Associação Propagadora Esdeva
Centro Universitário Academia – UniAcademia
Curso de Bacharelado em Sistemas de Informação
Trabalho de Conclusão de Curso – Artigo

Aumentando a Cobertura de Testes de Software através do Teste de Mutação

Tiago de Almeida Machado[1]
Centro Universitário Academia, Juiz de Fora, MG
Jacimar Fernandes Tavares[2]
Centro Universitário Academia, Juiz de Fora, MG

Linha de Pesquisa: Engenharia de Software

RESUMO

Através do teste de software espera-se contribuir para alcançar a qualidade do produto de software desenvolvido. O teste de mutação é uma técnica utilizada no teste de software para ajudar os desenvolvedores a determinar se o conjunto de testes criados detecta efetivamente todas as falhas possíveis. Este trabalho apresenta um estudo de caso da utilização da técnica de teste de mutação e a utilização de ferramental automatizado, para que, por meio de uma análise empírica, possam ser geradas propostas de novos testes ou melhorias nos já existentes. Observou-se, ao final, que a abordagem usada viabilizou um aumento na cobertura de código-fonte da aplicação que foi utilizada como exemplo.

Palavras-chave: Teste de mutação, Teste de software, Cobertura de teste.

1 INTRODUÇÃO

Atualmente, em boa parte do nosso tempo passamos interagindo com softwares através de dispositivos como celulares, computadores, televisores e até automóveis. Constantemente, lidamos com más experiências ocasionadas por defeitos nos softwares, desde as mais simples, como pequenas falhas de mensagens em aplicativos até falhas que podem causar sérios problemas, como o noticiado pela mídia envolvendo os carros de uma tradicional montadora, que entraram em combustão por causa de uma falha no software de monitoramento do motor (Folha de São Paulo, 2009).

Testes de Software, segundo (MYERS, 1979) descreve, é a execução de programas ou sistemas onde se tem a intenção de encontrar falhas. Os testes em softwares podem reduzir as ocorrências dessas falhas em ambientes de produção, e

---

[1] Discente do Curso de Bacharelado em Sistemas de Informação do Centro Universitário Academia – UniAcademia. Endereço: Rua Constantino Paleta, 100 ap.1302, Centro, Juiz de Fora. Celular: (32)98896-7131. E-mail: tiagoalmachado@outlook.com

[2] Docente do Curso de Bacharelado em Sistemas de Informação do Centro Universitário Academia. Orientador.

reduzir também a possibilidade de serem encontrados pelos usuários, pois como afirma em sua décima regra, quanto mais cedo um erro é localizado e resolvido em suas primeiras fases, sejam essas de especificação ou desenvolvimento, menor será seu custo de correção em relação aos encontrados em ambientes de produção.

Na literatura, diferentes tipos de testes de software são apresentados. Um deles é o teste de mutação (DEMILLO, LIPTON, & SAYWARD, 1978), que consiste na geração de variações do programa original, porém, com pequenas falhas inseridas propositalmente nessas variações. Os testes gerados serão dados como eficazes quando localizarem as falhas inseridas. Já os testes que não conseguiram encontrar as falhas inseridas devem ser melhorados até que possam encontrá-las.

Este trabalho tem como objetivo demonstrar que através da utilização da técnica de teste de mutação pode ser possível identificar as deficiências no conjunto de testes de uma aplicação e, somado a uma avaliação posterior, por parte dos envolvidos na definição dos testes criados, propor melhorias nos testes existentes, aumentando assim a porcentagem de cobertura deles. Através da técnica de teste de mutação espera-se responder à pergunta: “Somente com a morte dos mutantes será possível chegar a 100% de cobertura do código-fonte pelos testes?”.

O presente trabalho está dividido da seguinte forma: a primeira seção introduz os conceitos e os objetivos; a segunda seção apresenta o referencial teórico sobre teste de software, os critérios de cobertura de código-fonte e teste de mutação, além de abordar o ferramental utilizado no desenvolvimento do trabalho. *Stryker é uma* ferramenta utilizada para a criação automatizada de mutantes, e em complemento a ela a *Coverlet,* ferramenta utilizada para medir a cobertura de código, além de *Flunt,* aplicação testada no estudo de caso realizado, a qual foi submetida aos testes e critérios. Na terceira seção são apresentados trabalhos relacionados. Já a quarta seção mostra o emprego do teste de mutação através das ferramentas e o levantamento das propostas de melhoria nos testes unitários com a finalidade de aumentar a cobertura do código. A quinta e última seção discorre sobre os futuros trabalhos e apresenta as considerações finais.

2 REFERENCIAL TEÓRICO

Esta seção apresentará os conceitos necessários para o entendimento do que foi feito neste trabalho.

2.1 TESTE DE SOFTWARE

O teste de software é uma etapa no processo de desenvolvimento de software e tem como principal objetivo mitigar falhas que podem existir no sistema. Nesse ciclo verifica-se o comportamento do sistema, buscando saber se ele atende os requisitos especificados. "Teste de software é um elemento crítico da garantia de qualidade de software e representa a revisão final da especificação, projeto e geração de código" (PRESSMAN, 2002).

Com o objetivo de reduzir custos oriundos dos testes, é de suma importância a aplicação de técnicas que mostrem como testar e critérios de indiquem quando parar de testar, de forma planejada e sistemática, como afirma (DEMILLO, 1980).

Para (MYERS, 1979) há duas principais técnicas de teste. Teste funcional, comumente conhecida como teste de caixa-preta, onde o desenvolvedor ou testador não tem acesso ao código-fonte do sistema, sendo possível somente testar a entrada e saída, e o Teste estrutural, conhecido também por teste de caixa-branca, onde se tem total acesso ao código-fonte podendo realizar testes diretamente sobre ele. Este trabalho atua sobre testes estruturais.

Há também a técnica baseada em erros que segundo (DEMILLO, LIPTON, & SAYWARD, 1978) definem, trata-se de uma técnica que utiliza informações sobre os erros comumente cometidos do processo de desenvolvimento de um *software* para localizá-los. Esta técnica pode ser utilizada como critério para a análise de mutante ou também conhecido como Teste de mutação, outro foco deste trabalho. Sobre o planejamento dos testes, em (ROCHA, 2001) é dito que ele deve ser executado em diferentes fases do desenvolvimento do *software* e possui níveis de teste como teste de unidade, de integração, de sistema e de aceitação, dentre outros.

