UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CÂMPUS CORNÉLIO PROCÓPIO
DEPARTAMENTO DE COMPUTAÇÃO
ENGENHARIA DE SOFTWARE

PAULO RICARDO BUSCH

GERAÇÃO DE TESTES DE INTEGRAÇÃO PARA APIS WEB EM C#

TRABALHO DE CONCLUSÃO DE CURSO

CORNÉLIO PROCÓPIO
2021

PAULO RICARDO BUSCH

GERAÇÃO DE TESTES DE INTEGRAÇÃO PARA APIS WEB EM C#

Trabalho de Conclusão de Curso de graduação, apresentado ao curso de graduação de Engenharia de Software da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para a obtenção do título de Bacharel.

Orientador: Prof. Dr. André Takeshi Endo

CORNÉLIO PROCÓPIO
2021

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Cornélio Procópio
Departamento de Computação
Engenharia de Software

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

TERMO DE APROVAÇÃO

GERAÇÃO DE TESTES DE INTEGRAÇÃO PARA APIS WEB EM C#

por

PAULO RICARDO BUSCH

Este Trabalho de Conclusão de Curso de graduação foi julgado adequado para obtenção do Título de Bacharel e aprovado em sua forma final pelo Programa de Graduação em Engenharia de Software da Universidade Tecnológica Federal do Paraná.

Cornélio Procópio, 07 de dezembro de 2021

_____

Prof. Dr. André Takeshi Endo

_____

Prof. Dr. Cléber Gimenez Corrêa

_____

Prof. Dr. Paulo Augusto Nardi

"A Folha de Aprovação assinada encontra-se na Coordenação do Curso"

RESUMO

BUSCH, Paulo Ricardo. **Geração de Testes de Integração para APIs Web em C#.** 2021. 46 f. Trabalho de Conclusão de Curso (Graduação) – Engenharia de Software. Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2021.

A atividade de teste é um processo fundamental para a garantia de qualidade de um *software*, embora seja uma atividade custosa, podendo representar até 50% do custo total de um *software*. Neste sentido, é importante a adoção de soluções que apliquem testes automatizados. Principalmente para serviços disponibilizados na *Internet* em forma de APIs Web, porque podem impactar diversos consumidores. Muitos desses serviços são desenvolvidos utilizando a linguagem C#, que está se tornando cada vez mais conhecida. Este trabalho apresenta o desenvolvimento da ferramenta TesTool, que tem como foco principal a geração de código de teste de integração para APIs Web em C#. Para isso, foi utilizado o *framework* XUnit que possibilita a programação de testes para a plataforma .NET. Além disso, foram utilizados recursos da linguagem C# para geração de código. Espera-se como resultado melhorar a produtividade na codificação de testes automatizados por meio de várias funcionalidades providas pela ferramenta TesTool.

Palavras-chave: API Web. Teste de Integração. Geração de Teste. XUnit. TesTool.

ABSTRACT

BUSCH, Paulo Ricardo. **Generating Integration Tests for Web APIs in C#.** 2021. 46 f. Trabalho de Conclusão de Curso (Graduação) – Engenharia de Software. Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2021.

The testing activity is a fundamental process for the quality assurance of a software, although it is a costly activity, which can represent up to 50% of the total cost of a software. In this sense, it is important to adopt solutions that apply automated tests. Mainly for services made available on the Internet in the form of Web APIs, because they can impact different consumers. Many of these services are developed using the C# language, which is becoming more and more popular. This work presents the development of the TesTool tool, which has as main focus the generation of integration test code for Web APIs in C#. For this, the XUnit framework was used, which allows the programming of tests for the .NET platform. In addition, C# language resources were used for code generation. As a result, it is expected to improve productivity in the coding of automated tests through several functionalities provided by the TesTool tool.

Keywords: Web API. Integration Test. Test Generation. XUnit. TesTool.

LISTA DE TABELAS

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Nas últimas décadas inúmeros serviços foram disponibilizados na Web para atender a diversas necessidades de empresas e usuários. De acordo com informações do *site* Registro Br (REGISTROBR, 2021), desde o ano de 1996 até os dias atuais, o número de *sites* registrados vem numa tendência crescente, ultrapassando a marca de 4 milhões e demonstrando a importância das aplicações Web.

1.1 Problema

De acordo com MYERS et al. (2011), o teste de *software* pode ser entendido como um ou mais processos com o objetivo de certificar que um programa realiza o que foi projetado para fazer. Os autores afirmam que realizar testes ainda continua caro, representando aproximadamente 50% do custo total de desenvolvimento. Desta forma é importante considerar o uso de ferramentas para auxiliar no processo de teste de *software* para tornar o processo mais eficiente.

Yusifoğlu et al. (2015) destacam que existem muitos estudos na área de teste de unidade e de sistema, mas que existe uma carência de trabalhos sobre teste de integração, representando apenas 7% do total. Os autores também afirmam que existem poucas ferramentas para a aplicação deste tipo de teste e que é importante ter novos estudos na área. Diante disso, este trabalho realiza um estudo das ferramentas existentes para o teste de integração.

ARCURI (2020) define APIs (interface de programação de aplicações do inglês *application programming interfaces*) RESTful como serviços da Web que expõem funcionalidades em uma rede por meio de protocolos de comunicação. O autor ressalta que realizar testes nessas APIs exige um esforço considerável, porque é necessário configurar um ambiente controlado e gerar dados para serem consumidos na interação com a API. Diante disso, é válido estudar meios para simplificar a configuração e geração de testes, mais especificamente para C#, que é a linguagem foco deste projeto.

1.2 Justificativa

Para o desenvolvimento de *software* com qualidade, teste de *software* é um elemento fundamental (PRESSMAN, 2016). Neste sentido, foi realizado um estudo das ferramentas existentes, e o desenvolvimento de uma nova ferramenta de geração de código de teste para C#, agregando soluções já conhecidas com o propósito de oferecer funcionalidades que vão ajudar na implementação de testes de integração.

Realizar teste em APIs é um desafio, principalmente no início, quando é necessário realizar uma série de configurações (ARCURI, 2020). Considerando isto, a ferramenta desenvolvida prevê a geração automática de um projeto de teste com a configuração necessária para execução, e com uma estrutura que facilite o entendimento e evolução do projeto. Demais funcionalidades irão complementar o processo de teste, e farão uso dessa arquitetura.

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo deste trabalho é apresentar a ferramenta TesTool, que auxilia na geração de código de teste de integração para APIs Web em C#. A ferramenta recebe como entrada um projeto de API Web em C# e instruções por linha de comando, para produzir código de teste em xUnit (.NET FOUNDATION, 2021a).

1.3.2 Objetivos Específicos

A ferramenta permite gerar instâncias de objetos, necessários na interação com a API. Foi adotada uma convenção dos dados que serão atribuídos a cada propriedade (campos de classes em C#), de acordo com o nome e tipo.

Outra parte da solução permite gerar código de comparação de objetos com o uso de assertivas. Para cada propriedade que tiver o mesmo nome e tipo, será gerada uma assertiva.

Além disso, a ferramenta gera classes de teste tendo como entrada um *controller* (controlador) de API Web. Também é possível gerar projetos de teste a partir de projetos de API Web. Os objetivos anteriores foram utilizados nessas funcionalidades para gerar os artefatos de teste.

1.4 Organização do Texto

Os próximos capítulos estão organizados da seguinte forma: Capítulo 2 apresenta a fundamentação teórica com os elementos chave do trabalho e trabalhos relacionados. No Capítulo 3 é apresentado o desenvolvimento do projeto com as tecnologias utilizadas, uma explicação da arquitetura empregada e demonstração por meio de um exemplo. O Capítulo 4 mostra o estudo de caso realizado, com discussões acerca dos resultados. E, por fim, o Capítulo 5 apresenta a conclusão do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos chave do trabalho, que serão importantes para auxiliar na compreensão da ferramenta. A Seção 2.1 explica o que são APIs Web, identificando seus recursos e aplicações. Na Seção 2.2 é apresentado o teste de integração, mostrando seu conceito e importância na identificação de falhas de *software*. A Seção 2.3 explica a ferramenta XUnit, apresentando seus objetivos e características. Na Seção 2.4 são apresentados os modelos T4 e seus tipos. Na sequência, a Seção 2.5 elenca trabalhos relacionados com proposta similar.

2.1 API Web

De acordo com Sohan (2015), APIs Web são usadas como mecanismo de interconectividade para acessar serviços de *software* na *Internet* e traz como benefícios serviços melhores a um baixo custo para seus usuários. A implementação de APIs Web pode seguir vários protocolos, como SOAP (Protocolo de acesso a objetos simples ou do inglês *Simple Object Access Protocol*) e RESTful (Restrições de arquitetura definidas por Roy) (SOHAN, 2015). Muitas empresas usam APIs RESTful para expor seus serviços, como Twitter, Google, Amazon, etc. (ARCURI, 2020). Por ser amplamente utilizada, a estrutura RESTful será o foco deste estudo.

Uma forma de implementar uma API Web é seguindo a estrutura proposta pelo REST (Transferência Representacional de Estado do inglês *Representational State Transfer*) que estabelece diretrizes de como os recursos serão acessados por HTTP(S) em uma rede. Os recursos são identificados com URLs e podem ser manipulados usando a semântica do HTTP, por exemplo, solicitações GET para buscar dados, POST para criar novos dados, PUT / PATCH para modificar as informações e DELETE para removê-las. As entradas podem ser fornecidas por parâmetros nas URLs, cabeçalhos HTTP e no corpo da requisição. Os dados podem ser transferidos em qualquer formato; a forma mais comum é por JSON (ARCURI et al, 2020).

Uma das linguagens que permite a criação de APIs Web é o C# que é fortemente tipado e orientado a objetos (MICROSOFT, 2021a). Na Figura 1 é apresentado um exemplo de *controller* (controlador) que realiza o gerenciamento de desenvolvedores.

