ENGENHARIA DE SOFTWARE I

CAPÍTULO 4 - TESTES. SABE QUE SÃO ELES QUE GARANTEM A QUALIDADE DO *SOFTWARE* ?

Mauricio Antonio Ferste

INICIAR

Introdução

O desenvolvimento de sistemas de *software* envolve uma série de atividades de produção, nas quais as oportunidades de injeção de falhas humanas são enormes. Erros podem começar a acontecer, logo no começo do processo. Podem existir erros referentes à especificação de requisitos, que surgem no código durante a implementação, ou até mesmo, devido a problemas de infraestrutura. Muitos dos erros devem-se à incapacidade que os seres humanos têm de executar e comunicar com perfeição. Essa é uma ideia complexa, pois trata, especificamente, de pessoas e não de *software*.

Existe um senso comum que acredita ser fácil resolver problemas com comunicação ruim, mas não é. Vamos entender que, para compreender e sanar os erros na construção do *software*, por falhas técnicas, ou de comunicação, o desenvolvimento deve sempre ser acompanhado por uma atividade que possa garantir a qualidade.

Pensar em uma rotina de trabalho que insira processos, como os testes, bem definidos, faz parte da busca por qualidade. Os testes de *software* são técnicas utilizadas a fim de encontrar defeitos no sistema, de forma a assegurar maior credibilidade em sua utilização. Para garantir que o código tenha qualidade, parece lógico testar o *software*, porém, a fase que menos é priorizada é a de testes, ocasionando, muitas vezes, desnecessárias manutenções corretivas posteriores. Isso ocorre, geralmente, porque o prazo de entrega do sistema não considera uma fase coerente de testes.

Para aprender a lidar com esses desafios, vamos desenvolver esse conteúdo e absorver as informações necessárias para desenvolver a fase de testes. Para isso, vamos nos guiar por alguns questionamentos: como realizar testes? Quais fluxos existem? Que tipos de testes existem?

Acompanhe com atenção e bons estudos!

4.1 Testes de *software*

Teste é o ato destinado a mostrar que o programa faz o que foi proposto fazer, para descobrir defeitos, antes do uso (SOMERVILLE, 2011). Testar é um ato simples. Ao executar o programa que acaba de ser desenvolvido, para verificar sua saída, já estaríamos testando, mas não é isso que se deseja. Um teste profissional, de fato, deve ser organizado, ter fases e objetivos claros para cada nível. E também não deve ser feito apenas com o desenvolvedor, os testes devem envolver o cliente, os usuários, os analistas de requisitos, enfim, todos os envolvidos em seu desenvolvimento. Cada um nessa cadeia dará sua visão e terá seu ponto de vista, desde o mais técnico, do implementador, até o mais abstrato do usuário, que vivencia o dia a dia do sistema.

Por uma questão lógica, iniciaremos os estudos pelos testes de desenvolvimento e, paulatinamente, outros tipos de testes que envolvam o cliente e o usuário final. Mas é importante compreender, logo de partida, que teste não é a última fase do desenvolvimento do *software*, ele deve ocorrer em todos os momentos.

Talvez o maior problema para ser resolvido, em termos de testes, seja este: entender e passar para a equipe que o código deve ser aberto e inspecionado por todos e de várias formas, a todo momento. Quanto mais vistoriado for o código,

menos erros haverão. Essa inserção de testes no mundo real é um grande desafio, pois devemos conscientizar a equipe sobre sua aplicação, de forma que não seja somente mais uma atividade no fim do processo.

4.1.1 Testes de desenvolvimento

Os testes de desenvolvimento são todas as atividades realizadas pelos envolvidos durante o processo de desenvolvimento, podendo ser feito por desenvolvedores e implementadores, ou por equipes de testes. O importante aqui, é realizar o teste, utilizando um processo automatizado, seguindo o RUP, ou XP, pois o código deve ter um mínimo de qualidade, antes de sair da implementação para testes com uma equipe independente, ou com o cliente.

Na figura a seguir, podemos ver que existem etapas bem definidas para a produção de testes com coerência. Os testes devem ser feitos com um planejamento, para se pensar o que vai ser testado e preparar dados (SOMMERVILLE, 2011). Esses passos iniciais são, muitas vezes, negligenciados, mas são fundamentais para otimizar a execução dos testes, para que os resultados possam ser comparados.

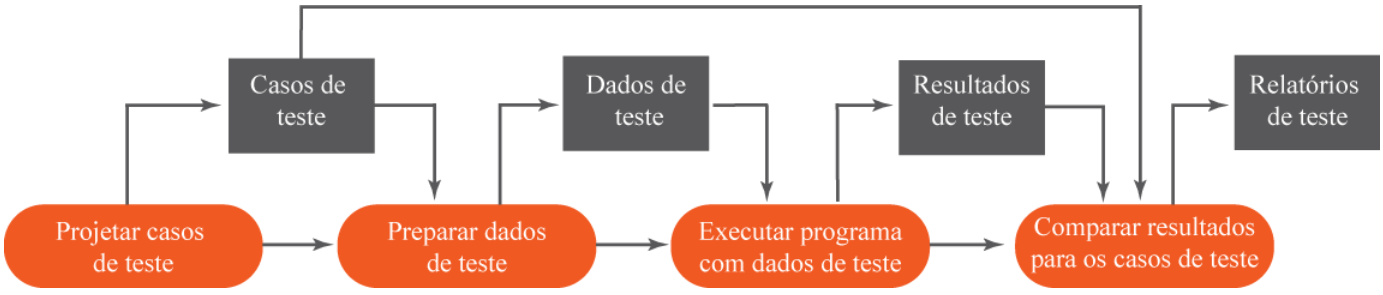


Figura 1 - Fluxo de teste proposto por Sommerville. Fonte: SOMMERVILLE, 2011, p. 147.

Durante o desenvolvimento, o teste pode ocorrer em três níveis de granularidade: unitário, componentes e sistema (SOMMERVILLE, 2011).

Teste unitário: antes de entender o que é um teste unitário, lembre-se que estamos falando de orientação a objetos, portanto, existem classes de objeto. Então, os testes unitários são os que feitos de forma individual, em classes, objetos, ou mesmo, métodos.

Teste de componentes: são testes aplicados a componentes, que devem ter um enfoque individual num determinado componente. Ou seja, deve-se testar o componente e seu funcionamento, quando ele é construído, por meio das junções de classes que o formam. Também testamos se a construção não gera erros novos no sistema, além de algum detectado no teste unitário.

Teste de sistema: entenda-se aqui que o teste de sistema não é com o cliente, chegaremos lá, mas no momento, temos a equipe de desenvolvimento desejando testar se tudo o que foi feito funciona corretamente em conjunto, tecnicamente falando.

Assim, em resumo, no teste unitário, focamos nos erros nas classes e objetos, que, quando unidos, formam componentes. Os objetos que formam os componentes, trocam mensagens entre si, e para garantir que não haja erro nelas, há o teste de componentes. E, por fim, quando se une os componentes para formar o sistema, eles podem apresentar erros entre si e, é aqui que entra o teste de sistema, que envolve várias integrações de componentes, tanto em nível de teste de componente, quanto de sistema, dá-se o nome de teste de integração.

4.1.2 Processo de testes durante o desenvolvimento

Existem várias formas de ver os testes durante o desenvolvimento. Estudamos os testes unitários, de componentes e de sistema, entretanto, eles podem ser vistos de outras formas, quanto ao grau com que interferem no código do sistema, tomando conhecimento do código implementado, ou somente considerando o sistema como um bloco único, verificando entradas e saídas. Dentre eles, vamos entender os testes de caixa preta e caixa branca, a seguir.

