QUALIDADE DE SOFTWARE: UTILIZAÇÃO DO TESTE DE UNIDADE

Bruno Furquim Gusmão[1]
Helder Luiz Ascenção Traci[2]
Anderson Francisco Talon[3]

Resumo: Este artigo visa demonstrar como a qualidade é algo indispensável quando falamos em desenvolvimento de sistemas. Para que um sistema tenha grande valor deve-se prezar a cima de tudo a qualidade do produto final, e para garantir qualidade do sistema, falando do código, é de grande importância utilizar os testes de unidade no sistema, minimizando o retrabalho e as possíveis falhas. Integrar essa filosofia de teste no processo de desenvolvimento agrega-se também em outras partes, como por exemplo a arquitetura do sistema, pois enquanto codifica-se, pensar que aquele código que escrevemos é testável ou não, torna-se uma prática comum e necessária, dessa forma é possível deixar o código limpo e desacoplado.

Palavras-chave: Desenvolvimento de sistemas. Desenvolvimento orientado por teste. Teste de unidade. Garantia de qualidade.

1 INTRODUÇÃO

Sabe-se que algumas pessoas pensam que teste de unidade é algo que leva tempo e atrapalha o desenvolvimento do sistema, não pensar em testes de unidade para o desenvolvimento pode ser um risco, dessa forma não alcançando a qualidade desejada. Segundo Aniche:

> Uma maneira para conseguir testar o sistema todo de maneira constante e contínua a um preço justo é automatizando os testes. Ou seja, escrevendo um programa que testa o seu programa. Esse programa invocaria os comportamentos do seu sistema e garantiria que a saída é sempre a esperada. (ANICHE, 2012, p. 3).

Imagine um desenvolvedor que a cada trecho de código escrito deva simular e realizar os testes para saber se o seu código funciona ou não, talvez o desenvolvedor realize inúmeros testes até encontrar a maneira correta, a melhor

[1]Aluno do Curso de Sistemas de Informação do Centro Universitário de Bauru. E-mail: bruno_furquim@msn.com.
[2]Aluno do Curso de Sistemas de Informação do Centro Universitário de Bauru. E-mail: helderdiin.work@gmail.com.
[3]Professor do Curso de Sistemas de Informação do Centro Universitário de Bauru. E-mail: anderson.talon@ite.edu.br.

solução para agilizar esse processo é criar um teste automatizado para esse problema, e com isso, sempre que alterar o código não precise simular e testar manualmente para saber se o trecho está funcional ou não. Sabe-se que testar um sistema não é barato e segundo Aniche:

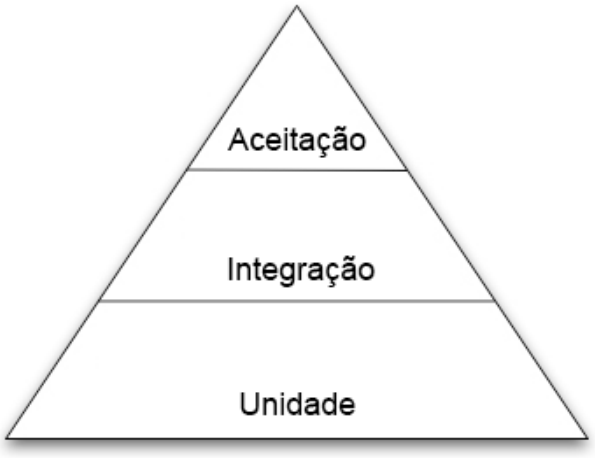
> Testar sai caro no fim, porque estamos pagando “a pessoa” errada para fazer o trabalho. Acho muito interessante a quantidade de tempo que gastamos criando soluções tecnológicas para resolver problemas “dos outros”. Por que não escrevemos programas que resolvam também os nossos problemas? (ANICHE, 2012, p. 3).

Já que todo desenvolvedor sabe que para minimizar insucessos no código que escreve deve-se realizar testes, por que o próprio desenvolvedor não realiza esses testes?

