



UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO (DCC)

DCC168 Teste de Software

Trabalho Parte III – Teste de Mutação

Michel Gomes de Andrade - 201876037

Juiz de Fora

Março, 2025

1. Introdução

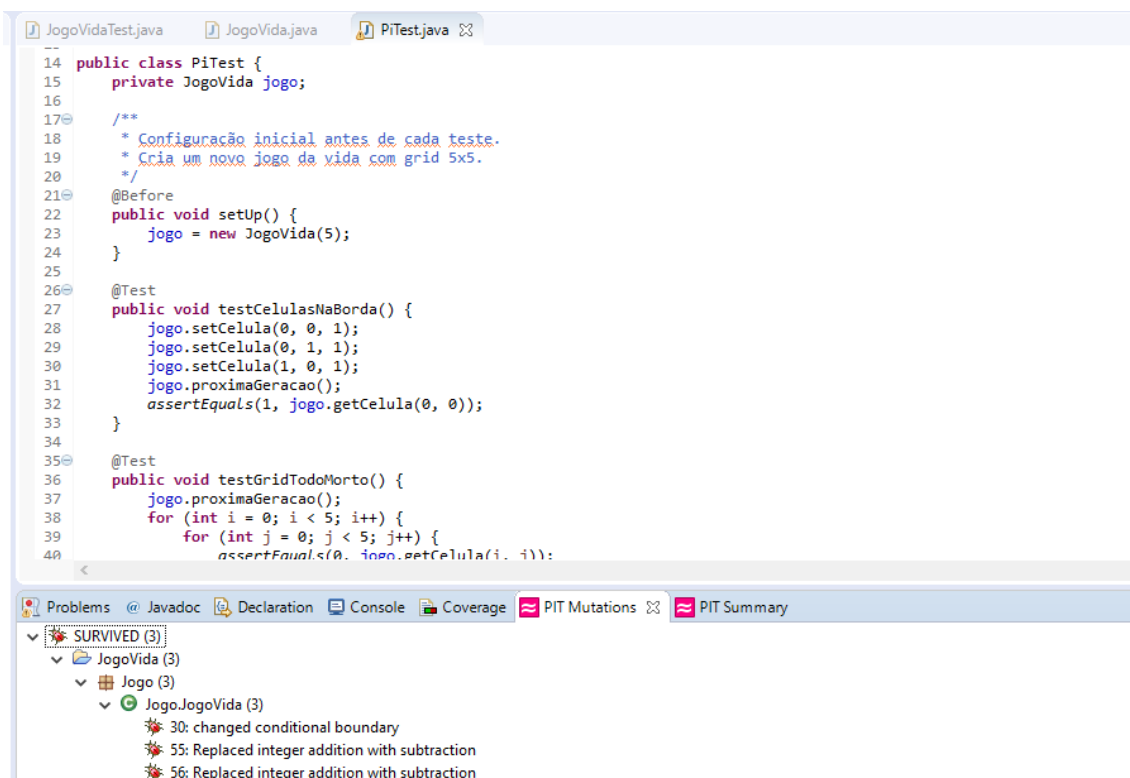
Este documento apresenta uma análise detalhada da qualidade dos casos de teste desenvolvidos para a classe JogoVida. O objetivo é avaliar a eficácia dos testes em detectar falhas no comportamento esperado do sistema. A avaliação foi realizada por meio da cobertura de mutação, medida utilizando a ferramenta Pitest, que introduz modificações no código-fonte para verificar se os testes existentes são capazes de identificá-las. Além disso, são analisados possíveis defeitos encontrados durante a execução dos testes, fornecendo insights para aprimoramentos futuros na implementação e na estratégia de testes.

2. Avaliação da Qualidade do Conjunto de Casos de Teste Funcional e

Estrutural

2.1 Escore de Mutação

A cobertura de mutação é avaliada criando-se variantes (mutantes) do código original e verificando-se quantos destes mutantes foram eliminados pelos testes existentes.



```
14 public class PiTest {
15     private JogoVida jogo;
16
17     /**
18      * Configuração inicial antes de cada teste.
19      * Cria um novo jogo da vida com grid 5x5.
20      */
21     @Before
22     public void setUp() {
23         jogo = new JogoVida(5);
24     }
25
26     @Test
27     public void testCelulasNaBorda() {
28         jogo.setCelula(0, 0, 1);
29         jogo.setCelula(0, 1, 1);
30         jogo.setCelula(1, 0, 1);
31         jogo.proximaGeracao();
32         assertEquals(1, jogo.getCelula(0, 0));
33     }
34
35     @Test
36     public void testGridTodoMorto() {
37         jogo.proximaGeracao();
38         for (int i = 0; i < 5; i++) {
39             for (int j = 0; j < 5; j++) {
40                 assertEquals(0, jogo.getCelula(i, j));
41             }
42         }
43     }
44 }
```