2.2 COBERTURA DE CÓDIGO

Segundo (SOARES E., 2006) a cobertura de código tem como objetivo verificar se o conjunto de teste exercita parte do código do *software***.** Ela é amplamente utilizada na fase de teste de unidade e integração. Em (SOARES E., 2006) são expostas as vantagens de se usar cobertura de código dentro do processo de desenvolvimento que são: (a) Orienta a administração de riscos, pois fornece dados sobre partes do código não executado pelos testes; e (b) Fornece informações de casos de testes redundantes.

Segundo (TIKIR, 2002) , desenvolvedores e testadores utilizam da cobertura de código para se assegurar que boa parte ou todas as linhas do código foram

executadas pelo menos uma vez durante os testes e segundo (CRAIG, 2002) existem pontos fracos e fortes:

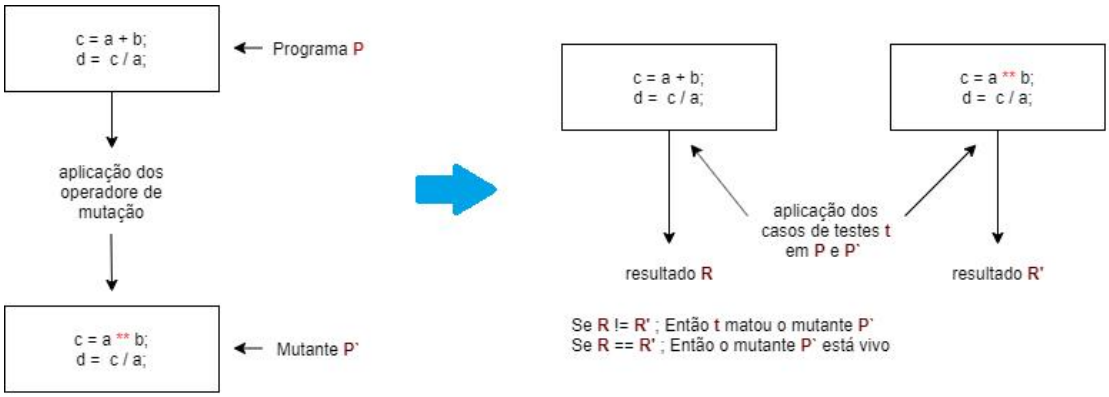
- Ponto fraco - para cada código executado pelo teste, não se assegura que ele atenda o cliente e ao requisito;
- Ponto forte - fornece um modo para determinar se os testes de unidade foram concluídos com sucesso e podem ser dados como terminados.

2.3 TESTE DE MUTAÇÃO

O teste de mutação consiste em criar um conjunto de programas com pequenas alterações, chamados de mutantes. O objetivo do teste de mutação é submeter os mutantes ao conjunto de testes já existentes no programa original, a fim destes identificarem diferenças de comportamentos entre o original dos seus mutantes, assim como proposto por (DEMILLO, LIPTON, & SAYWARD, 1978) . Quando os testes são capazes de revelar os defeitos inseridos na alteração, esses mutantes serão denominados “mortos” ou “mutantes mortos”, caso contrário “mutantes vivos”, portanto deverão ser melhorados de forma que consigam “matar” o mutante.

A FIGURA 1 exemplifica o processo do teste de mutação, onde é aplicada a mutação no programa P, gerando assim a mutação P` e quando submetidos ao teste T, podem apresentar resultados iguais entre P e P` ou não.

FIGURA 1– Processo do teste de mutação.

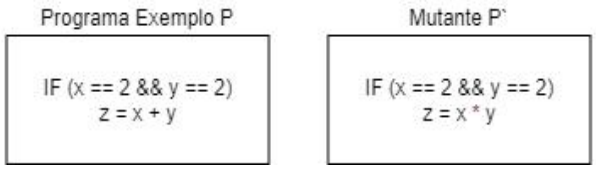


Fonte: (UMAR, 2006) traduzida pelo autor.

Em alguns casos a execução do mutante tem o mesmo resultado do original, chamado estes de “mutantes equivalentes”, que passam a não contribuir com a

avaliação da eficácia do teste. A FIGURA 2 exemplifica um mutante equivalente, onde independente do conjunto de teste o resultado de Z sempre será 4.

FIGURA 2 – Mutação equivalente



Fonte: (UMAR, 2006) traduzida pelo autor

(DEMILLO, 1980) , propõem uma medida quantitativa para adequação do conjunto de testes, definindo assim, um escore de mutação, que é o produto dos mutantes mortos sobre os mutantes gerados excluindo-se os mutantes equivalentes. Esse escore pode variar de 0 a 1, e quanto maior for, melhor será a eficácia do conjunto teste, portanto o escore de mutação definirá o término do teste de mutação quando seu valor chegar a 1 ou próximo de 1, e assim podendo definir que o conjunto de teste está adequado.

$$escore(P,T) = \frac{DM(P,T)}{M(P) - EM(P)}$$

onde:

P – Produto “Programa Original”

T – Conjunto de teste

DM(P,T) – Número de mutantes mortos

M(P) – Número de mutantes gerados

EM(P) – Número de mutantes equivalentes

2.4 FERRAMENTA STRYKER E SEUS COMPLEMENTOS

Para realização deste trabalho será utilizada a ferramenta *Stryker* na versão 0.16.1(beta), não havendo versão estável disponível para a linguagem Csharp, que é usada como apoio para a geração de mutantes. Nela são feitas substituições de sinais, operadores matemáticos, métodos e constantes no código-fonte do programa original para a geração das falhas nos programas mutantes. Das classes de operandos do *Stryker* se destaca os operadores aritmético, igualdade e booleanos. A lista completa dos operadores e mais informações sobre os seus objetivos podem ser vistas em seu site oficial (STRYKER).

Stryker gera um arquivo de *output*, onde podem ser extraídas informações, como o escore de mutação, mutantes mortos, mutantes sobreviventes, a cobertura dos testes, os *erros de execução*, os *erros de compilação* e os *totais de mutantes*.

2.5 FLUNT: A APLICAÇÃO USADA NO ESTUDO DE CASO.

Com a finalidade de demonstrar a técnica de teste de mutação, está sendo usada a aplicação Flunt, que é uma implementação do padrão de projeto *Notification* descrito por (FOWLER, 2004) . Esse padrão de projeto permite encaminhar mensagens da camada de negócios para a camada de apresentação, com a reduzida complexidade da aplicação.

Hoje, a aplicação Flunt é amplamente utilizada pela comunidade tendo em registro mais de 95 mil downloads em seu site oficial (NUGET), contendo 1869 linhas de códigos distribuídas entre 14 classes, 2 interfaces e 174 métodos.