O teste de unidade foca a verificação dos menores elementos testáveis do software, o mesmo é aplicado aos componentes representados no modelo de implementação para verificar se os fluxos de controle e de dados estão cobertos e se eles funcionam conforme o esperado, portanto, usar testes de unidade garante que o código seja executado como o esperado, quando utilizamos testes no sistema como um todo, a garantia de que adicionar uma nova funcionalidade no sistema sem grandes impactos negativos é maior.

Quando inicia-se a implementação de testes de unidade dentro de um sistema, leva-se em conta o que a pirâmide de testes vem dizer em relação ao uso de todos os tipos de testes dentro do sistema.

Figura 1 - Pirâmide de testes.



Fonte: Elaborada pelos autores

Quando utiliza-se o padrão da pirâmide, garante-se uma boa implementação de testes dentro do sistema de forma adequada. Uma base grande de testes de unidade, pois são testes rápidos e poderosos, com respostas à impactos quase que imediato, já que consegue-se descobrir falhas nas unidades assim que roda-se os testes. Para Feathers (2013, p. 13), “Um teste de unidade que leve 1/10 de segundo para ser executado é um teste lento”. Um pouco menos de testes de integração, e somente o necessário para validação dos itens feitos com testes de interface, por serem um pouco mais lentos e não dependerem apenas do nosso sistema.

2 NÍVEIS DE TESTE

Uma dúvida comum entre os desenvolvedores é sobre o que é cada nível de teste e como definir esses níveis dentro do sistema. Saber o que são e como separar esses níveis é importante para escrever um teste fiel. Aniche acredita que:

> Apesar de parecer uma discussão boba, é importante que desenvolvedores usem os mesmos termos para se comunicar; isso facilita e acelera o entendimento. (ANICHE, 2014).

No geral utiliza-se três níveis de teste, sendo eles: unidade, integração e aceitação.

2.1 UNIDADE

Entende-se por unidade a menor parte isolada testável dentro do sistema, não existe um conceito final do que é uma unidade, isso varia de acordo com a forma que o sistema foi desenvolvido ou até mesmo da arquitetura e dos padrões escolhidos. Uma unidade por exemplo pode ser uma classe chamada Vendas, ou até mesmo um método chamado calcularTotal.

2.2 INTEGRAÇÃO

Pode-se definir como integração o teste que envolve mais de uma parte do sistema, que pode ser um serviço externo que é consumido pelo sistema. Um teste de uma classe que se conecta com o banco de dados por exemplo, seria um teste de integração, pois envolve partes internas e externas do sistema.

2.3 ACEITAÇÃO

Teste de aceitação, também chamados de teste de sistema, são os testes em que se garante o funcionamento completo da ferramenta, quando se tem todas as unidades e partes do sistema trabalhando juntas.

3 TÉCNICAS DE TESTE DE SOFTWARE

Existem diversas maneiras de testar um sistema, contudo existem as técnicas que mais se utiliza em sistemas desenvolvidos sobre linguagens de programação. Sabe-se que o principal objetivo das técnicas de teste é: encontrar falhas no sistema.

Existem duas técnicas bem conhecidas, são elas: Teste de Caixa-Branca (Técnica estrutural) e Teste Caixa-Preta (Teste comportamental). Sobre as técnicas de teste, Pressman informa:

> Teste caixa-preta, também chamado de teste comportamental, focaliza os requisitos funcionais do software. As técnicas de teste caixa-preta permitem derivar séries de condições de entrada que utilizarão completamente todos os requisitos funcionais para um programa [...] Diferentemente do teste de caixa-branca, que é executado antecipadamente no processo de teste… (PRESSMAN, 2011, p. 401).

3.1 TESTE DE CAIXA-BRANCA

A técnica teste de caixa-branca consiste em avaliar o comportamento interno do componente de sistema. Utilizando essa técnica de teste avaliamos os aspectos: teste de condição, fluxo de dados, ciclos e teste de caminhos lógicos.

O teste é desenvolvido analisando o código fonte e elaborando casos de testes que cubram todas as possibilidades do componente de sistema, com isso todas as variações originadas por condições serão testadas.

