aquele número:

```
0 references
public class ConversorDeNumeroRomanoTest
{
    0 references
    public void DeveEntenderOSimboloI()
    {
        ConversorDeNumeroRomano romano = new ConversorDeNumeroRomano();
        int numero = romano.Converte("I");

        Assert.Equal(1, numero);
    }
}
```

TDD - Test Driven Development

Veja que nesse momento, esse código não compila; a classe **ConversorDeNumeroRomano**, bem como o método **Converte()** não existem.

Para resolver o erro de compilação, é necessário criar classe, mesmo que sem uma implementação real:

```
2 references
public class ConversorDeNumeroRomano
{
    1 reference
    public int Converte(string numeroEmRomano)
    {
        return 0;
    }
}
```

TDD - Test Driven Development

De volta ao teste, ele agora compila. Ao executá-lo, o teste falha. Mas não há problema, isso já era esperado. Para fazê-lo passar, introduziremos ainda uma segunda regra: **o código escrito deve ser sempre o mais simples possível**.

```
1 reference
public int Converte(string numeroEmRomano)
{
    return 1;
}
```

Desenvolvedores, muito provavelmente, não ficarão felizes com essa implementação, afinal ela funciona apenas para um caso. Mas isso não é problema, afinal a implementação não está pronta, ainda estamos trabalhando nela.

TDD - Test Driven Development

Um próximo cenário seria o símbolo V. Nesse caso, o algoritmo deve retornar 5. Novamente começando pelo teste:

```
[Fact(DisplayName = "Deve entender o simbolo V")]
0 references
public void DeveEntenderOSimboloV()
{
    var romano = new ConversorDeNumeroRomano();
    var numero = romano.Converte("V");

    Assert.Equal(5, numero);
}
```

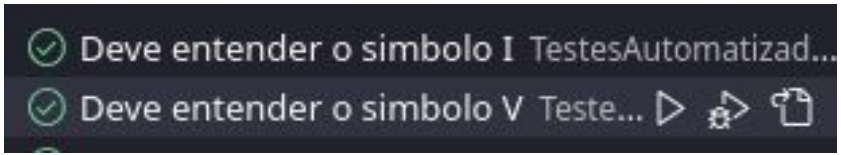
TDD - Test Driven Development

Esse teste também falha. Novamente faremos a implementação mais simples que resolverá o problema. Podemos, por exemplo, fazer com que o método Converte() verifique o conteúdo do número a ser convertido: se o valor for “I”, o método retorna 1; se o valor for “V”, o método retorna 5:

```
2 references
public int Converte(string numeroEmRomano)
{
    if(numeroEmRomano.Equals("I"))
        return 1;

    else if(numeroEmRomano.Equals("V"))
        return 5;

    return 0;
}
```

TDD - Test Driven Development

Os testes passam (os dois que temos).



Poderíamos repetir o mesmo teste e a mesma implementação para os símbolos que faltam (X, L, C, M, ...).

Mas nesse momento, já temos uma primeira definição sobre nosso algoritmo: quando o numeral romano possui apenas um símbolo, basta devolvermos o inteiro associado a ele.

TDD - Test Driven Development

Ao invés de escrever um monte de ifs para cada símbolo, é possível usar um switch, que corresponde melhor ao nosso cenário.

Melhorando ainda mais, ao invés do switch, é possível guardar os símbolos em uma tabela, ou no C#, em um Dictionary entre o algarismo e o inteiro correspondente à ele. Sabendo disso, vamos nesse momento alterar o código para refletir a solução:

```
2 references
public class ConversorDeNumeroRomano
{
    1 reference
    private static Dictionary<string, int> tabela =
        new Dictionary<string, int>() {
        {"I", 1},
        {"V", 5},
        {"X", 10},
        {"L", 50},
        {"C", 100},
        {"D", 500},
        {"M", 1000}
    };

    2 references
    public int Converte(string numeroEmRomano)
    {
        return tabela[numeroEmRomano];
    }
}
```

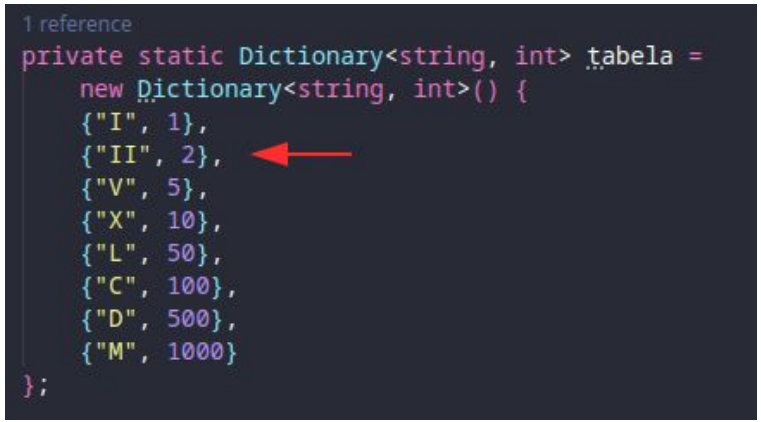
TDD - Test Driven Development

Ambos os testes continuam passando. Passaremos agora para um segundo cenário: dois símbolos em sequência, como por exemplo, “II” ou “XX”. Começando novamente pelo teste, temos:

```
[Fact(DisplayName = "Deve entender o simbolo II")]
0 references
public void DeveEntenderDoisSimbolosComoII()
{
    var romano = new ConversorDeNumeroRomano();
    var numero = romano.Converte("II");
    Assert.Equal(2, numero);
}
```

TDD - Test Driven Development

Para fazer o teste passar de maneira simples, é possível simplesmente adicionar os símbolos “II” na tabela:



O teste passa. Mas, apesar de simples, essa não parece uma boa ideia de implementação: seria necessário incluir todos os possíveis símbolos nessa tabela, o que não faz sentido.

TDD - Test Driven Development

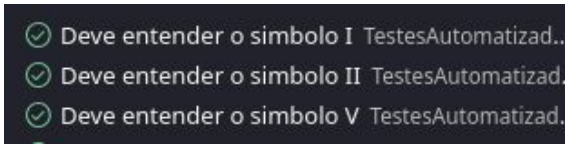
É hora de refatorar esse código novamente. Uma possível solução seria iterar em cada um dos símbolos no numeral romano e acumular seu valor, ao final, retorna o valor acumulado.

Para isso, é necessário mudar a tabela para guardar somente os símbolos principais da numeração romana. Uma possível implementação deste algoritmo seria:

```
3 references
public class ConversorDeNumeroRomano
{
    1 reference
    private static Dictionary<char, int> tabela =
        new Dictionary<char, int>() {
        {'I', 1},
        {'V', 5},
        {'X', 10},
        {'L', 50},
        {'C', 100},
        {'D', 500},
        {'M', 1000}
    };

    3 references
    public int Converte(string numeroEmRomano)
    {
        var acumulador = 0;

        for(int i = 0; i < numeroEmRomano.Length; i++)
        {
            acumulador += tabela[numeroEmRomano[i]];
        }

        return acumulador;
    }
}
```

TDD - Test Driven Development

Os três testes continuam passando.



E, dessa forma, resolvemos o problema de dois símbolos iguais em seguida. O próximo cenário são quatro símbolos, dois a dois, como por exemplo, “XXII”, que deve resultar em 22. O teste:

```
[Fact(DisplayName = "Deve entender o simbolo XXII")]
0 references
public void DeveEntenderQuatroSimbolosDoisADoisComoXXII()
{
    var romano = new ConversorDeNumeroRomano();
    var numero = romano.Converte("XXII");
    Assert.Equal(22, numero);
}
```

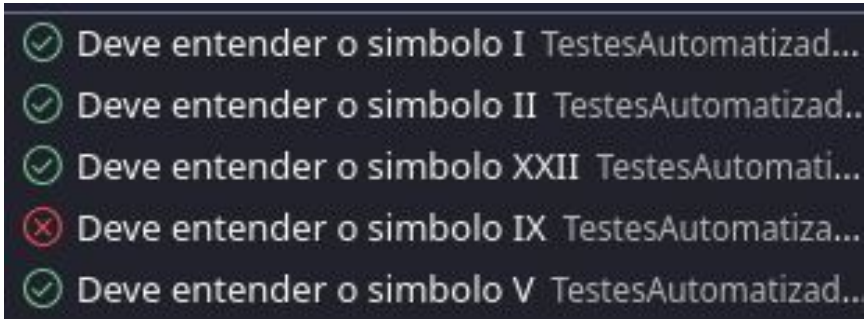
TDD - Test Driven Development

Esse teste já passa sem que precisemos fazer qualquer alteração. O algoritmo existente até então já resolve o problema. Vamos então para o próximo cenário: números como “IV” ou “IX”, onde não basta apenas somar os símbolos existentes. Novamente, começando pelo teste:

```
[Fact(DisplayName = "Deve entender o simbolo IX")]
0 references
public void DeveEntenderQuatroSimbolosDoisADoisComoIX()
{
    var romano = new ConversorDeNumeroRomano();
    var numero = romano.Converte("IX");

    Assert.Equal(9, numero);
}
```

TDD - Test Driven Development

Antes de iniciar a refatoração, vamos verificar se novo teste passa.