3 TRABALHOS RELACIONADOS

(SOARES I. W., 2000) faz um experimento onde é utilizada a ferramenta *Proteum* (DELAMARO, MALDONADO, & JINO, 2007) e o critério de análise de mutantes e, ao final, é feito um levantamento das 4 classes de operadores que geram o maior número de mutantes equivalentes e as 4 classes que geram menos.

(JORGE, 2002) abordou as técnicas de testes funcionais, estruturais e baseados em erro, com ênfase no critério de análise de mutantes e outros dois critérios que derivam deste, mutação fraca (HOWDEN, 1982) e a mutação firme (WOODWARD & HALEWOOD, 1988). Já em (SOARES I. W., 2000) e (JORGE, 2002) são estabelecidas classes de operadores essenciais, ou seja, operadores que geram uma menor porcentagem de mutantes equivalentes podendo assim reduzir o custo empregado para análise de mutantes.

(YANO, 2004) elaborou um estudo da utilização do teste de mutação em programas com paradigma funcional. Nele é utilizada e apresentada a ferramenta *Proteum/SML,* modificada especificamente para esse paradigma.

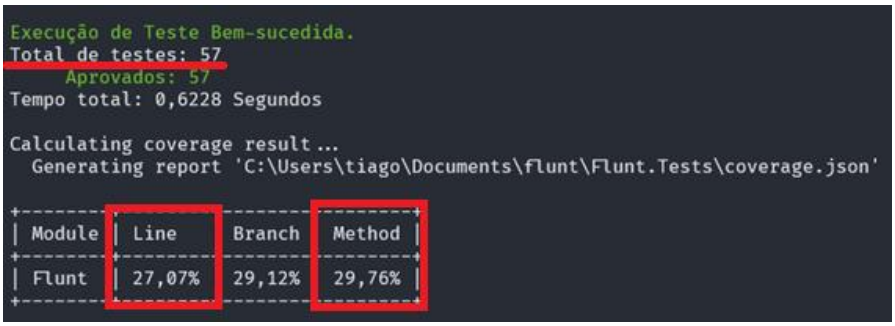
Em (SOARES E. , 2006) é mostrado um estudo de caso sobre processo de análise de cobertura alinhado ao processo de desenvolvimento de *software,* que realizou uma definição sobre cobertura de código e levanta pontos essenciais para execução da análise deste critério.

Em (CAMPANHA, 2010) foi conduzido um estudo sobre a utilização do teste de mutação em 2 programas de paradigmas diferentes, estruturado e orientado a objeto. Para execução de seu estudo foram utilizadas 2 ferramentas para geração de mutantes: *Proteum* (DELAMARO, MALDONADO, & JINO, 2007) e *MuClipse* (SMITH & WILLIANS, 2009) , assim como este trabalho utilizou da análise de mutantes para propor melhoria nos testes em ambos paradigmas. Nos trabalhos de (CAMPANHA, 2010) e (YANO, 2004) são analisados os escores de mutação anterior e posterior às propostas de melhorias nos testes, porém em nenhum houve uma demonstração do aumento da cobertura do código-fonte pelos testes, como neste trabalho demonstra.

4 DESENVOLVIMENTO DO TRABALHO

No desenvolvimento do estudo de caso deste trabalho, o projeto de código-fonte do software testado e analisado com o apoio ferramental é colocado diante de uma primeira bateria de testes, visando analisar a sua condição atual. Após a primeira execução do ferramental, descobriu-se o seguinte cenário (FIGURA 3):

FIGURA 3 - Teste de cobertura da aplicação Flunt com o apoio de Coverlet.



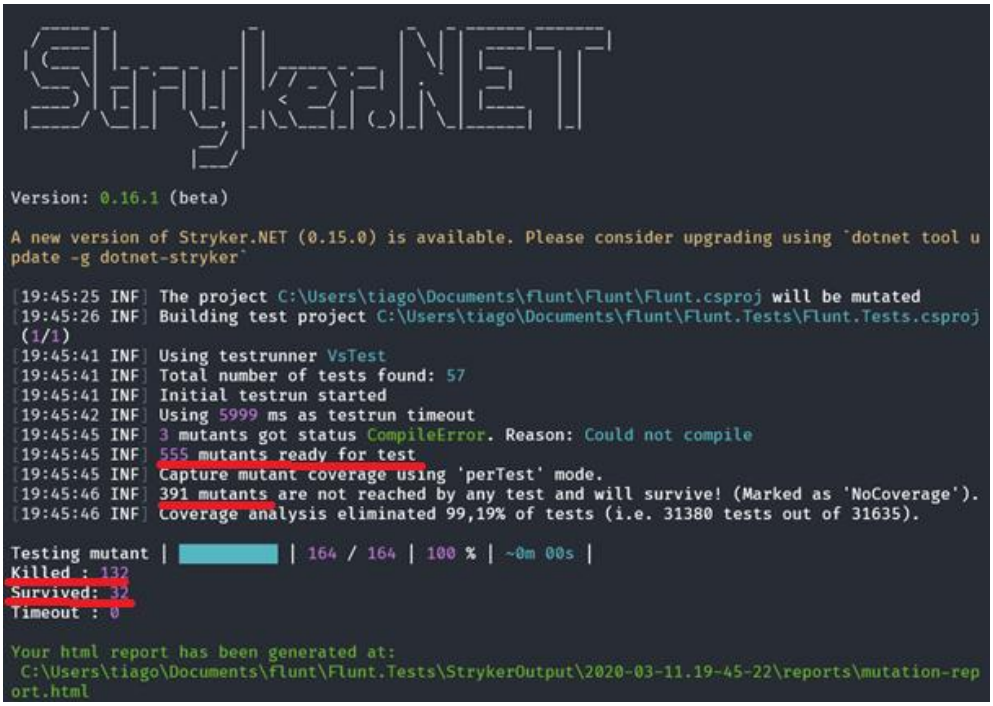
Fonte: Autor.

- Foram encontradas implementações de 57 testes unitários no projeto de código-fonte da aplicação.
- 29,76% dos métodos da aplicação são cobertos pelos testes unitários existentes.
- O conjunto de testes cobre 27,07% das linhas de código da aplicação.

Ainda com apoio do ferramental outras informações puderam ser geradas, como demonstrado na FIGURA 4:

- Em relação ao projeto de código 555 mutações foram geradas, de forma automática pelo ferramental usado.
- 391 mutações continuaram sendo sobreviventes. Isso significa que os testes existentes (57) são insuficientes para cobertura desejada do código-fonte.
- O conjunto de testes unitários já existentes eliminou 132 mutações, ou seja, conseguiram identificar as alterações inseridas pelos operadores de mutação.
- O conjunto de testes unitários já existentes não foram capazes de identificar a diferença entre o programa original e 32 mutações.