3.2 TESTE DE CAIXA-PRETA

Está técnica também é chamada de teste funcional ou teste comportamental, onde o componente de sistema testado é abordado como se fosse uma caixa-preta,

ou seja, sem considerar o desempenho do componente. Os dados de entrada são fornecidos, os testes são executados e o resultado é comparado à um resultado já esperado.

O componente testado pode ser um método, um programa, uma função ou até mesmo uma funcionalidade, pode-se utilizar o teste caixa-preta em todas as fases do teste, inclusive na fase do teste de unidade.

4 DESENVOLVIMENTO ORIENTADO POR TESTES

Considerado o precursor da prática de desenvolvimento orientado por testes (em inglês *Test driven development*, mais conhecido pela sigla TDD), Beck diz:

> Desenvolvimento orientado por testes (TDD) é uma abordagem evolutiva para o desenvolvimento que combina teste-primeiro desenvolvimento, onde você escreve um teste antes de escrever código de produção suficiente para realizar esse teste e refatoração. (BECK, 2010).

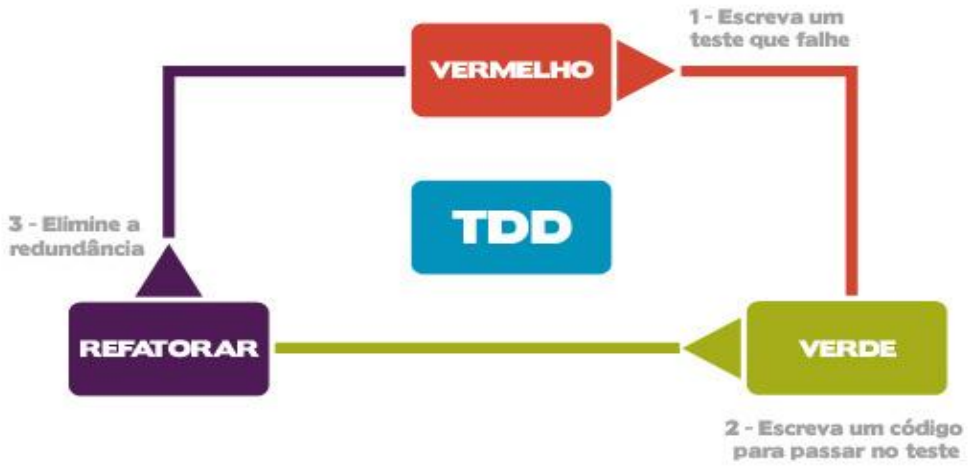
TDD é uma prática de desenvolvimento de sistemas que está se tornando cada vez mais utilizada. A ideia dessa prática é a seguinte: antes mesmo de escrever o código escreva seu teste, ao escrever o teste antes, o programador garante que o sistema desenvolvido está em funcionamento e sem erros. O TDD é baseado no conceito *Test First Programming* (utilizado na metodologia de desenvolvimento *eXtreme Programming*, ou da forma como é mais conhecida XP), porém o interesse é grande e vem sendo utilizado independente da metodologia XP.

4.1 UTILIZANDO O TDD

Algo importante no início da prática do TDD, é dar "passos de bebês" na construção dos testes, não assuma ou evite assumir testes complexos no começo, deixe esses testes complicados para o fim.

Red, green, refactor. Para o TDD, nada mais é que: Escreva um teste falho e faço-o compilar, claro que não passará corretamente pelo teste, afinal apenas escreve-se o teste (*red*). Implemente o requisito e agora execute o teste (*green*). Por fim, refatore o código (*refactor*).

Figura 2 - Ciclo de vida do TDD



Fonte: Devmedia.[4]

4.2 EXEMPLO DE UTILIZAÇÃO DO TDD

Pode-se mostrar através de exemplos a utilização do TDD. Na Figura 3 temos um método que, dado um número romano, retorna o seu valor.