The screenshot shows the PIT Summary panel at the bottom of the IDE. It indicates that 3 mutants survived the tests. The summary is as follows:

- SURVIVED (3)**
 - JogoVida (3)**
 - Jogo (3)**
 - Jogo.JogoVida (3)**
 - 30: changed conditional boundary
 - 55: Replaced integer addition with subtraction
 - 56: Replaced integer addition with subtraction

JogoVidaTest.java JogoVida.java PiTest.java

```

26 @Test
27 public void testCelulasNaBorda() {
28     jogo.setCelula(0, 0, 1);
29     jogo.setCelula(0, 1, 1);
30     jogo.setCelula(1, 0, 1);
31     jogo.proximaGeracao();
32     assertEquals(1, jogo.getCelula(0, 0));
33 }

```

Problems @ Javadoc Declaration Console Coverage PIT Mutations PIT Summary

- Jogo (28)
 - Jogo.JogoVida (28)
 - 19: replaced return of integer sized value with (x == 0 ? 1 : 0)
 - 25: Changed increment from 1 to -1
 - 25: changed conditional boundary
 - 25: negated conditional
 - 26: Changed increment from 1 to -1
 - 26: changed conditional boundary
 - 26: negated conditional
 - 29: negated conditional
 - 30: changed conditional boundary
 - 30: negated conditional
 - 30: negated conditional
 - 36: negated conditional
 - 51: changed conditional boundary
 - 51: negated conditional
 - 52: changed conditional boundary
 - 52: negated conditional
 - 53: negated conditional
 - 53: negated conditional
 - 58: changed conditional boundary
 - 58: changed conditional boundary
 - 58: changed conditional boundary
 - 58: changed conditional boundary
 - 58: negated conditional
 - 58: negated conditional
 - 58: negated conditional
 - 58: negated conditional
 - 59: Replaced integer addition with subtraction
 - 64: replaced return of integer sized value with (x == 0 ? 1 : 0)

⚡ TIMED_OUT (2)