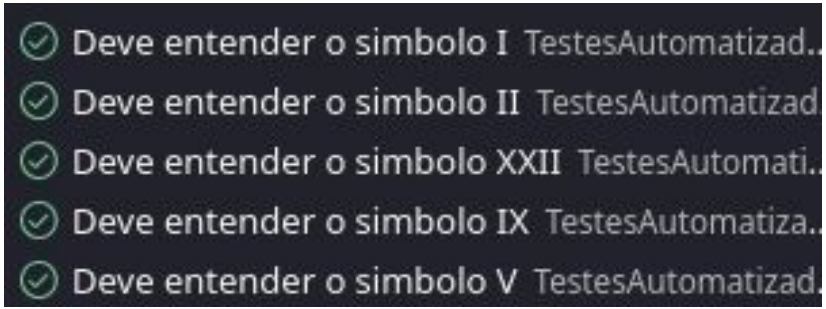
Como previamos o teste não passa, para fazer esse teste passar, é necessário pensar um pouco melhor sobre o problema. Repare que os símbolos em um numeral romano, da direita para a esquerda, sempre crescem. Quando um número a esquerda é menor do que seu vizinho a direita, esse número deve então ser subtraído ao invés de somado no acumulador. Vamos implementar esse novo algoritmo:

TDD - Test Driven Development

```
5 references
public int Converte(string numeroEmRomano)
{
    var acumulador = 0;
    var ultimoVizinhoDaDireita = 0;

    for(int i = numeroEmRomano.Length - 1; i >= 0; i--)
    {
        // pega o inteiro referente ao simbolo atual
        var atual = tabela[numeroEmRomano[i]];

        // se o da direita for menor, o multiplicaremos
        // por -1 para torná-lo negativo
        var multiplicador = 1;

        if(atual < ultimoVizinhoDaDireita)
            multiplicador = -1;

        acumulador += tabela[numeroEmRomano[i]] * multiplicador;

        // atualiza o vizinho da direita
        ultimoVizinhoDaDireita = atual;
    }
    return acumulador;
}
```

TDD - Test Driven Development

Os testes agora passam.



Mas veja, existe código repetido ali. Na linha em que o acumulador é somado com o valor atual, **tabela[numeroEmRomano[i]]** pode ser substituído pela variável atual. Melhorando o código, temos:

```
acumulador += atual * multiplicador;
```

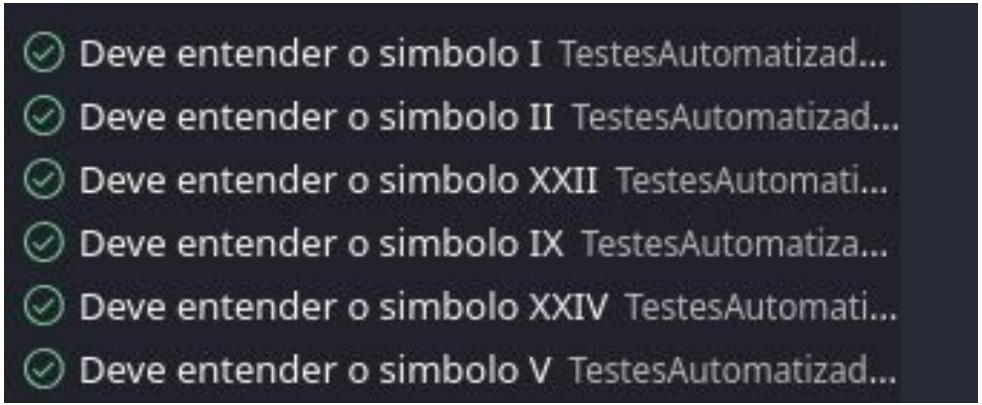
TDD - Test Driven Development

A refatoração foi feita com sucesso, já que os testes continuam passando. Ao próximo cenário: numeral romano que contenha tanto números “invertidos”, como “IV”, e dois símbolos lado a lado, como “XX”. Por exemplo, o número “XXIV”, que representa o número 24.

```
[Fact(DisplayName = "Deve entender o simbolo XXIV")]
0 references
public void DeveEntenderQuatroSimbolosDoisADoisComoXXIV()
{
    var romano = new ConversorDeNumeroRomano();
    var numero = romano.Converte("XXIV");

    Assert.Equal(24, numero);
}
```

TDD - Test Driven Development

Esse teste já passa; o algoritmo criado até então já atende o cenário do teste. Ainda há outros cenários que poderiam ser testados, mas já fizemos o suficiente para discutir sobre o assunto.



TDD - Test Driven Development

❑ Refletindo sobre o assunto

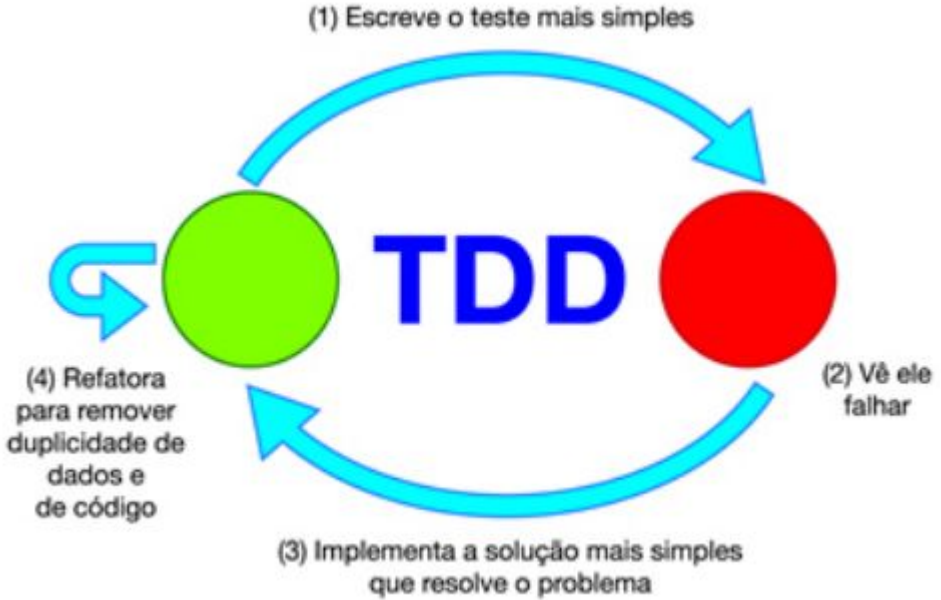
De maneira mais abstrata, o ciclo que foi repetido ao longo do processo de desenvolvimento da classe acima foi:

- Escrevemos um teste de unidade para uma nova funcionalidade.
- Vimos o teste falhar.
- Implementamos o código mais simples para resolver o problema.
- Vimos o teste passar.
- Melhoramos (refatoramos) nosso código quando necessário.

TDD - Test Driven Development

Esse ciclo de desenvolvimento é conhecido por **Test-Driven Development (TDD)**, ou, **Desenvolvimento Guiado pelos Testes**.

A ideia é simples: o desenvolvedor deve começar a implementação pelo teste e, deve o tempo todo, fazer de tudo para que seu código fique simples e com qualidade.



TDD - Test Driven Development

Esse ciclo é também conhecido como **ciclo vermelho-verde-refatora** (ou red-green-refactor).

O primeiro passo é escrever um teste que falha. A cor vermelha representa esse teste falhando.

Em seguida, fazemos ele passar (a cor verde representa ele passando).

Por fim, refatoramos para melhorar o código que escrevemos.

TDD - Test Driven Development

❑ Quais as vantagens?

Muitos praticantes de TDD afirmam que executar esse ciclo pode ser muito vantajoso para o processo de desenvolvimento. Alguns deles:

- Foco no teste e não na implementação. Ao começar pelo teste, o programador consegue pensar somente no que a classe deve fazer, e esquece por um momento da implementação. Isso o ajuda a pensar em melhores cenários de teste para a classe sob desenvolvimento.

- Código nasce testado. Se o programador pratica o ciclo corretamente, isso então implica em que todo o código de produção escrito possui ao menos um teste de unidade verificando que ele funciona corretamente.

TDD - Test Driven Development

- **Melhor reflexão sobre o design da classe.** No cenário tradicional, muitas vezes a falta de coesão ou o excesso de acoplamento é causado muitas vezes pelo desenvolvedor que só pensa na implementação e acaba esquecendo como a classe vai funcionar perante o todo. Ao começar pelo teste, o desenvolvedor pensa sobre como sua classe deverá se comportar perante as outras classes do sistema.

  O teste atua como o primeiro cliente da classe que está sendo escrita. Nele, o desenvolvedor toma decisões como o nome da classe, os seus métodos, parâmetros, tipos de retorno, e etc. No fim, todas elas são decisões de design e, quando o desenvolvedor consegue observar com atenção o código do teste, seu design de classes pode crescer muito em qualidade.

TDD - Test Driven Development

- **Simplicidade.** Ao buscar pelo código mais simples constantemente, o desenvolvedor acaba por fugir de soluções complexas, comuns em todos os sistemas. O praticante de TDD escreve código que apenas resolve os problemas que estão representados por um teste de unidade. Quantas vezes o desenvolvedor não escreve código desnecessariamente complexo?

Uma pergunta que pode vir a cabeça é:

"No modelo tradicional, onde os testes são escritos depois, o desenvolvedor não tem os mesmos benefícios?"

TDD - Test Driven Development

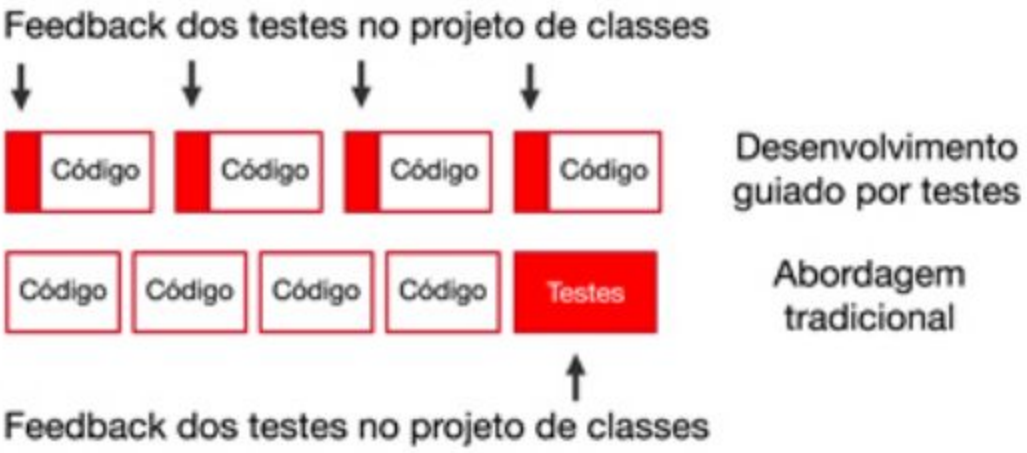
A resposta é **sim**. Mesmo desenvolvedores que escrevem testes depois podem obter as mesmas vantagens.

