INTERFACE TECNOLÓGICA

PRINCÍPIO DA INVERSÃO DE DEPENDÊNCIA NA QUALIDADE DE SOFTWARE:
Aplicação da injeção de dependência no desenvolvimento de software

INVERSION OF DEPENDENCY PRINCIPLE IN THE SOFTWARE QUALITY:
Application of the dependency Injection in the software development field

Bruna Piserà Gonçalves – pissara2@gmail.com
Faculdade de Tecnologia de Taquaritinga – Taquaritinga – SP – Brasil

Oswaldo Lazaro Mendes – oswaldo.lazaro@fatectq.edu.br
Faculdade de Tecnologia de Taquaritinga – Taquaritinga – SP – Brasil

DOI: 10.31510/infa.v19i1.1362
Data de submissão: 08/03/2022
Data do aceite: 29/05/2022
Data da publicação: 30/06/2022

RESUMO

A sociedade moderna cada vez mais demanda softwares para a resolução de atividades cotidianas. Por conta disso, as ferramentas computacionais mudam frequentemente. Para que tais mudanças ocorram, é imprescindível a escrita de testes automáticos que validem se o comportamento anterior à alteração continua sendo plausível após. Visto isso, o presente trabalho teve como objetivo apresentar a importância dos testes de unidade no desenvolvimento de software e como o princípio da inversão de controle e da injeção de dependência viabilizam a escrita dos cenários de testes e suas respectivas implementações. Para isso foi realizada uma pesquisa exploratória bibliográfica, bem como, exemplos empíricos através do desenvolvimento de códigos pelo autor. Com isso, pode-se observar que a ausência da aplicação do princípio de inversão de dependência e injeção de dependência torna inviável a realização do teste de unidade além de atingir um alto acoplamento no código. Concluiu-se, portanto, que com a utilização de tais princípios é possível garantir um desenvolvimento livre de dependências, com boa legibilidade, alta qualidade, e coesão em seu processamento atual e futuro, tornando o teste de unidade uma prática factível no que tange o desenvolvimento de software.
Palavras-chave: Teste. Unidade. Dependência. Acoplamento. Inversão.

ABSTRACT

Modern society more and more demands software to solve its daily problems. Because of that, computational tools change frequently. For these changes to be able to happen, it's essential to automate tests that assert whether the behavior before the changes remains the same afterward. Hence, the present paper aims to present the importance of the unit tests when it comes to software development and how the inversion of control and dependency principle enables the development of test scenarios and their respective implementations. To achieve that, a

bibliographical research was made. Also, empirical examples were drawn up by the author. After that, it is possible to observe that the absence of these principles turns the unit test writing into an unfeasible task - besides having a high coupling code. It's clear from this paper that the use of the Inversion of Control and Dependency Injection principles helps to guarantee a free-dependency development, with good readability, high quality, and cohesion for the time being and also the future of the computational tool under development. Thus making the unit tests a feasible assignment in the software development field.
Keywords: Test. Unit. Dependency. Coupling. Inversion.

1 INTRODUÇÃO

Winters et al (2020) afirma que testar sempre foi uma parte da programação de um sistema. Na verdade, mesmo os iniciantes em programação colocam manualmente alguns dados iniciais para garantir que o programa está performando corretamente. Por muito tempo, os testes de software lembravam esse método manual de inserir e averiguar a saída do código. Entretanto, desde o começo do milênio, a abordagem de testes de software na indústria mudou drasticamente a fim de acompanhar o tamanho e a complexidade dos sistemas modernos.

Para Delamaro et al (2016), a construção de um software não é uma tarefa trivial. Na verdade, de sua perspectiva, construir software pode ser uma tarefa bastante complexa, variando de acordo com as características e dimensões do sistema arquitetado. Como consequência, a construção da ferramenta computacional está sujeita a vários tipos de inconvenientes que na maioria das vezes acabam resultando em um produto distinto daquele que se almejava inicialmente.

Uma pesquisa conduzida por Tassey, G em 2002 na National Institute of Standards & Technology (NIST), afirma que o negligenciamento de testes de Software custava, naquele ano, para a economia norte-americana cerca de $22.2 and $59.5 bilhões de dólares, com aproximadamente metade desses custos entre os desenvolvedores na forma de correção de problemas e a outra metade pelos usuários das ferramentas na forma de prevenção de falhas e mitigação de esforços (TASSEY, 2002).