Teste caixa preta ou funcional

Teste caixa preta se importa somente com o resultado do teste, ou seja, com a saída, e não em como a estrutura trabalha para chegar neste resultado (PRESSMAN, 2016).

VOCÊ SABIA?

Existem diversas plataformas de teste que podem ser utilizadas, elas promovem a construção de testes e depois, permitem que se repitam os mesmos testes, de forma rápida e fácil. É um evidente investimento de tempo inicial, mas com grandes vantagens no fim, pois quanto maior é o sistema, mais difícil é testá-lo, e ter uma plataforma que realiza parte dos testes, minimiza custos e tempo utilizados.

Há desvantagens no teste em caixa preta. A primeira é que depende de uma boa formatação dos requisitos, pois deles que virão os resultados para comparação, ou seja, requisito bem escrito é fundamental, não somente para uma boa implementação, mas também para os testes. Outra desvantagem é que dificilmente consegue testar todas as entradas possíveis.

Existem vários formatos clássicos de teste de caixa preta, que são: partições em classe de equivalência, que veremos a seguir, grafos de causa-efeito e testes de condições de fronteira.

Teste de caixa branca

O teste de caixa branca examina o sistema, considerando aspectos internos e, ao contrário do de caixa preta, não considera somente as interfaces do sistema, mas todos os fluxos internos possíveis. O teste é mais complexo, pois, por ser baseado na implementação, considera aspectos do código, e caso estes mudem, o teste automaticamente também deverá ser alterado (PRESSMAN, 2016).

Atenção, pois pode parecer uma desvantagem utilizar um teste de caixa branca, já que ele atua diretamente no código, melhor seria usar logo um de caixa preta, pois é mais simples e não depende do código. Então seria mais fácil de implementar? Não é bem assim, os testes detectam erros excludentes entre si, ou seja, certos erros são detectados por uma caixa branca e não por um caixa preta, pois seus objetivos são diferentes. Tome cuidado, pois os testes de caixa preta, normalmente, são mais fáceis de fazer, pois não envolvem um nível tão complexo

de conhecimento de código, mas não vão detectar alguns problemas críticos de sistema (MYERS, 2004). Alguns exemplos de testes de caixa branca são a cobertura lógica de Myers (2004) e o método dos caminhos básicos.

No método dos caminhos básicos, é proposta a criação de um grafo contendo todos os caminhos possíveis, este método desdobrado, na prática, define que se deve contar o total de condicionais e de laços do sistema e somar um no total, para então definir a sua complexidade.

VOCÊ SABIA?

Existem várias denominações para diferentes tipos de testes, além das que já estudamos, listamos outros, a seguir.

Teste de regressão: utilizado quando diferentes versões de *software* são lançadas, verifica se o teste realizado antes da alteração, continua retornando como certo, depois da alteração.

Teste de carga: visa definir o limite de dados que o sistema suporta, por isso o termo “carga”. É indicado para avaliar requisitos não funcionais também.

Teste de *stress*: é uma variação do teste de carga, visando avaliar condições especiais, como falta de memória ou *hardware* ineficiente.

Teste de segurança: utiliza sistemas específicos para realizar ataques, detectar vulnerabilidades, ou qualquer outro tipo de ameaça que se necessite testar.

Como testar o teste?

Para saber se o teste vai funcionar, precisamos estabelecer uma metodologia. Ela deve analisar a efetividade de um teste, pois quando ele é escolhido e montado, como saber se ele cobriu todas as possibilidades desejadas?

Primeiro, ao implementar um teste de qualidade, existe uma dependência total de requisitos do sistema. O teste unitário reflete, além da qualidade do código, a qualidade das regras de negócio que foram definidas. São elas que devem ser testadas para validar o código. Começamos a testar o que é certo, e o que é errado, ou seja, verificamos resultados positivos e negativos.

Por exemplo, em um mundo ideal, pense que tudo seria testado. Se existe uma classe com um método, ele deve ser testado para os valores corretos e aceitos. E também deve ser testado para os que não são aceitos, cobrindo todo o escopo possível de testes. Obviamente, examinar tudo, ou seja, o teste exaustivo de todas as possibilidades, para todo o sistema, seria uma estratégia. Então, se não ocorrer falha, significa que o sistema é perfeito?

Bem, se isto fosse possível, e acredite, muitas vezes é, seria muito custoso, o que inviabiliza o valor final, ou, até mesmo, o prazo de um projeto. Logo, temos de adotar alguma técnica para mitigar os riscos e também minimizar o tempo gasto com testes. A seguir, temos alguns exemplos destas técnicas.

Definição de partição, ou partição de equivalência

A técnica de partição, tipo caixa branca, visa diminuir o número de testes para serem feitos. Vamos ver um exemplo: precisamos testar se um método está validando corretamente a idade de alguém com mais de 18 (dezoito) anos. Bem, neste caso, teríamos de testar todos os valores acima de 18. Logo, o teste tem de retornar uma solução verdadeira para 18, 19, 20, 21, … e assim por diante, da mesma forma tem de retornar falso para 17, 16, 15, 14, ...

Ou seja, são muitos testes para fazer. Logo, devemos pensar em simplificar o processo. Uma regra envolve a partição de valores.

Veja que o teste tem de ser verdadeiro, para qualquer valor igual, ou acima de 18, e falso para qualquer valor abaixo de 18. Logo, basta testar esta relação com 18, pois o número maior tem de retornar verdadeiro para o teste, e também 17, pois o valor abaixo de 17, tem de retornar falso.

Definição de valor limite

O teste de valor limite é mais uma diretriz na qual se afirma que, quando for testar um domínio de valores, devemos usar os valores limites, pois a possibilidade de encontrar um erro é maior (MYERS, 2004). Ou seja, no exemplo que vimos acima, como já é analisado o valor 18, já temos um teste de valor limite.

Elaborando o caso de testes baseados em diretrizes

O uso de teste de valor limite e partições de equivalência é um método eficaz para elaborar testes, uma vez tendo sido estabelecidas as partições de entrada e relacionando-as às partições de saída, fica mais fácil estabelecer o conjunto de valores, esperado para um conjunto de valores de entrada. Observe na figura a seguir, que, para uma entrada específica, uma determinada saída deve ser almejada.