FIGURA 4 - Primeira execução do Stryker na aplicação Flunt.



Fonte: Autor.

Na continuidade do estudo de caso, ao se aplicar a fórmula de escore de DeMillo nas informações extraídas (FIGURA 4), tem-se um resultado aproximado de 0.237, como demonstrado a seguir.

$$escore(P,T) = \frac{555}{132 - 0}$$

$$escore(P,T) \cong 0.237$$

No *output* do ferramental, demonstrado pela FIGURA 5, pode-se observar que há classes que obtiveram escores de mutação abaixo de 0,5: Contract.cs, DecimalValidationContract.cs, DoubleValidationContract.cs,

FloatValidationContract.cs, GuidValidationContract.cs, IntValidationContract.cs, e LongValidationContract.cs. Outras duas classes obtiveram escores abaixo de 0,8: DateTimeValidationContract.cs, e TimeSpanValidationContract.cs, porém todas as classes em destaque na FIGURA 5 têm total relevância para o desenvolvimento deste trabalho, dado que para a morte total dos mutantes será necessário propor melhorias nos testes unitários existentes em todas as classes abaixo de 1.0 (100%) em escore de mutação, a fim de aumentar seus escores de mutação e consequentemente a cobertura do código-fonte pelos conjuntos de teste.

FIGURA 5 - Primeiro *output* do Stryker na aplicação Flunt.

File / Directory	Mutation score	# Killed	# Survived	# Timeout	# No coverage	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
Validations	23.65	131	32	0	391	0	3	131	423	557
BoolValidationContract.cs	100.00	2	0	0	0	0	0	2	0	2
Contract.cs	40.00	2	0	0	3	0	0	2	3	5
DateTimeValidationContract.cs	78.26	18	5	0	0	0	0	18	5	23
DecimalValidationContract.cs	18.68	17	6	0	68	0	0	17	74	91
DoubleValidationContract.cs	8.79	8	3	0	80	0	0	8	83	91
FloatValidationContract.cs	8.79	8	3	0	80	0	0	8	83	91
GuidValidationContract.cs	50.00	4	0	0	4	0	0	4	4	8
IntValidationContract.cs	10.99	10	6	0	75	0	0	10	81	91
LongValidationContract.cs	4.60	4	3	0	80	0	0	4	83	87
ObjectValidationContract.cs	100.00	7	0	0	0	0	0	7	0	7
StringValidationContract.cs	92.31	36	2	0	1	0	3	36	3	42
TimeSpanValidationContract.cs	78.95	15	4	0	0	0	0	15	4	19

Fonte: Stryker

4.1 ANÁLISES DAS CLASSES COM MUTANTES VIVOS

Ao analisar a classe *Contract.cs* é observado o método *Join* como pode ser visto na FIGURA 6, onde existem três mutantes vivos, nos dois primeiros foi modificado o operador != (diferente) do código-fonte original por sua negação (mutante 4) e por == (Igual, mutante 3) ambos na linha 16. Na última mutação é feita uma negação do método *Invalid* (mutante 5) na linha 20.

FIGURA 6 - Join da classe *Contract*

```
14 public Contract Join(params Notifiable[] items)
15 {
16     if (3 items == null 4 !(items != null) items != null)
17     {
18         foreach (var notifiable in items)
19         {
20             if (5 !(notifiable.Invalid) notifiable.Invalid)
21                 AddNotifications(notifiable.Notifications);
22         }
23     }
24     return this;
25 }
```

Fonte: Stryker

Para este método foi observado a ausência de testes unitários o que passa a ser impossível a morte dos mutantes. Para resolver os dois primeiros mutantes (3 e 4) foram propostos dois testes, como demonstrado na FIGURA 7. No primeiro teste *Join_Should_DontUniteNotifications_When_NullItems* linha 14, é fornecido um contrato nulo para ser adicionado a outro já existente e espera-se que esse contrato não seja modificado por um outro nulo. Já o segundo teste *Join_Should_JoinNotifications_When_NotNullItems* linha 28 tem como objetivo validar a junção de dois contratos válidos não nulos.

FIGURA 7 - Testes propostos a *Contract*.

```
12 [TestMethod]
13 [TestCategory("ContractValidation")]
   0 referências
14 public void Join_Should_DontUniteNotifications_When_NullItems()
15 {
16     _dummy = new Dummy();
17
18     var contract = new Contract()
19         .Requires()
20         .IsDigit(_dummy.stringProp, nameof(_dummy.stringProp), "Value is not a digit")
21         .Join(null);
22
23     Assert.AreEqual(1, contract.Notifications.Count);
24 }
25
26 [TestMethod]
27 [TestCategory("ContractValidation")]
   0 referências
28 public void Join_Should_JoinNotifications_When_NotNullItems()
29 {
30     _dummy = new Dummy();
31     _dummy.stringProp = "A";
32
33     var firstContract = new Contract()
34         .Contains(_dummy.stringProp, "1", nameof(_dummy.stringProp), "The value contains the digit 1");
35
36     var contract = new Contract()
37         .Requires()
38         .IsDigit(_dummy.stringProp, nameof(_dummy.stringProp), "Value is not a digit")
39         .Join(firstContract);
40
41     Assert.AreEqual(2, contract.Notifications.Count);
42 }
```

Fonte: Flunt

Para a morte do último mutante são propostos os testes da FIGURA 8. Nela é possível encontrar dois outros testes

Join_Should_DontUniteNotifications_When_ContractValid na linha 46 e *Join_Should_JoinNotifications_When_ContractInvalid* na linha 64, onde o primeiro teste verifica que os contratos sem notificações não devem ser juntados a outros contratos com notificações. O segundo tem o objetivo de testar que os dois contratos com notificações devem se juntar.