Um desenvolvedor pode perceber que está com dificuldades de escrever um teste e descobrir que o design da classe que implementou tem problemas, ele pode também conseguir abstrair a implementação e escrever bons cenários de teste.

Mas há uma diferença, que faz toda a diferença: a quantidade de vezes que um programador praticante de TDD recebe feedback sobre esses pontos e a quantidade que um programador que não pratica TDD recebe.

TDD - Test Driven Development

Veja a figura abaixo. O praticante de TDD escreve um pouco de testes, um pouco de implementação e recebe feedback. Isso acontece ao longo do desenvolvimento de maneira frequente. Já um programador que não pratica TDD espera um tempo (às vezes longo demais) para obter o mesmo feedback.



TDD - Test Driven Development

Ao receber feedback desde cedo, o programador pode melhorar o código e corrigir problemas a um custo menor do que o programador que recebeu a mesma mensagem muito tempo depois. Todo programador sabe que alterar o código que ele acabara de escrever é muito mais fácil e rápido do que alterar o código escrito 3 dias atrás.

No fim, TDD apenas maximiza a quantidade de feedback sobre o código que está sendo produzido, fazendo o programador perceber os problemas antecipadamente e, por consequência, diminuindo os custos de manutenção e melhorando o código.

TDD - Test Driven Development

❑ Um pouco da história de TDD

TDD ficou bastante popular após a publicação do livro **TDD: By Example, do Kent Beck**, em **2002**.

O próprio **Kent Beck** afirma que TDD não foi uma ideia totalmente original. Ele conta que em algum momento de sua vida, que leu em algum dos livros de seu pai (que também era programador), sobre uma técnica de testes mais antiga, onde o programador colocava na fita o valor que ele esperava daquele programa, e então o programador desenvolvia até chegar naquele valor.

TDD - Test Driven Development

Ele próprio conta que achou a ideia estúpida. Qual o sentido de escrever um teste que falha? Mas resolveu experimentar. Após a experiência, ele disse que **“as pessoas sempre falavam pra ele conseguir separar o que o programa deve fazer, da sua implementação final, mas que ele não sabia como fazer, até aquele momento em que resolveu escrever o teste antes.”**

Daquele momento em diante, Kent Beck continuou a trabalhar na ideia. Em 1994, ele escreveu o seu primeiro framework de testes de unidade, o SUnit (para Smalltalk). Em 1995, ele apresentou TDD pela primeira vez na OOPSLA (conferência muito famosa da área de computação, já que muitas novidades tendem a aparecer lá).

TDD - Test Driven Development

Já em **2000**, o **JUnit** surgiu e **Kent Beck,** junto com **Erich Gamma,** publicou o artigo chamado de **“Test Infected”**, que mostrava as vantagens de se ter testes automatizados e como isso pode ser viciante.

Finalmente em **2002**, Kent Beck lançou seu livro sobre isso, e desde então a prática tem se tornado cada vez mais popular entre os desenvolvedores.

A história mais completa pode ser vista na **Wiki da C2** ou na palestra do **Steve Freeman** e **Michael Feathers** na **QCON** de **2009**.

TDD - Test Driven Development

Duas histórias muito conhecidas cercam Kent Beck e o nascimento dos primeiros frameworks de testes unitários. O **SUnit**, para **Smalltalk**, não era um código distribuído em arquivos da maneira usual.

Beck na época atuava como consultor e tinha o hábito de recriar o framework junto com os programadores de cada cliente.

A criação do JUnit também é legendária: a primeira versão foi desenvolvida numa sessão de programação pareada entre o Kent Beck e o Erich Gamma em um voo Zurique-Atlanta.

TDD - Test Driven Development

Conseguimos identificar diversos **benefícios** no uso do **TDD** como:

- Testes Automatizados e Contínuos
- Identificação Precoce de Problemas
- Testes como Documentação
- Melhor Design de Classes

E todos eles impactam positivamente o processo de desenvolvimento de software.

O TDD é uma abordagem que, embora possa demandar um tempo inicial para se adaptar, traz benefícios substanciais ao longo do ciclo de vida do desenvolvimento do software, resultando em código mais robusto, confiável e de melhor qualidade.

BDD - Behavior Driven Development

O **BDD**, ou **Behavior-Driven Development** é uma abordagem de desenvolvimento de software que se concentra em **descrever o comportamento esperado** de acordo com os **requisitos de negócio** usando uma linguagem mais próxima à **linguagem natural**.

Para melhor entendimento dos requisitos de negócio, é utilizado **conversas estruturadas** sobre exemplos de uso e comportamento da funcionalidade, buscando o **entendimento compartilhado**.

Portanto, a ideia principal da construção do BDD é possibilitar que as funcionalidades do sistema sejam escritas em **linguagem natural**, possibilitando que todos os stackholders envolvidos estejam conversando na mesma lingua.

BDD - Behavior Driven Development

Basicamente, o BDD utiliza um conceito de metodologia que prioriza o compartilhamento pela equipe de desenvolvimento, pelo time da qualidade e pelo pessoal da área de negócios. Como resultado, tem-se um produto que responde à expectativa do cliente, com a otimização do tempo de todos os envolvidos no processo.

O **BDD**, é um complemento ou uma **evolução** do **TDD**, que é o Test Driven Development. Embora muitas pessoas acreditem que ele veio para substituir o TDD, criado por **Kent Beck.**

Isso porque, basicamente, o TDD propõe a elaboração de testes simples antes da definição do código. Mas como testar e o que avaliar?

BDD - Behavior Driven Development

Foi assim que, em meados do ano **2000**, **Dan North** apresentou a metodologia do BDD pela primeira vez.

A sua intenção era trazer pessoas não técnicas para o entendimento da testagem das funcionalidades dos programas, mantendo os testes.

Vem daí outro cunho comportamental da metodologia, pois quando se desenvolve um software corre-se o risco de deixar de lado algum conceito usado pela equipe de negócios ou ainda manter-se estritamente técnico.

BDD - Behavior Driven Development

Por consequência, não é apenas a equipe de desenvolvedores que escreve os cenários de testes. Sendo assim, três agentes interagem para criar o produto: o **Product Owner (PO)**, o **Quality Analyst (QA)** e o **Developer**. É o que **Georgie Dinwiddie** chamou de **"regra dos três amigos"**. Assim, se obtém melhores resultados na descrição dos testes.

❑ BDD é automação de testes?

Se você é um analista de teste e, algum dia, já pensou o que é BDD, tenho certeza de que a primeira coisa que veio à sua mente foi: **"Preciso instalar o Cucumber!!!"**

BDD - Behavior Driven Development

E após fazer a instalação, e conseguir criar o seu primeiro teste automatizado usando o Cucumber em parceria com alguma linguagem de programação, você com certeza pensou: **“Cara, eu estou automatizando em BDD!”**

Na verdade, **não**! Você “apenas” criou um teste automatizado, a partir de uma documentação executável desenvolvida em **Gherkin**, na qual é suportado pelo **Cucumber**. Ou seja, não houve aplicação do BDD nesse caso!

Mas por que muitos confundem ou associam o que é BDD unicamente a testes automatizados com Cucumber?

BDD - Behavior Driven Development

Tanto o **BDD** quanto o **Cucumber** têm uma pequena similaridade: **a utilização do Gherkin como padrão de escrita de critérios de aceitação**. E é nesse ponto que toda a confusão se inicia.

Não existe uma obrigatoriedade de escrever os critérios de aceitação no BDD utilizando Gherkin. Porém, **existe um aconselhamento por ser a linguagem que mais se aproxima da descrição do comportamento do usuário com a aplicação.**

Você pode automatizar testes de software utilizando **Cucumber/Gherkin** dentro da metodologia **BDD**. Mas você não está fazendo BDD, por estar apenas aplicando automação de testes com o Cucumber. É importante que essa separação de conceitos esteja clara para que você tenha sucesso no entendimento e na aplicação do que é BDD.

BDD - Behavior Driven Development

❑ Afinal, o que é o BDD?

Como dissemos antes BDD, é um **processo colaborativo** que envolve múltiplos membros do time, trabalhando em conjunto com o PO (Product Owner) para descobrir e refinar os requisitos usando, para isso, **conversas estruturadas** sobre exemplos de uso e comportamento de um sistema ou funcionalidade, buscando o entendimento compartilhado.

A alma do BDD, portanto, está na **conversa**, no **alinhamento constante** e, principalmente, no **entendimento compartilhado** entre todos os membros do time que estão envolvidos na parte de **regras de negócios** do desenvolvimento de uma história, geralmente **QAs, DEVs** e **POs**.

BDD - Behavior Driven Development

Para entendermos melhor essas conversas estruturadas vamos a alguns exemplos:

O BDD é dividido basicamente em 3 partes:

- História de usuário
- Funcionalidade
- Cenários

O processo de desenvolvimento do BDD se baseia na escrita de **cenários de testes**. Cada cenário é um exemplo escrito para ilustrar um aspecto específico de comportamento esperado da aplicação.

BDD - Behavior Driven Development

Para a escrita do BDD utilizamos uma **DSL (Domain-Specific Language)** conhecida como **Gherkin**, sendo uma linguagem estruturada que se inicia com a sintaxe **Given**, **When** e **Then (Dado que, Quando, Então)**.

- **Dado que (Given):** que especificam as pré-condições para que ocorra a ação de interesse do cenário.
- **Quando (When):** cuja função é especificar os eventos que devem ocorrer para que o cenário seja executado.
- **Então (Then):** que especificam as expectativas a respeito dos resultados da execução dos eventos do cenário.