Outra pesquisa, desta vez em 2020, conduzida por Krasner, H. para o Consórcio de Informação e Qualidade de Software (CISQ, na sigla em Inglês) evidencia que a indústria de tecnologia da Informação não melhorou muito em 10 anos. A pesquisa mostra que 47% dos projetos de Software encontravam-se em vulnerabilidade devido a fatores como: orçamento, atrasos, e a produção de executáveis com baixa qualidade (KRASNER, 2020).

Durante o processo de desenvolvimento de um software, inicialmente o código de cada componente deve ser testado de forma isolada dos outros componentes do sistema, sendo esse

tipo de testes conhecido como testes de unidade (PFLEEGER, 2004). Com outras palavras, Thomas et al (2004) diz que os testes de unidade objetivam, principalmente, identificar erros na implementação da unidade, seja logicamente ou de não conformidade com os requisitos.

Segundo Winters et al (2020), existem muitos fatores que contribuem para uma equipe de software negligenciar os seus testes, mas um deles é o tempo. O ato de escrever testes pode tomar o mesmo tempo - ou mais - do que implementar uma funcionalidade.

Seemann e Deursen (2019) alegam que ter um código bem estruturado, coeso e com um baixo acoplamento facilita a diminuir a complexidade na escrita de testes. Nesse sentido, o conceito da Injeção de dependência pode auxiliar na escrita de componentes mais coesos e testáveis. Diminuindo o tempo e o esforço dos testes.

O presente artigo objetiva discorrer sobre a importância dos testes de unidade automáticos durante o desenvolvimento de uma ferramenta computacional e como um código que faz uso do princípio da Inversão de Controle e da Injeção de dependência viabiliza a escrita dos cenários e suas respectivas implementações. Para isso, foram realizadas revisões bibliográficas, bem como exemplos empíricos através do desenvolvimento de códigos de exemplo pelo autor.

2 FUNDAMENTAÇÃO TEÓRICA

A atual seção apresentará um embasamento teórico sobre os seguintes assuntos: teste de unidade, *mock*, princípio da inversão de controle e injeção de dependência. Tópicos esse que estarão correlacionados na seção 4 – Resultados e discussões – com o intuito de atingir o objetivo proposto inicialmente no presente trabalho.

2.1 Teste de unidade

De acordo com Sommerville (2011) o teste é destinado a mostrar que um programa faz o que é proposto a fazer e para descobrir os defeitos do programa antes do uso. Ao testar um sistema, dados fictícios são utilizados em sua execução. Os resultados dessa execução são analisados em busca de alguma anomalia, erros ou informações sobre os atributos não funcionais do sistema.

Testes de unidade, segundo Valente (2020) são testes automatizados de pequenas unidades de código, normalmente classes, as quais são testadas de forma isolada do restante do sistema. O teste unitário é um programa que invoca a função de uma classe para garantir que a

função (ou método) tenha os resultados esperados como saída. Desse modo, quando se faz uso do teste de unidade, o código do programa pode ser segmentado em dois grupos: um grupo de classes – que desenvolve os requisitos do sistema – e um grupo de teste. Com outras palavras, Seshadri e Green (2014) apontam que o teste de unidade é o conceito de tomar uma única função ou parte de nosso código e escrever asserções e testes para assegurar que funcionará da maneira prevista.

De acordo com Sommerville (2011) deve-se escrever dois tipos de casos de teste. O primeiro cenário de teste objetiva espelhar o comportamento normal de um programa e deve assegurar que o componente executa aquilo que fora imaginado.. Já o segundo tipo deve ser baseado em testes empíricos, e é a partir deles que os problemas mais corriqueiros se manifestam. Deve-se fazer uso de entradas atípicas com o intuito de averiguar que estas são corretamente processadas e que não farão o componente falhar.

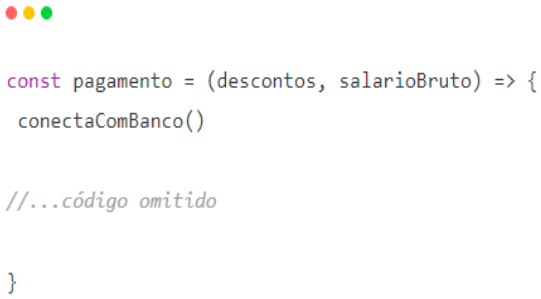
De outro ponto de vista Hetzel (1987) define teste como: “o processo de executar um programa ou sistema com a finalidade de encontrar erros. Teste é a medida de qualidade do software”. Um sistema testável é um sistema que permite testar efetivamente partes individuais do sistema. Contudo, é difícil testar uma classe quando a mesma depende de outra, pois cada classe é vinculada a tipos de armazenamentos específicos. A injeção de dependência auxilia na solução desse problema (BETTS, ET. AL, 2013).