Figura 1: Exemplo de Controller em C#

```
10  namespace Tasks.API.Controllers
11  {
12      public class DevelopersController : TasksControllerBase
13      {
14          private readonly IDeveloperService _developerService;
15
16          public DevelopersController(IDeveloperService developerService)...
20
21          [HttpGet]
22          public async Task<Result<IEnumerable<DeveloperListDto>>> ListAsync([FromQuery] PaginationDto paginationDto)...
26
27          [HttpGet("{id}")]
28          public async Task<Result<DeveloperDetailDto>> GetAsync([FromRoute] Guid id)...
32
33          [HttpPost]
34          public async Task<Result> CreateAsync([FromBody] DeveloperCreateDto developerDto)...
38
39          [HttpPut("{id}")]
40          public async Task<Result> UpdateAsync([FromBody] DeveloperUpdateDto developerDto, [FromRoute] Guid id)...
45
46          [HttpDelete("{id}")]
47          public async Task<Result> DeleteAsync([FromRoute] Guid id)...
51      }
52  }
```

Fonte: Autoria Própria

Na imagem é possível perceber a exposição de vários recursos. O primeiro método anotado com o atributo HttpGet (Linha 21) permite consultar a lista de desenvolvedores cadastrados. Neste método os resultados podem ser paginados por meio de parâmetros *query string* (parâmetros de consulta) que são recebidos no objeto anotado com FromQuery. O segundo método também anotado com HttpGet (Linha 27) permite consultar um desenvolvedor específico por meio do parâmetro id (identificador único), anotado com FromRoute que deve ser passado como segmento da URL (MICROSOFT, 2021a).

Existem também recursos responsáveis por alterar as informações. No método anotado com o atributo HttpPost (Linha 33) é possível fazer o cadastro de um novo desenvolvedor, passando as informações no corpo da requisição, que são recebidas no objeto anotado com FromBody. No penúltimo método anotado com HttpPut (Linha 39) é possível atualizar um desenvolvedor, passando o seu id e as informações na requisição. No último método (Linha 46) pode ser feita a remoção de um desenvolvedor passando o id (MICROSOFT, 2021a).

2.2 Teste de Integração

Teste de *software* é uma etapa crucial do ciclo de vida do *software* que é necessária para identificar falhas a serem corrigidas. O teste pode ser realizado em níveis de unidade, integração e de sistema (BERTOLINO, 2005):

- **Teste de unidade:** responsável por testar a menor parte do *software*, envolvendo seus módulos ou componentes. Esse teste foca na lógica interna de processamento e nas estruturas de dados, podendo ser conduzido de forma paralela (PRESSMAN, 2016);
- **Teste de integração:** são testes desenvolvidos para descobrir erros associados às interfaces na integração das unidades de um *software* (PRESSMAN, 2016);
- **Teste de sistema:** exercita o sistema como um todo, por meio de vários tipos de teste, como de recuperação, segurança, esforço, desempenho e disponibilidade (PRESSMAN, 2016).

O teste de integração procura identificar problemas na combinação entre as unidades de um *software* (PRESSMAN, 2016). Entretanto, muitas vezes é impraticável e até mesmo impossível encontrar todas as falhas de um programa devido ao grande número de testes que seriam necessários. Por isso, foram desenvolvidas técnicas que exercitam o *software* sob diferentes perspectivas para selecionar testes que têm maior probabilidade de encontrar errors (MYERS et al, 2011):

- **Caixa preta:** pode ser interpretado como um teste *data-driven* (orientado a dados) ou baseado em *input / output-driven* (dirigido por entradas e saídas). É necessário enxergar o programa como uma caixa preta, interagindo com suas interfaces sem se preocupar com seu comportamento interno. O objetivo é identificar circunstâncias em que o programa não reage de acordo com as especificações;
- **Caixa branca:** é uma estratégia *logic-driven* (dirigida por lógica) que considera o comportamento interno do programa e permite identificar fluxos de execução que precisam ser testados. Com esta abordagem é possível obter uma melhor cobertura de testes, mas é necessário ter acesso ao código fonte do *software*.

Um dos *frameworks* que permite a escrita de vários tipos de teste, inclusive de integração, é o xUnit (.NET FOUNDATION, 2021a) que será abordado mais detalhadamente na próxima seção. A título de exemplo, na Figura 2 é apresentado um teste de integração desenvolvido em xUnit para testar o método de consulta por id da *controller* (controlador) de desenvolvedores que foi introduzido na Figura 1.

Figura 2: Exemplo de Teste de Integração em xUnit

```
21      [Fact]
22      public async void GetByIdAsync()
23      {
24          var developer = EntitiesFactory.NewDeveloper().Save();
25
26          var (status, result) = await Request.GetAsync<ResultTest<DeveloperDetailDto>>(
27              new Uri($"{Uri}/{developer.Id}")
28          );
29
30          var developerResult = result.Data;
31          Assert.Equal(Status.Success, status);
32          Assert.Equal(developer.Id, developerResult.Id);
33          Assert.Equal(developer.Name, developerResult.Name);
34          Assert.Equal(developer.Login, developerResult.Login);
35          Assert.Equal(developer.CPF, developerResult.CPF);
36      }
```

Fonte: Autoria Própria

Previamente à execução do método de teste, é gerada uma nova instância da API e o banco de dados é configurado simulando um ambiente real. O atributo Fact sinaliza ao xUnit que se trata de um método de teste. Na primeira linha do método (Linha 24), um novo registro é salvo na base de dados, na sequência é realizada uma chamada à API, por meio do método GetAsync (Linha 26), passando o id do registro. Por fim, o resultado da chamada é validado com assertivas do xUnit (.NET FOUNDATION, 2021a).

2.3 Framework xUnit

XUnit é uma ferramenta de código aberto desenvolvida em C# que pode ser usada para testar aplicativos da plataforma .NET. Foi escrito pelo inventor do NUnit v2 e é licenciado pela Apache 2. A ferramenta infere diversas configurações que no NUnit ou MSTest tinham que ser implementadas explicitamente, por exemplo a classe de teste é reconhecida automaticamente se houver algum método de teste (ADEWOLE, 2018).

XUnit suporta dois tipos de teste, Facts *e* Theories. Facts testam apenas um cenário sem parâmetros. Theories podem testar vários cenários, sendo essencialmente para testes parametrizados. Na Figura 3 é apresentado um exemplo de classe de teste unitário. Os atributos Fact e Theory são utilizados para indicar ao

framework que se tratam de métodos de teste. O atributo Theory deve vir acompanhado do atributo InlineData, que serve para especificar os parâmetros que serão passados para o método de teste (ADEWOLE, 2018).

Figura 3: Exemplo de Teste de Unidade em xUnit

```
 7  public class DeveloperTests : BaseTest
 8  {
 9      public DeveloperTests(TasksFixture fixture) : base(fixture) { }
10
11      [Fact]
12      public void HashingPasswordTest()
13      {
14          var password = "321654";
15          var expectedHash = "1VPRSEeaJokUzst3sriOag==";
16
17          var developer = EntitiesFactory.NewDeveloper(password: password).Get();
18
19          Assert.Equal(expectedHash, developer.PasswordHash);
20      }
21
22      [Theory]
23      [InlineData("123", "123")]
24      [InlineData("123", "679875")]
25      public void ValidatePasswordTest(string expected, string actual)
26      {
27          var developer = EntitiesFactory.NewDeveloper(password: expected).Get();
28          var expectedValid = expected.Equals(actual);
29
30          var valid = developer.ValidatePassword(actual);
31
32          Assert.Equal(expectedValid, valid);
33      }
34  }
```

Fonte: Autoria Própria

No primeiro método de teste (Linha 12), o objetivo é validar se a senha está sendo criptografada ao instanciar um objeto do tipo Developer passando a senha no construtor. O atributo InlineData especifica os parâmetros que serão recebidos no método de teste, é importante notar que no primeiro conjunto são passadas senhas equivalentes (Linha 23) e no segundo conjunto (Linha 24) são fornecidas senhas divergentes, isto será considerado na validação. Primeiro são declaradas variáveis que serão utilizadas no teste. Em seguida, é gerada uma nova instância do objeto, por meio do método NewDeveloper (Linha 17). Por fim, são utilizadas as assertivas do xUnit para verificar se a senha foi criptografada corretamente (.NET FOUNDATION, 2021a).

O segundo método (Linha 25), se trata de um teste parametrizado com o objetivo de testar a validação de senha. Nesse teste, são recebidos dois parâmetros passados pelo atributo InlineData, o primeiro com a senha esperada, e o segundo com a senha informada. O teste começa com uma chamada ao método

NewDeveloper, passando a senha esperada (Linha 27), na sequência é declarada uma variável auxiliar para identificar se o resultado deve ser positivo. Em seguida, o método ValidatePassword é chamado (Linha 30) e por fim o resultado é validado com as assertivas do xUnit (.NET FOUNDATION, 2021a).

Devido à experiência prévia do autor com esta ferramenta, e por suas características favoráveis, este foi o *framework* utilizado para realizar testes de integração.

2.4 Modelos T4

Uma tecnologia que facilita o processo de geração de código, que foi integrada com a TesTool são os Modelos T4. Trata-se de uma ferramenta que permite combinar estruturas de controle com blocos de texto para gerar uma saída de texto processada. A lógica de controle é escrita em fragmentos utilizando a linguagem do programa, em C# ou Visual Basic. O arquivo de saída pode ser um texto de qualquer tipo, como uma página Web, um arquivo de recurso ou código fonte em qualquer linguagem. Existem dois tipos de modelos T4: tempo de execução e tempo de *design* (MICROSOFT, 2021e):

- **Tempo de execução:** também conhecido como modelo pré-processado, é executado no aplicativo para produzir cadeias de caracteres de texto. Este tipo exige a utilização da IDE Visual Studio para que o template seja traduzido para a linguagem utilizada. O processador *TextTemplatingFilePreprocessor* é normalmente empregado para esta finalidade;
- **Tempo de design:** define parte do código fonte e outros recursos do aplicativo. Geralmente vários modelos leem os dados de um único arquivo ou banco de dados de entrada. Por exemplo, a entrada pode ser um arquivo de configuração XML, sempre que houver uma edição deste arquivo os modelos de texto atualizam parte do código de saída.

No template T4, a lógica de controle deve estar envolvida entre <# … #>. O operador de igualdade poderá ser combinado <#= … #> para indicar a renderização de um texto dinâmico. O operador de adição poderá ser acrescentado <#+ … #> para indicar a declaração de propriedades de entrada do template. Na Figura 4 é apresentado um exemplo de template de código T4. Na primeira linha é definida a

linguagem de controle utilizada. Nas linhas 2 e 3 são declarados os *namespaces* necessários (MICROSOFT, 2021e).