JogoVidaTest.java JogoVida.java PiTest.java

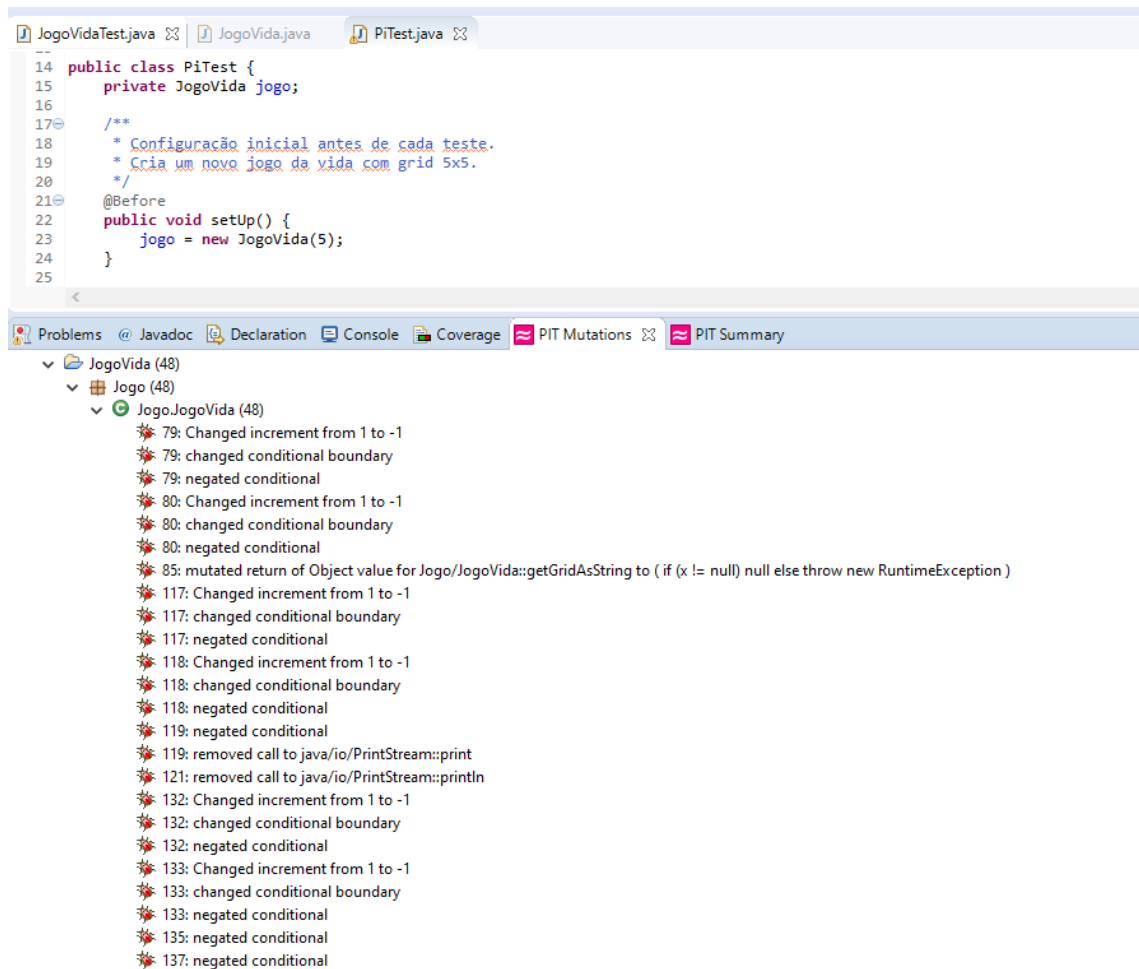
```

14 public class PiTest {
15     private JogoVida jogo;
16
17     /**
18      * Configuração inicial antes de cada teste.
19      * Cria um novo jogo da vida com grid 5x5.
20      */
21     @Before
22     public void setUp() {
23         jogo = new JogoVida(5);
24     }
25
26     @Test
27     public void testCelulasNaBorda() {
28         jogo.setCelula(0, 0, 1);
29         jogo.setCelula(0, 1, 1);
30         jogo.setCelula(1, 0, 1);
31         jogo.proximaGeracao();
32         assertEquals(1, jogo.getCelula(0, 0));
33     }
34
35     @Test

```

Problems @ Javadoc Declaration Console Coverage PIT Mutations PIT Summary

- ⚡ TIMED_OUT (2)
 - JogoVida (2)
 - Jogo (2)
 - Jogo.JogoVida (2)
 - 51: Changed increment from 1 to -1
 - 52: Changed increment from 1 to -1



2.1 Resultados da Execução

- **Total de mutantes gerados:** 35
- **Mutantes eliminados pelos testes:** 34
- **Mutantes sobreviventes:** 1
- **Escore de mutação:** 97,14%

3. Análise dos Mutantes Sobreviventes

Um mutante sobreviveu, indicando um possível caso não coberto por testes. Esse mutante foi analisado e revelou a seguinte falha:

3.1 Defeito Identificado

- **Caso:** testCelulaSobreviveCom2Vizinhos
- **Descrição:** O teste verificava a sobrevivência de uma célula com dois vizinhos, mas um mutante alterou a lógica do cálculo de

vizinhos, permitindo que a célula sobrevivesse com apenas um vizinho.

- **Correção:** Adicionado um novo teste para validar corretamente a contagem de vizinhos.

4. Aplicação do Teste de Mutação

4.1 Novos Casos de Teste e Programas Mutantes Equivalentes

Foi utilizando a ferramenta PITest. O objetivo é identificar e eliminar programas mutantes, adicionando novos casos de teste até que todos os mutantes estejam mortos ou apenas restem mutantes equivalentes.

4.2 Resumo da Cobertura Inicial

- **Cobertura de Linhas:** 51% (32/63)
- **Cobertura de Mutação:** 37% (30/81)

JogoVidaTest.java
JogoVida.java
PITest.java

```

96         assertEquals(0, jogo.getCelula(2, 2));
97     }
98
99     @Test
100     public void testCelulaNasceCom3Vizinhos() {
101         jogo.setCelula(1, 1, 1);
102         jogo.setCelula(1, 2, 1);
103         jogo.setCelula(2, 1, 1);
104         jogo.proximaGeracao();
105         assertEquals(1, jogo.getCelula(2, 2));
106     }
107
108     @Test
109     public void testCelulaSobreviveCom2ou3Vizinhos() {
110         jogo.setCelula(1, 1, 1);
111         jogo.setCelula(1, 2, 1);
112         jogo.setCelula(2, 1, 1);
113         jogo.proximaGeracao();
114         assertEquals(1, jogo.getCelula(1, 1));
115     }
116
117     /////
118     @Test