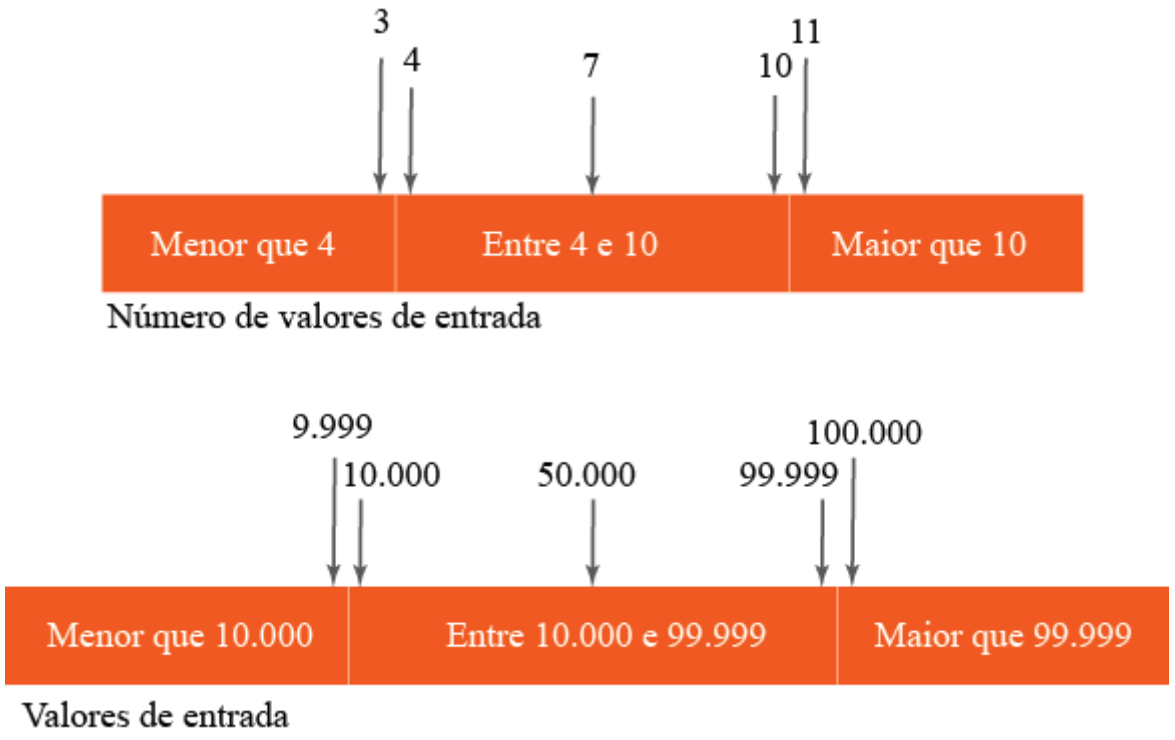


Figura 2 - Exemplo da criação de partições de teste. Fonte: SOMMERVILLE, 2011, p. 147.

Ainda segundo Sommerville (2011), temos que ter atenção a alguns detalhes e diretivas aplicáveis aos testes, como:

- não testar utilizando somente um valor, programadores, em geral, pensa em sequências, logicamente, pois um valor único, ou conjunto único, pode viciar os resultados;
- derivar testes para, quando necessário, acessar mais elementos aleatórios em uma lista, dando assim, mais flexibilidade ao teste;
- utilizar massas de testes de tamanhos e sequências diferentes.

VOCÊ QUER LER?

No livro "*The art of software testing*", Glenford Myers (2004) trouxe ao leitor um conhecimento mais avançado sobre testes, abordando as técnicas de *software* no ano de 1979. Este livro é considerado pelos profissionais da área, o marco inicial do teste de *software* moderno e, até hoje, é uma referência para quem quer entender como, historicamente, se inseriu uma metodologia e um processo para teste durante o desenvolvimento.

Em seu livro, Whitaker (2002) reforçou as ideias de Sommerville, com mais alguns conselhos, como:

- escolher entradas que forcem o sistema a gerar todas as mensagens de erros possíveis;
- projetar entradas que gerem *overflow*, ou estouros, propositalmente;
- repetir os mesmos testes mais de uma vez, esperando os mesmos resultados;
- obrigar a criação de saídas inválidas, com resultados muito grandes, ou muito pequenos e fora, até, do tipo de estruturas usadas.

Assim, seguindo essas premissas, temos o primeiro escopo de como devemos organizar os testes.

4.1.3 Desenvolvimento dirigido a testes

TDD, ou *Test-Driver Development*, ou então, desenvolvimento dirigido a testes, é uma técnica de desenvolvimento, não de testes. Então, é uma forma de implementar o código, já totalmente coberto por testes? Também não é exatamente isto. *Test-Driver Development*, ou TDD, como chamaremos daqui para frente, é uma técnica de implementação, na qual, primeiro cria-se o teste, com o

requisito, para, então, criar a unidade de implementação que atende. Este é um processo comum de desenvolvimento para realizar o TDD. Temos, então, o ciclo de vida, a seguir, para usar o TDD.

1. Escreva o teste.
2. Execute o teste (não há código de implementação, o teste não é aprovado).
3. Escreva apenas o código de implementação suficiente para fazer o teste passar.
4. Execute todos os testes.
5. Refatorar.
6. Repetir.

Na prática, deve ser escrito com os requisitos em mãos, pensar no que o teste fará, o que ele vai cobrir, em termos de requisitos. No caso ideal, para qualquer requisito devem-se implementar os testes, antes mesmo de existir a classe. Como o teste não funcionará, então existe a obrigação de criar a classe para executar o código, conforme o que for solicitado pelo teste, este sim é seu código. Veja que você não está testando e, sim, também implementando a classe para seu teste. Veja na figura abaixo, a construção de um teste unitário com a ferramenta *Java Junit*.

```
class AlunoTest {

void testInscriçãoValido() {

        assertTrue(new AlunoValidar().validarAg("123456"));

}

void testInscriçãoInValido() {

        assertFalse(new AlunoValidar().validarAg("12345"));

        assertFalse(new AlunoValidar().validarAg("1234567"));

}
```

Figura 3 - Código de teste unitário, criado pra testar um método, quanto ao tamanho da entrada. Fonte: Elaborada pelo autor, 2018.

Então, para executar este código, será necessário implementar uma classe, segundo a figura a seguir, que mostra como será o início de nosso código.

```
public class AlunoValidar {

    public boolean validarAg(String ag) {

        boolean retorno = true;

        if (ag.length() != 6) {

            retorno = false;

        }

        return retorno;

    }

}
```

Figura 4 - Implementação da classe em si, que fará a verificação das regras de negócio. Fonte: Elaborada pelo autor, 2018.

Supondo que o código executou corretamente para aquele requisito, então se deve partir para a próxima implementação de teste, senão, ele deverá ser corrigido até atingir o ponto no qual o teste faça sentido. Veja que cada método deve ter uma asserção sobre estar correto e errado, não é regra, mas a tendência é de não criar vários por método, para não tornar confuso o processo. Este processo envolve muito código, e para realizá-lo, existem várias ferramentas que podem auxiliar.

VOCÊ SABIA?

As ferramentas de integração contínua, Jenkins, Hudson, Travis e Bamboo, devem ser usadas para puxar o código do repositório, compilá-lo e executar testes, em uma definição utilizada de forma aberta. O repositório realiza a construção do *software* e já realiza os testes. O conceito de integração contínua tem crescido e, embora sua implementação dependa da configuração de vários sistemas, seus resultados automatizam o processo de desenvolvimento, o que é muito útil

para a equipe, pois minimiza passos necessários para gerar novas *releases* com qualidade. Usar integração contínua é uma mudança de cultura na empresa, e como tal, pode receber algum tipo de resistência, mas seus benefícios são comprovados.

Este processo deve continuar desta forma, até implementar todos os testes para que não exista a necessidade mais de implementar código e para que eles sejam executados sem erros. Neste ponto o processo estará concluído.

Outro ponto interessante é que, além do código, o TDD auxilia para validar os requisitos, com o teste precoce, falhas nos requisitos aparecem cedo, pois, em qualquer dúvida existente, o requisito é questionado e as definições do sistema são revistas logo no começo, evitando, assim, o retrabalho.

VOCÊ O CONHECE?

O engenheiro de *software* Kent Beck, é o criador do *Extreme Programming* e do *Test Drive Development*, é um dos 17 agilistas originais do manifesto ágil de 2001, e escreveu diversos livros sobre engenharia de *software* e testes. Formado em Ciência da Computação, pela Universidade do Oregon, em 1987, Kent trabalha no Facebook e é um dos maiores nomes do desenvolvimento na atualidade.

Para aplicar o TDD, existem algumas boas práticas, de forma a otimizar o desenvolvimento. Listamos a seguir.