BDD - Behavior Driven Development

As **Estórias BDD** e os cenários são escritos usando o padrão composto por palavras-chave:

- **Como:** Papel que desempenha alguma ação no produto em teste.
- **Eu quero:** Realizar alguma ação no produto sobre teste.
- **Para:** Algum resultado específico, ou que algo seja feito.
- **Dado:** Pré condições para o meu teste.
- **Quando:** Realizo alguma ação no meu produto em teste.
- **Então:** Alguma coisa deve acontecer.

Em alguns casos, para compor cenários e **Estórias BDD** mais complexas, a letra **“E”** é usada.

BDD - Behavior Driven Development

❑ Exemplo de história de usuário com cenários

Como: cliente do banco “I-bank” tenho o app do banco instalado no meu celular.
Eu quero: ser capaz de controlar a minha conta corrente a partir do app.
Para: ter melhor controle sobre meu dinheiro.

Cenário 1: realizar transferências da conta corrente

Dado: que eu esteja logado na minha conta do banco.
E: possua um limite de conta maior que 0.
Quando: eu tentar realizar a transferência de um valor maior que 0.
E: menor que o limite de transferência pelo app.
E: menor que o limite da minha conta.

BDD - Behavior Driven Development

Então: o banco deverá autorizar e realizar a transferência do dinheiro;
E: eu devo receber um sms confirmando a transação bancária.

Essa **“ausência de detalhes”** é de propósito. Ou seja, não queremos detalhar muito os nossos cenários, a ponto de eles ficarem exatamente iguais os casos de teste clássicos cheios de informação e impossíveis de se manterem atualizados.

BDD - Behavior Driven Development

❑ Exemplo 02

Como lojas “EDmais”
Eu quero permitir frete grátis para clientes
Para compra acima de R$100,00 de modo que o cliente seja estimulado a comprar mais.

Cenário 1: Taxa de frete será cobrada para compras abaixo de R$ 100,00

Dado que o cliente esteja realizando uma compra abaixo de R$100,00
Quando o cliente calcular seu frete
Então será exibido um valor de frete a ser cobrado

BDD - Behavior Driven Development

Como toda metodología, o BDD também possui um processo a ser seguido, passando pela **descoberta**, **definição**, **formalização** e **entrega** a automação dos testes aqui é uma opção, e não uma obrigação.

Porém, se estamos falando de times ágeis, nos quais o foco está, na maioria das vezes, em entregar de forma ágil, com o máximo de qualidade possível, é recomendável a automação de testes, tanto no nível unitário (usando TDD) quanto no nível funcional (usando ATDD).

A imagem abaixo ilustra, resumidamente, o processo do BDD.

BDD - Behavior Driven Development

BDD - Behavior Driven Development

1. Descoberta

- **Quando:** cerimônia de refinamento (ou Grooming)
- **Participantes:** QAs, DEVs, POs e quaisquer outros que possam contribuir com o refinamento da história.
- **Objetivo:** na fase de descoberta, o PO apresentará a história que fará parte de uma sprint, falando sobre a visão a nível de negócio, as funcionalidades, fluxos e regras de negócio que já foram mapeados por ele, para que todos os envolvidos possam ter o mesmo entendimento sobre a finalidade da história em questão.

BDD - Behavior Driven Development

Após a apresentação, todos os envolvidos iniciarão uma conversa estruturada, utilizando técnicas de BDD (que veremos mais à frente) para levantar exemplos de uso e comportamento do usuário com a funcionalidade em questão

A ideia é que sejam geradas dúvidas e que cada resposta possa virar uma nova regra de negócio, um critério de aceitação ou até mesmo uma nova história de usuário na fase de definição.

BDD - Behavior Driven Development

2. Definição

- **Quando:** cerimônia de refinamento (ou Grooming)
- **Participantes:** QAs, DEVs, POs e quaisquer outros que possam contribuir com o refinamento da história.
- **Objetivo:** com todas as perguntas necessárias, o objetivo é definir quais delas se tornaram regras de negócios, critérios de aceitação ou novas histórias.

Essas definições, logicamente, se darão por meio da conversa e alinhamento entre os envolvidos na cerimônia de refinamento. A ideia é que, após a definição, todos os itens levantados possam ser formalizados na fase seguinte.

BDD - Behavior Driven Development

3. Formalização

- **Quando:** geralmente, por precisar de um pouco mais de esforço para se construir uma nova história ou critérios de aceitação, é recomendado que a formalização seja realizada fora da cerimônia de refinamento.

- **Responsável:** QAs, DEVs, POs ou qualquer um que seja responsável pela criação dos artefatos.

- **Objetivo:** nesse caso, o objetivo é transcrever todos os itens levantados em uma linguagem amigável e de fácil entendimento por todos.

BDD - Behavior Driven Development

No caso dos critérios de aceitação, recomenda-se o uso do Gherkin. Porém, pode ser utilizado qualquer outro formato que seja realmente entendível por todos, como Protótipos, Wireframes (no caso de aplicações web ou mobile) ou Wiremocks (no caso de APIs).

BDD - Behavior Driven Development

4. Entrega

- **Quando:** geralmente, durante a cerimônia de review e durante todo o tempo em que a história estiver em produção.
- **Responsável:** QAs, DEVs, POs ou qualquer um possa apresentar o software de valor desenvolvido.
- **Objetivo:** após a execução de todo o desenvolvimento e testes da história, podemos então apresentar para o PO, durante a cerimônia de review para validação do entregável e posteriormente para produção.

Ideal que a funcionalidade seja monitorada após a promoção para o ambiente de produção, para que possam ser gerados feedbacks relacionados à utilização dos clientes na aplicação.

BDD - Behavior Driven Development

❑ Quais técnicas de BDD podemos utilizar?

Agora que você sabe melhor o que é BDD, saiba que existem várias técnicas que possibilitam o levantamento de requisitos, tais como specification workshops, discovery workshop, example mapping, 3 amigos, dentre outras.

Porém, vamos focar aqui nas duas principais técnicas de BDD:

- Example Mapping.
- 3 amigos.

BDD - Behavior Driven Development

❑ Example Mapping

Como dito anteriormente, dentro do processo de BDD, mais especificamente nas fases de descoberta e definição, será apresentada a história e, com isso, dúvidas sobre o exemplo de uso serão levantadas para que, posteriormente, possam se tornar uma regra de negócio e critérios de aceitação.

No caso, o Example Mapping é uma técnica que engloba essas duas fases e, de uma forma estruturada, ajuda a levantar todos os requisitos possíveis para cobrir toda a história.

BDD - Behavior Driven Development

BDD - Behavior Driven Development

Durante a apresentação do PO sobre a história e as regras de negócios levantadas previamente, os demais envolvidos escreverão em um post it as dúvidas que surgirem (cada dúvida em um post it diferente).

Após o fim da apresentação, será iniciada uma conversa de forma que todas as dúvidas sejam respondidas. Assim, as respostas deverão ser escritas em post its, conforme especificidade (critérios de aceitação, regras de negócios ou histórias).

O **time-box** padrão é de **25 minutos** para cada história, mas o recomendado é que no início não fique preso a ele, somente se atente para não fazer com que a reunião fique demorada, de acordo com o feeling da equipe. A reunião deve acabar quando todos os membros concordarem que a cobertura levantada para a história foi a máxima possível.

BDD - Behavior Driven Development

❑ 3 Amigos

A técnica **“3 amigos”** não é absolutamente diferente do “Example Mapping”. Porém, acredito ser mais simples e de fácil implementação. A diferença primordial é basicamente a não presença dos post its.

O PO continua apresentando a história, e **as dúvidas e respostas são realizadas de forma mais dinâmica, sem necessariamente aguardar o fim da apresentação para levantá-las ou para definir os critérios**.

É, de fato, mais dinâmico e menos “rígido”. Entretanto, é necessário que os envolvidos tenham o máximo de comprometimento em, de fato, conseguirem tirar o máximo de cobertura possível da história.

BDD - Behavior Driven Development

BDD - Behavior Driven Development

O ideal é que se inicie com Example Mapping, para conseguir entender a dinâmica e os outputs.

Quando estiverem confortáveis, fazer a migração para a “3 amigos”. Assim, acredito, terão mais sucesso na aplicação das técnicas.

❑ Mas e a automação de teste?

Sim! A automação está presente aqui, mas, como dito anteriormente, é opcional.

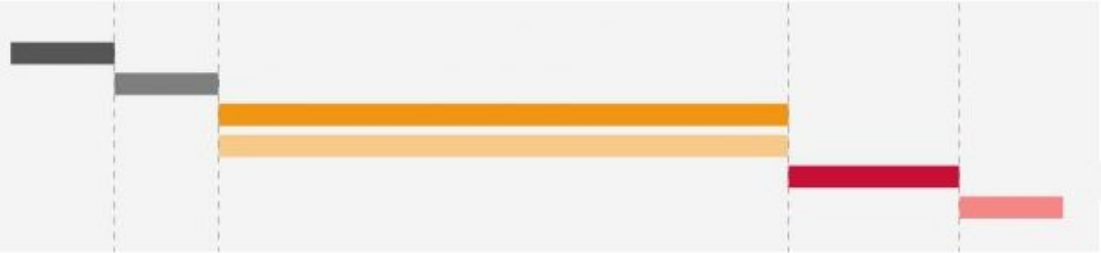
Se aplicado corretamente, o Behavior Driven Development abrirá uma excelente janela para que se possa criar os scripts de testes automatizados.

BDD - Behavior Driven Development

Isso acontecerá seja usando **Cucumber**, **Rest Assurance**, **Robot Framework**, **Postman** ou qualquer outra ferramenta de testes automatizados, de forma paralela com o desenvolvimento, agilizando não somente a execução dos testes, mas também a entrega da história para o PO e, consequentemente, para o ambiente de produção.

O único ponto de atenção é: se atentar para que os critérios de aceitação que serão escritos (geralmente por QA) não fiquem orientados a implementação ou aos testes que serão realizados, mas, sim, ao comportamento do usuário, que é, o que prega o BDD.