Figura 4: Exemplo de Template de Código T4

```
1  <#@ template language="C#" #>
2  <#@ import namespace="System.Text" #>
3  <#@ import namespace="TesTool.Core.Models.Templates.Comparator" #>
4  <#
5      foreach (var @namespace in Namespaces)
6      {
7  #>
8  using <#= @namespace #>;
9  <#
10     }
11 #>
12
13 namespace <#= ComparatorNamespace #>
14 {
15     public class <#= ComparatorClassName #>
16     {
17 <#
18     if (SourceClassName != TargetClassName)
19     {
20 #>
21         public void Equals(<#= TargetClassName #> source, <#= SourceClassName #> target) => Equals(target, source);
22 <#
23     }
24 #>
25         public void Equals(<#= SourceClassName #> source, <#= TargetClassName #> target)
26         {
27             AssertExtensions.AreEqualObjects(source, target);
28         }
29     }
30 }
31 <#+
32     public string ComparatorNamespace { get; set; }
33     public string ComparatorClassName { get; set; }
34     public string SourceClassName { get; set; }
35     public string TargetClassName { get; set; }
36     public string[] Namespaces { get; set; }
37 #>
```

Fonte: Autoria Própria

Entre as linhas 4 e 11 existe uma estrutura de repetição que itera os namespaces definidos na linha 36, renderizando cada um na linha 8. Na linha 13 é declarado o namespace do arquivo. O nome da classe é renderizado na linha 15. Entre as linhas 17 e 24, existe uma verificação de segurança, para evitar a duplicação dos métodos, nas linhas 21 e 25. Em cada método, são especificados os tipos dos parâmetros de forma dinâmica, por meio das propriedades (MICROSOFT, 2021e).

2.5 Trabalhos Relacionados

Existem estudos na área que propõem algoritmos e ferramentas de geração de código. Em FRASER et al. (2011), é apresentada uma solução para geração de testes unitários parametrizados, em que são produzidas pré e pós condições para um código existente. Os autores fizeram uma avaliação da abordagem por meio de um protótipo desenvolvido para Java. Foram utilizadas 5 bibliotecas para passarem pela ferramenta, e tiveram um aumento significativo na quantidade de código testado. Por outro lado, foi necessário um tempo considerável para o processamento dos resultados e o código gerado tinha baixa legibilidade.

No trabalho de ARCURI (2020) é realizado um estudo comparativo do uso de teste de caixa branca e caixa preta utilizando a ferramenta EvoMaster para gerar testes JUnit para APIs RESTful. O autor utilizou a ferramenta em oito APIs, executando 10 vezes cada uma, classificando em teste de caixa branca e caixa preta. Os resultados apontaram que houve um aumento importante na cobertura de código quando utilizado o teste de caixa branca. ARCURI também destaca que o teste de caixa branca teve um custo maior, pois foi necessário mais esforço de desenvolvimento comparado à abordagem de caixa preta. Mas, para realizar o teste de caixa branca, a linguagem utilizada na API precisa ser suportada pela ferramenta.

WIEDERSEINER et al. (2010) realizaram um estudo de caso para propor uma ferramenta de geração de código de teste para um *software* industrial de grande escala. Os autores consideraram utilizar ferramentas de geração de código de caixa preta como TestGen4J e NModel, mas concluíram que não atendiam às suas necessidades, porque precisavam de uma ferramenta com interface gráfica e que permitisse realização de teste em pares em NUnit. Propuseram então, o desenvolvimento da ferramenta AutoBBUT para contemplar estes requisitos. Em várias telas é possível delimitar os parâmetros a serem gerados na entrada e definir validações a serem aplicadas na saída. Desta forma a AutoBBUT foi eficiente na geração de testes automatizados em NUnit para o *software* industrial.

A ferramenta Microsoft Pex foi analisada por HONFI et al. (2020) para gerar código de teste de caixa branca em projetos *open source.* Para o estudo, foram selecionados aleatoriamente 10 projetos populares disponíveis no GitHub escritos em C#. Na análise dos autores, foram levadas em consideração algumas métricas, por exemplo: a complexidade ciclomática do código dos projetos; a cobertura de código resultante dos testes; etc. No entanto, os autores não discutiram sobre os

resultados alcançados, se concentrando em apresentar as informações coletadas. Os gráficos evidenciam que a maioria dos métodos tem cobertura próxima a 0% e a 100%, também indicam muitos métodos não testados pela ferramenta.

O Postman é uma ferramenta comercial amplamente conhecida que funciona como cliente de API podendo fazer consultas em REST, SOAP e GraphQL. Com ela, é possível criar scripts de teste de caixa preta e simular requisições. Postman tem uma série de funcionalidades: uma delas permite gerar rotas automaticamente a partir da documentação da API o que agiliza o trabalho de teste (POSTMAN, 2021). Entretanto, não é possível fazer a análise de cobertura de código e realizar teste baseado em caixa cinza (combinação de caixa preta e branca) ou branca, porque a ferramenta atua de forma independente dos projetos.

2.6 Considerações Finais

Neste capítulo foram tratados e explicados os elementos chave necessários para entendimento do problema que este trabalho aborda. Também foram elencadas ferramentas que procuram resolver os problemas relacionados a este trabalho. A solução proposta pelos autores FRASER et al. (2011) e ARCURI (2020) permite a geração de código, mas fica limitado à linguagem Java.

As ferramentas EvoMaster e AutoBBUT propostas por ARCURI (2020) e WIEDERSEINER et al. (2010) respectivamente, produzem testes para os *frameworks* JUnit e NUnit, mas não são capazes de gerar testes em XUnit. A ferramenta Postman é amplamente utilizada para testar APIs Web, mas os testes produzidos precisam ser executados em uma sequência pré definida, o que pode gerar falsos positivos.

O aumento no número de serviços disponibilizados na Web e a constante demanda por *software* de qualidade reforça a importância da utilização de ferramentas de teste automatizado, principalmente nas APIs. No Capítulo 3 será apresentada a ferramenta desenvolvida neste trabalho.

3. DESENVOLVIMENTO DO PROJETO

Este capítulo irá apresentar detalhes da implementação da ferramenta. A Seção 3.1 mostra os recursos considerados no desenvolvimento do projeto. Na Seção 3.2 é explicada a arquitetura empregada, por meio de imagens com um diagrama e os comandos suportados. Por fim, a Seção 3.3 apresenta um exemplo aplicado a um projeto *open source*, explicando a estrutura de pasta, arquivos e códigos produzidos.

3.1 Recursos Utilizados

Nesta seção, serão descritas as tecnologias e ferramentas que foram utilizadas para o desenvolvimento da ferramenta TesTool.

C#: (pronuncia-se "See Sharp") é uma linguagem de programação moderna, orientada a objetos fortemente tipada permitindo que os desenvolvedores criem muitos tipos de aplicativos seguros e robustos que são executados na plataforma .NET (MICROSOFT, 2021b).

NET Core: é um *framework* de código aberto provido pela Microsoft e pela comunidade *open source* que foi uma evolução do .NET, oferecendo suporte para os sistemas operacionais Windows, Linux e Mac (MICROSOFT, 2021c). NET Core é a estrutura base para programação da ferramenta em C#.

Reflection: com reflection é possível criar dinamicamente instâncias de objetos em tempo de execução, invocar seus métodos ou acessar suas propriedades e campos. Também é possível identificar atributos no código usando C# (MICROSOFT, 2021d).

Modelos T4: é um recurso de geração de código que utiliza um *template* como entrada *e* pode ter alguma lógica de controle. O *template* geralmente é utilizado para gerar arquivos de configuração e consulta à banco de dados (MICROSOFT, 2021e). Mais detalhes na Seção 2.4.

XUnit: é um *framework* de código aberto desenvolvido em C# para testar aplicativos da plataforma .NET. Foi escrito pelo inventor do NUnit v2 e é licenciado pela Apache 2 (.NET FOUNDATION, 2021b). Os códigos gerados pela ferramenta TesTool utilizam este *framework* de teste. Mais informações na Seção 2.3.

Roslyn: é uma plataforma de compilador para .NET composto por diversas APIs para análise e gerenciamento de código (GITHUB, 2021a). Este componente de *software* foi utilizado para realizar a análise do sistema em teste e também para realizar a manutenção do código de teste.

EntityFramework: é um ORM (mapeador de objeto relacional ou do inglês *object relational mapper*), que oferece suporte para consultas LINQ (MICROSOFT, 2021f), controle de alterações, atualizações e migrações de esquema desenvolvido para a plataforma .NET (MICROSOFT, 2021g). Por ser amplamente utilizado em conjunto com aplicações Web, este *framework* será integrado com o projeto de teste, para oferecer controle sobre o estado do banco de dados.

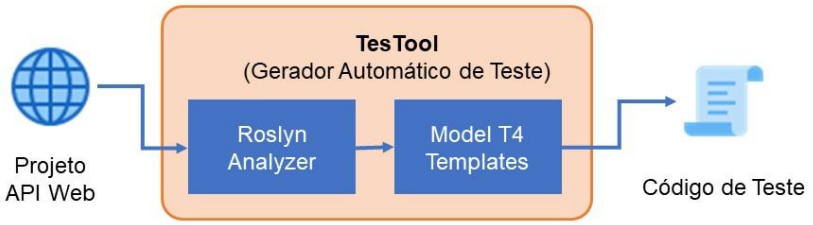
Bogus: é um *framework* de fabricação de objetos fictícios em tempo de execução (GITHUB, 2021b). Foi utilizado para gerar as entradas necessárias nos testes da API Web.

Visual Studio: é um ambiente de desenvolvimento integrado ou *Integrated Development Environment* (IDE) que pode ser usado para editar, depurar, compilar e publicar código com uma série de recursos para desenvolvimento de *software* (MICROSOFT, 2021h). Devido à experiência com o uso da IDE, foi o editor de código utilizado para o desenvolvimento da ferramenta TesTool.

3.2 Arquitetura

O fluxo básico de execução da ferramenta se inicia com um projeto API Web que deve ser fornecido como entrada. Primeiro é realizada uma análise do código fonte por meio do compilador Roslyn, que identifica classes controladoras que servirão de entrada para o próximo estágio. Em seguida, um conjunto de templates T4 são utilizados, em combinação com os dados coletados na fase anterior, para produzir o código final. Na Figura 5 é possível ver esse processo na forma de diagrama.