- **Separar implementação de teste**: sempre manter, em separado, o teste do código do projeto, principalmente, não deve existir ligação entre os dois (baixo acoplamento).
- **Usar convenções**: em qualquer desenvolvimento, o uso de convenções auxilia a organizar as unidades em seu projeto, uma pasta para código e outra para o teste, como visto acima, um termo que identifique a classe de teste de sucesso e um para a classe de erro, e assim por diante. De preferência procure por padrões

conhecidos de nomenclatura, como iniciar ou terminar sua classe de teste com a palavra “teste”.

- **Processo**: seguir o TDD à risca e somente escrever um código “válido”. Quando o teste está falhando, se não for possível, não utilize o TDD em todo o projeto, mas em um componente apenas, por exemplo, mas, neste componente, haja desta forma. Parecem simples, entretanto, os implementadores são sempre tentados a terminar o código, antes de seguir adiante, e não é este o caso. Depois de ter todos os testes prontos, haverá um momento para retornar e refatorar o código e, também, será o momento de terminar as implementações que restarem.
- Toda vez que qualquer parte do código de implementação é alterada, todos os testes devem ser executados, para garantir a integridade do código.
- Utilizar ferramentas de integração contínua.

VOCÊ QUER LER?

BDD é uma variação do TDD, focada em comportamento, se for escrever um caso de teste usando TDD, o teste acaba sendo orientado para uma saída interna. Já usando BDD, notamos algumas diferenças, sendo a principal que o sistema acaba respondendo ao teste com um comportamento similar ao que o usuário final receberia. Você pode conhecer mais sobre isso no endereço <https://www.agilealliance.org/glossary/bdd (https://www.agilealliance.org/glossary/bdd)> (AGILE ALLIANCE, 2018). Um bom exemplo de como utilizar o BDD, seria ler a documentação de sistemas, como a que se encontra no endereço: <https://cucumber.io/ (https://cucumber.io/)>(CUCUMBER, 2018).

Até aqui, utilizamos várias técnicas, notoriamente conhecidas por facilitar e garantir uma boa testabilidade de código, em nível de caixa branca. O teste é necessário e de responsabilidade do desenvolvedor e, além disso, existem várias ferramentas reconhecidas que podem ser usadas.

Mas não é só isso. O teste não é somente de responsabilidade do testador, é de toda a equipe. Veremos a seguir, níveis mais altos e abstratos de teste.

4.2 Casos de testes de *software*

Até o momento, nos dedicamos bastante ao desenvolvimento de testes focados nas estruturas do sistema, todos eles são muito importantes para a construção da implementação. Mas, em determinado momento, será necessário unir essas estruturas para elaborar componentes para o sistema e, depois, montar uma *release*, ou versão do sistema em si, e, neste momento, outros problemas podem acontecer. Não porque nossa abordagem tenha sido errada, mas sim, devido à forma como os componentes vão se comunicar entre si e compartilhar recursos.

Vamos conhecer mais esses testes, para cobrir a fase em que a implementação já está em um elevado grau de completude e se permite ser testada como um todo, tanto pela equipe interna, quanto por uma externa, ou, até, pelo próprio usuário.

Um caso de teste serve como base para que a equipe realize os testes manualmente, ou de forma automatizada, conforme veremos a seguir. Ele é fortemente dependente dos requisitos do projeto. É por meio do caso de teste, feito normalmente na forma de Casos de Uso (formato tradicional) ou Histórias de Usuário (mais aberto).

Os testes devem mostrar todos os caminhos percorridos no sistema por uma funcionalidade, como demonstrado na figura acima. Nela, vemos o exemplo de Caso de Uso, por exemplo, devem ser testados os fluxos normais e, também, os fluxos alternativos. Ou seja, todo o caminho que pode existir em uma funcionalidade e, para isso, devemos usar os requisitos, não o código. Ou seja, um requisito bem desenvolvido tem de ser testável.

Os casos de testes são normalmente denominados como ações relacionadas às funcionalidades, como "*Validar o cadastrar disto...*" ou "*Preencher aquilo*". A criação dos casos de teste baseia-se em um planejamento anterior, com uma validação posterior dos resultados que devem ser obtidos em cada caso de teste.

Se a *release* do sistema atender aos testes, pode então ser elevada a um patamar para ser testada pelo usuário final, em seu contexto normal de uso, o que não se consegue no ambiente de desenvolvimento normal, que veremos a seguir, no teste de usuário.

4.2.1 Testes de release

Por *release*, entenda-se uma versão de sistema que será disponibilizada para teste, visando já uma análise da viabilidade do produto, para atender aos objetivos, para os quais, foi proposta sua criação. Como, a essa altura, a equipe de desenvolvimento já trabalhou muito sobre o sistema, a situação ideal é que a *release* seja testada por uma equipe externa, que vai analisar os resultados sem vícios, pois uma pessoa tem a tendência de repetir os mesmos testes e outra pessoa, ou grupo, tende a ampliar a quantidade e amplitude de testes.

O objetivo principal do teste de *release*, segundo Sommerville (2011), é o de convencer o usuário e cliente de que o sistema é bom o suficiente para ser usado. Nesse contexto, o teste torna-se um teste de caixa preta, pois não nos interessa mais o código em si, mas sim o atendimento aos requisitos sem erros, utilizando casos de teste.

4.2.2 Testes de usuário

O teste de usuário, ou de cliente é o momento em que, após a *release* com algumas sugestões feitas pelo cliente, considera-se que já existe maturidade para estabelecer uma estratégia junto a usuários finais, dentro do ambiente produtivo, a fim de verificar como o sistema se comporta em situações reais, com infraestrutura real e direcionado para o uso (GALLOTTI, 2017).

Pode, ainda, também segundo Sommerville (2011), ser o teste dividido em três diferentes níveis, que acompanhamos a seguir.

Teste *alfa*: neste teste, um ou mais usuários normais do sistema trabalham em conjunto com a equipe de desenvolvimento e de testes, ajudando a testar as funcionalidades e aconselhando a equipe, para otimizar os trabalhos de desenvolvimento e aproximar o máximo possível os requisitos escritos e utilizados pela equipe, das práticas vivenciadas no ambiente produtivo.

Teste *beta*: este é o teste que ocorre quando uma *release* do sistema, que pode estar pronta ou não, é disponibilizada no ambiente de trabalho real. É interessante que esse grupo de usuários tenha experiência no negócio e também no uso de sistemas para compreender que alguns pontos podem não funcionar conforme o esperado. A vantagem de um teste *beta* ser realizado é que, nele, muita evidência

referente à infraestrutura vem à tona. É muito difícil para o desenvolvedor testar nos mais variados ambientes, e isso pode ser conseguido em um ambiente de teste *beta*.

Teste de aceitação de sistema: conhecido também por ser parte da homologação de um sistema, é o momento final, quando os usuários, junto com o cliente, testam se o sistema pode ser usado, é normalmente realizado em um intervalo definido de tempo e determina se o sistema tem alguma restrição e se pode ser homologado e colocado em produção.

Em teoria, segundo Sommerville (2011), o teste de aceitação define se o sistema será, ou não pago, entretanto, na realidade esta situação não é nada simples. Um sistema é feito para agregar valor ao trabalho do cliente, ou seja, é um investimento, e, desde o momento em que é iniciada sua implementação, gera uma grande expectativa, pois usuários são treinados, problemas tem de ser resolvidos, enfim, sua vinda é esperada.