Figura 3 - Exemplo de utilização do TDD.

```
public class ConverteNumRomano {
      public int converte(String numRomano) {
         return 0;
      }
   }
```

Fonte: Elaborada pelos autores

Na Figura 4, temos a funcionalidade que deve retornar 1 quando passamos o algarismo I como parâmetro. Temos, então, o primeiro caso de testes.

O teste irá falhar, já que não houve alteração no método de converter. Sabe-se que é importante esse passo ao longo do desenvolvimento, para que tenhamos certeza que o teste foi codificado e se o mesmo é capaz de encontrar erros.

[4] Disponível em: <http://www.devmedia.com.br/tdd-fundamentos-do-desenvolvimento-orientado-a-testes/28151> acesso em ago. 2015.

Figura 4 - Exemplo de utilização do TDD.

```
public void testeConverteNumRomano_DeveConverterOAlgarismoI(){
      ConverteNumRomano conversor = new ConverteNumRomano();

      //O retornar ou deve retornar 1 quando é passado o algarismo I
      int valor = conversor.converte("I");
      Assert.equals(1, valor);
}
```

Fonte: Elaborada pelos autores

No TDD é exigido que seja escrito o código somente para validar o teste, ou seja, não é necessário se preocupar em implementar o método para qualquer funcionalidade a qual não se tenha testes. Portanto, retornando 1 do método o problema será resolvido.

Figura 5 - Exemplo de utilização do TDD.

```
public int converte(String numeroRomano) {
      return 1;
}
```

Fonte: Elaborada pelos autores

Ao executar o teste, verifica-se que ele não irá falhar, pois está sendo retornado o que ele espera.

O exemplo mostrado é simples, não é necessário realizar uma refatoração no mesmo. Portando, pode-se retornar ao passo 1 e escrever testes para cada algarismo romano.

4.3 PORQUÊ UTILIZAR O TDD?

"Código limpo que funciona" (JEFFRIES *apud* BECK, 2010), é uma frase dita por Ron Jeffries sobre o TDD.

Sabe-se que o TDD não é apenas uma técnica para testar o código criado, e sim uma técnica para construir sistemas, pois através do TDD é possível definir classes, métodos, etc. E com isso o desenvolver torna-se mais objetivo e com mais qualidade.

Utilizando o TDD é uma das boas práticas para aumentar a qualidade do código utilizando a refatoração, removendo o código duplicado e dando mais performance ao mesmo, consegue-se também melhorar padrões e com isso alterar as funcionalidades do sistema sem comprometer o comportamento do mesmo.

Os problemas (ou bugs) podem ser descobertos mais cedo utilizando está técnica, visto que será testado de imediato o que foi escrito.

Pode-se dizer que com o uso desta técnica é possível ter como resultado confiabilidade no que foi desenvolvido, dessa forma reduzindo muito a quantidade de bugs no sistema criado.

Pensa-se que utilizando o TDD, o tempo gasto com a criação do teste é muito grande, porém se levarmos em conta que após o desenvolvimento do sistema o mesmo será testado por uma outra equipe e o tempo gasto por essa equipe será muito maior do que utilizar a técnica do TDD pelo desenvolvedor.

5 INVERSÃO DE CONTROLE E INJEÇÃO DE DEPENDÊNCIA

Quando desenvolve-se pensando em testes de unidade, é indispensável aplicar Inversão de Controle (em inglês *Inversion of Control*, conhecido pela sigla IoC) e Injeção de Dependência (em inglês *Dependency Injection*, conhecida pela sigla DI) no sistema desenvolvido.

5.1 INVERSÃO DE CONTROLE

É um famoso padrão de desenvolvimento usado em sua grande maioria com o propósito de minimizar o acoplamento do sistema. Quando trabalha-se com sistemas grandes, é comum e extremamente importante pensarmos em manutenabilidade e escalabilidade de código, logo, ter um sistema monolítico será uma grande barreira no caminho para atingir esses dois objetivos e uma barreira também para criar testes fiéis e que garantam o funcionamento do sistema. Exemplificando esse cenário, Soares ressalta:

> Se para criar uma instância da ItemPedido você precisasse instanciar outro objeto como por exemplo DetalhesDoItem e um dia esse comportamento fosse mudado? Novamente teríamos que mudar a classe Pedido. (SOARES, 2015).