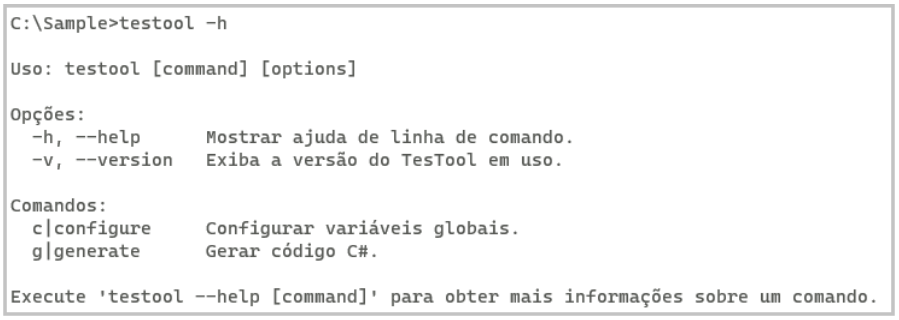
Figura 5: Representação da Arquitetura



Fonte: Autoria Própria

A interação com o usuário acontece por meio de linha de comando ou *command line interface* (CLI) que possui uma documentação básica integrada. As rotinas mais comuns foram otimizadas com comandos complementares para permitir a configuração por escopo de trabalho. Na Figura 6 é possível identificar a relação de comandos primários suportados; para alguns será necessário passar parâmetros extras que serão apresentados nas demais figuras.

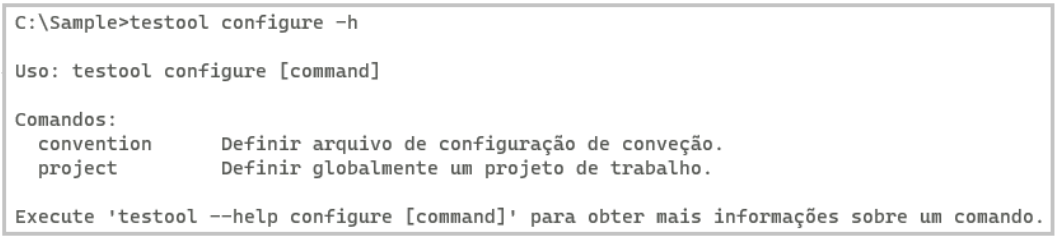
Figura 6: Comandos Primários do TesTool



Fonte: Autoria Própria

Na Figura 7 é possível ver os sub-comandos que podem ser usados com o comando *configure.* Estes comandos têm como objetivo configurar de forma global a localização de alguns artefatos necessários para o uso da ferramenta, de modo a não precisar passar nas outras operações esses mesmos parâmetros, tornando o uso da ferramenta mais simples.

Figura 7: Sub-comandos de Configuração



Fonte: Autoria Própria

Um arquivo de configuração de convenção poderá ser criado e vinculado à ferramenta por meio do comando *convention* acompanhado do diretório do arquivo apresentado na Figura 8. Também é possível gerar automaticamente esse arquivo por meio da *flag* (sinalizador) --*init* que irá criar um arquivo com convenções padrões implementadas. Este arquivo poderá definir regras para geração de dados ao produzir instâncias de objetos para testes. Por exemplo, poderá ser configurada a geração de email sempre que uma propriedade contiver “email” em seu nome, o mesmo vale para telefone, nome de pessoas, etc.

Figura 8: Comandos de Configuração de Convenção

```
C:\Sample>testool configure convention -h

Uso: testool configure convention [arguments] [options]

Argumentos:
  <CONFIGURATION_PATH>      Diretório do arquivo de configuração.

Opções:
  -h, --help      Mostrar ajuda de linha de comando.
  -i, --init      Gera arquivo de configuração padrão.
```

Fonte: Autoria Própria

Outro sub-comando de configuração é o *project,* que permite configurar um projeto de trabalho, exige como parâmetro o diretório do projeto de API Web que é mostrado na Figura 9. Este valor será condicionado pelo diretório corrente. Por exemplo, se o diretório atual for nos testes de um projeto A, então o diretório do projeto A será utilizado, para testes em um projeto B, será considerado o diretório da Web API B. Outros comandos definem automaticamente este valor, ele será útil para configurar a ferramenta em uma nova instalação.

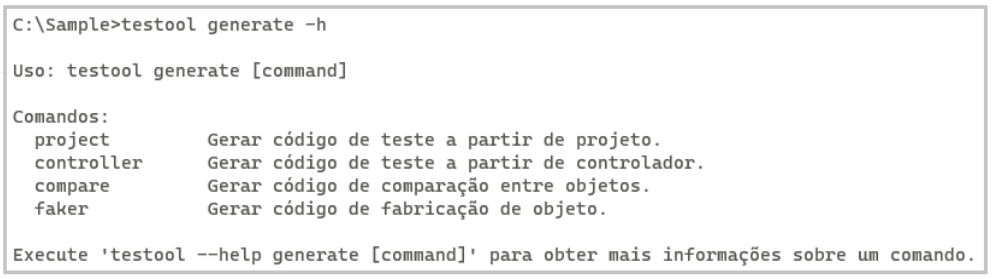
Figura 9: Comandos de Configuração de Projeto

```
C:\Sample>testool configure project -h

Uso: testool configure project [arguments] [options]

Argumentos:
  <PROJECT_PATH>  Diretório do projeto.

Opções:
  -h, --help      Mostrar ajuda de linha de comando.
```

Fonte: Autoria Própria

Na Figura 10 é possível ver os sub-comandos que são suportados pelo comando *generate,* responsáveis pela geração de código de teste. O sub-comando *project* deve ter prioridade de execução, pois irá criar a estrutura básica do projeto de teste, que será considerada pelos demais sub-comandos. Todos esses sub-comandos têm uma hierarquia de chamadas, o sub-comando *project* irá invocar logicamente o comando *controller* para cada controlador identificado no projeto API Web, e este por sua vez irá invocar os comandos *compare* e *faker* dependendo do contexto.

Figura 10: Sub-comandos de Geração de Código



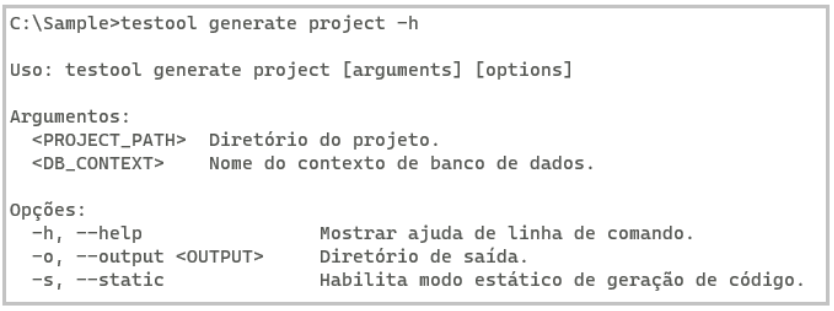
```
C:\Sample>testool generate -h

Uso: testool generate [command]

Comandos:
  project         Gerar código de teste a partir de projeto.
  controller      Gerar código de teste a partir de controlador.
  compare         Gerar código de comparação entre objetos.
  faker           Gerar código de fabricação de objeto.

Execute 'testool --help generate [command]' para obter mais informações sobre um comando.
```

Fonte: Autoria Própria

Para gerar um projeto de teste de integração para uma API Web, o sub-comando *project* pode ser utilizado. Na Figura 11 é possível identificar os parâmetros necessários, como diretório do projeto e nome do contexto de banco de dados. A ferramenta irá tentar inferir esses valores, caso não tenham sido informados. Se isto não for possível, será apresentada uma mensagem de erro.

Para o sub-comando *project* pode ser especificado um diretório de saída por meio da *flag* --*output,* caso contrário, o projeto de teste será gerado no diretório atual. Além disso, é possível adicionar a *flag* --*static* para produzir valores estáticos para os objetos de interação com a API Web, de forma que não sejam produzidos de forma aleatória em tempo de execução.

Figura 11: Geração de Projeto de Teste



```
C:\Sample>testool generate project -h

Uso: testool generate project [arguments] [options]

Argumentos:
  <PROJECT_PATH>  Diretório do projeto.
  <DB_CONTEXT>    Nome do contexto de banco de dados.

Opções:
  -h, --help                Mostrar ajuda de linha de comando.
  -o, --output <OUTPUT>     Diretório de saída.
  -s, --static              Habilita modo estático de geração de código.
```

Fonte: Autoria Própria

Outro sub-comando de geração de código é o *controller* que é responsável por criar uma classe de teste para o controlador informado. Este sub-comando irá criar um método de teste para cada *endpoint* identificado na análise do código e também irá gerar arquivos complementares de comparação e fabricação de objetos, que serão necessários para a realização do teste. Deve ser informado de forma obrigatória o nome do controlador, e opcionalmente a entidade relacionada. As *flags* --*output* e --*static* explicadas no sub-comando anterior também podem ser utilizadas.

Figura 12: Geração de Teste para Controlador

```
C:\Sample>testool generate controller -h

Uso: testool generate controller [arguments] [options]

Argumentos:
  <CONTROLLER>    Nome da classe controlador.
  <ENTITY>        Nome da classe entidade.

Opções:
  -h, --help                Mostrar ajuda de linha de comando.
  -o, --output <OUTPUT>     Diretório de saída.
  -s, --static              Habilita modo estático de geração de código.
```

Fonte: Autoria Própria

Existe um sub-comando para realizar a comparação entre objetos correlacionados. É necessário informar o nome das classes a serem comparadas, caso elas não existam, uma mensagem de erro será mostrada, o mesmo acontece se as classes não tiverem nenhuma propriedade similar. É possível alterar o diretório de saída e definir o modo estático de geração de código, como pode ser visto na Figura 13. Para este sub-comando, o modo estático produzirá assertivas de comparação entre cada propriedade, caso contrário, isto será realizado em tempo de execução por meio de *reflection* que é a configuração padrão.