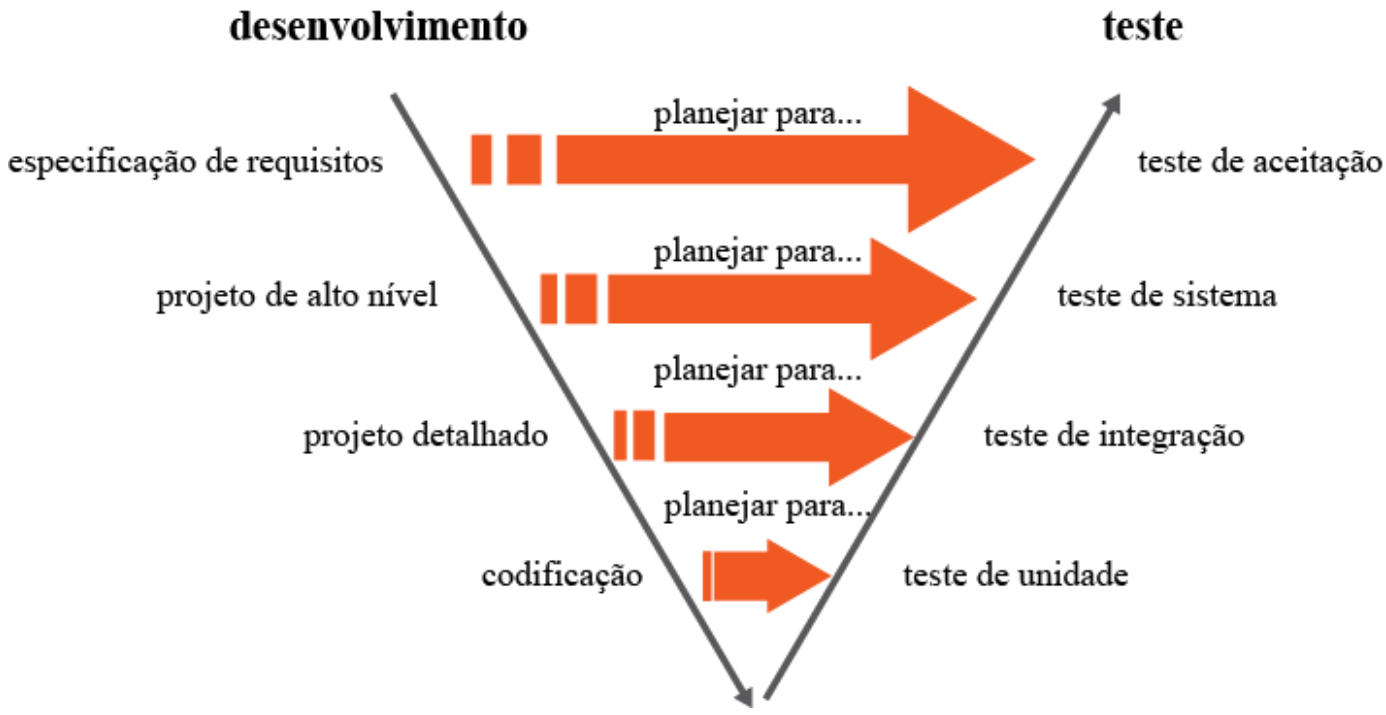


Figura 5 - Fluxo de um Caso de Uso, no qual pode ser baseada a construção de um caso de teste. Fonte: PRESSMAN, 2016, p. 328.

Na figura acima, temos a análise em V do teste de *software*. É um fluxo de gestão baseado no processo em cascata e ilustra de uma forma bem clara a relação entre os diferentes momentos de desenvolvimento e teste, a ideia é que, do nível mais baixo, para o mais alto, o sistema ganhe em termos de abstração. De forma contrária, mais abaixo, existe mais cuidado com o código em si, e menos com o requisito. Do lado esquerdo, o entendimento do nível de projeto para codificação e, do lado direito, o nível de teste que mais se enquadra para aquele nível de desenvolvimento.

Este é o modelo clássico de teste amarrando à especificação de requisitos com os testes de aceitação, o projeto de todo o sistema em um alto nível, com o teste de todo o sistema, um projeto mais específico com testes de componentes, ou de integração, e no nível mais baixo, o código implementado com o teste unitário.

4.2.3 Encerrando o processo de testes

Ao final dos testes, é muito importante analisar todo o processo, comparar os resultados obtidos com os esperados pelo sistema, e gerar algum tipo de relatório para as divergências, coletando, avaliando os testes e os erros encontrados, priorizando os erros que devem ser tratados e classificando os demais para tratamento posterior. A situação ideal é sempre que não ocorram erros, mas nesta situação, a evidência do tratamento dos erros é importante, tanto para a equipe, quanto para o cliente.

No encerramento do processo, também devemos verificar quais as lições aprendidas, para mitigar a ocorrência dos mesmos erros no futuro, verificar os pontos críticos em testes e registrar, preferencialmente, guardar e comparar com projetos anteriores e futuros, visando gerar uma base interna histórica.

Quanto ao cliente, a expectativa é a de que o sistema esteja concluído sem erros. Mas, muitas vezes, o sistema é aceito parcialmente, pois postergar sua entrega, na maior parte dos casos, significa que o cliente continuará a conviver com um problema que o sistema deveria solucionar. Essas são situações específicas, que devem ser analisadas em cada caso.

4.3 Evolução de *software*

Quando consideramos o *software*, do ponto de vista leigo, não imaginamos que ele tenha um ciclo de vida, um início, meio e um fim, embora não seja um meio físico, o que tornaria este raciocínio mais simples. A verdade é que o *software* nasce e morre. Mesmo sendo funcional todo o tempo, suas funcionalidades podem não se enquadrar mais na realidade do momento. Por exemplo, um *software* criado para Windows, em uma versão de 20 anos atrás, dificilmente será atrativo em sua forma de funcionar hoje, ou seja, embora, provavelmente, atinja objetivos, não deve funcionar em conjunto com *softwares* mais atuais e sua apresentação visual pode não agradar, além de ser quase impossível de modificar, se isso for necessário.

4.3.1 Processo de evolução de software

Atualmente, todas as áreas de trabalho tem algum tipo de relação com *software* e utilizam algum sistema. Esse contexto é real, atual e plenamente visível. Logo, pode-se dizer que tecnologia da informação é necessária, hoje, para qualquer sucesso em qualquer empreendimento moderno.

Por essa razão, as empresas empregam cada vez mais recursos em *softwares* competitivos, que tragam velocidade e melhorem processos. O processo de mudança e evolução em um *software* é inevitável, embora se deseje atender, o mais rapidamente possível, ao cliente, a verdade é que o *software*, como qualquer produto, tem um tempo de vida. Segundo Sommerville (2011), esse ciclo de vida do *software* é definido por quatro fases distintas, que destacamos a seguir.

Desenvolvimento inicial: esta fase reflete toda a construção inicial do sistema, ou seja, desde o momento em que são levantados os requisitos, até o momento em que é feita a primeira entrega, quando é concluído o projeto com a primeira implantação.

Evolução: posteriormente à entrega inicial, pode-se dizer que o sistema está pronto, porém, após esta entrega, devido ao próprio uso do *software*, o cliente detecta que algumas situações, mesmo tendo sido definidas, não atendem plenamente às necessidades. Nesse caso, o sistema passa pela evolução, que é o momento no qual, o cliente solicita alterações para o desenvolvedor.

Serviço: posteriormente a um período, no qual o *software* é muito utilizado e, também, sofre muitas alterações, o sistema atinge o momento de serviço. Agora, ele está estável, não necessita mais de alterações e o seu uso é contínuo, embora tenda a cair.

Interrupção gradual: gradualmente, devido a necessidades da empresa e novas tecnologias, o sistema tende a ser abandonado e trocado por outra versão, ou talvez, outro sistema.

Normalmente, existem vários fatores relevantes, relacionados a preço, custo, treinamento, que antecipam, ou postergam essa entrega, mas a verdade é que todo o sistema, devido à própria evolução tecnológica, tende a cair em desuso, um dia.