FIGURA 8 - Testes propostos a *Contract*.

```
44 [TestMethod]
45 [TestCategory("ContractValidation")]
   0 referências
46 public void Join_Should_DontUniteNotifications_When_ContractValid()
47 {
48     _dummy = new Dummy();
49     _dummy.stringProp = "A";
50
51     var firstContract = new Contract()
52         .Contains(_dummy.stringProp, "A", nameof(_dummy.stringProp), "The value contains the digit 1");
53
54     var contract = new Contract()
55         .Requires()
56         .IsDigit(_dummy.stringProp, nameof(_dummy.stringProp), "Value is not a digit")
57         .Join(firstContract);
58
59     Assert.AreEqual(1, contract.Notifications.Count);
60 }
61
62 [TestMethod]
63 [TestCategory("ContractValidation")]
   0 referências
64 public void Join_Should_JoinNotifications_When_ContractInvalid()
65 {
66     _dummy = new Dummy();
67     _dummy.stringProp = "A";
68
69     var firstContract = new Contract()
70         .Contains(_dummy.stringProp, "B", nameof(_dummy.stringProp), "The value contains the digit 1");
71
72     var contract = new Contract()
73         .Requires()
74         .IsDigit(_dummy.stringProp, nameof(_dummy.stringProp), "Value is not a digit")
75         .Join(firstContract);
76
77     Assert.AreEqual(2, contract.Notifications.Count);
78 }
```

Fonte: Autor

Ao detalhar a classe *DateTimeValidationContract.cs* o ferramental nos mostra seu código-fonte como pode ser visto na FIGURA 9. Nela é possível ver no método *IsGreaterThan,* linha 7, e seu mutante sobrevivente na linha 9. Este substituiu a comparação de <= (menor igual) do código-fonte original por < (menor, mutante 9) e mesmo assim os testes que validam este fragmento de código-fonte não conseguiram notar a mutação.

FIGURA 9 – Método *IsGreaterThan* da aplicação Flunt.

```
7  public Contract IsGreaterThan(DateTime val, DateTime comparer, string property, string message)
8  {
9      if (9 val < comparer val <= comparer)
10         AddNotification(property, message);
11
12     return this;
13 }
```

Fonte: Stryker

Para que seja possível a morte do mutante o teste deve identificá-lo como falha, portanto são analisados os testes unitários existentes (FIGURA 10) do método *IsGreaterThan*. Nas primeiras verificações (da linha 20 a 30) o teste utiliza datas maiores e nas posteriores(da linha 32 a 41) com datas menores em relação a uma data fornecida para comparação. Com esses dados é possível notar a ausência da validação com valores iguais, e com isso pode-se propor uma melhoria nos testes, como na FIGURA 11 linha 46, onde foi acrescentada validação.

FIGURA 10 – Testes unitários existente no método *IsGreaterThan* da aplicação Flunt.

```
15 public void IsGreaterThan()
16 {
17     _dummy = new Dummy();
18     _dummy.dateTimeProp = new DateTime(2005, 5, 15, 16, 0, 0);
19
20     var wrong = new Contract()
21         .Requires()
22         .IsGreaterThan(_dummy.dateTimeProp, _dummy.dateTimeProp.AddMilliseconds(1),
23             nameof(_dummy.dateTimeProp), "Date 1 should be greater than Date 2")
24         .IsGreaterThan(_dummy.dateTimeProp, _dummy.dateTimeProp.AddSeconds(1),
25             nameof(_dummy.dateTimeProp), "Date 1 should be greater than Date 2")
26         .IsGreaterThan(_dummy.dateTimeProp, _dummy.dateTimeProp.AddMinutes(1),
27             nameof(_dummy.dateTimeProp), "Date 1 should be greater than Date 2");
28
29     Assert.AreEqual(false, wrong.Valid);
30     Assert.AreEqual(3, wrong.Notifications.Count);
31
32     var right = new Contract()
33         .Requires()
34         .IsGreaterThan(_dummy.dateTimeProp, _dummy.dateTimeProp.AddMilliseconds(-2),
35             nameof(_dummy.dateTimeProp), "Date 1 is not greater than Date 2")
36         .IsGreaterThan(_dummy.dateTimeProp, _dummy.dateTimeProp.AddSeconds(-2),
37             nameof(_dummy.dateTimeProp), "Date 1 is not greater than Date 2")
38         .IsGreaterThan(_dummy.dateTimeProp, _dummy.dateTimeProp.AddMinutes(-2),
39             nameof(_dummy.dateTimeProp), "Date 1 is not greater than Date 2");
40
41     Assert.AreEqual(true, right.Valid);
42 }
```

Fonte: Flunt

FIGURA 11 - Teste proposto para o método *IsGreaterThan* da aplicação Flunt.

```
44 [TestMethod]
45 [TestCategory("DateTimeValidation")]
   0 referências
46 public void IsGreaterThan_Should_InvalidAndWithNotifications_When_Date2IsEquals()
47 {
48     _dummy = new Dummy();
49     _dummy.dateTimeProp = new DateTime(2005, 5, 15, 16, 0, 0);
50
51     var equal = new Contract()
52         .Requires()
53         .IsGreaterThan(_dummy.dateTimeProp, _dummy.dateTimeProp,
54             nameof(_dummy.dateTimeProp), "Date 1 and Date 2 are equals");
55
56     Assert.AreEqual(false, equal.Valid);
57     Assert.AreEqual(1, equal.Notifications.Count);
58 }
```

Fonte: Autor

Os métodos *IsLowerThan,* FIGURA 12, linha 25, e *IsBetween,* FIGURA 13, linha 41 da classe *DateTimeValidationContract.cs*, contêm os mesmos problemas do método *IsGreaterThan*, porém, com suas respectivas validações. Assim foi proposto a mesma técnica de melhoria nos testes, necessitando apenas adicionar um teste para cada um, contendo a validação de valores iguais.

FIGURA 12 - *IsLowerThan* da classe *DateTimeValidationContract.cs* e mutante.

```
23 public Contract IsLowerThan(DateTime val, DateTime comparer, string property, string message)
24 {
25     if (15 val > comparer~~val >= comparer~~)
26         AddNotification(property, message);
27
28     return this;
29 }
```