Figura 13: Geração de Código de Comparação

```
C:\Sample>testool generate compare -h

Uso: testool generate compare [arguments] [options]

Argumentos:
  <SOURCE_CLASS_NAME>  Nome da classe de origem.
  <TARGET_CLASS_NAME>  Nome da classe de destino.

Opções:
  -h, --help                Mostrar ajuda de linha de comando.
  -o, --output <OUTPUT>     Diretório de saída.
  -s, --static              Habilita modo estático de geração de código.
```

Fonte: Autoria Própria

O sub-comando *faker* possui derivações como *entity* e *model,* para gerar instâncias de entidades de banco de dados, e objetos de transporte de dados, necessários na interação com a API Web. A Figura 14 apresenta o menu de ajuda para o sub-comando faker com a relação de sub-comandos suportados.

Figura 14: Sub-comandos de Fabricação de Objetos

```
C:\Sample>testool generate faker -h

Uso: testool generate faker [command]

Comandos:
  entity     Gerar código de fabricação de entidade de banco de dados.
  model      Gerar código de fabricação de modelo de transporte de dados (DTO).

Execute 'testool --help generate faker [command]' para obter mais informações sobre um comando.
```

Fonte: Autoria Própria

Os sub-comandos de *faker* utilizam a biblioteca Bogus para definir os valores para cada propriedade dos objetos, de forma dinâmica. A diferença entre os sub-comandos está no código gerado, para *entity* haverá herança de uma classe base com funcionalidades específicas de persistência, que para *model* não é necessário. A Figura 15 apresenta a documentação dos dois sub-comandos.

Figura 15: Geração de Código de Fabricação de Objetos

```
C:\Sample>testool generate faker entity -h

Uso: testool generate faker entity [arguments] [options]

Argumentos:
  <CLASS_NAME>    Nome da classe a ser fábricada.

Opções:
  -h, --help                Mostrar ajuda de linha de comando.
  -o, --output <OUTPUT>     Diretório de saída.
  -s, --static              Habilita modo estático de geração de código.
```

```
C:\Sample>testool generate faker model -h

Uso: testool generate faker model [arguments] [options]

Argumentos:
  <CLASS_NAME>    Nome da classe a ser fábricada.

Opções:
  -h, --help                Mostrar ajuda de linha de comando.
  -o, --output <OUTPUT>     Diretório de saída.
  -s, --static              Habilita modo estático de geração de código.
```

Fonte: Autoria Própria

O projeto da ferramenta TesTool está disponível como um projeto *open source* no github em: https://github.com/paulobusch/testool.

3.3 Exemplo de Uso da Ferramenta

Para demonstrar o uso da ferramenta, será utilizado um projeto *open source* desenvolvido pelo autor deste trabalho disponível em: https://github.com/paulobusch/netcore-tasks-api. A aplicação consiste em um sistema de gerenciamento de tarefas, com *endpoints* básicos, para autenticação, gerenciamento de projetos, usuários e tarefas. A ferramenta TesTool está disponível publicamente na plataforma NuGet (MICROSOFT, 2021i) e pode ser instalada localmente por meio do comando *dotnet tool install global dotnet-testool-globaltool*. Para gerar um projeto de teste de integração, deverá ser utilizado o comando *testool generate project* informando o diretório da API, como ilustrado na Figura 16.

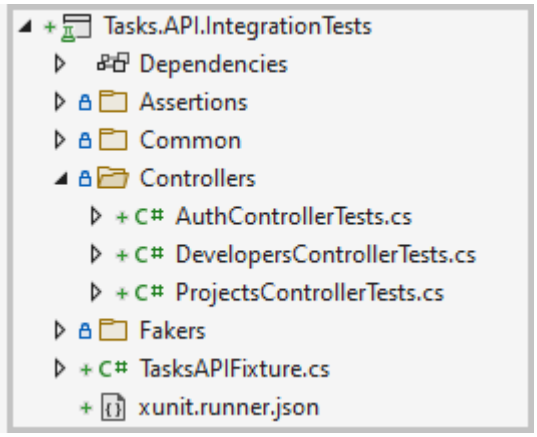
Figura 16: Comando para Gerar Projeto de Teste de Integração

```
C:\Sample\netcore-tasks-api>testool generate project "Tasks.API"
Escaneando projeto.
Criando projeto Tasks.API.IntegrationTests.
Instalando pacotes.
Gerando arquivos.
Código gerado com sucesso. Realize uma revisão antes de comitar.
```

Fonte: Autoria Própria

Na Figura 17, é apresentada a estrutura de pastas e arquivos que foi gerada pela ferramenta. Na pasta Controller, as classes de teste foram criadas, uma para cada controlador; o conteúdo desses arquivos será explicado detalhadamente. O arquivo xunit.runner.json possui configurações do XUnit; nele é desativada a execução paralela dos testes para evitar conflito no acesso a recursos, como banco de dados.

Figura 17: Estrutura de Pastas

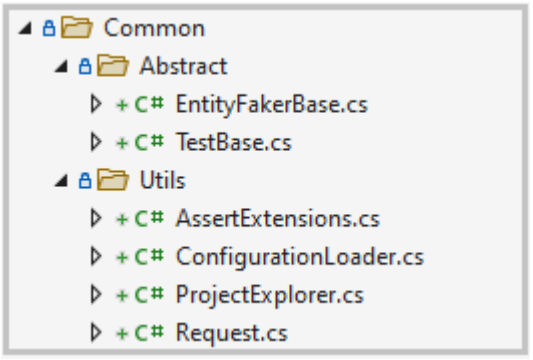


Fonte: Autoria Própria

A classe TasksAPIFixture configura a inicialização da API em modo teste, e configura os serviços necessários. Esta classe será injetada como dependência em todas as classes de teste, para fornecer um cliente de consulta HTTP, assim como o contexto de banco de dados e fábricas.

Na Figura 18 é apresentado o conteúdo da pasta Common, que possui arquivos com classes auxiliares e utilizadas pelo projeto de teste. A classe TestBase recebe no construtor uma instância de TasksAPIFixture e também a URL padrão que será utilizada pela classe de teste do controlador. Esta classe replica as propriedades de TasksAPIFixture para facilitar o uso. Se o projeto de API necessitar de autenticação, terá um método de *login* e *logout*.

Figura 18: Pasta Common



Fonte: Autoria Própria

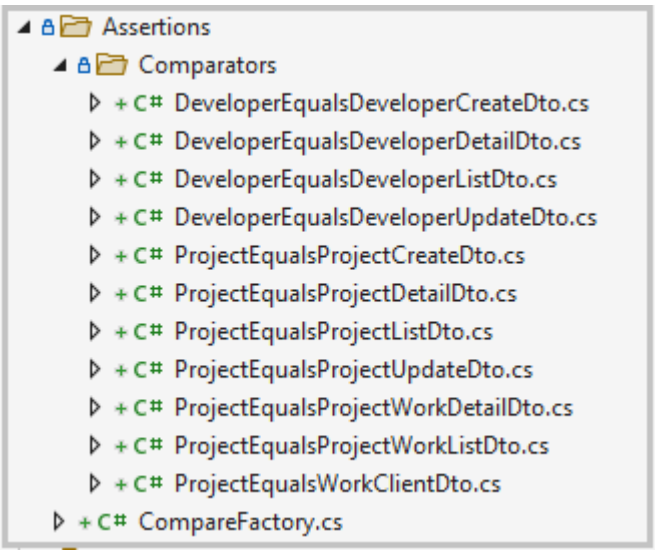
Todas as classes de fabricação de entidade herdam de EntityFakerBase, que possui métodos para persistência no banco de dados. Esta classe recebe no construtor uma instância do contexto de banco de dados. A classe AssertExtensions possui métodos de comparação entre objetos por meio de *reflection,* e é utilizada nas classes de comparação que serão explicadas posteriormente.

A classe ConfigurationLoader é responsável por carregar as configurações utilizadas pela API, na classe TasksAPIFixture; ela é utilizada para carregar o arquivo *appsettings.Test.json*. A classe ProjectExplorer é utilizada em conjunto para localizar o diretório do projeto da API Web onde deve estar localizado o arquivo de configuração. Se este arquivo não existir, então será lançada uma exceção, pois o teste exige uma configuração específica.

A classe Request encapsula um cliente de consulta HTTP, e implementa métodos para requisições REST, para facilitar a comunicação com a API.

A pasta Assertions concentra arquivos de comparação entre classes. A classe CompareFactory possui um método para cada classe de comparação, funcionando como uma fábrica para simplificar o uso nos métodos de teste.

Figura 19: Pasta Assertions



Fonte: Autoria Própria

A Figura 20 apresenta um exemplo de classe de comparação produzida; nela existem dois métodos Equals, para permitir uma comparação bidirecional. Um dos métodos utiliza a classe AssertExtensions (Linha 12) para realizar a comparação entre os objetos em tempo de execução; se a *flag --static* fosse utilizada, haveria uma assertiva de comparação, para cada propriedade equivalente entre os objetos.

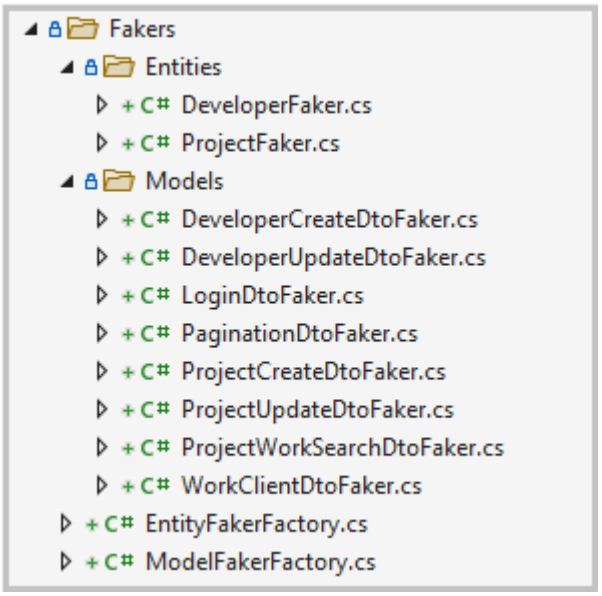
Figura 20: Exemplo de Classe de Comparação

```
5   namespace Tasks.API.IntegrationTests.Assertions.Comparators
6   {
7       public class DeveloperEqualsDeveloperCreateDto
8       {
9           public void Equals(DeveloperCreateDto source, Developer target) => Equals(target, source);
10          public void Equals(Developer source, DeveloperCreateDto target)
11          {
12              AssertExtensions.AreEqualObjects(source, target);
13          }
14      }
15  }
```

Fonte: Autoria Própria

A pasta Fakers contém os fakers de entidades e modelos de transferência de dados. Existem duas classes de fábrica, uma para cada tipo, sendo: EntityFakerFactory e ModelFakerFactory.