4.3.2 Dinâmica de evolução de programas

Conforme mencionado anteriormente, existe uma fase de desenvolvimento inicial, concluída com a primeira implantação, posteriormente as três outras fases, que são de evolução, serviço e interrupção gradual, consideram em maior, ou menor, quantidade, a manutenção de sistemas, que é o que podemos considerar a evolução de programas.

Essa fase posterior à entrega, necessita de cuidados específicos, por melhor que sejam os testes, é um conhecimento empírico. Todo sistema vai ter uma evolução e vai mudar, devido a adaptações e novas necessidades do cliente, é o que se chama de dinâmica de evolução de programas, mencionada por Gallotti (2017).

Em 1970, o pesquisador Meir Lehman, estabeleceu, junto com o cientista da computação László Bélády, as oito leis da evolução de um *software*, que podiam ser aplicados, de forma geral, a qualquer *software*, em momentos de tomada de decisão, de planejamento, desenvolvimento e manutenção (LEHMAN; RAMIL; SANDLER, 2001; SOMMERVILLE, 2011). No quadro a seguir, podemos visualizar um resumo das leis de Lehman, compiladas por Sommerville (2011).

Lei	Descrição
Mudança Contínua	Um programa usado em um ambiente real deve mudar necessariamente, ou tornar-se progressivamente menos útil.
Complexidade crescente	À medida em que um programa muda, sua estrutura tende a se tornar mais complexa. Recursos extras devem ser dedicados para preservar e simplificar a estrutura.
Evolução de programa de grande porte	A evolução de programa é um processo auto-regulável. Atributos de sistemas como tamanho, tempo entre versões e número de erros reportados é quase invariável em cada versão de sistema.
Estabilidade organizacional	Durante o ciclo de vida de um programa, sua taxa de desenvolvimento é quase constante e independente de recursos dedicados ao desenvolvimento do sistema.
Conservação de familiaridade	Durante o ciclo de vida de um sistema, mudanças incrementais, em cada versão, são quase constantes.
Crescimento contínuo	A funcionalidade oferecida pelos sistemas deve aumentar continuadamente para manter a satisfação do usuário.
Qualidade em declínio	A qualidade dos sistemas entrará em declínio, a menos que eles sejam adaptados a mudanças em seus ambientes operacionais.
Sistema de *feedback*	Os processos de evolução incorporam sistemas de *feedback*, com vários agentes e *loops* e você deve tratá-los como sistemas de *feedback*, para conseguir aprimoramentos significativos de produto.

Quadro 1 - As oito leis de Lehman, criadas para identificar as mudanças ocorridas com os softwares, durante seu ciclo de vida. Fonte: SOMMERVILLE, 2011, p. 169.

A manutenção é inevitável e, enquanto um sistema for útil, deve ser mantido. Isso por que o futuro é sempre incerto, novos requisitos surgem, o ambiente produtivo muda, infraestrutura muda, com isso, passa a existir a necessidade de gerenciar as mudanças.

O sistema, então, tem de continuar agregando valor para não ser descontinuado. É nesse sentido, que Lehman e Bélády desenvolveram as leis que regulam o crescimento e alteração de sistemas, válidas para permitir o uso contínuo do sistema, ou a ampliação de sua vida útil (LEHMAN; RAMIL; SANDLER, 2001).

4.4 Manutenção de *software*

Anteriormente, mencionamos a decadência do *software*, o início do uso do sistema, até o momento em que ele não pode ser mais utilizado, por algum motivo. Entretanto, em seu período de vida útil, por assim dizer, ele pode ter de passar por alterações. Essas modificações podem ser desejadas ou indesejadas, mas, para que o *software* seja utilizado, tem de ser realizadas, a fim de permitir que

o *software* evolua, se adapte a alguma nova realidade, ou, até mesmo, seja corrigido de algum problema. Vamos estudar um pouco mais sobre essas manutenções, a seguir.

4.4.1 Manutenção de programas

O pesquisador Pfleeger (2004), lista vários tipos de manutenção que são, geralmente, reconhecidos. Mas é normal que haja nomes diferentes para os mesmos tipos de manutenção. Vamos definir os principais tipos de manutenção, utilizando os nomes mais comumente conhecidos.

A manutenção de *software* engloba três atividades: manutenção corretiva (reparo de defeitos no *software*); manutenção adaptativa (adaptação do *software* a um ambiente operacional diferente); e manutenção evolutiva (manutenção para adicionar funcionalidades ao *software*, ou modificá-la).

Manutenção evolutiva é a mais desejada, refere-se à implementação de novos requisitos, evolução do sistema, melhorias e mais valor para o usuário e cliente, acrescentando novas funcionalidades, melhorias de desempenho, ou, até, alterações no código, buscando conseguir mais legibilidade.

Manutenção adaptativa significa alterações no sistema, devido a mudanças de ambiente, ou infraestrutura, como uma nova plataforma, novos processadores, o uso de um novo navegador, ou sistema operacional, que não era previsto na definição inicial do sistema.

Manutenção corretiva lamentavelmente significa erros que foram para a produção e que tem de ser tratados. Ninguém está livre dessa situação, embora ela não seja desejada.

VOCÊ SABIA?

É preferível investir o maior esforço na implementação de um sistema do que correr riscos de mudanças futuras. Geralmente adicionar uma nova funcionalidade, após a liberação, é mais caro, por várias razões, como:

- a equipe pode ter mudado e não apresentar mais especialistas na área;

- imagine se houve vícios no desenvolvimento, neste caso, eles ficam cada vez mais complicados de serem resolvidos, pois o sistema ficará mais complexo, com o passar do tempo;
- normalmente, a equipe que desenvolve, tende a ser mais experiente que a equipe que realiza manutenção;
- a tecnologia pode ter mudado e alterar posteriormente, pode ser complexo, por exemplo, a linguagem pode ter sido descontinuada, ter sua arquitetura revista. Temos vários exemplos assim, em linguagens modernas.

Outro tipo de manutenção que, infelizmente, é pouco utilizada, é a Manutenção Preventiva. Ela normalmente ocorre por ônus do próprio desenvolvedor, ou da equipe, e seu baixo uso se dá porque é complexo explicar para o cliente que o *software* tem algum detalhe que, preventivamente, deveria ser alterado e que o cliente deve pagar por isto.

Essa postura, por outro lado, somente faz crescer o número de manutenções corretivas.

4.4.2 Gerenciamento de programas legados

Por fim, no ciclo de vida do produto, o sistema, conforme visto anteriormente, tende a passar por uma interrupção no seu uso. Entretanto, como o sistema agrega valor para a empresa, assim como ocorre com o ativo, se desfazer dele tem o mesmo impacto que teria, se fosse com qualquer outro bem. Sem ele, processos serão afetados e as pessoas tendem a sair da sua área de conforto, o que provoca transtornos na produtividade, que pode afetar os processos e o financeiro, para a empresa.

Sobre isso, Pressman (2016, p. 428) tem o seguinte ponto de vista:

> Sistemas de informação desenvolvidos há 20 anos sofreram dezenas de gerações de mudanças e são quase impossíveis de serem mantidos, havendo risco de falharem; [...]
>
> Aplicações de engenharia, por causa da idade e estado de reparo não são satisfatoriamente entendidas, a estrutura interna desses sistemas é desconhecida; [...]

> Sistemas de controle de fábricas, usinas de energia, tráfego aéreo e outros, exibem comportamentos inesperados e algumas vezes inexplicáveis, mas não podem sair de operação pois não há nada para substituí-los.

É comum encontrar sistemas que forçam seus usuários a realizar processos *batch*, criar macros complexas em linguagens já ultrapassadas, mas, processos que resolvem o fluxo diário da empresa com sucesso, e qual a razão de mudar?