Fonte: Stryker

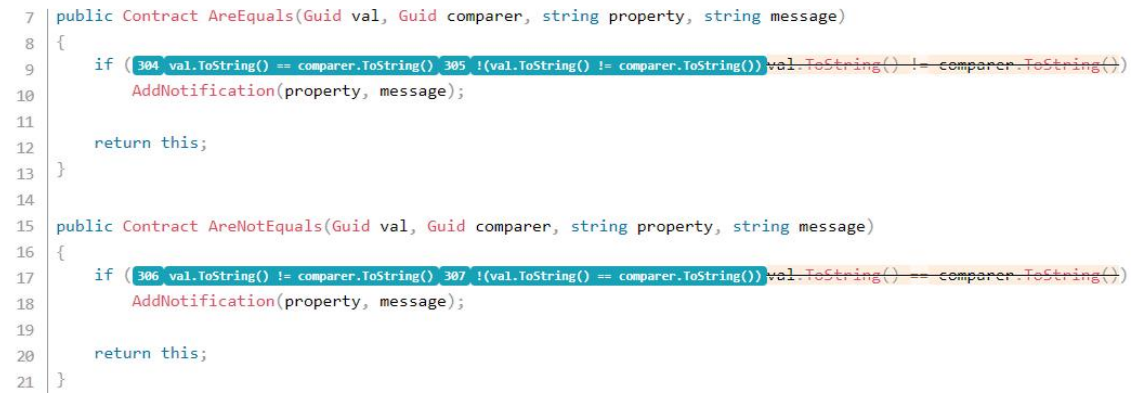
FIGURA 13 - *IsBetween* da classe *DatetimeValidationContract.cs* e mutantes.

```
39 public Contract IsBetween(DateTime val, DateTime from, DateTime to, string property, string message)
40 {
41     if (!(21 val > from~~val >= from~~ && 23 val < to~~val <= to~~))
42         AddNotification(property, message);
43
44     return this;
45 }
```

Fonte: Stryker

Ao abrir o output referente à classe <u>*GuidValidationContract.cs*</u> é possível notar a existência de quatro mutantes vivos com os identificadores 304, 305 na linha 9 e 306, 306 na linha 17, como pode ser visto na FIGURA 14 nos métodos *AreEquals* linha 7, e *AreNotEquals*, linha 15.

FIGURA 14 - *AreEquals* e *AreNotEquals* da classe *GuidValidationContract.cs* e mutantes.



```
7  public Contract AreEquals(Guid val, Guid comparer, string property, string message)
8  {
9      if (304 val.ToString() == comparer.ToString() 305 !(val.ToString() != comparer.ToString()) val.ToString() != comparer.ToString())
10         AddNotification(property, message);
11
12     return this;
13 }
14
15 public Contract AreNotEquals(Guid val, Guid comparer, string property, string message)
16 {
17     if (306 val.ToString() != comparer.ToString() 307 !(val.ToString() == comparer.ToString()) val.ToString() == comparer.ToString())
18         AddNotification(property, message);
19
20     return this;
21 }
```

Fonte: Stryker

Para a morte dos mutantes foram propostos dois novos testes unitários (FIGURA 15). No primeiro teste *AreEquals_Should_InvalidAndWithNotifications_When_Distinct* (linha 15) são solucionados os dois primeiros mutantes (304 e 305). Nele é fornecido o mesmo *Guid* para comparação no método *AreEquals*. Para a morte dos outros mutantes (306 e 307) restantes da classe *GuidValidationContract.cs* foi adicionado o segundo teste unitário *AreNotEquals_Should_InvalidAndWithNotifications_When_Same linha 29* onde são utilizados dois *guids* diferentes para comparação no método *AreNotEquals.*

FIGURA 15 - Testes propostos para classe *GuidValidationContract.cs d*a aplicação.

```
13 [TestMethod]
14 [TestCategory("GuidValidation")]
   0 referências
15 public void AreEquals_Should_InvalidAndWithNotifications_When_Distinct()
16 {
17     _dummy = new Dummy();
18     _dummy.guidProp = Guid.NewGuid();
19
20     var wrong = new Contract()
21         .Requires()
22         .AreEquals(_dummy.guidProp, Guid.NewGuid(), nameof(_dummy.guidProp), "Guid1 and Guid2 are distincts");
23
24     Assert.AreEqual(false, wrong.Valid);
25     Assert.AreEqual(1, wrong.Notifications.Count);
26 }
27
28 [TestMethod]
29 [TestCategory("GuidValidation")]
   0 referências
30 public void AreNotEquals_Should_InvalidAndWithNotifications_When_Same()
31 {
32     var guid = Guid.NewGuid();
33     _dummy = new Dummy();
34     _dummy.guidProp = guid;
35
36     var wrong = new Contract()
37         .Requires()
38         .AreNotEquals(_dummy.guidProp, guid, nameof(_dummy.guidProp), "Guid1 and Guid2 are same");
39
40     Assert.AreEqual(false, wrong.Valid);
41     Assert.AreEqual(1, wrong.Notifications.Count);
42 }
```

Fonte: Autor.

A classe *StringValidationContract.cs* mesmo tendo escore de mutação inicialmente superior a 0,8 possui alguns mutantes que devem ser mortos, há dois mutantes vivos (502 e 506) um para o método *HasMinLen,* linha 34, e *HasMaxLen,* linha 42. Na FIGURA 16, foram substituídos os operadores < (menor) do código-fonte original por <= (menor igual) no mutante 502 e > (maior) também do código original por >= (maior igual) no mutante 506. Para a solução desses mutantes foram propostos mais dois testes (FIGURA 17), onde os métodos MinLen_Should_ValidAndWithoutNotifications_When_Equal (linha 139) e MaxLen_Should_ValidAndWithoutNotifications_When_Equal (linha 174) têm o intuito de verificar o comportamento do método quando informado dois valores iguais para comparação.

FIGURA 16 – Métodos *HasMinLen e HasMaxLen* da classe *StringValidationContract.cs.*

```
34 public Contract HasMinLen(string val, int min, string property, string message)
35 {
36     if (string.IsNullOrEmpty(val) || 502 val.Length <= min ~~val.Length < min~~)
37         AddNotification(property, message);
38
39     return this;
40 }
41
42 public Contract HasMaxLen(string val, int max, string property, string message)
43 {
44     if (string.IsNullOrEmpty(val) || 506 val.Length >= max ~~val.Length > max~~)
45         AddNotification(property, message);
46
47     return this;
48 }
```

Fonte: Stryker.