2.1.1 *Mock*

De acordo com Valente (2020) o *mock* é um objeto que emula o objeto real, mas apenas para permitir o teste do programa. Fowler (2006) diz que mocks são simulações pré-programadas com expectativas de formar uma especificação das chamadas que devem receber. Eles podem lançar uma exceção se receberem uma chamada que não esperam e são verificados durante a verificação para garantir que receberam todas as chamadas que esperavam.

Ao escrever testes, às vezes é necessário simular partes do sistema para tornar os testes possíveis e os resultados reproduzíveis. *Mocks* são representações ou unidades falsas que simulam as ações de unidades reais. Por exemplo, na Figura 1 há o método *pagamento()* que é encarregado de transacionar um pagamento. Dentro dela há um outro método que faz conexão com o banco de dados, conexão essa que compõe o processo de pagamento. Essas funções estão dentro de uma classe maior que tem relação com pagamentos (NASCIMENTO, 2021):

Figura 1 – Exemplo mock



Fonte: Nascimento, 2021.

Para testar o método *pagamento()* não obrigatoriamente precisa-se ter o método *conectComBanco()* como dependência, independentemente do que ocorre dentro do método *conectComBanco()*, deve ser possível testar o método *pagamento()*. Desse modo, é necessário mockar ou criar um *mock*, simulando assim o comportamento da função *conectaComBanco()* sem precisar executar de fato a função (NASCIMENTO, 2021).

2.2 Princípio da inversão de dependência

O princípio da inversão de dependência faz parte do SOLID. De acordo com Betts et. al (2013) o SOLID é um acrônimo composto pelos seguintes princípios: princípio da Responsabilidade Única, princípio do aberto/fechado, princípio da substituição de Liskov, princípio da segregação de interface e princípio da inversão de dependência. Martin (2019) diz que tais princípios são guias para desenvolver estruturas de código que sejam fáceis de compreender, resistentes à mudanças e que sejam também a base de componentes que possam ser utilizados por vários softwares distintos.

Entretanto, o uso desses princípios deve ser feito apenas quando modificações são necessárias e as necessidades dessas modificações evidenciam *bad smells* (quando o código é mal projetado ou quando a implementação escolhida estava errada (Martin, 2019)) presentes no design do código.

O princípio da injeção de dependência aborda que (BETTS, ET. AL, 2013):

a) Módulos de alto nível não devem depender de módulos de baixo nível. Os dois devem depender de abstrações.

b) As abstrações não devem depender de detalhes. Os detalhes devem depender de abstrações.

Pode-se definir a interação que ocorre entre dois módulos como conectividade, e a intensidade dessa conectividade, como acoplamento. Ao se apontar que dois módulos estão

vigorosamente acoplados significa que há entre eles um alto grau de conectividade, isto é, um deles tem um conhecimento impróprio da implementação do outro, podendo estar com um alto nível de dependência desse outro módulo (MYERS, 1978).

O princípio da inversão de dependência trata que sistemas flexíveis não tem dependências de código fonte, eles apenas se referem a abstrações. Desse modo os sistemas são flexíveis o suficiente para que suas implementações possam mudar diminuindo o impacto em dependentes. Isso pode ser resumido como: não se deve depender de nada que seja concreto. Depender de elementos concretos é arriscado. Esse perigo transcorre do fato que abstrações são mais estáveis que implementações. Portanto, as abstrações são mais confiáveis no que tange a dependência (MARTIN, 2019).

De acordo com Aniche (2015) a ideia é: sempre que uma classe for depender de outra, ela deve depender sempre de outro módulo mais estável do que ela mesma. Se *A* depende de *B*, logo, o intuito é que *A* seja menos estável que *B*. Mas *B* depende de *C*. Logo, a ideia é que *C* seja mais estável que *B*. Isto é, as classes devem sempre buscar a estabilidade, dependendo de módulos mais estáveis que ela mesmo.