Esses sistemas, chamados de legados fazem isso, estão totalmente fora da curva tecnológica, mas ainda promovem soluções para a empresa, na maior parte dos casos, estão tão integrados ao cotidiano da empresa, que passam a ser vistos como parte dela.

;o

:sa iniciou suas atividades na década de 1970 e desenvolveu seu sistema de ERP em uma linguagem, que siderada antiga. O sistema funcionava apenas localmente, sendo acessado via terminal e respondia e às necessidades dos clientes.

oo, devido à mudança de tecnologias na sociedade, o sistema deixou de ser atrativo, as vendas caíram e o empresa viu-se diante de algumas questões:

- continuar a manter a clientela antiga, com o sistema existente, atualizando no que fosse possível, o sistema dentro do que a tecnologia nativa permitisse;
- considerar a hipótese de desenvolver um sistema inteiramente novo, em uma linguagem para *web*, acessível via *mobile*, utilizando nuvem;
- criar um híbrido, parte no modelo antigo, parte em nova tecnologia, para não ter de reescrever tudo.

uma solução perfeita, qualquer uma das possibilidades já foi usada com sucesso e insucesso. Para tomar o, deve-se fazer uma análise de mercado, conhecer as necessidades de seus clientes, para verificar a deles em comprar uma nova versão, ou, simplesmente, uma antiga mas “atualizada”, que pode atender às es. A empresa, então, optou por um modelo híbrido, manteve o sistema antigo, mas a base passou a ser or um barramento de serviços e disponibilizada na *web* para venda, também, de um novo produto, com e outro ciclo de vida.

Veja na figura a seguir, que existem várias características que devem ser consideradas em um *software* legado. Sua manutenção passa a ser um custo, pois não integra mais com novas tecnologias, seus desenvolvedores passam a ter um valor alto e mantêm a empresa em um nicho tecnológico com poucas opções. Por outro lado, o processo de convencimento na troca desses sistemas é difícil, normalmente apoiado por usuários já condicionados ao processo estabelecido. Sua troca, muitas vezes, parece até "dolorosa", dada a quantidade de problemas que os seus usuários encontram para se adaptar.

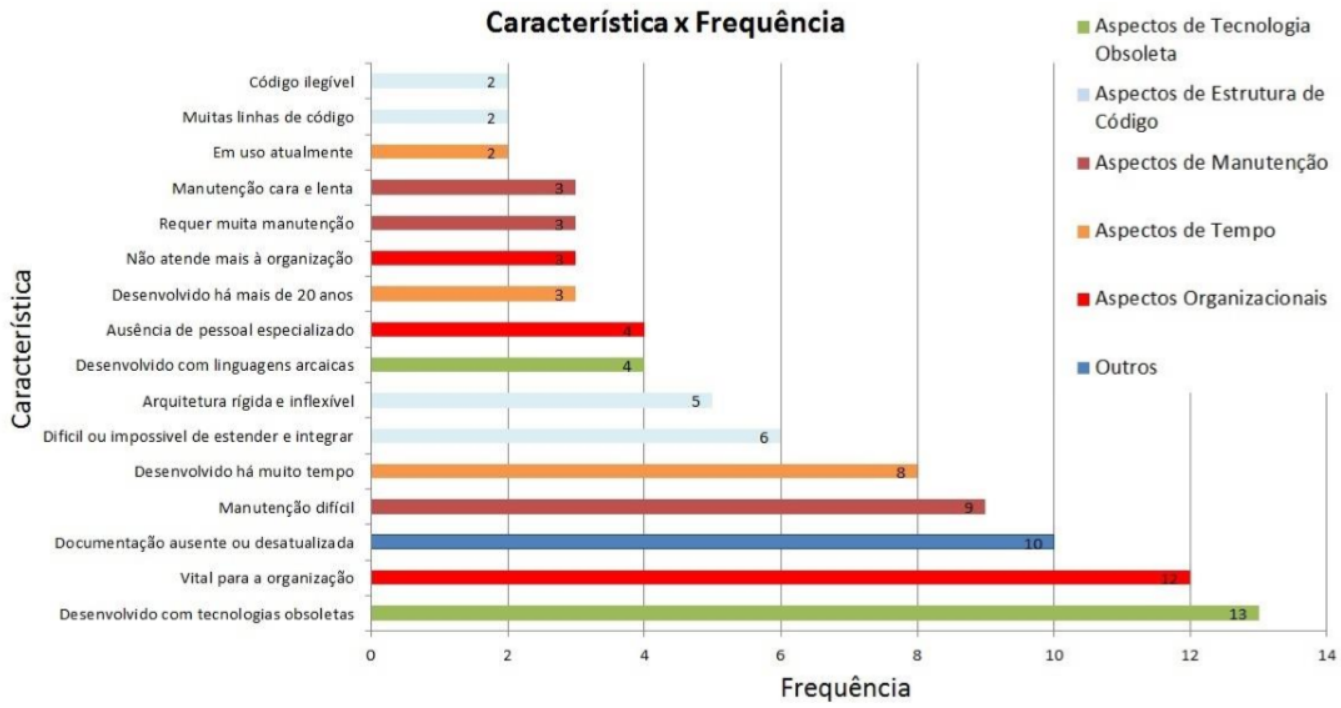


Figura 6 - Comparativo sobre as principais características de sistemas legados. Fonte: BORDIN, 2016, p. 2.

Existem várias opções, para troca total do sistema, alteração de módulos específicos, ou, até mesmo, manter o sistema inalterado, arcando com custos de manutenção. A complexidade aqui, é elevada e praticamente, não existe uma regra e fórmula para a decadência de um sistema e a troca por um novo. Cada caso deve ser considerado de forma particular, com o planejamento e necessidades da empresa.