Dessa forma, temos a seguinte timeline a ser seguida:



Grooming

1. História dividida em pequenas tarefas
2. Criação dos critérios de aceitação
3. Clara definição de "pronto"
4. Entendimento compartilhado

Planning

1. Planejamento das execuções das tarefas
2. Estimativas das tarefas (criadas na Grooming)
3. Clara definição de "pronto"
4. Entendimento compartilhado

Desenvolvimento das tarefas

1. Entrega da funcionalidade ou novo produto no ambiente de teste
2. Código com testes unitários executados

Automação dos testes

1. Scripts de testes automatizados criados com base nos critérios de aceitação

Execução dos testes e ajuste de erros

1. Detecção de bugs
2. Relatório de execução

Review

1. Entrega de valor para o PO

**Referente à próximas sprints*

BDD - Behavior Driven Development

❑ BDD, TDD e ATDD: o que são e quais as diferenças?

Mas além de saber o que é BDD, é essencial que você, enquanto Product Owner, também saiba o que é TDD, ATDD e as principais diferenças entre os três conceitos. Então, vamos às explicações!

❑ TDD

Nesse modelo, os desenvolvedores criam códigos fontes de um determinado teste automatizado com o propósito que esse falhe para, somente depois, codificar a função necessária para o teste ser validado. O principal objetivo com o Desenvolvimento Orientado a Testes é garantir a qualidade do software.

BDD - Behavior Driven Development

❑ ATDD

Abreviação do termo em inglês **Acceptance Test-Driven Development, ATDD** é **Desenvolvimento Orientado a Testes de Aceitação**.

Nessa ferramenta de desenvolvimento ágil, todos os critérios e requisitos de aceite para cada história são levantados de forma colaborativa, ilustrando também possíveis cenários de testes.

Para isso, no entanto, é necessário envolver todo o time, incluindo o Product Owner, desenvolvedores, designers, entre outros profissionais que, por meio de reuniões fazem o levantamento dos problemas que precisam ser sanados com o software que está sendo desenvolvido, expectativas do cliente, e outros pontos relacionados.

BDD - Behavior Driven Development

Lembra da técnica “3 amigos” que citamos? Pois então, ela tende a ser bastante utilizada no conceito ATDD para obtenção das perguntas e respostas necessárias para a evolução do projeto.

❑ Principais diferenças entre BDD, TDD e ATDD

Para entender as principais diferenças entre **BDD**, **TDD** e **ATDD**, a primeira ideia que você precisa ter em mente é que o Behavior Driven Development foi criado com o intuito de substituir o Test Driven Development.

Na verdade, o BDD veio para complementar o conceito já aplicado no TDD, deixando essa ferramenta de desenvolvimento ágil ainda mais completa, dinâmica e precisa.

BDD - Behavior Driven Development

Com isso em mente, saiba que as três técnicas visam o uso de testes de código, evolução e integração contínua. Em outras palavras, todas são orientadas a testes.

Entretanto, apesar do ponto de semelhança, há uma diferença sutil entre elas. Por exemplo, no Test Driven Development os testes são escritos e validados com o objetivo de que funcionem corretamente. No Behavior Driven Development a descrição é sobre como um determinado problema no software deve se comportar.

O Acceptance Test-Driven Development, por sua vez, contempla também a idealização de diferentes cenários possíveis de realizar os testes.

BDD - Behavior Driven Development

❑ Quais são as vantagens ao escolher o BDD?

Mas se após conhecer todas essas ferramentas de desenvolvimento ágil você resolveu ficar com a BDD, saiba que pode obter uma série de vantagens com o seu uso, desde que esse seja feito corretamente.

Entre os benefícios do Behavior Driven Development que mais se destacam estão:

Visão global do projeto: Por meio do uso do BDD é possível desenhar cenários e testar a solução antes do projeto estar finalizado. Com isso, os profissionais têm uma visão ampla de tudo o que pode acontecer e de falhas que podem ocorrer, se antecipando a elas previamente à entrega do produto. Isso evita erros e fracassos posteriores, bem como otimiza o alcance de resultados mais expressivos e significativos.

BDD - Behavior Driven Development

Melhoria da comunicação: Devido a sua característica de reunir os envolvidos para bater diferentes possibilidades, a estratégia Behavior Driven Development melhorar a comunicação do time e dos profissionais envolvidos.

Em outras metodologias, geralmente, o trabalho é individual, o que pode impactar na comunicabilidade, gerar ruídos nas conversações e, consequentemente, afetar o resultado do projeto.

Documentações dinâmica: Os frameworks do BDD viabilizam a emissão de documentações automaticamente. Essa dinâmica, por sua vez, otimiza o tempo do PO e da sua equipe, tornando a geração documental muito mais prática e rápida.

BDD - Behavior Driven Development

❑ Vamos para parte prática do BDD

Para configurar o nosso ambiente, vamos precisar instalar a biblioteca do **SpecFlow** que é um framework inspirado no **Cucumber**, que vai nos ajudar na escrita de testes em **BDD** com **Gherkin**.:

Com o SpecFlow instalado, vamos adicionar uma extensão que é utilizada para escrever arquivos .feature que suporte a linguagem **Gherkin**

BDD - Behavior Driven Development

Para instalar a extensão basta:

1. Abrir o Visual Studio Code.
2. Vá para a barra lateral esquerda e clique no ícone de **"Extensions"** (ou use Ctrl + Shift + X).
3. Pesquise por **"Cucumber (Gherkin) Full Support"**.
4. Instale a extensão oferecida pela **alexkrechik**.

BDD - Behavior Driven Development

Vamos demonstrar esse passo usando o exemplo do cenário de identificar o maior e o menor produto por preço no carrinho de compras, que implementamos na seção de testes de unidade. Agora vamos utilizar a linguagem estruturada suportada pelo **Gherkin,** para montar os cenários de testes, para isso, vamos criar um arquivo chamado **MaiorMenorProduto.feature**.

```
Feature: Maior e Menor Produto por Preço no Carrinho de Compras

  Como um cliente interessado em tomar decisões com base no preço dos produtos
  Eu quero ser capaz de identificar o maior e o menor produto no meu carrinho de compras
  Para tomar decisões informadas de compra

  Scenario: Identificar o maior e o menor produto por preço no carrinho de compras
    Given Eu adicionei alguns produtos ao meu carrinho de compras
    When Eu seleciono a opção para identificar o maior e o menor produto por preço no carrinho de compras
    Then Devo ver as informações do produto mais caro e mais barato no meu carrinho de compras

    And Eu devo ver o maior produto exibido primeiro
    And Eu devo ver o menor produto exibido depois do maior produto
```

BDD - Behavior Driven Development

Neste cenário, estamos utilizando os passos **Given**, **When**, e **Then** para descrever o **estado inicial**, a **ação** que estamos realizando, e as **verificações** que esperamos realizar.

Agora precisamos realizar a implementação dos passos de teste associados ao **cenário de Gherkin** no código.

Vamos criar a implementação para o cenário de **“Identificar o maior e o menor produto por preço no carrinho de compras”**, será bem parecido com o que fizemos na implementação de **testes de unidade** anteriormente.

BDD - Behavior Driven Development

Vou criar a classe de passos de teste **MaiorMenorSteps.cs**

```
using TechTalk.SpecFlow;
using TestesAutomatizados;
using Xunit;

[Binding]
0 references
public class MaiorMenorSteps
{
    0 references
    private CarrinhoDeCompras carrinho;
    0 references
    private MaiorEMenor algoritmo;

}
```

BDD - Behavior Driven Development

Agora vamos implementar os métodos associados aos passos Gherkin. Lembre-se de que os métodos devem ser marcados com as anotações do SpecFlow (**[Given]**, **[When]**, **[Then]**).

Aqui montamos o cenário **Given (Dado que):**

```
[Given(@"Eu adicionei alguns produtos ao meu carrinho de compras")]
0 references
public void GivenEuAdicioneiAlgunsProdutosAoMeuCarrinhoDeCompras()
{
    carrinho = new CarrinhoDeCompras();
    carrinho.Adiciona(new Produto("Jogo de pratos", 70.0M));
    carrinho.Adiciona(new Produto("Geladeira", 450.0M));
    carrinho.Adiciona(new Produto("Liquidificador", 250.0M));
}
```

BDD - Behavior Driven Development

Repare que na **Notação Given**, colocamos exatamente a descrição do arquivo **MaiorMenorProduto.feature**

MaiorMenorProduto.feature

```
Scenario: Identificar o maior e o menor produto por preço no carrinho de compras
  Given Eu adicionei alguns produtos ao meu carrinho de compras
```

MaiorMenorSteps.cs

```
[Given(@"Eu adicionei alguns produtos ao meu carrinho de compras")]
```

É assim que o **SpecFlow** consegue **associar** o **comportamento** ao **teste.**

BDD - Behavior Driven Development

Vamos implementar o passo **When (Quando),** nesse passo temos que realizar uma **ação** no produto de teste.

```
[When(@"Eu seleciono a opção para identificar o maior e o menor produto por preço no carrinho de compras")]
0 references
public void WhenEuSelecionoAOpcaoParaIdentificarOMaiorEOmenorProdutoPorPrecoNoCarrinhoDeCompras()
{
    algoritmo = new MaiorEMenor();
    algoritmo.Encontra(carrinho);
}
```

BDD - Behavior Driven Development

Agora realizamos a validação do nosso teste **Then (Então).**

Aqui estamos validando se o produto mais caro realmente possui o valor maior que o mais barato.

```
[Then(@"Devo ver as informações do produto mais caro e mais barato no meu carrinho de compras")]
0 references
public void ThenDevoVerAsInformacoesDoProdutoMaisCaroEMaisBaratoNoMeuCarrinhoDeCompras()
{
    var produtoMaisCaro = algoritmo.Maior;
    var produtoMaisBarato = algoritmo.Menor;

    Assert.NotNull(produtoMaisCaro);
    Assert.NotNull(produtoMaisBarato);

    Assert.True(produtoMaisCaro.Valor > produtoMaisBarato.Valor);
}
```

BDD - Behavior Driven Development

Repare que se rodarmos os testes nesse momento, os testes falham.