Segundo Valente (2020) um nome mais intuitivo para o princípio seria: prefira Interfaces a Classes**.** Para ele, tal princípio recomenda que uma classe cliente deve estabelecer dependências prioritariamente com abstrações e não com implementações concretas, pois abstrações (interfaces) são mais estáveis do que implementações concretas (classes). A ideia é então trocar (ou inverter) as dependências: em vez de depender de classes concretas, clientes devem depender de interfaces.

2.3 Injeção de dependência

Importante esclarecer que o princípio de inversão de dependência (visto no tópico anterior) é distinto da injeção de dependência. A injeção de dependência é a ideia de ter os parâmetros no construtor, e alguém, geralmente um framework, automaticamente injetar essas dependências para você (ANICHE, 2015). Em outros termos, Fowler (2004) aponta que a injeção de dependência consiste em tornar possível que uma determinada funcionalidade seja inserida em uma classe, sem que a mesma saiba como ela implementa as suas funcionalidades ou como ela foi criada.

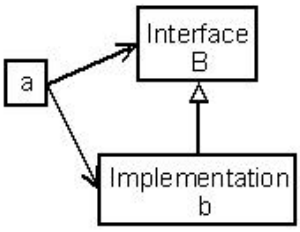
De acordo com Walls (2008), ao aplicar a Injeção de Dependência (*Dependency Injection – DI*), os objetos recebem suas dependências em tempo de criação por alguma entidade externa que coordena cada objeto no sistema. Em outras palavras, as dependências são

injetadas nos objetos. Por isso, DI significa uma inversão de responsabilidade com relação a como um objeto obtém referências a objetos colaboradores.

De acordo com Prasanna (2009) a injeção de dependência oferece uma alternativa inovadora através do princípio de Hollywood: "Não nos chame. Nós vamos lhe chamar", quer dizer que um componente não precisa ter conhecimento de suas dependências nem solicitá-las diretamente. A injeção de dependência é capaz de resolver vários problemas quando uma aplicação é projetada e construída em torno dela.

Segundo Razina e Janzem (2007) a injeção de dependência é um padrão que permite ao programador injetar objetos em uma classe usando um contêiner configurado externamente, em vez de permitir que a classe instancie os objetos diretamente. Suponha que há uma classe, *a*, que contém um objeto de classe *b*. A classe *b* é uma implementação da interface *B*. Isso faz com que a classe *a* tenha uma dependência da classe *b*, assim como a interface *B*, de acordo com a Figura 2.

Figura 2 - Exemplo para injeção de dependência



Fonte: Adaptado de Razina e Janzem, 2007.

A injeção de dependência remove a dependência da classe *a* da classe *b* adicionando um contêiner e tornando responsável pela pesquisa de dependência. Esse contêiner é a interface *B*. Se não quiser mais usar *b*, mas quiser usar uma nova classe, *c*, pode-se facilmente fazer a substituição sem alterar a classe *a* (RAZINA E JANZEM, 2007).

A injeção de dependência também facilita os testes. Como pode-se observar na Figura 2, testar a classe *"a"* inevitavelmente leva a testar a implementação *"b"*. Se a injeção de dependência for aplicada ao se desenvolver um sistema, pode-se introduzir uma implementação simulada da classe *b* no programa e testar de forma isolada a classe. Permitindo testes mais fáceis, ou testabilidade. Como a testabilidade é um dos componentes da manutenção, isso nos leva acreditar que o padrão de injeção de dependência melhora a manutenção do código (RAZINA E JANZEM, 2007).

3 PROCEDIMENTOS METODOLÓGICOS

Para o desenvolvimento desse artigo foi realizada uma pesquisa bibliográfica, originando do levantamento de referências teóricas já estudadas e publicadas (GIL, 2007). De acordo com Vergara (2000), a partir de material já elaborado é desenvolvida a pesquisa bibliográfica, material esse que é composto, essencialmente, de artigos científicos e livros que são essenciais para o levantamento de informações básicas sobre os aspectos direta e indiretamente ligados à nossa temática.

O presente estudo trata sobre a inviabilidade de garantir a qualidade de software, ou seja, de realizar testes de unidade sem a aplicação do princípio da inversão de dependência juntamente com a injeção de dependência. Desse modo foi realizada uma pesquisa exploratória bibliográfica com a visão de diversos autores sobre os tópicos relacionados. Bem como, a apresentação de um código desenvolvido pelo autor, mostrando de maneira prática como resolver tal inviabilidade utilizando o princípio da inversão de dependência.