Figura 21: Pasta Fakers



Fonte: Autoria Própria

A Figura 22 apresenta um exemplo de classe de configuração de objeto de banco de dados, para fabricação, utilizando o *framework* Bogus. A classe herda de EntityFakerBase, que possui métodos pré definidos de persistência. No construtor são definidas regras para geração de valores para cada propriedade. Na linha 11, a propriedade Id é configurada, para que seu valor seja gerado aleatoriamente. Na linha 12 a propriedade Title é configurada, de modo que seu valor seja produzido com base na API Lorem, o mesmo acontece para a propriedade Description.

Figura 22: Exemplo de Classe Entity Faker

```
5   namespace Tasks.API.IntegrationTests.Fakers.Entities
6   {
7       public class ProjectFaker : EntityFakerBase<Project>
8       {
9           public ProjectFaker(TasksContext context) : base(context)
10          {
11              RuleFor(p => p.Id, f => f.Random.Guid())
12                  .RuleFor(p => p.Title, f => f.Lorem.Word())
13                  .RuleFor(p => p.Description, f => f.Lorem.Paragraph());
14          }
15      }
16  }
```

Fonte: Autoria Própria

Para as classes de configuração, como a mostrada na Figura 22, o uso da *flag --static* resultaria na atribuição explícita do valor gerado no lugar da expressão lambda, o processamento deste valor acontece no momento em que o arquivo é

gerado. A Figura 23 apresenta uma classe de configuração para modelos de transferência de dados com as mesmas configurações que no exemplo anterior, exceto pela herança de EntityFakerBase, que é substituída diretamente por Faker.

Figura 23: Exemplo de Classe Model Faker

```
4   namespace Tasks.API.IntegrationTests.Fakers.Models
5   {
6       public class ProjectCreateDtoFaker : Faker<ProjectCreateDto>
7       {
8           public ProjectCreateDtoFaker()
9           {
10              RuleFor(p => p.Id, f => f.Random.Guid())
11                  .RuleFor(p => p.Title, f => f.Lorem.Word())
12                  .RuleFor(p => p.Description, f => f.Lorem.Paragraph());
13          }
14      }
15  }
```

Fonte: Autoria Própria

Como foi mostrado na Figura 17 , os testes são gerados na pasta Controllers. Na Figura 24 é apresentada a classe DevelopersControllerTests, que realiza testes de integração para a controller DevelopersController, que foi introduzida na Figura 1. Esta classe herda da classe TestBase, que exige alguns parâmetros no construtor, como a URL referente ao controlador em teste que é passada explicitamente. Foram produzidos métodos de teste para cada *endpoint* identificado no controlador, seguindo um padrão de nomenclatura, de acordo com o método HTTP utilizado.

Figura 24: Exemplo de Classe de Teste

```
12  namespace Tasks.API.IntegrationTests.Controllers
13  {
14      public class DevelopersControllerTests : TestBase
15      {
16          public DevelopersControllerTests(TasksAPIFixture fixture) : base(fixture, "api/developers") { }
17
18          [Fact]
19          public async Task ShouldReturnManyAsync()...
33
34          [Fact]
35          public async Task ShouldReturnByIdAsync()...
46
47          [Fact]
48          public async Task ShouldCreateAsync()...
60
61          [Fact]
62          public async Task ShouldUpdateAsync()...
75
76          [Fact]
77          public async Task ShouldDeleteAsync()...
88      }
89  }
```

Fonte: Autoria Própria

Na Figura 25 é apresentado o método responsável por testar e validar a consulta de desenvolvedores. Na linha 21, utilizando a fábrica de entidades, um novo registro é gerado e salvo na base de dados. Na linha seguinte a fábrica de modelos é utilizada para gerar dados de paginação, que na linha 24 é passado como parâmetro para ser serializado como *query string* na URL de consulta. Nesta linha é

realizada a requisição à API, que devolve dois parâmetros, o primeiro referente ao status da requisição, e o segundo o resultado desserializado em forma de objeto genérico.

Figura 25: Método de Teste ShouldReturnManyAsync

```
18  [Fact]
19  public async Task ShouldReturnManyAsync()
20  {
21      var developer = EntityFactory.Developer().Save();
22      var paginationDtoRequest = ModelFactory.PaginationDto().Generate();
23
24      var (response, result) = await Request.GetAsync<Result<IEnumerable<DeveloperListDto>>>(Uri, paginationDtoRequest);
25
26      response.EnsureSuccessStatusCode();
27      var listResponse = result?.Data;
28      Assert.NotEqual(default, listResponse);
29      Assert.NotEmpty(listResponse);
30      var modelResponse = listResponse.Single(d => d.Id == developer.Id);
31      CompareFactory.DeveloperEqualsDeveloperListDto().Equals(developer, modelResponse);
32  }
```

Fonte: Autoria Própria

A partir da linha 26, ocorre a validação dos resultados da requisição. O método *EnsureSuccessStatusCode* verifica se houve sucesso na realização da consulta, na linha seguinte uma variável é declarada para armazenar a lista retornada. Nas linhas 28 e 29, são utilizadas assertivas do XUnit, para verificar a lista. Na linha 30, o registro salvo na base de dados é recuperado, e utilizado na próxima linha, para ser comparado com o registro persistido. Isto é realizado por meio da fábrica de classes de comparação, que na linha 31 gera uma instância capaz de comparar os objetos.

A Figura 26 apresenta o método de teste para consulta de um desenvolvedor pelo Id. Primeiro, um registro é salvo na base de dados, por meio da fábrica de entidades. Na linha 39, uma requisição é realizada na API, passando como parâmetro de URL, o id do registro persistido. Na linha 41 o status da requisição é validado. Na sequência uma variável é declarada para armazenar o registro retornado pela API, que na próxima linha é verificado por meio de assertivas do XUnit. Por fim, na linha 44, o registro obtido é comparado com o que foi salvo na base de dados, com o uso da fábrica de classes de comparação.

Figura 26: Método de Teste ShouldReturnByIdAsync

```
34  [Fact]
35  public async Task ShouldReturnByIdAsync()
36  {
37      var developer = EntityFactory.Developer().Save();
38
39      var (response, result) = await Request.GetAsync<Result<DeveloperDetailDto>>(new Uri($"{Uri}/{developer.Id}"));
40
41      response.EnsureSuccessStatusCode();
42      var modelResponse = result?.Data;
43      Assert.NotNull(modelResponse);
44      CompareFactory.DeveloperEqualsDeveloperDetailDto().Equals(developer, modelResponse);
45  }
```

Fonte: Autoria Própria

O controlador possui um método para cadastro de desenvolvedor, desta forma foi produzido um método de teste para verificar se o registro está sendo salvo corretamente. Na Figura 27 é apresentado este método, que inicia com a criação de uma instância do objeto que precisa ser passado no corpo da requisição, isto é feito automaticamente pela classe Request na linha 52. Nas linhas 54 e 55, o resultado da requisição é validado. Na linha 57, o registro que foi persistido durante a requisição é recuperado pelo Id informado. Na última linha, os objetos são comparados para verificar se os valores correspondem aos informados previamente.

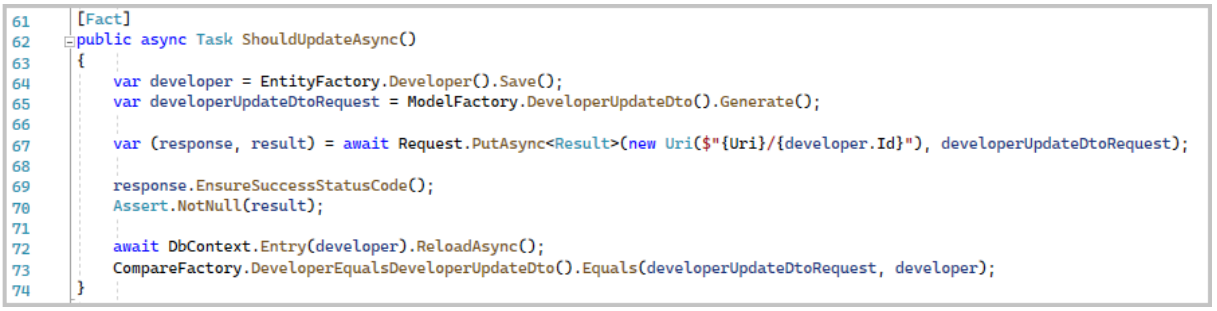
Figura 27: Método de Teste ShouldCreateAsync

```
47  [Fact]
48  public async Task ShouldCreateAsync()
49  {
50      var developerCreateDtoRequest = ModelFactory.DeveloperCreateDto().Generate();
51
52      var (response, result) = await Request.PostAsync<Result>(Uri, developerCreateDtoRequest);
53
54      response.EnsureSuccessStatusCode();
55      Assert.NotNull(result);
56
57      var developer = await DbContext.Developers.FindAsync(developerCreateDtoRequest.Id);
58      CompareFactory.DeveloperEqualsDeveloperCreateDto().Equals(developerCreateDtoRequest, developer);
59  }
```

Fonte: Autoria Própria

Na Figura 28 é apresentado o método de teste para atualização de desenvolvedor. Na linha 64 um novo registro é persistido na base de dados, em seguida, um objeto necessário para requisição é gerado. A requisição é realizada na linha 67, em que é passado como parâmetro de URL o Id do desenvolvedor e no corpo da requisição é enviado o objeto com os novos dados a serem salvos. As linhas 69 e 70 concentram a validação do resultado da requisição. Na linha 72, o registro persistido previamente, é atualizado, para que na linha seguinte a comparação possa ser realizada, com as novas informações.