Inverter o controle de uma classe quer dizer que ela não terá mais conhecimento de algumas coisas, como por exemplo instanciar uma outra classe. Quando inverte-se o controle não quer dizer que não existe ninguém cuidando dessa tarefa, o controle apenas é passado para alguém cuidar dessa parte, segundo Silveira:

> Mas há uma questão não resolvida. Inversão de Controle não significa falta de controle. Alguém precisa estar no controle. Mesmo usando um framework de DI, alguém precisa inicia-lo, dar o passo de inicialização e chamar explicitamente o primeiro bean. É preciso um código inicial, que não será Inversão de Controle, para que o restante da aplicação esteja com o controle invertido. (SILVEIRA, 2011, p. 107).

Para exemplificar o uso de IoC, pense que exista uma classe chamada FinalizarPedido e ao invés de instanciar-se dentro dessa classe uma outra classe chamada CarrinhoDeCompras, leva-se a responsabilidade de instanciar a classe CarrinhoDeCompras para uma outra classe, deixando a classe FinalizarPedido limpa e com a única preocupação que ela realmente deveria ter, que é usar a classe CarrinhoDeCompras, sem ter que instancia-la.

5.2 INJEÇÃO DE DEPENDÊNCIA

É um padrão de projeto para a aplicação de IoC. Com uma ótima resposta ao desacoplamento e ao isolamento de componentes, utilizar DI quando foca-se em testes é uma ótima prática. Mantendo a ideia de IoC, a classe não ter que se preocupar em instanciar uma outra classe, o que a DI propõe é receber a dependência no momento da construção da classe. Receber os objetos já instanciados é a solução proporcionada pela DI, como afirma Palmeira:

> Note que, dentro do método executa, fazemos a instanciação do ContatoDAO para invocar o método exclui. Como vimos, isso não é uma boa prática, pois deixa as classes mais acopladas e dificulta na hora de realizar os testes de unidade. A solução, como já indicamos, é apenas receber os objetos já instanciados. (PALMEIRA, 2015).

Quando aplica-se esses e outros padrões em um sistema, o desenvolvedor tem que tomar cuidado para não utiliza-los de forma incorreta ou de forma descontrolada e excessiva, da mesma forma que padrões de projetos, convenções de programação e *frameworks* ajudam no desenvolvimento, eles podem atrapalhar, complicando alguns pontos simples ou levando muito mais tempo do que o necessário, sem nenhuma vantagem aparente, Soares explica:

> Afinal, usar qualquer Pattern de forma obsessiva é ruim, chamamos isso de Patternite “Quando o programador está tão obcecado por Patterns que usa em todo o seu projeto” chega um momento que para realizar um “Hello World” você constrói 6 camadas, 1 factory, com IoC, D.I, Observer/Listener, Orientado a serviço, com MVVM, etc. Assim o seu sistema não lhe trás vantagens, pelo ao contrário, pode até ficar lento e principalmente de difícil entendimento. (SOARES, 2015).

Para exemplificar a aplicação de DI, imagine uma classe Alunos que precise fazer conexão à um bando de dados para realizar uma consulta dos alunos cadastrados no sistema, e para isso deve-se utilizar a classe AlunosDAO, ao invés de instanciar a classe AlunosDAO dentro da classe Alunos, recebe-se essa dependência no momento da construção da classe Alunos, assim, a classe Alunos não precisa saber instanciar a classe AlunosDAO, apenas usar ela para fazer as consultas necessárias ao bando de dados do sistema. Com isso evita-se o acoplamento do sistema utilizando as vantagens da aplicação de DI.