4 RESULTADOS E DISCUSSÃO

O conceito de Injeção de dependência torna-se indubitavelmente prático quando os testes de unidade, especialmente os *Mocks*, são desenvolvidos.

Para exemplificar, uma classe denominada *ClientHandler* foi criada, utilizando arbitrariamente a sintaxe do Java. Tal classe, como o seu nome já diz, será responsável por gerenciar a inserção de um novo cliente na base de dados da aplicação. Antes de inserir, será feito apenas um controle negocial: verificar se já existe algum cliente com o nome e o *ID* fornecido pelo usuário. Para tanto, o código deverá conectar-se à base de dados. Portanto, nota-se que a classe terá uma dependência: a classe que conecta, procura e persiste um cliente. No exemplo, tal classe será a *ClientRepository.*

A priori não é utilizado o conceito da injeção de dependência. A classe *ClientRepository* é instanciada manualmente. Conforme pode ser visto na linha 5, da Figura 3.

Figura 3 - Código de exemplo Parte I

```
3   public class ClientHandler {
4   @       public void save(Client clientToSave) {
5               ClientRepository repository = new ClientRepository();
6
7               Client clientFound = repository.getClientByNameAndId(
8                   clientToSave.name,
9                   clientToSave.id
10              );
11
12              if(clientFound == null) {
13                  repository.save(clientToSave);
14              } else {
15                  throw new IllegalArgumentException("Cliente já existe");
16              }
17      }
18  }
```

Fonte: O autor, 2022.

Para fins de legibilidade e compilação, a classe funciona - até então, nada prejudicial. Entretanto, os problemas começam no momento de testar unitariamente a classe *ClientHandler*. A essência do teste unitário reside no fato de querer testar-se a menor unidade do código, que normalmente é um método. Eles devem garantir o comportamento esperado do método testado, dadas certas condições iniciais particulares.

Em um sistema, muitas pequenas peças (banco de dados, filas, servidores de arquivo, e muitas outras) se integram a fim de realizar uma tarefa. O cerne do teste de unidade é a abstração. Em um teste de unidade, não é demonstrado se o método conseguirá conectar-se à uma instância de banco de dados (real ou *in-memory*), fila, etc. pelo contrário, assume-se condições para essas integrações e verifica se o comportamento esperado está de fato acontecendo. Por exemplo: diga-se que uma pessoa queira testar se o seu guarda-chuva funciona. Não é necessário esperar uma chuva ocorrer para garantir que o comportamento - que nesse caso é garantir que o aparato protege o usuário de se molhar - é o esperado. É possível *simular (mockar)* a chuva, colocando o utensílio sob um chuveiro.

Nesse sentido, *Mockar* as condições iniciais (que no caso da classe *ClientHandler* é a conexão com o banco de dados) é muito importante. Afinal, se isso não fosse feito, o teste não seria trivial; isto é, a fim de testar um simples método, seria necessário subir uma instância de banco de dados local ou remota e simular conexões que foram bem e mal sucedidas. Tudo isso para garantir o comportamento de uma rotina.

Do jeito que a classe foi escrita na Figura 3 fica impossível simular a conexão e a existência do cliente na base de dados, pois a classe *ClientRepository* - que é a responsável por

conectar, buscar e inserir o cliente- está sendo instanciada manualmente. Não está flexível. Não há como, de fora da classe, fornecer uma instância de *mock* para esta dependência. O teste de unidade não pode, portanto, ser escrito. E isso é um problema.

Ao refatorar a classe, fazendo com que o atributo do *ClientRepository* seja passado (injetado) para a classe através do construtor, garante-se a flexibilidade dos possíveis valores dessa dependência, inclusive a possibilidade de *mock.* A classe ficará de acordo com a Figura 4.