Porque no arquivo de especificação do teste **(MaiorMenorProduto.feature)**, atribuímos condições para que esse teste passe utilizando a palavra chave **And (E)**

```
Scenario: Identificar o maior e o menor produto por preço no carrinho de compras
  Given Eu adicionei alguns produtos ao meu carrinho de compras
  When Eu seleciono a opção para identificar o maior e o menor produto por preço no carrinho de compras
  Then Devo ver as informações do produto mais caro e mais barato no meu carrinho de compras

  And Eu devo ver o maior produto exibido primeiro
  And Eu devo ver o menor produto exibido depois do maior produto
```

BDD - Behavior Driven Development

Então vamos codificar essas condições

```
[Then(@"Eu devo ver o maior produto exibido primeiro")]
0 references
public void ThenEuDevoVerOMaiorProdutoExibidoPrimeiro()
{
    var produtosNoCarrinho = carrinho.OrdenarPorMaisCaro();
    var indiceMaiorProduto = produtosNoCarrinho.IndexOf(carrinho.ObterProdutoMaisCaro());

    Assert.True(indiceMaiorProduto == 0);
}

[Then(@"Eu devo ver o menor produto exibido depois do maior produto")]
0 references
public void ThenEuDevoVerOMenorProdutoExibidoDepoisDoMaiorProduto()
{
    var produtosNoCarrinho = carrinho.OrdenarPorMaisCaro();
    var indiceMenorProduto = produtosNoCarrinho.IndexOf(carrinho.ObterProdutoMaisBarato());
    var indiceMaiorProduto = produtosNoCarrinho.IndexOf(carrinho.ObterProdutoMaisCaro());

    Assert.True(indiceMenorProduto > indiceMaiorProduto);
}
```

BDD - Behavior Driven Development

Para realizar esse teste criei alguns métodos da classe **CarrinhoDeCompras.cs**

```
2 references
public Produto ObterProdutoMaisCaro()
{
    var algoritmo = new MaiorEMenor();
    algoritmo.Encontra(this);

    return algoritmo.Maior;
}
```

```
1 reference
public Produto ObterProdutoMaisBarato()
{
    var algoritmo = new MaiorEMenor();
    algoritmo.Encontra(this);

    return algoritmo.Menor;
}
```

```
2 references
public List<Produto> OrdenarPorMaisCaro()
{
    var produtosOrdenados = Produtos.OrderByDescending(item => item.Valor).ToList();
    return produtosOrdenados;
}
```

BDD - Behavior Driven Development

Agora os testes passaram.

Com o **BDD** os testes estão presentes em todos os ciclos do desenvolvimento, não só para codificação, mas pela experimentação da ideia desde sua concepção.

Perceba que a forma de escrita do BDD agora nos fornece padrões que nos permitem estabelecer uma **linguagem ubíqua** de análise do software muito próxima da maneira como o **usuário entende o negócio**. A partir deste **contexto de trabalho** você conseguirá construir um vínculo muito mais claro e objetivos com todos os **stackholders** envolvidos.

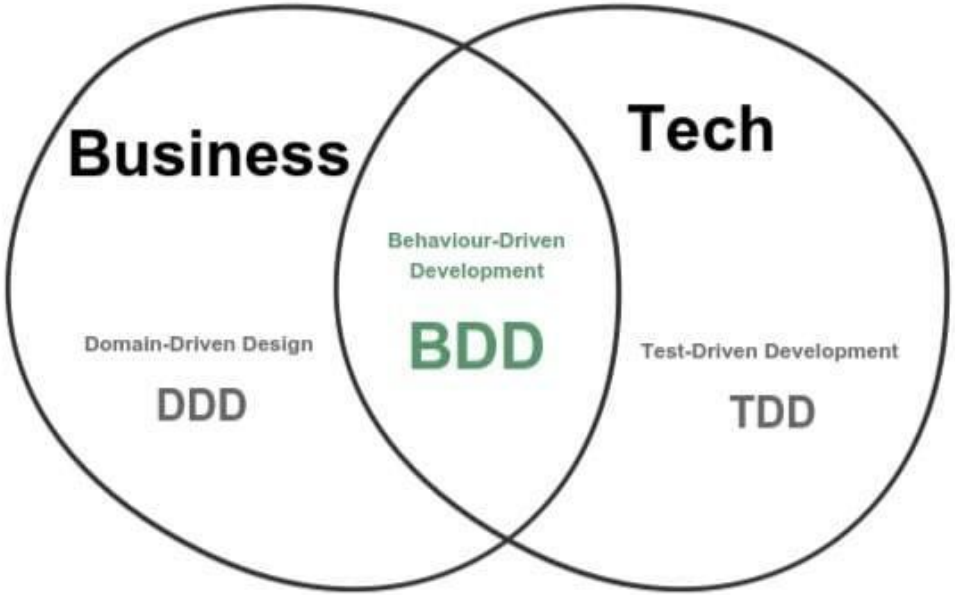
BDD - Behavior Driven Development

Agora a documentação e entendimento do contexto da funcionalidade que normalmente é uma dificuldade para o desenvolvimento de software, passa a ser parte presente do sistema. O documento é **“vivo”**, sempre sendo atualizado junto às funcionalidades e executável nas especificações de testes.

```
Feature: Maior e Menor Produto por Preço no Carrinho de Compras

  Como um cliente interessado em tomar decisões com base no preço dos produtos
  Eu quero ser capaz de identificar o maior e o menor produto no meu carrinho de compras
  Para tomar decisões informadas de compra

  Scenario: Identificar o maior e o menor produto por preço no carrinho de compras
    Given Eu adicionei alguns produtos ao meu carrinho de compras
    When Eu seleciono a opção para identificar o maior e o menor produto por preço no carrinho de compras
    Then Devo ver as informações do produto mais caro e mais barato no meu carrinho de compras

    And Eu devo ver o maior produto exibido primeiro
    And Eu devo ver o menor produto exibido depois do maior produto
```

BDD - Behavior Driven Development

Concluindo, o **BDD** é uma prática de desenvolvimento de software que consiste em escrever o comportamento do aplicativo em linguagem natural antes de testar e desenvolver.



BDD - Behavior Driven Development

O **BDD** seria outra versão do **TDD**, com testes baseados em comportamento escritos antes do desenvolvimento. Como o TDD, o BDD consiste em 3 fases **Given (Dado que), When (Quando), Then (Então):**

1. Começamos descobrindo colaborativamente o escopo do comportamento exigido pela história.

2. Uma vez que concordamos com esse comportamento, escrevemos a especificação em linguagem legível para negócios.

3. Finalmente, automatizamos a especificação escrita para verificar se o sistema realmente se comporta como esperado.

BDD - Behavior Driven Development

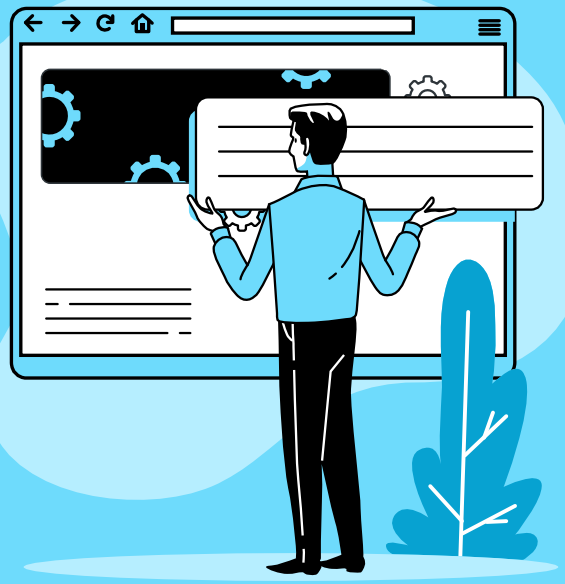
❑ Em suma:

- **BDD:** Prática de Desenvolvimento de Software.
- **Cucumber:** Ferramenta que converte texto em etapas para automação.
- **Gherkin:** É um conjunto de regras gramaticais que envolvem palavras-chave que permitem que o Cucumber transforme texto escrito em linguagem natural em etapas para automação. **Given**, **When**, **And**, **Then** são algumas dessas palavras-chave.

5.

Cultura e Adoção

Educação e conscientização sobre a importância dos testes automatizados.



Cultura e Adoção

Será que testar software é realmente importante?

Nesta sessão vamos discutir um pouco sobre as consequências de software não testado e a importância da **cultura** e **adoção** de testes automatizados.

Software de má qualidade é um problema para a sociedade como um todo, já que hoje softwares estão em todos os lugares.

Estimular uma cultura de testes é essencial para o sucesso, vamos falar sobre as razões pelas quais investir em cultura de testes vá beneficiar o seu time e empresa.

Porque não testamos ?

Imagine a seguinte situação, um médico ao longo de uma cirurgia, nunca abre mão de qualidade. Se o paciente falar para ele: “Doutor, o senhor poderia não lavar a mão e terminar a cirurgia 15 minutos mais cedo?”, tenho certeza que o médico negaria na hora.

Ele saberia que chegaria ao resultado final mais rápido, mas a chance de um problema é tão grande, que simplesmente não valeria a pena.

Em nossa área, acontece justamente o contrário. Qual desenvolvedor nunca escreveu um código de má qualidade de maneira consciente? Quem nunca escreveu uma “gambiarra"? Quem nunca colocou software em produção sem executar o mínimo suficiente de testes para tal?

Porque não testamos ?

Não há desculpas para não testar software. E a solução para que seus testes sejam sustentáveis é automatizando. Testar é divertido, aumenta a qualidade do seu produto, e pode ainda ajudá-lo a identificar trechos de código que foram mal escritos ou projetados.