Figura 28: Método de Teste ShouldUpdateAsync

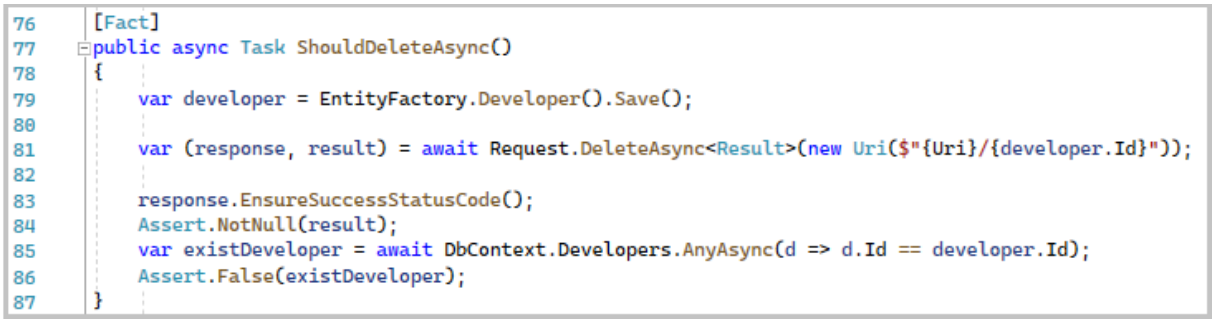


```
61  [Fact]
62  public async Task ShouldUpdateAsync()
63  {
64      var developer = EntityFactory.Developer().Save();
65      var developerUpdateDtoRequest = ModelFactory.DeveloperUpdateDto().Generate();
66
67      var (response, result) = await Request.PutAsync<Result>(new Uri($"{Uri}/{developer.Id}"), developerUpdateDtoRequest);
68
69      response.EnsureSuccessStatusCode();
70      Assert.NotNull(result);
71
72      await DbContext.Entry(developer).ReloadAsync();
73      CompareFactory.DeveloperEqualsDeveloperUpdateDto().Equals(developerUpdateDtoRequest, developer);
74  }
```

Fonte: Autoria Própria

O último método gerado é responsável por testar a remoção de um desenvolvedor. O método inicia com um registro sendo persistido na base de dados. Em seguida, uma requisição HTTP é realizada, utilizando o verbo DELETE, passando como parâmetro de URL o Id do registro. Nas linhas 83 e 84 a resposta da requisição é válida. Na linha 85, uma consulta é realizada no banco de dados, para verificar se o registro foi realmente removido, na linha seguinte uma assertiva do XUnit é utilizada para confirmar isso.

Figura 29: Método de Teste ShouldDeleteAsync



```
76  [Fact]
77  public async Task ShouldDeleteAsync()
78  {
79      var developer = EntityFactory.Developer().Save();
80
81      var (response, result) = await Request.DeleteAsync<Result>(new Uri($"{Uri}/{developer.Id}"));
82
83      response.EnsureSuccessStatusCode();
84      Assert.NotNull(result);
85      var existDeveloper = await DbContext.Developers.AnyAsync(d => d.Id == developer.Id);
86      Assert.False(existDeveloper);
87  }
```

Fonte: Autoria Própria

Vale destacar que durante o processo de geração de código, podem ser identificadas dependências que não podem ser resolvidas pela ferramenta, como por exemplo, o processo de autenticação utilizado e parâmetros de URL desconhecidos. Para estes casos, o código é comentado com o prefixo TODO acompanhado de uma observação, para que na revisão isso seja tratado. Neste exemplo, e eventualmente em outros projetos serão necessários ajustes específicos nas classes geradas, e no controle de persistência, para que os testes sejam executados corretamente.

Na Figura 30 é apresentada a classe TestBase, que precisou ser ajustada, para realizar o processo de autenticação adequadamente. Para esta API, foi necessário obter uma instância do serviço de autenticação para gerar o token, e passar no cabeçalho das requisições, de forma padrão. O código poderá ser

adaptado dependendo da necessidade, por exemplo, poderá ser necessário definir papéis (Roles) específicos para testar cada controlador, desta forma, um parâmetro poderá ser adicionado no método para identificar isto.

Figura 30: Fluxo de Autenticação que Necessário

```
9   namespace Tasks.API.IntegrationTests.Common.Abstract
10  {
11      public class TestBase : IClassFixture<TasksAPIFixture>
12      {
13          public readonly Uri Uri;
14          public readonly Request Request;
15          public readonly TasksContext DbContext;
16
17          public readonly ModelFakerFactory ModelFactory;
18          public readonly EntityFakerFactory EntityFactory;
19          public readonly CompareFactory CompareFactory;
20
21          public TestBase(TasksAPIFixture fixture, string url)...
32          protected virtual void Login()
33          {
34              // TODO: add your auth flow
35              // Request.Client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer|Basic", token);
36          }
37
38          protected virtual void Logout()...
42      }
43  }
```

Fonte: Autoria Própria

Na Figura 31 é exibido um exemplo de método de teste para o controlador *ProjectsController* que possui um parâmetro de URL não resolvido. Neste cenário, a solução implementada foi salvar um novo registro na base de dados, referente ao parâmetro necessário.

Figura 31: Parâmetro de URL Faltante

```
121 [Fact]
122 public async Task ShouldUpdateWorkProjectAsync()
123 {
124     var project = EntityFactory.Project().Save();
125     var workClientDtoRequest = ModelFactory.WorkClientDto().Generate();
126
127     // TODO: Resolve unsafe route parameters
128     var (response, result) = await Request.PutAsync<Result>(new Uri($"{Uri}/{project.Id}/works/{{workId}}"), workClientDtoRequest);
129
130     response.EnsureSuccessStatusCode();
131     Assert.NotNull(result);
132
133     await DbContext.Entry(project).ReloadAsync();
134     CompareFactory.ProjectEqualsWorkClientDto().Equals(workClientDtoRequest, project);
135 }
```

Fonte: Autoria Própria

4 ESTUDO DE CASO

Este capítulo irá apresentar o estudo de caso realizado com a ferramenta TesTool. A Seção 4.1 explica os projetos utilizados na condução do estudo, bem como os resultados obtidos. Em seguida, a Seção 4.2 realiza uma discussão sobre o uso da ferramenta, abordando as características dos testes produzidos, otimizações que poderiam ser aplicadas, ganho em produtividade e limitações da ferramenta.

4.1 Condução do Estudo

Esta seção irá apresentar um estudo de caso realizado com três sistemas *open source* disponíveis publicamente no GitHub. O objetivo é avaliar a utilização da ferramenta TesTool em diferentes APIs Web. O estudo foi realizado em três projetos implementados pelo autor. Algumas informações relativas aos projetos podem ser vistas na Tabela 1.

Tabela 1: APIs Web Utilizadas no Estudo de Caso

API Web	Linhas de código	Controladores	Endpoints
Vehicle	3.030	7	36
Tasks	2.077	3	17
Shop	690	3	11

Fonte: Autoria Própria

A seguir, cada API Web será descrita em detalhes.

Vehicle: foi desenvolvido com o objetivo de ser uma plataforma de venda de veículos na *Internet.* Dentre as principais funcionalidades, estão o gerenciamento de veículos, anúncios, marcas e modelos de veículos. A API Web desse sistema foi implementada, utilizando a linguagem C# com banco de dados MySQL (ORACLE, 2021). O código fonte está disponível em: https://github.com/paulobusch/vehicle-api.

Para criar o projeto de teste de integração, foi utilizado o comando apresentado na Figura 32. Neste caso foi necessário especificar o contexto de banco de dados a ser utilizado, porque a aplicação possui mais de um.

Figura 32: Comando para Gerar Testes para o Projeto Vehicle

```
C:\Sample\vehicle-api>testool generate project "Vehicle.API" "VehicleMutationsDbContext"
Escaneando projeto.
Criando projeto Vehicle.API.IntegrationTests.
Instalando pacotes.
Gerando arquivos.
Código gerado com sucesso. Realize uma revisão antes de comitar.
```

Fonte: Autoria Própria

Após a conclusão do processo, foi necessário revisar o código gerado, para arrumar parâmetros e códigos não resolvidos pela ferramenta. Uma configuração de teste precisou ser definida no projeto de API Web para especificar um banco de dados a ser utilizado. Devido a construção do projeto, que utilizou scripts SQL escritos à mão, foi preciso desenvolver um script específico para execução dos testes. Também foi necessário configurar o fluxo de autenticação.

Depois de tudo revisado, foram requeridos mais alguns ajustes, como a alteração de parâmetros enviados à API, que continham chaves estrangeiras inválidas, também propriedades com informações incorretas, que passam por algum tipo de verificação na API. Durante este processo, também foram identificados defeitos na API, que foram corrigidos neste momento.

Tasks: se trata do mesmo projeto utilizado como exemplo na seção anterior, a aplicação possui funcionalidades para gerenciamento de tarefas, projetos e usuários. A linguagem de implementação utilizada na API foi o C# e o banco de dados também foi o MySQL (ORACLE, 2021). O comando para geração do projeto pode ser visto na Figura 33. O código fonte pode ser consultado em: https://github.com/paulobusch/netcore-tasks-api.

Figura 33: Comando para Gerar Testes para o Projeto Tasks

```
C:\Sample\netcore-tasks-api>testool generate project "Tasks.API"
Escaneando projeto.
Criando projeto Tasks.API.IntegrationTests.
Instalando pacotes.
Gerando arquivos.
Código gerado com sucesso. Realize uma revisão antes de comitar.
```

Fonte: Autoria Própria

Depois da execução do comando, foi preciso revisar o código gerado e também configurar a conexão com um banco de dados de teste e o fluxo de autenticação. Neste caso não foi necessário criar script de banco de dados, porque o recurso de migração do *entity framework* foi utilizado. Da mesma forma que o projeto anterior, foram necessárias correções pontuais em alguns testes.

Shop: é uma aplicação bem simples para controle de estoque para uma loja *online*. O projeto possui funcionalidades para manutenção de clientes e produtos e tem as mesmas características que os anteriores, também implementado com C# e utilizando o MySQL (ORACLE, 2021) como banco de dados. Para gerar o projeto de teste, o mesmo comando foi utilizado, exceto por um parâmetro extra que foi especificado, neste caso o --output, para indicar a pasta de saída. O projeto está disponível em: https://github.com/paulobusch/domain-driven-design-api.

Figura 34: Comando para Gerar Testes para o Projeto Shop

```
C:\Sample\domain-driven-design-api>testool generate project "src\DDD.API" --output "tests"
Escaneando projeto.
Criando projeto DDD.API.IntegrationTests.
Instalando pacotes.
Gerando arquivos.
Código gerado com sucesso. Realize uma revisão antes de comitar.
```

Fonte: Autoria Própria

Ao finalizar a criação do projeto de teste, também foram necessárias tratativas no código gerado, exceto pelo fluxo de autenticação, que neste caso não foi preciso configurar, pois a aplicação não utiliza nenhum. Foram identificados alguns problemas nas migrações existentes, que foram corrigidos para executar os testes.