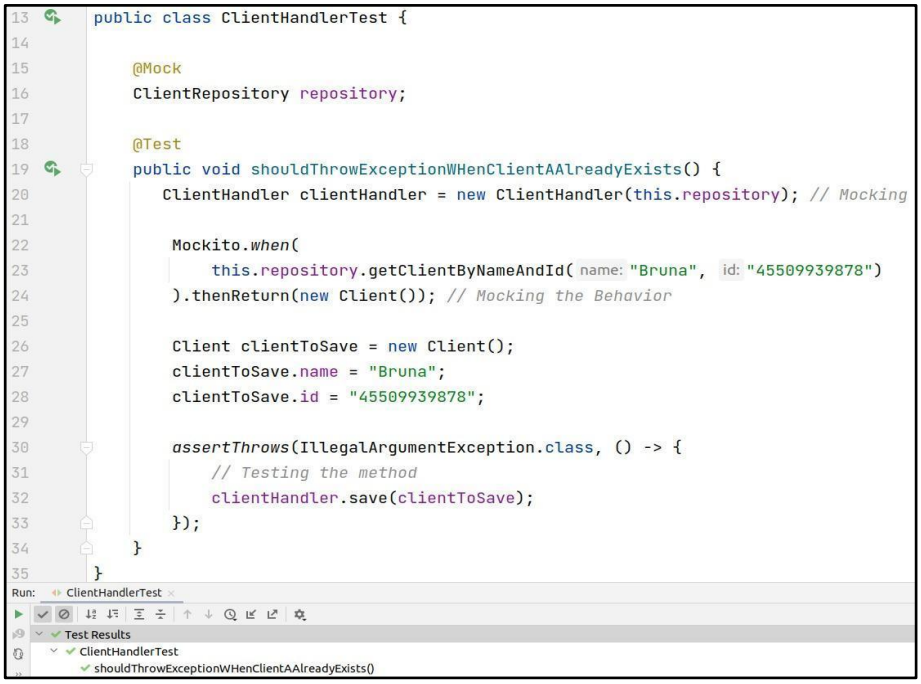
Figura 4 - Código de exemplo Parte II

```
3   public class ClientHandler {
4
5       ClientRepository repository;
6
7       ClientHandler(ClientRepository clientRepository) {
8           this.repository = clientRepository;
9       }
10
11  @   public void save(Client clientToSave) {
12          Client clientFound = this.repository.getClientByNameAndId(
13              clientToSave.name,
14              clientToSave.id
15          );
16
17          if(clientFound == null) {
18              this.repository.save(clientToSave);
19          } else {
20              throw new IllegalArgumentException("Cliente já existe");
21          }
22      }
23  }
```

Fonte: O autor, 2022.

Agora, já é possível *simular* a conexão com o banco. Conforme pode ser visto na Figura 5.

Figura 5 - Código de exemplo Parte III



Fonte: O autor, 2022.

Em suma, ao utilizar-se a inversão de dependência, a classe não mais regula as suas dependências, mas as recebe. Isso fez com que a *ClientHandler* fosse passível de ser testada unitariamente, garantindo assim além de uma boa legibilidade, uma alta qualidade e coerência em seu processamento atual e futuro.

5 CONCLUSÃO

Com a disseminação da computação a nível organizacional e pessoal, testar software é fundamental. As ferramentas computacionais estão cada vez mais presentes no cotidiano da sociedade moderna. Por fazerem parte de quase todos os aspectos da civilização, os softwares precisam mudar frequentemente, a fim de se adaptarem às novas demandas que a sociedade impõe. Para que o software mude com confiança, os testes de unidade são inegociáveis, posto que eles garantem que o que funcionava continua funcionando após as alterações. Mesmo assim, muitos projetos de software os negligenciam. Os motivos são vários: falta de conhecimento, má gestão, prazos apertados e, sobretudo, o tempo.

Testar leva tempo. E tempo demanda recurso. Recurso gera custo. Por isso, quanto mais tempo levar para os testes serem escritos, maior será o custo no orçamento do projeto.

Entretanto, a onerosidade na escrita dos testes unitários é sinal de que o código poderia ser refatorado. Isso porque os testes de unidade estão intrinsecamente ligados à qualidade da codificação. Se uma unidade (e por unidade, nesse contexto, podemos entender como classes e métodos) for altamente acoplada, então as simulações das condições iniciais dos cenários de testes (os famigerados *Mocks*) tornam-se dificílimas, se não impossíveis.

Nesse sentido, a prática da Inversão de Controle e o seu padrão de projeto denominado Injeção de dependência torna-se inexoravelmente importante no que tange a escrita dos testes: como já abordado, ao testar a unidade, o principal objetivo é simular as condições iniciais e verificar se a solução particular (ou o comportamento) é o esperado. Não é importante, neste momento, testar as integrações. Em suma, assumem-se os dados de entrada e verifica-se o comportamento e os dados de saída.