6 QUESTIONÁRIO

Foi realizada uma pesquisa quantitativa com dezesseis colaboradores de duas empresas do ramo de tecnologia da informação, onde os colaboradores responderam a um questionário contendo sete questões referentes ao uso do teste de unidade em seu trabalho. Com as respostas obtidas com o questionário espera-se ter dados para que seja possível afirmar alguns pontos:

Para avaliar a melhora significativa no número de problemas encontrados no sistema com a utilização de testes de unidade, foram utilizadas duas questões, sendo elas: “1) Quantos problemas eram encontrados por semana no sistema antes da aplicação do teste de unidade?” e “2) Quantos problemas são encontrados por semana no sistema depois da aplicação do teste de unidade?”.

Para avaliar se existiu e de quanto foi o tempo gasto a mais para a escrita dos testes de unidade, foi utilizada uma questão: “3) Com a utilização do teste de unidade, o seu tempo de desenvolvimento aumentou? Se sim, em quantos por cento?”. Com esses dados, consegue-se comparar com os dados das questões um e dois, e chegar-se a uma conclusão sobre a qualidade do desenvolvimento quando aplica-se testes de unidade.

Para avaliar se as pessoas seguem o padrão da pirâmide, e se garantem de fato que a unidade funciona independente de outras partes do sistema, foi utilizada uma questão: “4) Você aplica mais testes de unidade do que testes de integração?”.

Para avaliar se os testes escritos pelos entrevistados estão fazendo sentido ou não, foi utilizada uma questão: “5) Utiliza técnicas como Caixa-Branca ou Caixa-Preta para a escrita dos testes de unidade?”, pois em nada ajuda escrever testes de unidade que não estão testando realmente o código escrito.

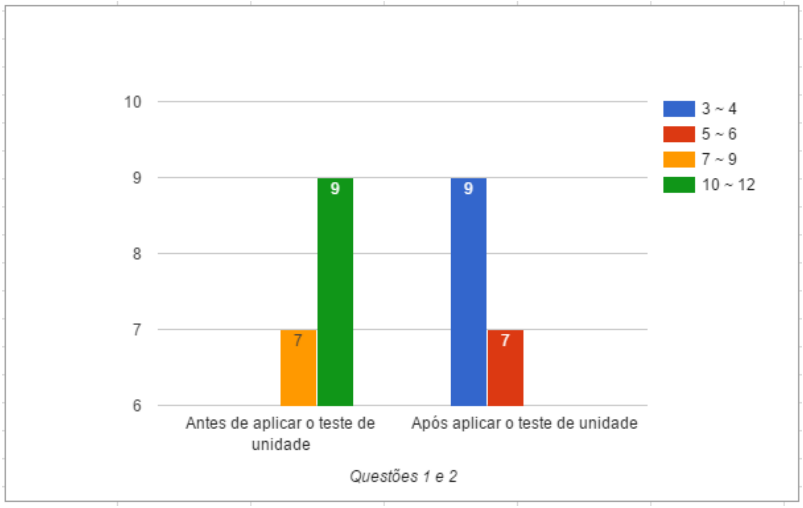
Para avaliar se os entrevistados praticam o TDD e descobrir se está agregando valor ou não, foi utilizada uma questão: “6) Utiliza no desenvolvimento a abordagem do TDD? Em uma escala de 0 até 10, o TDD ofereceu ganho positivo na qualidade do produto final?”.

E, para descobrir se os entrevistados utilizam na arquitetura dos projetos as aplicações de padrões de desenvolvimento para uma escrita fiel dos testes de unidade, foi utilizada uma questão: “7) Aplica padrões de desenvolvimento como Inversão de Controle e Injeção de Dependência?”.

7 RESULTADOS

Ao analisar os dados obtidos através das questões um e dois, pode-se afirmar que existiu uma baixa significativa no número de problemas encontrados no sistema.