Algumas métricas foram coletadas durante o processo, como tempo que a TesTool gastou para gerar os projetos de teste, tempo para executar os testes, esforço humano necessário para correções e cobertura de código resultante. Na Tabela 2 é possível verificar os valores coletados, e perceber que o tempo que a ferramenta levou para gerar o código é menor que meio minuto. O projeto com mais linhas de código, neste caso Vehicle levou menos tempo para executar.

Tabela 2: Resultados do Estudo de Caso

API Web	Tempo de Geração de Código (Segundos)	Tempo de Execução de Teste (Segundos)	Esforço Humano (Tempo em Correções)	Cobertura de Linhas de Código
Vehicle	14,5	5,6	06:15:00	80,6%
Tasks	27,2	5,3	02:23:00	84,5%
Shop	20,1	1,2	01:35:00	83,1%

Fonte: Autoria Própria

Analisando a Tabela 2 é possível notar que o tempo para executar os testes, e o esforço humano é proporcional a quantidade de *endpoints* que cada projeto tem, vide Tabela 1. É possível perceber também que a cobertura de código para todos os projetos foi superior a 80%, evidenciando que a utilização do teste de caixa branca foi eficaz, fornecendo uma boa cobertura de código comparado com o que é praticado no mercado (STEVE CORNETT, 2021). Também foram observados alguns erros nos projetos avaliados, mas não foram considerados nesta análise porque alguns já haviam sido identificados e corrigidos anteriormente por meio de testes de integração existentes nas soluções.

4.2 Discussão

Uma observação importante sobre o código que a ferramenta produz é que ele testa apenas o aspecto funcional da API Web, verificando se os *endpoints* estão sendo executados com sucesso e validando as informações que foram persistidas. Regras de negócio, validações e restrições podem não ser testadas completamente pela ferramenta, pois é gerado código para testar um cenário específico de execução. No entanto, o desenvolvedor poderá complementar os testes produzidos aproveitando a arquitetura empregada para codificar novos testes.

Outro aspecto que deve ser discutido é o trabalho necessário para corrigir os testes gerados. Ao produzir um projeto de teste a partir de uma API Web, existem alguns trechos de código que a ferramenta não consegue resolver, e estes são comentados para que sejam corrigidos manualmente. O fato é que alguns desses artefatos serão produzidos durante o processo de geração de código. Portanto, existem algumas otimizações que poderiam ser aplicadas para minimizar o tempo necessário para realizar tratativas no código.

Por outro lado, a ferramenta agiliza de maneira considerável o trabalho de escrita de código de teste de integração. Devido ao pouco tempo disponível para o desenvolvimento do projeto, não foram realizadas análises mais detalhadas de quanto tempo seria necessário para implementar manualmente os testes de integração, para relacionar com o tempo gasto para gerar automaticamente por meio da ferramenta. Mas é possível afirmar, com base em experiências anteriores do autor, que o trabalho necessário utilizando a ferramenta, chega a ser três vezes menor.

A ferramenta possui algumas restrições de uso, como a linguagem de programação C#, e a necessidade de utilização do *entity framework* como mecanismo de persistência. Além disso, o código gerado é implementado em XUnit e segue uma arquitetura própria. Ademais, projetos que já possuem teste de integração, que foram desenvolvidos por outras ferramentas, ou que foram programados manualmente, não poderão ser migrados e mantidos pela ferramenta. Desta forma, a ferramenta não poderá ser utilizada para estes cenários.

5 CONCLUSÃO

Neste trabalho, foram elencados diversos estudos e recursos que contribuíram para o desenvolvimento da ferramenta TesTool, para a geração automática de código de teste de integração para APIs Web. Um estudo de caso foi realizado com a ferramenta, utilizando diferentes projetos para validar a sua capacidade de auxiliar no processo de codificação. Neste processo foram coletadas algumas medidas a partir de métricas importantes para evidenciar as contribuições da solução para projetos que utilizaram a ferramenta.

Foram apresentados vários artefatos que explicam de maneira detalhada todos os elementos da ferramenta desenvolvida. Entre elas a arquitetura, imagens com a documentação da ferramenta e imagens dos códigos fonte produzidos. Um projeto de API Web foi utilizado para exemplificar a utilização da ferramenta, explicando a estrutura de pastas gerada, testes que foram produzidos e a parametrização dos comandos.

A utilização da ferramenta traz vantagens e desvantagens no processo de desenvolvimento. Entre as vantagens pode-se citar: agilidade na produção de testes de integração para APIs Web; facilidade de manutenção e implementação de novos testes; boa cobertura de código de teste. Por outro lado, vale destacar algumas desvantagens na utilização da ferramenta, como: restrições de suporte a aplicações Web; necessidade de correções no código produzido; teste apenas de um caminho de execução possível.

Durante o processo de desenvolvimento, e em discussões com o orientador do trabalho, foram definidas algumas funcionalidades que seriam importantes para contribuir ainda mais com o processo de teste. Pode se citar a possibilidade de realizar migração de projetos de teste já existentes para serem suportados pela ferramenta; a capacidade de atualizar classes de teste já implementadas; uma configuração para gerar apenas o projeto com classes essenciais, sem produzir classes de teste e algumas otimizações nos códigos produzidos.

REFERÊNCIAS

ADEWOLE, Ayobami. **C# and .NET Core Test-Driven Development**: Dive into TDD to create flexible, maintainable, and production-ready .NET Core applications. 2 ed. Packt Publishing, 2018.

AMMANN, Paul; OFFUTT, Jeff. **Introduction to software testing**. Cambridge University Press, 2016.

ARCURI, Andrea. **Automated Black-and White-Box Testing of RESTful APIs With EvoMaster**. IEEE Software, v. 38, n. 3, p. 72-78, 2020.

ARCURI, Andrea; GALEOTTI, Juan P. **Testability transformations for existing APIs**. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). IEEE, 2020. p. 153-163.

BERTOLINO, Antonia; MARCHETTI, Eda. **A brief essay on software testing**. Software Engineering, 3rd edn. Development process, v. 1, p. 393-411, 2005.

FIELDING, Roy Thomas. **Architectural styles and the design of network-based software architectures**. University of California, Irvine, 2000.

GITHUB. **Roslyn Analyzers**. Disponível em: <https://github.com/dotnet/roslyn-analyzers>. Acesso em: 14 de nov. de 2021a.

GITHUB. **Bogus for .NET: C#, F#, and VB.NET**. Disponível em: <https://github.com/bchavez/Bogus>. Acesso em: 6 de ago. de 2021b.

HONFI, Dávid; MICSKEI, Zoltán. **White-box software test generation with Microsoft Pex on open source C# projects**: A dataset. Data in Brief, v. 31, p. 105962, 2020.

MICROSOFT. **ASP.NET Web APIs:** Build secure REST APIs on any platform with C#. Disponível em: <https://dotnet.microsoft.com/apps/aspnet/apis>. Acesso em: 07 de ago. de 2021a.

MICROSOFT. **Um tour pela linguagem C#**. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/tour-of-csharp>. Acesso em: 07 de ago. de 2021b.

MICROSOFT. **Introdução ao .NET**. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/core/introduction>. Acesso em: 07 de ago. de 2021c.

MICROSOFT. **Reflexão (C#)**. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/reflection>. Acesso em: 07 de ago. de 2021d.

MICROSOFT. **Geração de código e modelos de texto T4**. Disponível em: <https://docs.microsoft.com/pt-br/visualstudio/modeling/code-generation-and-t4-text-templates?view=vs-2019>. Acesso em: 07 de ago. de 2021e.

MICROSOFT. **Introdução a consultas LINQ (C#)**. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/linq/introduction-to-linq-queries>. Acesso em: 21 de nov. de 2021f.

MICROSOFT. **Documentação do Entity Framework**. Disponível em: <https://docs.microsoft.com/pt-br/ef>. Acesso em: 07 de ago. de 2021g.

MICROSOFT. **Visão geral do Visual Studio**. Disponível em: <https://docs.microsoft.com/pt-br/visualstudio/get-started/visual-studio-ide?view=vs-2019>. Acesso em: 07 de ago. de 2021h.

MICROSOFT. **NuGet**. Disponível em: <https://www.nuget.org>. Acesso em: 21 de nov. de 2021i.

MYERS, Glenford J.; SANDLER, Corey; BADGETT, Tom. **The art of software testing**. John Wiley & Sons, 2011.

ORACLE. **MySQL**. Disponível em: <https://www.mysql.com>. Acesso em: 14 de nov. de 2021.

POSTMAN. **Postman API Client**. Disponível em: <https://www.postman.com/product/api-client>. Acesso em: 07 de ago. de 2021.

PRESSMAN, Roger; MAXIM, Bruce. **Engenharia de Software-8ª Edição**. McGraw Hill Brasil, 2016.

REGISTROBR. **Estatísticas - Registro.br**. Disponível em: <https://registro.br/dominio/estatisticas>. Acesso em: 6 de ago. de 2021.

SOHAN, S. M.; ANSLOW, Craig; MAURER, Frank. **A case study of web API evolution**. In: 2015 IEEE World Congress on Services. IEEE, 2015. p. 245-252.

STEVE CORNETT. **Minimum Acceptable Code Coverage**. Disponível em: <https://www.bullseye.com/minimum.html>. Acesso em: 8 de dez. de 2021.

WIEDERSEINER, Christian et al. **An open-source tool for automated generation of black-box xunit test code and its industrial evaluation**. In: International Academic and Industrial Conference on Practice and Research Techniques. Springer, Berlin, Heidelberg, 2010. p. 118-128.

YUSIFOĞLU, Vahid Garousi; AMANNEJAD, Yasaman; CAN, Aysu Betin. **Software test-code engineering:** A systematic mapping. Information and Software Technology, v. 58, p. 123-147, 2015.

.NET FOUNDATION. **About xUnit.net**. Disponível em: <https://xunit.net>. Acesso em: 07 de ago. de 2021a.

.NET FOUNDATION. **Getting Started with xUnit.net**. Disponível em: <https://xunit.net/docs/getting-started/netfx/visual-studio>. Acesso em: 15 de ago. de 2021b.