Não existe desenvolvedor que **não saiba** que a solução para o problema é testar seus códigos. A pergunta é: por que não testamos?

Não testamos, porque testar sai caro. Imagine o sistema em que você trabalha hoje. Se uma pessoa precisasse testá-lo do começo ao fim, quanto tempo ela levaria? Semanas? Meses? Pagar um mês de uma pessoa a cada mudança feita no código (sim, os desenvolvedores também sabem que uma mudança em um trecho pode gerar problemas em outro) é simplesmente impossível.

Porque não testamos ?

Testar sai caro, no fim, porque estamos pagando “a pessoa” errada para fazer o trabalho.

Acho muito interessante a quantidade de tempo que gastamos criando soluções tecnológicas para resolver problemas “dos outros”. Por que não escrevemos programas que resolvam também os nossos problemas?

Vamos listar algumas razões que geralmente acontece em empresas que não possui a cultura e adoção de testes automatizados, para que você consiga identificar e compreender as desvantagens de atuar neste tipo de ambiente.

Porque não testamos ?

Falta de Conscientização: Alguns desenvolvedores podem não estar totalmente conscientes da importância dos testes ou dos benefícios que eles proporcionam. Eles podem subestimar o impacto positivo que os testes têm na qualidade do software.

Pressão por Prazos: Em ambientes onde os prazos são apertados e há uma pressão constante para entregar rapidamente, os testes podem ser negligenciados em prol da entrega rápida do código.

Cultura Organizacional: Em algumas empresas, a cultura pode não priorizar os testes automatizados. Se os líderes e equipes não enfatizam a importância dos testes, os desenvolvedores podem não se sentir incentivados a realizá-los.

Porque não testamos ?

Complexidade do Código ou Arquitetura: Em sistemas complexos ou mal estruturados, pode ser mais difícil escrever testes automatizados eficazes. Isso pode desencorajar os desenvolvedores a investir tempo nessa área.

Falta de Integração nos Processos de Desenvolvimento: Se os testes não forem integrados diretamente no fluxo de trabalho de desenvolvimento, os desenvolvedores podem ver os testes como uma tarefa separada e adicional, em vez de uma parte essencial do processo.

Porque não testamos ?

É importante destacar que a falta de testes automatizados não é necessariamente por falta de vontade dos desenvolvedores, mas muitas vezes é resultado de uma combinação de fatores, incluindo cultura organizacional, recursos disponíveis e conhecimento técnico.

É crucial que as empresas incentivem e apoiem a prática de testes automatizados, reconhecendo sua importância na entrega de software de alta qualidade.

Agora vamos falar dos benefícios que adotar a cultura de testes automatizados causa no dia a dia do desenvolvedor.

Benefícios

Garantia de Qualidade: Os testes automatizados ajudam a garantir a qualidade do software. Uma cultura que valoriza os testes automatizados implica em um compromisso com a entrega de produtos de alta qualidade aos clientes.

Eficiência e Rapidez: Automatizar os testes permite uma verificação mais rápida e eficiente do software. Isso acelera o ciclo de desenvolvimento, permitindo que novas funcionalidades sejam implementadas e lançadas mais rapidamente.

Economia de Tempo e Recursos: Ao substituir parte dos testes manuais por testes automatizados, a empresa economiza tempo e recursos. Isso reduz a dependência de equipes grandes realizando testes repetitivos e permite que os recursos humanos se concentrem em tarefas mais criativas e de maior valor agregado.

Benefícios

Detecção Precoce de Problemas: Testes automatizados podem identificar problemas assim que o código é alterado, permitindo correções imediatas. Isso ajuda a evitar que bugs se acumulem e se tornem mais difíceis e caros de corrigir no futuro.

Manutenção da Confiança do Cliente: Clientes confiam em software estável e de alta qualidade. Uma cultura que valoriza testes automatizados está alinhada com a expectativa do cliente em relação a produtos confiáveis e consistentes.

Facilidade de Mudança e Inovação: Com testes automatizados robustos, as equipes podem realizar mudanças no software com mais confiança. Isso encoraja a inovação e a experimentação, sabendo que há uma rede de segurança (os testes automatizados) para detectar possíveis problemas.

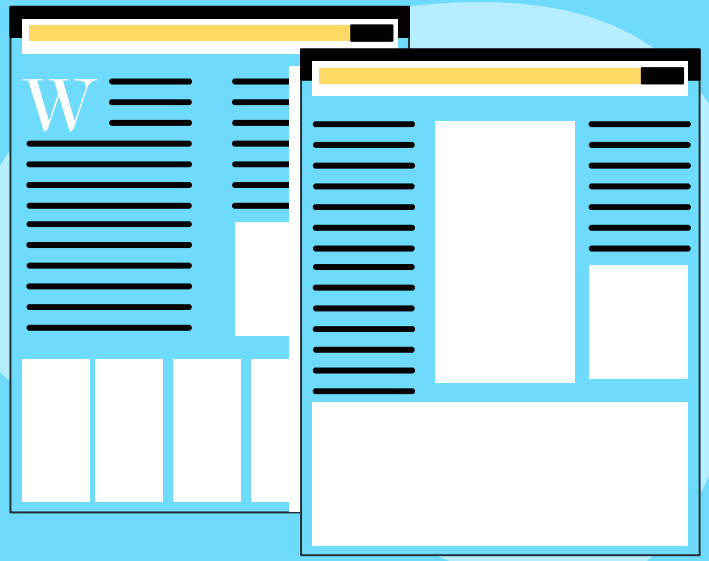
Conclusão

Em resumo, ter uma cultura e adoção de testes automatizados não apenas melhora a qualidade do software, mas também influencia positivamente a eficiência, a agilidade, a confiança do cliente e a capacidade de inovação de uma empresa de desenvolvimento de software.

É uma abordagem que não só aprimora o produto final, mas também transforma a maneira como as equipes trabalham e entregam valor aos clientes.

6. Outros Tipos de Testes

Testes de aceitação, regressão, carga e segurança.



Tipos de Testes

Existem diferentes tipos de testes que podem ser usados para abordar aspectos específicos do software.

Aqui estão alguns dos testes mais específicos:

- Testes Funcionais
- Testes de Aceitação do Usuário (UAT - User Acceptance Testing)
- Testes de Regressão
- Testes de Performance
- Testes de Usabilidade
- Testes de Segurança

Testes Funcionais

Os **Testes Funcionais** são uma categoria de testes que visam validar se um sistema ou software funciona conforme o esperado, de acordo com os requisitos funcionais estabelecidos para ele. Eles se concentram nas funcionalidades específicas do software e verificam se essas funções estão operando corretamente.

❑ Características de Testes Funcionais:

Foco nas Funcionalidades: Os testes funcionais examinam se cada funcionalidade do software está operando conforme o esperado, executando ações específicas e verificando se os resultados correspondem ao previsto.

Testes Funcionais

Baseados em Requisitos: São desenvolvidos com base nos requisitos funcionais do software, ou seja, as expectativas de como o software deve se comportar em termos de funcionalidade.

Independência da Implementação: Geralmente, os testes funcionais são Independente em relação à implementação. Eles se concentram apenas no que o software deve realizar, não nas peculiaridades de como foi implementado.

Testes Funcionais

Os testes funcionais desempenham um papel crucial na garantia da qualidade do software, assegurando que ele atenda às expectativas e funcione conforme o desejado pelos usuários finais.

- **O que são:** Testes que verificam a interação entre diferentes unidades de código, módulos ou sistemas.
- **Objetivo:** Verificar se as funções e recursos do software estão em conformidade com as expectativas do usuário e as especificações.
- **Benefícios:** Garantia de que o software realiza as ações esperadas.

Testes de Aceitação

Testes de Aceitação do Usuário ou **UAT** (User Acceptance Testing), são testes realizados pelo cliente, com dados do cliente. Os resultados desses testes irão determinar se o cliente está de acordo ou não com a implementação realizada. Se estiver de acordo, o sistema pode entrar em produção. Se não estiver de acordo, os devidos ajustes devem ser realizados.

Por exemplo, quando se usa métodos ágeis, uma história somente é considerada completa após passar por testes de aceitação, realizados pelos usuários, ao final de um sprint.

Testes de Aceitação

Testes de aceitação possuem duas características que os distinguem de todos os testes que estudamos antes. Primeiro, são testes manuais, realizados pelos clientes finais do sistema. Segundo, eles não constituem exclusivamente uma atividade de verificação (como os testes anteriores), mas também uma atividade de validação do sistema.

Lembre-se da Seção de Introdução: **verificação** testa se fizemos o sistema corretamente, isto é, de acordo com a sua especificação e/ou requisitos. Já **validação** testa se fizemos o sistema correto, isto é, aquele que o cliente pediu e precisa.

Testes de Aceitação

Testes de aceitação podem ser divididos em duas fases. Testes alfa são realizados com alguns usuários, mas em um ambiente controlado, como a própria máquina do desenvolvedor. Se o sistema for aprovado nos testes alfa, pode-se realizar um teste com um grupo maior de usuários e não mais em um ambiente controlado. Esses testes são chamados de testes beta.

- **O que são:** Testes conduzidos pelos usuários finais para confirmar se o software está pronto para ser aceito e utilizado.
- **Objetivo:** Validar se o software atende aos requisitos de negócios e às necessidades dos usuários.
- **Benefícios:** Garantir que o produto final atenda às expectativas dos usuários finais.

Testes de Regressão

Os **Testes de Regressão** são uma parte essencial do processo de garantia de qualidade no desenvolvimento de software. Eles são projetados para verificar se as mudanças realizadas no código não afetaram negativamente as funcionalidades existentes.

Testes de regressão tem um único objetivo que é **Garantir Estabilidade**, verificando se as alterações recentes no código introduziram falhas ou quebraram funcionalidades previamente testadas e funcionando corretamente.

Testes de Regressão

Algumas das características dos testes de regressão são:

Foco nas Funcionalidades Críticas: Priorizam testes em funcionalidades cruciais ou áreas do sistema mais suscetíveis a regressões.