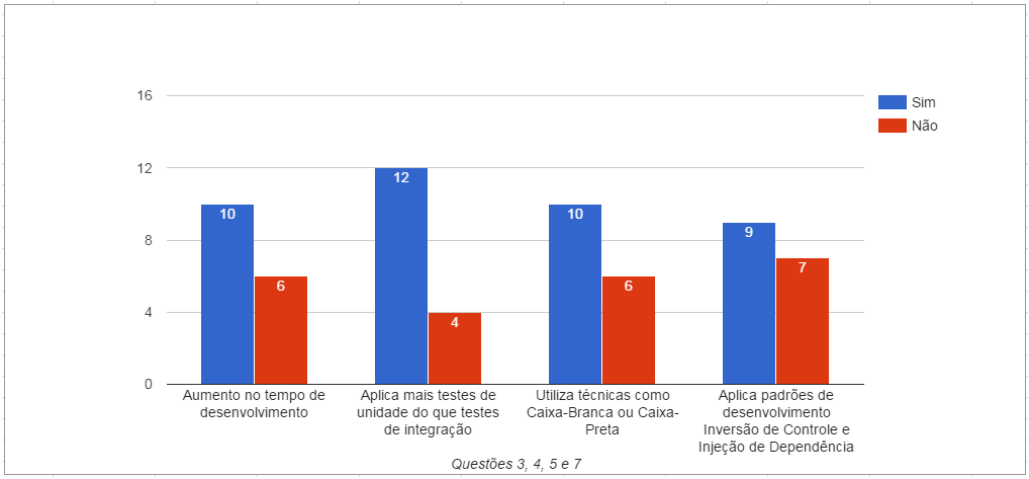
Figura 6 - Gráfico para representação dos dados do questionário.



Fonte: Elaborada pelos autores

Com base na análise dos dados obtidos através da questão três, houve um aumento no tempo do desenvolvimento, visto que com a aplicação dos testes de unidade é normal esse aumento de tempo, por conta do tempo investido para pensar e escrever os testes em cima daquela unidade escrita. Com os dados da questão quatro, é possível descobrir que a grande parte das pessoas que aplicam teste de unidade, se preocupam mais com a unidade do que com a integração delas com as outras partes do sistema, utilizando, conscientemente (ou não), a pirâmide de testes. Com os dados da questão cinco, é possível identificar que os entrevistados utilizam técnicas para escrever os testes de unidade, assim garantindo um teste coeso e fiel ao código escrito. Com os dados da questão sete, é notado que um pouco mais da metade das pessoas entrevistadas aplicam padrões de desenvolvimento, que garante mais ainda a qualidade dos testes escritos.

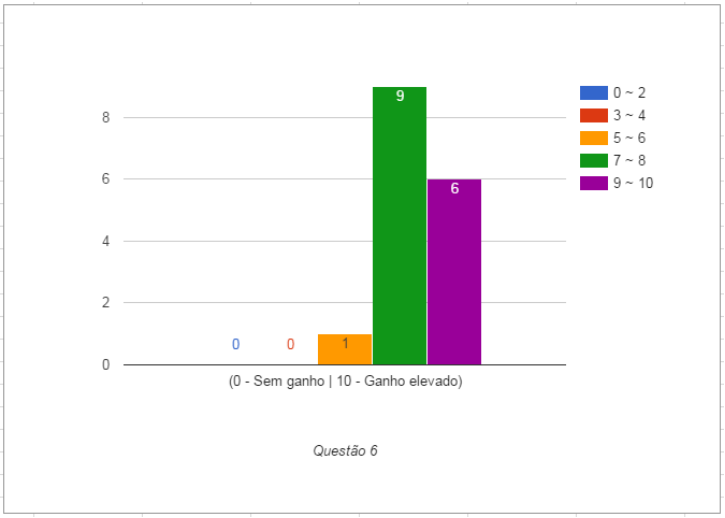
Figura 7 - Gráfico para representação dos dados do questionário.



Fonte: Elaborada pelos autores

Com base na análise dos dados obtidos através da questão seis, pode-se identificar que todos os entrevistados aplicam o TDD para guiar o desenvolvimento do sistema e que a maioria acredita ter ganhos positivos na qualidade do produto final.

Figura 8 - Gráfico para representação dos dados do questionário.