FIGURA 17 - Testes propostos para a classe *StringValidationContract.cs*.

```
137 [TestMethod]
138 [TestCategory("StringValidation")]
    0 referências
139 public void MinLen_Should_ValidAndWithoutNotifications_When_Equal()
140 {
141     var right = new Contract()
142         .Requires()
143         .HasMinLen("Equal", 5, "string", "String len is less than permited");
144
145     Assert.IsTrue(right.Valid);
146     Assert.AreEqual(0, right.Notifications.Count);
147 }
    :
172 [TestMethod]
173 [TestCategory("StringValidation")]
    0 referências
174 public void MaxLen_Should_ValidAndWithoutNotifications_When_Equal()
175 {
176     var right = new Contract()
177         .Requires()
178         .HasMaxLen("Equal", 5, "string", "String len is more than permited");
179
180     Assert.IsTrue(right.Valid);
181     Assert.AreEqual(0, right.Notifications.Count);
182 }
```

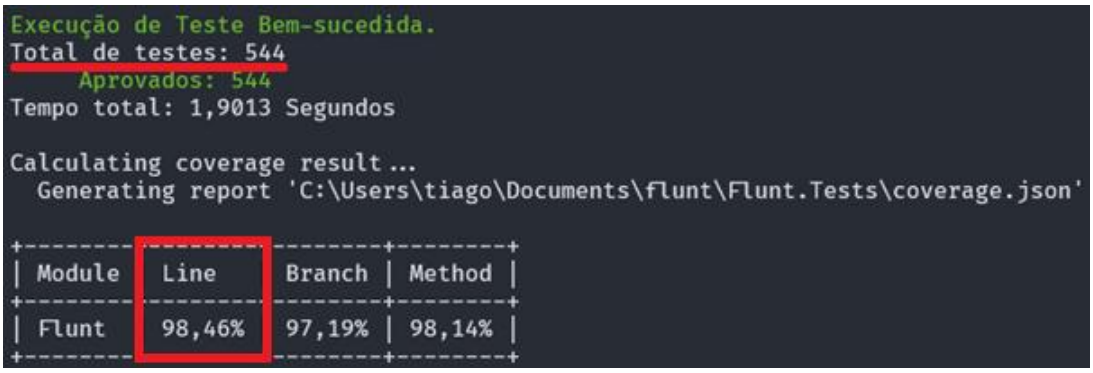
Fonte: Autor.

As classes: *DecimalValidationContract.cs*, DoubleValidationContract.cs, *LongValidationContract.cs*, *IntValidationContract.cs*, *FloatValidationContract.cs* e *TimeSpanValidationContract.cs*, contêm os mesmos métodos *IsGreaterThan*, *IsLowerThan* e *IsBetween* da classe *DateTimeValidationContract.cs*, porém com suas respectivas comparações por tipo. As melhorias propostas aos testes desta classe também se enquadram nelas, pois, seus testes contêm as mesmas deficiências, assim foram gerados testes semelhantes para essas classes.

5 RESULTADOS

Após as análises realizadas nas classes da FIGURA 5, bem como a criação dos testes resultantes dessas análises, novamente foi-se usado o ferramental automatizado em uma segunda bateria de execuções. Os testes unitários que antes eram contabilizados em 57 passaram a ser 544 com uma cobertura de 98.46% das linhas sendo cobertas por esse novo conjunto de testes (FIGURA 18), em comparação ao percentual de 27,07% na primeira execução do ferramental (FIGURA 4).

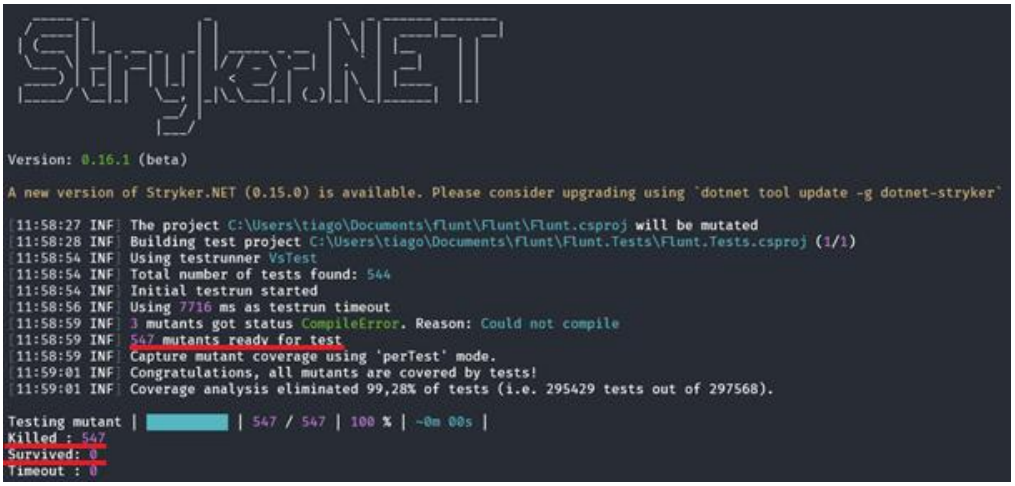
FIGURA 18 -Teste de cobertura da aplicação Flunt na segunda bateria de teste.



Fonte: Autor.

Na FIGURA 19, pode-se notar diferenças de informações sobre a primeira e a segunda bateria de testes. Ao final, em resumo, tem-se as informações da TABELA 1, onde é feita a uma comparação dos resultados na execução do ferramental antes das propostas de melhorias no conjunto de testes.

FIGURA 19 - Segunda execução do Stryker na aplicação Flunt.



Fonte: Autor

TABELA 1 - Paralelo entra a primeira e segunda execução do Stryker na aplicação Flunt.

	Primeira bateria	Segunda bateria
Número de mutantes	555	547
Mutantes com testes	164	547
Mutantes vivos	32	0
Mutantes mortos	132	547

Fonte: Autor.

Ao aplicar-se novamente a fórmula de escore de DeMillo tem-se o resultado 1.0, maior escore possível.

$$escore(P,T) = \frac{547}{547 - 0}$$

$$escore(P,T) = 1.0$$

Pode-se também validar as informações geradas com o output gerado pelo ferramental (FIGURA 20), onde podem ser observadas as classes DecimalValidationContract.cs, DoubleValidationContract.cs, FloatValidationContract.cs, GuidValidationContract.cs, IntValidationContract.cs, LongValidationContract.cs, DateTimeValidationContract.cs, TimeSpanValidationContract.cs da aplicação testada que foram verificadas com baixo escore de mutação e que seus testes deveriam ser analisados. Agora com a pontuação de 100% de escore de mutação.

FIGURA 20 – Segundo o*utput* do Stryker na aplicação Flunt.

File / Directory	Mutation score	# Killed	# Survived	# Timeout	# No coverage	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
Validations	100.00% 100.00	546	0	0	0	0	3	546	0	549
BoolValidationContract.cs	100.00% 100.00	2	0	0	0	0	0	2	0	2
Contract.cs	100.00% 100.00	5	0	0	0	0	0	5	0	5
DateTimeValidationContract.cs	100.00% 100.00	21	0	0	0	0	0	21	0	21
DecimalValidationContract.cs	100.00% 100.00	89	0	0	0	0	0	89	0	89
DoubleValidationContract.cs	100.00% 100.00	89	0	0	0	0	0	89	0	89
FloatValidationContract.cs	100.00% 100.00	89	0	0	0	0	0	89	0	89
GuidValidationContract.cs	100.00% 100.00	8	0	0	0	0	0	8	0	8
IntValidationContract.cs	100.00% 100.00	89	0	0	0	0	0	89	0	89
LongValidationContract.cs	100.00% 100.00	89	0	0	0	0	0	89	0	89
ObjectValidationContract.cs	100.00% 100.00	7	0	0	0	0	0	7	0	7
StringValidationContract.cs	100.00% 100.00	39	0	0	0	0	3	39	0	42
TimeSpanValidationContract.cs	100.00% 100.00	19	0	0	0	0	0	19	0	19

Fonte: Stryker.