Reutilização de Testes Existentes: Muitas vezes, os testes de regressão são baseados em casos de teste existentes, concentrando-se nas partes do sistema mais propensas a serem afetadas por alterações.

Execução Automatizada: A automação é frequentemente empregada nos testes de regressão para garantir que eles possam ser executados de maneira rápida e consistente sempre que houver alterações no código.

Testes de Regressão

Tipos de testes de regressão:

Testes Unitários: Garantem que as alterações feitas em um módulo não afetem as unidades individuais do software.

Testes de Integração: Verificam se as mudanças introduzidas não quebraram a integração entre os componentes.

Testes Funcionais: Certificam-se de que as funcionalidades principais permaneçam operacionais após as mudanças.

Testes de Desempenho: Asseguram que o desempenho do sistema não tenha sido comprometido pelas alterações.

Testes de Regressão

Benefícios dos testes de regressão:

Detecção Precoce de Problemas: Identificação rápida de falhas após alterações no código.

Preservação da Qualidade: Garantia de que as funcionalidades existentes permaneçam intactas mesmo com atualizações ou modificações no software.

Testes de Regressão

Os testes de regressão desempenham um papel vital na manutenção da estabilidade e na garantia de que o software permaneça funcional, mesmo após as modificações e atualizações frequentes durante o ciclo de vida do desenvolvimento.

- **O que são:** Testes que verificam se as alterações recentes no código afetaram negativamente o funcionamento de funcionalidades existentes.
- **Objetivo:** Garantir que as atualizações ou novas implementações não quebraram funcionalidades previamente testadas e funcionando.
- **Benefícios:** Evitar regressões ao garantir a estabilidade do software após modificações.

Testes de Performance

Os **Testes de Desempenho** são uma categoria de testes que visam avaliar o desempenho de um sistema sob condições específicas, como carga de usuários, volume de dados ou demanda de processamento.

O objetivo principal é identificar como o sistema se comporta em situações de uso intenso e garantir que ele mantenha seu **desempenho aceitável**.

Testes de Performance

Características dos Testes de Desempenho:

Simulação de Carga: Os testes simulam situações de uso intenso para avaliar o comportamento do sistema sob pressão.

Medição de Métricas: Métricas como tempo de resposta, tempo de carregamento, uso de recursos (CPU, memória, rede) são monitoradas e analisadas.

Identificação de Gargalos: Procuram por áreas do sistema que podem se tornar gargalos em situações de carga alta.

Testes de Performance

Tipos de Testes de Desempenho:

Testes de Carga: Avaliam o comportamento do sistema ao aumentar gradualmente a carga até atingir seu limite, identificando onde o desempenho começa a degradar.

Testes de Estresse: Avaliam como o sistema se comporta em situações extremas ou acima de sua capacidade máxima, identificando o ponto de falha.

Testes de Volume: Verificam como o sistema lida com grandes volumes de dados, identificando possíveis problemas de desempenho ao lidar com grandes conjuntos de informações.

Testes de Resiliência: Avaliam a capacidade do sistema de se recuperar de falhas ou picos de carga, verificando sua resiliência e estabilidade.

Testes de Performance

Características dos Testes de Desempenho:

Simulação de Carga: Os testes simulam situações de uso intenso para avaliar o comportamento do sistema sob pressão.

Medição de Métricas: Métricas como tempo de resposta, tempo de carregamento, uso de recursos (CPU, memória, rede) são monitoradas e analisadas.

Identificação de Gargalos: Procuram por áreas do sistema que podem se tornar gargalos em situações de carga alta.

Testes de Performance

Benefícios dos Testes de Desempenho:

Identificação Antecipada de Problemas: Identificação precoce de gargalos e problemas de desempenho antes do lançamento do sistema em ambiente de produção.

Garantia de Qualidade do Desempenho: Assegurar que o sistema funcione de maneira eficiente e estável mesmo sob condições de uso intenso.

Testes de Performance

Os testes de desempenho são essenciais para garantir que um sistema possa lidar com demandas reais, mantendo um desempenho aceitável e proporcionando uma experiência positiva ao usuário final.

- **O que são:** Testes que avaliam como o software se comporta em termos de velocidade, escalabilidade e estabilidade sob diferentes condições de carga.
- **Objetivo:** Identificar gargalos de desempenho e problemas que possam surgir em situações de uso intenso.
- **Benefícios:** Garantir que o software funcione de maneira eficiente e estável em condições de uso variadas.

Testes de Usabilidade

Os **Testes de Usabilidade** são uma parte crucial do desenvolvimento de produtos, focados em entender como os usuários interagem e percebem um produto ou sistema.

O objetivo principal é garantir que o produto seja fácil de usar, eficiente e ofereça uma experiência positiva aos usuários.

Testes de Usabilidade

Características dos Testes de Usabilidade:

Centrados no Usuário: Colocam os usuários no centro do processo, observando suas interações, comportamentos e percepções.

Métodos Diversificados: Podem incluir entrevistas, questionários, observações diretas, testes de navegação, entre outros métodos para compreender diferentes aspectos da experiência do usuário.

Identificação de Problemas de Usabilidade: Permitem identificar pontos problemáticos no design ou na usabilidade do produto.

Testes de Usabilidade

Benefícios dos Testes de Usabilidade:

Melhoria da Experiência do Usuário: Identificação e correção de problemas de usabilidade para tornar o produto mais intuitivo e fácil de usar.

Aumento da Satisfação do Cliente: Oferece aos usuários uma experiência mais agradável, o que pode resultar em maior satisfação e fidelidade.

Testes de Usabilidade

Os testes de usabilidade são essenciais para garantir que um produto atenda às necessidades e expectativas dos usuários finais, ajudando a criar soluções mais eficientes e agradáveis de usar.

- **O que são:** São focados em entender como os usuários interagem e percebem um produto ou sistema.
- **Objetivo:** Garantir que o produto seja fácil de usar, eficiente e ofereça uma experiência positiva aos usuários.
- **Benefícios:** Oferecer aos usuários do sistema uma experiência mais agradável, o que pode resultar em maior satisfação e fidelidade.

Testes de Segurança

Os **Testes de Segurança**, são uma categoria de testes focados em identificar vulnerabilidades e pontos fracos nos sistemas de software, redes ou infraestrutura.

O objetivo é detectar **falhas de segurança** antes que sejam exploradas por indivíduos mal-intencionados.

Testes de Segurança

Características dos Testes de Segurança:

Análise Profunda: Examinam minuciosamente o sistema em busca de vulnerabilidades, explorando diferentes vetores de ataque.

Simulação de Ataques: Simulam ações que um invasor real poderia realizar para comprometer a segurança do sistema.

Melhoria Contínua: Os testes de segurança geralmente são realizados periodicamente, pois as ameaças e as técnicas de ataque estão em constante evolução.

Testes de Segurança

Tipos de Testes de Segurança:

SAST - Teste de Segurança Estático: SAST (Static Application Security Testing) é uma técnica de teste de segurança estática que verifica o código-fonte de um aplicativo em busca de vulnerabilidades e problemas de segurança potenciais. Diferentemente dos testes dinâmicos, que envolvem a execução do código para identificar falhas, o SAST analisa o código sem executá-lo, durante a fase de desenvolvimento.

DAST - Teste de Segurança Dinâmico: DAST (Dynamic Application Security Testing) é uma abordagem de teste de segurança que avalia a segurança de um aplicativo enquanto ele está em execução. Em contraste com o SAST, que analisa o código-fonte sem executá-lo, o DAST testa a aplicação em tempo real, interagindo com ela como um usuário faria.

Testes de Segurança

IAST - Teste de Segurança Interativo: IAST (Interactive Application Security Testing) é uma abordagem de teste de segurança que combina elementos do SAST e do DAST. Ele é conhecido por realizar a análise da aplicação de forma interativa e em tempo real, integrando-se ao ambiente de execução do aplicativo.

SCA - Análise de Composição de Software: SCA (Software Composition Analysis) é uma prática de segurança que visa identificar e gerenciar as bibliotecas de terceiros e componentes de código aberto usados em um aplicativo. O objetivo é identificar possíveis vulnerabilidades nessas dependências externas para garantir a segurança e a integridade do software.

Testes de Segurança

Pentest: O Pentest ou Teste de Intrusão simula ataques de cybercriminosos para avaliar a segurança de um sistema, rede ou aplicativo.

O objetivo é identificar e explorar vulnerabilidades de segurança, permitindo correções antes de possíveis ataques reais.

No final dos testes é feito um relatório detalhado das vulnerabilidades encontradas, juntamente com recomendações para mitigação e correção.

Testes de Segurança

Benefícios dos Testes de Segurança:

Prevenção de Ataques: Identificação e correção antecipada de falhas de segurança, reduzindo o risco de exploração por cybercriminosos.

Melhoria da Postura de Segurança: Aumento da conscientização sobre a importância da segurança e implementação de medidas para proteger o sistema.

Testes de Segurança

Os testes de segurança são cruciais para garantir que um sistema esteja protegido contra ameaças externas e internas, ajudando a manter a integridade, confidencialidade e disponibilidade dos dados e recursos.

- **O que são:** Testes que avaliam se o software possui falhas de segurança ou que foram desenvolvidos utilizando princípios e práticas de desenvolvimento seguro .

- **Objetivo:** Detectar falhas de segurança antes que sejam exploradas por indivíduos mal-intencionados.

- **Benefícios:** Garantir que o software funcione de maneira segura, mesmo sofrendo ataques de cybercriminiosos.

Conclusão

Esses são apenas alguns dos testes mais específicos que podem ser aplicados durante o ciclo de vida do desenvolvimento de software para garantir a qualidade e a funcionalidade do produto final.

Cada um tem seu propósito e contribui para garantir um software mais confiável e eficaz.

Obrigado!

Alguma Pergunta ?

- willian_brito00@hotmail.com
- linkedin.com/in/willian-ferreira-brito
- github.com/willian-brito