Fonte: Elaborada pelos autores

8 CONCLUSÃO

Desenvolver um sistema sem utilizar testes é um problema que a maioria das empresas sofrem à longo prazo. Com retrabalhos em excesso, manutenções desnecessárias e falhas à todo instante, são exemplos claros de um sistema sem uma cobertura ideal de testes, principalmente os de unidade. Com uma distribuição correta da quantidade à ser feita de cada tipo de teste, pode-se garantir muito mais precisamente o sucesso do produto final. Com o foco na base dos níveis de teste, encontramos o teste de unidade, um teste rápido e que garante muitos dos impactos que com o teste manual, poderia ser descoberta uma falha muito mais tarde, talvez até mesmo em um ambiente de produção num cliente em potencial. Ao analisar o questionário respondido, pode-se afirmar que essa é a realidade de quem aplica os testes de unidade no desenvolvimento do sistema.

Para agregar ainda mais ao sistema, existe a abordagem de desenvolvimento orientada por testes, o TDD, que traz o pensamento de teste de unidade, antes mesmo de escrevermos o nosso código, ajudando ainda mais no desenvolvimento seguro do sistema. Com o resultado do questionário, foi possível descobrir que a prática de TDD é feita entre os entrevistados, e grande parte deles dizem ter obtido ganho positivo na qualidade do produto final.

Pode-se até aplicar técnicas como os testes de caixa-preta e os testes de caixa-branca, que deixam mais claro como escrever o teste, porém com um sistema

altamente acoplado, isso não será de muito proveito, para solucionar essa necessidade, aparece a inversão de controle com a injeção de dependência, para garantir que os testes escritos serão mais coesos e que garantem um funcionamento das unidades do sistema. Ao cruzar as respostas dos entrevistados é possível confirmar que as técnicas de teste e padrões de projeto são utilizadas durante o desenvolvimento do sistema, apoiando na escrita dos testes.

SOFTWARE QUALITY: USE OF UNIT TEST

Abstract: This article demonstrates how the quality is something indispensable to software development. For a system has value should appreciate the quality of final product, to ensure software quality. In terms of code, it has great importance to use unit tests in all parts of the system, minimizing rework and possible failures. Integrating the test philosophy in the development process with other parts, like system architecture, while it encodes, thinking if the code is stable or not, becomes common and necessary practice, creating a clean and decoupled code.

Keywords: Software development. Test driven development. Unit test. Quality assurance.

REFERÊNCIAS

ANICHIE, Maurício. **Teste-Driven Development:** Teste e Design no Mundo Real. 1. ed. São Paulo: Casa do Código, 2012.

ANICHIE, Maurício. **Unidade, integração ou sistema? Qual teste fazer?**. Disponível em: <http://blog.caelum.com.br/unidade-integracao-ou-sistema-qual-teste-fazer/>.
Acesso em: 21 ago. 2015.

BECK, Kent. **TDD Desenvolvimento Guiado por Testes**. 1. ed. Porto Alegre: Bookman Editora, 2010.

FEATHER, Michael. **Trabalho Eficaz com Código Legado**. 3. ed. Porto Alegre: Bookman Editora, 2013.

PALMEIRA, Alessandro; LOSNAK, Ana. **Entenda a injeção de dependência nos frameworks MVC**. Disponível em: <http://blog.caelum.com.br/ioc-e-di-para-frameworks-mvc/>. Acesso em: 19 jul. 2015.

PRESSMAN, Roger. **Engenharia de Software:** Uma abordagem profissional. 7. ed. Porto Alegre: McGraw-Hill, 2011.

SILVEIRA, Paulo *et al.* **Introdução À Arquitetura e Design de Software -** Uma Visão Sobre a Plataforma Java. Estados Unidos da América: ELSEVIER USA, 2011.

SOARES, Marcos Paulo Lopes. **Inversão de Controle (IoC) e Injeção de Dependência (DI) -** Diferenças. Disponível em: <http://www.linhadecodigo.com.br/artigo/3418/inversao-de-controle-ioc-e-injecao-de-dependencia-di-diferencas.aspx>. Acesso em: 5 out. 2015.