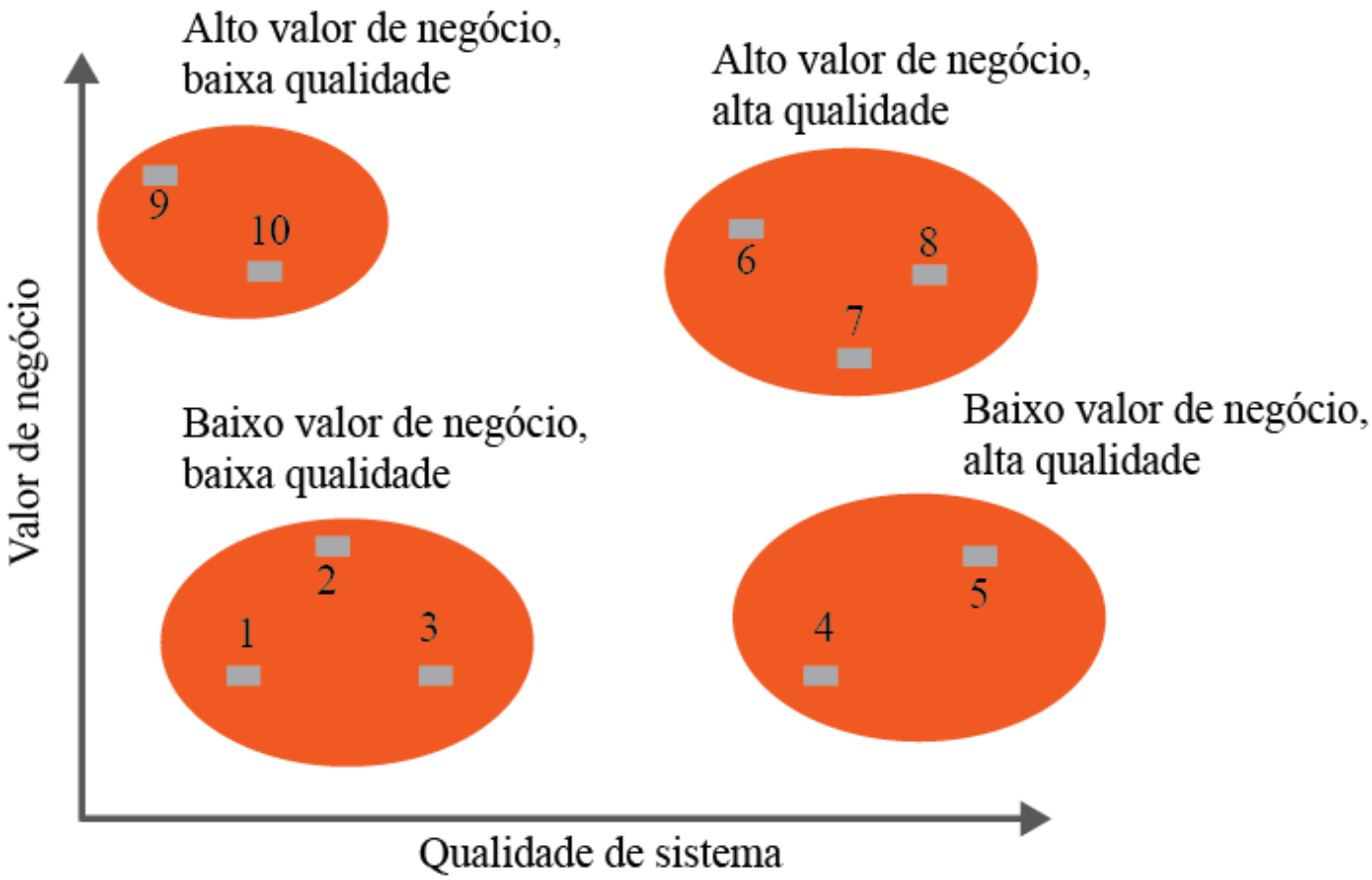


Figura 7 - Exemplo de análise de sistemas legados. Fonte: SOMMERVILLE, 2011, p. 178.

Na figura acima, temos um exemplo que sugere que a continuidade, ou não, do *software,* depende do valor de negócio agregado para o sistema e, também, da qualidade com que se encontra o legado, ou seja:

- se o valor agregado for baixo e qualidade baixa: manter o sistema custará muito e não trará o retorno desejado;
- se o valor agregado for alto e a qualidade baixa: problema! O sistema ainda é importante para agregar valor para a empresa, mas é complexo de se manter. Neste caso tem de se verificar se pode haver uma evolução, ou se existe possibilidade de substituir por outro, o que, talvez, seja a melhor opção;
- se o valor agregado for baixo e a qualidade alta: enquanto não gerar manutenção alta, pode ser continuado, mas deve ser descartado assim que algum *software* mais atual for adquirido, seu custo não vale o retorno;
- se o valor agregado for alto e a qualidade alta: deve ser mantido naturalmente, pois sua qualidade é aceitável e traz bom retorno

para a empresa, logo não existe razão para mudar.

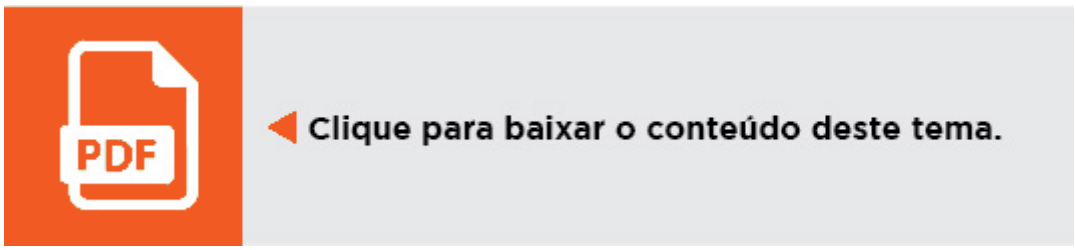
Não existe uma regra, mas características que devem ser observadas e uma análise aprofundada sobre o retorno de investimento que o *software* traz para a empresa. Essa análise deve contar com a maior quantidade de dados possível para mitigar problemas na troca, antecipá-la ou evitá-la.

Síntese

Chegamos ao final deste conteúdo! Entendemos aqui, que testar um sistema é muito mais que verificar se, ao final da codificação, o sistema opera conforme se imagina, ou não. Existe toda uma metodologia para testes, com técnicas específicas que devem ser usadas durante a implementação, pela equipe de desenvolvimento. Os testes devem ser feitos também com o cliente e o usuário. Aprendemos, também, que o sistema tem um tempo de vida e, ao ser criado, ele vai sofrer algum tipo de manutenção em seu ciclo de vida e entrar em processo de obsolescência.

Neste capítulo, você teve a oportunidade de:

- entender como funcionam os testes em desenvolvimento, caixa branca e preta;
- estudar o ciclo TDD e seu processo de desenvolvimento;
- conhecer o processo de evolução e o ciclo de vida de um *software;*
- compreender os sistemas legados;
- entender a manutenção de *software*.



Bibliografia

AGILE ALLIANCE. Behavior Driven Development (BDD). **Portal Agile Alliance**, 2018. Disponível em: <https://www.agilealliance.org/glossary/bdd (https://www.agilealliance.org/glossary/bdd)>. Acesso em: 06/09/2018.

BORDIN, A. S. Mapeamento de Características de Sistemas Legados. **Anais** do 8º Salão Internacional de Ensino, Pesquisa e Extensão. Bagé, Rio grande do Sul: Universidade Federal do Pampa, 2016.

CUCUMBER. Tools & techniques that elevate teams to greatness. **Portal Cucumber**, 2018. Disponível em: <https://cucumber.io/ (https://cucumber.io/)>. Acesso em: 06/09/2018.

GALLOTTI, G. M. A. **Qualidade de *Software***. São Paulo: Pearson, 2017.

LEANDRO LIMA. **Selenium WebDriver Config e Downloads - Parte 1 de 5**. Canal Leandro Lima, YouTube, publicado em 23 de ago. de 2017. Disponível em: <https://www.youtube.com/watch?v=2UhRVicIzGI&list=PLkXFme-9nWWbzJ41maUuIxO4ojgI6aIu- (https://www.youtube.com/watch?v=2UhRVicIzGI&list=PLkXFme-9nWWbzJ41maUuIxO4ojgI6aIu-)>. Acesso em: 06/09/2018.

LEHMAN, M. M.; RAMIL, J. F.; SANDLER, U. **An Approach to Modelling Long-term Growth Trends in Software Systems**. Proc. Int. Conf. on Software Maintenance, Florença, Itália, 2001.

MYERS, G. J. **The art of Software Testing**. Hoboken (NJ, EUA): John Wiley & Sons, 2004.

PRESSMAN, R. **Engenharia de *Software***. 8. ed. Porto Alegre: AMGH, 2016. Disponível em: <https://Ânimabrasil.blackboard.com/webapps/blackboard/content/listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset (https://Ânimabrasil.blackboard.com/webapps/blackboard/content/listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset)>. Acesso em: 24/05/2018.

PFLEEGER, S. L. **Engenharia de *Software* - Teoria e Prática**. 2. ed. São Paulo: Pearson Addison Wesley, 2004.

SOMMERVILLE, I. **Engenharia de *Software***. 9. ed. São Paulo: Pearson Education, 2011.

WHITTAKER, J. **How to break software**: a practical guide to testing. Upper Saddle River (NJ, EUA): Pearson, 2002.