Considerando o cenário apresentado, o trabalho resumiu-se em, através de revisões bibliográficas sobre o tema, demonstrar que a Inversão de Controle e a Injeção de dependência estão intimamente integradas à facilidade de se testar unitariamente um código. Para comprovar empiricamente essa correlação e causalidade, a autora desenvolveu, didaticamente, códigos de exemplos a fim de apresentar o problema que a falta da Injeção de dependência causa, e como isto está ligado ao tempo (e consequentemente, orçamento) dos projetos de software.

REFERÊNCIAS

ANICHE, M. **Orientação a objetos e SOLID para ninjas: projetando classes flexíveis.** São Paulo: Casa do Código, 2015.

BETTS, ET. AL. **Dependency injection with unity.** Washington: Microsoft, 2013.

DELAMARO, M; MALDONADO, J. C; JINO, M. **Introdução ao Teste de Software.** 2. ed - Rio de Janeiro: Elsevier, 2016.

FOWLER, M. **Inversion of Control Containers and the Dependency Injection pattern.** 2004 .Disponível em:< https://martinfowler.com/bliki/InversionOfControl.html>.acesso em: janeiro de 2022.HETZEL, William. **Guia completo do teste de software**. Rio de Janeiro: Campus, 1987.

GIL, A.C. **Métodos e técnicas de pesquisa social.** São Paulo: Atlas, 2007.

KRASNER, H. **The Cost of Poor Software Quality in the US: A 2020 Report.** Consortium for information & Software Quality (CISQ), Janeiro de 2021. Disponível em: <https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf>. acesso em: fevereiro de 2022.

MARTIN, R. C. **Arquitetura Limpa:** O Guia do Artesão para Estrutura e Design de Software. 1. ed. Rio de Janeiro: Alta Books Editora, 2019.

MYERS, G. J. **Composite/Structured Design.** 1. ed. New York: Van Nostrand Reinhold, 1978.

NASCIMENTO, F. **Mocks e Stubs em testes: o que são e quais as diferenças.** Disponível em:<https://www.alura.com.br/artigos/testes-com-mocksstubs?gclid=CjwKCAiAgvKQBhBbEiwAaPQw3HGjoGhsqNvYKiXEEfw0d7W1FlHL3nor-Htne-IXWGl14pW0lbuRNhoCdgIQAvD_BwE>. acesso em: fevereiro de 2022.
PFLEEGER, S. L. **Engenharia de software**: teoria e prática. 2. ed. São Paulo: Prentice Hall, 2004.

PRASANNA, D. R. **Dependency injection:** design patterns using spring and guice. Nova York: Manning Publications, 2009.

RAZINA, E; JANZEM, D. S. **Effects of Dependency Injection on Maintainability**. 11º Internacional Conference: Software Engineering and Applications. Cambridge, MA, USA. Novembro, 2007.

SEEMANN, M; DEURSEN, V. S. **Dependency Injection: principles, practices and patterns.** Shelter Island, Nova York: Manning Publications Co, 2019.

SESHADRI, S. GREEN, B. **Desenvolvendo com angularJs.** São Paulo: Novatec, 2014.

SOMMERVILLE, I. **Engenharia de software.** 9 ed. São Paulo: Pearson Prentice Hall, 2011.

TASSEY, D. **The Economic Impacts of Inadequate Infrastructure for Software Testing.** National Institute of Standards & Technology (NIST), Maio de 2002. Disponível em: <https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf>. acesso em: fevereiro de 2022.

THOMAS, J. YOUNG, M. BROWN, K. GLOVER, A **Java Testing Patterns**. Indianápolis: John Wiley, 2004.

VALENTE, M. T. **Engenharia de Software Moderna.** 1 ed. Minas Gerais: independente,2020.

VERGARA, S. C. **Projetos e relatórios de pesquisa em administração.** 3.ed. Rio de Janeiro: Atlas, 2000.

WALLS, Craig; BREIDENBACH, Ryan**. Spring em Ação.** 2. ed. Rio de Janeiro: Alta Books, 2008.

WINTERS; MANSHRECK, T; HYRUM, W. **Software Engineering at Google - lessons learned from programming overtime.** 1. ed - California: O'Reilly Media, 2020.