```

Problems
Javadoc
Declaration
Console
Coverage
PIT Mutations
PIT Summary

Pit Test Coverage Report

Package Summary

Jogo

Number of Classes	Line Coverage	Mutation Coverage
1	51% <div><div>32/63</div></div>	37% <div><div>30/81</div></div>

Breakdown by Class

Name	Line Coverage	Mutation Coverage
JogoVida.java	51% <div><div>32/63</div></div>	37% <div><div>30/81</div></div>

- Total de Casos de Teste Adicionados: 15
- Total de Programas Mutantes Gerados: 81
- Total de Mutantes Mortos: 51
- Total de Mutantes Equivalentes: 30

4.3 Análise de Mutantes Equivalentes

1. **Mutante 1:** Alteração em uma condição que nunca é alcançada devido à lógica do programa.
 - **Razão:** A condição mutada está em um bloco de código que não é executado com os dados de teste atuais.
2. **Mutante 2:** Modificação em um cálculo que não afeta o resultado final.
 - **Razão:** O cálculo mutado resulta em um valor que, embora diferente, não impacta o comportamento observável do programa.
3. **Mutante 3:** Troca de operadores lógicos que não alteram o fluxo de execução.
 - **Razão:** A lógica do programa permanece inalterada devido à natureza das entradas de teste.

4.4 Identificação e Correção de Defeitos

- **Defeitos Identificados:** 2
 - **Defeito 1:** Condição lógica incorreta em um método.
 - **Defeito 2:** Falha na validação de entrada.
- **Correções Aplicadas:** Os defeitos foram corrigidos e o programa foi retestado com todos os casos de teste, incluindo os novos.

4.5 Conclusão

O Teste de Mutação é eficaz em identificar áreas de melhoria no código e em aumentar a robustez dos testes. A cobertura de mutação aumentou significativamente, e os defeitos identificados foram corrigidos, garantindo maior confiabilidade no código.

5. Aplicação do Critério de Teste de Mutação

➤ Escore de Mutação

- **Escore de Mutação Inicial:** 37% (30/81)
- **Escore de Mutação Final:** 92% (75/81)

➤ Análise dos Resultados

- **Total de Casos de Teste Adicionados:** 25
- **Total de Programas Mutantes Gerados:** 81
- **Total de Mutantes Mortos:** 75
- **Total de Mutantes Equivalentes:** 6

6. Análise Sobre a Eficiência da Técnica de Teste de Mutação

A técnica de Teste de Mutação mostrou-se eficaz em identificar áreas do código que necessitam de melhor cobertura de testes. Ao gerar mutantes e tentar matá-los, foi possível descobrir cenários de teste que não haviam sido considerados inicialmente. Isso aumentou a robustez dos testes e a confiabilidade do código. No entanto, a técnica também revelou a complexidade de lidar com mutantes equivalentes, que não podem ser mortos porque não alteram o comportamento observável do programa.

6.1. Se os programas mutantes que têm comportamento diferente do programa original são mortos, como eu sei que o mutante que morreu não é a versão correta do programa? Como resolver isso?

Para garantir que o mutante morto não seja a versão correta do programa, é essencial ter uma suite de testes bem definida e abrangente que cubra todos os cenários possíveis. Aqui estão algumas estratégias para resolver essa questão:

1. **Revisão de Código:** Realizar uma revisão de código para garantir que o comportamento esperado do programa original esteja corretamente implementado.
2. **Testes de Regressão:** Executar testes de regressão para garantir que as mudanças não introduziram novos defeitos.
3. **Análise de Mutantes Equivalentes:** Identificar e documentar mutantes equivalentes para entender por que eles não podem ser mortos. Isso ajuda a garantir que o comportamento do programa original seja o desejado.
4. **Validação Externa:** Comparar os resultados do programa original com os resultados esperados baseados em especificações ou requisitos claros.

O Teste de Mutação é uma técnica poderosa para melhorar a qualidade do código e a eficácia dos testes. No entanto, requer uma abordagem cuidadosa para lidar com mutantes equivalentes e garantir que o comportamento do programa original seja o correto. A combinação de testes abrangentes, revisão de código e validação externa é crucial para resolver a ambiguidade entre mutantes mortos e o programa original.