Desta forma, pode-se concluir o processo de teste, pois o conjunto de testes mostrou-se adequado para a aplicação base do estudo de caso, apesar de não se alcançar 100% de cobertura (FIGURA 18, 98.46%), já que o ferramental não conseguiu gerar todos os possíveis mutantes, como ilustrado no exemplo da FIGURA 21, onde podem ser observadas três mutações (510, 511 e 509) linha 52, deixando de gerar uma quarta com a comparação “if(string.IsNullOrEmpty(val) && val.Length == len)“. A nova comparação geraria um novo mutante vivo, consequentemente um novo teste que obrigaria a fornecer uma “string” nula, que por final geraria a cobertura sobre a trecho do código “string.IsNullOrEmpty(val)". Há também casos que não existem mutantes para os operadores como por exemplo “if (val is null)” ou expressões regulares.

FIGURA 21 - Fragmento do *output* do Stryker

```
50  public Contract HasLen(string val, int len, string property, string message)
51  {
52      if (502 string.IsNullOrEmpty(val) && val.Length != len
            503 !(string.IsNullOrEmpty(val) || val.Length != len) string.IsNullOrEmpty(val)
        || 501 val.Length == len val.Length != len)
53          AddNotification(property, message);
54
55      return this;
56  }
```

Fonte: Stryker.

6 CONSIDERAÇÕES FINAIS

Este trabalho buscou apresentar o processo de testes e a utilização da técnica de teste de mutação, onde com o apoio ferramental e uma análise empírica chegou-se aos resultados demonstrados. Após propostas as melhorias no conjunto de testes da aplicação aqui usada como exemplo, foi possível notar o aumento da cobertura de código-fonte junto com o escore de mutação, mas somente com a morte dos mutantes não foi capaz de chegar a 100% de cobertura de código. Destaca-se que o teste de mutação não é substituto para o teste de cobertura e sim um complemento e que os testes gerados neste artigo serão submetidos à base principal de código-fonte da aplicação *Flunt* para que a comunidade de desenvolvedores possa usufruir de seus benefícios.

A utilização do critério de análise de mutantes, em geral, necessita da intervenção humana e exige esforço, aumentando assim o custo da atividade de teste. Assim como Barbosa (1998) definiu um procedimento de determinar um conjunto essencial de classes de operadores de mutação em C, reduzindo em 65,9% do custo de teste, preservando um alto escore de mutação, propõe-se, em trabalhos futuros, que seja submetido às classes de operandos do *Stryker* no mesmo procedimento de Barbosa.

ABSTRACT

Through software testing, it is expected to contribute to achieving software quality. Mutation testing is a technique used in software testing to help developers determine if the test they have created effectively detects all possible failures. This work presents a case study about the use of the mutation test technique and automated tools, so that, through an empirical analysis, new tests or improvements can be generated. Finally, the approach used enabled an increase in coverage of the application`s source code that was used as an example.

Keywords: Mutation test, Software test, Test coverage.

REFERÊNCIAS

CAMPANHA, D. **Teste de mutação nos paradigmas procedural e OO: uma avaliação no contexto de estrutura de dados**. São Paulo: USP. 2010.

CRAIG, R. D. (2002). **Systematic Software Testing**. London: Artech House Publishers. 2002.

DELAMARO, M., MALDONADO, J., & JINO, M. **Introdução ao Teste de Software**. 2007.

DEMILLO, R. A. **Mutation Analysis as a Tool for Software Quality Assurance**. Los Alamitos: IEEE Computer Society Press. 1980.

DEMILLO, R. A., LIPTON, R. J., & Sayward, F. G. **Hints on Test Data Selection: Help for the Practicing Programmer**. IEEE Computer. 1978.

HOWDEN, W. **Weak Mutation testing and completeness of test sets. IEEE Transactions on Software Engineering**. 1982

JORGE, R. F. **Teste de Mutação: Subsídios para a redução do custo de aplicação**. São Carlos: USP. 2002.

MYERS, G. **The Art of Software Testing**. New York. 1979.

PRESSMAN, R. S. **Engenharia de Software**. Rio de Janeiro: McGraw-Hill. 2002.

ROCHA, A. R. **Qualidade de software – Teoria e prática. São Paulo: Prentice Hall**.

SMITH, B., & WILLIAMS, L. **On guiding the augmentation of an automated test suite via mutation analysis**. 2009.

SOARES, E. **Um Processo de Análise de Cobertura alinhado ao Processo de Desenvolvimento de Software em Aplicações Embarcadas**. Recife: UFPE. 2006.

SOARES, I. W. **Equivalência de Programas e o Teste de Software: Resultados de um Experimento de Aplicação do Critério Análise de Mutantes**. 2000.

TIKIR, M. M. **Efficient Instrumentation for Code Coverage Testing. Roma: ACM SIGSOFT Software Engineering**. 2002.

UMAR, M. **An evaluation of mutation operators for Equivalent mutants. London: Department of Computer Science King's College**. 2006.

WOODWARD, M. R., & Halewood, K. (1988). **From weak to strong, dead or alive? An analysis of some mutation testing issues**. 1988.

YANO, T. **Estudo do teste de mutação em programas funcionais SML**. São Paulo: USP. 2004.

SOBRÉ, E. **Defeito em software causou incêndio no Onix, diz vice da GM**. Folha de São Paulo. (2009). Disponível em https://www1.folha.uol.com.br/mercado/2019/11/defeito-em-software-causou-incendio-no-onix-diz-vice-da-gm.shtml. Acesso em 01 de 05 de 2020

FOWLER, M. (2004). **Notification**. Disponível em: https://martinfowler.com/eaaDev/Notification.html. Acesso em 01 de 05 de 2020

NUGET. **Flunt**. Disponível em: https://www.nuget.org/packages/Flunt/. Acesso em 16 de 05 de 2020

STRYKER. **Mutators**. Disponível em: https://stryker-mutator.io/. Acessao em 02 de 02 de 2020