Universidade de Brasília
Departamento de Ciência da Computação
Trabalho 2 - Métodos em Programação - 201600

Desenvolvimento Orientado a Testes
Relatório de testes

Aluno: Felipe Nascimento Rocha
Matrícula: 17/0050084
Professor: Jan Medonca Correa

Setembro
2019

Conteúdo

1 Introdução

A busca de novos métodos de desenvolvimento para melhoria de desempenho e eficiência sempre foi uma busca constante no cenário da programação. Dessa forma, o desenvolvimento orientado a testes foi inventado na Engenharia de *Software* com intuito de reduzir o tempo de *debug* e programação como um todo.

A ideia principal de forma superficial, consiste em codificar um módulo de testes antes de programar o programa principal, e começar a desenvolver para que o código passe nos testes realizados.

Neste trabalho, a ideia é desenvolver uma função que retorna o resultado de uma matriz baseado nas regras do jogo da velha, e os testes serão de acordo com essa função.

2 Testes realizados

2.1 Criação do Jogo

O primeiro teste criado consistia na validação da criação de um jogo dado uma matriz passada.

A ideia é que existe uma classe chamada JogoDaVelha que recebe uma matriz 3x3 que é copiada para dentro de um objeto instanciado. A função que recebe a matriz é o próprio construtor da classe.

O primeiro teste recebe uma matriz 3x3 apenas com 0's. Depois é realizado uma checagem dentro do objeto de JogoDaVelha para saber se os valores dentro do objeto são iguais aos passado pela matriz teste:

```
// Matriz teste:
    int jogoTeste[3][3] = {
        {0, 0, 0},
        {0, 0, 0},
        {0, 0, 0}};
// Matriz passada para o construtor da classe JogoDaVelha:
    JogoDaVelha jogo(jogoTeste);
    for (int i = 0; i <= 2; i++)
    {
        for (int j = 0; i <= 2; i++)
        {
// Teste: checa se os elementos da matriz sao 0s
            ASSERT_EQ(0, jogo.matriz[i][j]);
        }
    }
```

2.2 Obter Valores

Esse teste será responsável por uma função que irá auxiliar na coesão do código. A função *int getValor(int linha, int coluna)* é responsável por retornar o valor inteiro contido na linha i e coluna j da matriz do jogo. A ideia é poupar repetição de código.

O teste é realizado com um jogo teste válido de uma matriz do jogo da velha. Depois é realizado um teste para cada posição da matriz baseado no jogo teste passado, o teste será positivo caso todas os valores retornados pela função sejam iguais aos da matriz teste:

```
/* Teste para garantir que a funcao getValor(int linha, int coluna)
* funcione corretamente.
* Obs.: linhas e colunas comecam em 0 */

    int jogoTeste[3][3] = {
        {1, 2, 0},
        {1, 1, 2},
        {2, 0, 1}};
    JogoDaVelha jogo(jogoTeste);
    ASSERT_EQ(1, jogo.getValor(0, 0)); // Linha 1 coluna 1 = 1
    ASSERT_EQ(2, jogo.getValor(0, 1)); // Linha 1 coluna 2 = 2
    ASSERT_EQ(0, jogo.getValor(0, 2)); // Linha 1 coluna 3 = 0

    ASSERT_EQ(1, jogo.getValor(1, 0)); // Linha 2 coluna 1 = 1
    ASSERT_EQ(1, jogo.getValor(1, 1)); // Linha 2 coluna 2 = 1
    ASSERT_EQ(2, jogo.getValor(1, 2)); // Linha 2 coluna 3 = 2

    ASSERT_EQ(2, jogo.getValor(2, 0)); // Linha 3 coluna 1 = 2
    ASSERT_EQ(0, jogo.getValor(2, 1)); // Linha 3 coluna 2 = 0
    ASSERT_EQ(1, jogo.getValor(2, 2)); // Linha 3 coluna 3 = 1
```

2.3 Obter Resultado

O método *int getResultado()* é responsável por retornar os valores de um jogo da velha baseada em uma tabela de relacionamento, depois serão adicionados testes para cada estado que serão codificados até que passem em todos os testes:

```
/*
*          Especificacoes:
*              jogoImpossivel  -> retorna -2
*              jogoIndefinido  -> retorna -1
*              jogoEmpatado    -> retorna 0
*              vitoriaX        -> retorna 1
*              vitoriaO        -> retorna 2
*/
```

É válido ressaltar que o fluxo de codificação é:

```
1. Escrever um teste ->
2. Realizar o teste e falhar ->
3. Codificar suficientemente para que o teste passe ->
4. Escrever o proximo teste
```

2.3.1 Vitória X

O primeiro teste checa em uma matriz de inteiros válida do jogo da velha (em que 1's representam o jogador X e 2's representam o jogador O) se o vencedor é o jogador X.

No primeiro teste, vamos testar se a função *int getResultado()* retorna 1 em casos que o jogador X é vitorioso, para isso, foram codificadas funções que checam se as colunas, linhas, ou diagonais possuem o valor 1, e retornam positivo para esse caso:

```
/* Espera-se que caso o vencedor do jogo seja o X(1), a funcao retorne 1.
* Linha com X */
    int jogoTeste1[3][3] = {
        {1, 1, 1},
        {2, 2, 0},
        {0, 0, 0}};
    JogoDaVelha jogo1(jogoTeste1);
    ASSERT_EQ(1, jogo1.getResultado());

// Coluna com X
    int jogoTeste2[3][3] = {
        {0, 1, 0},
        {2, 1, 0},
        {2, 1, 0}};
    JogoDaVelha jogo2(jogoTeste2);
    ASSERT_EQ(1, jogo2.getResultado());

// Diagonais com X:

// Principal:
    int jogoTeste3[3][3] = {
        {1, 0, 0},
        {2, 1, 0},
        {2, 0, 1}};
    JogoDaVelha jogo3(jogoTeste3);
    ASSERT_EQ(1, jogo3.getResultado());
// Secundaria:
    int jogoTeste4[3][3] = {
        {0, 2, 1},
        {2, 1, 0},
```

```
        {1, 0, 0}};
    JogoDaVelha jogo4(jogoTeste4);
    ASSERT_EQ(1, jogo4.getResultado());
```

2.3.2 Vitória O

O segundo teste teve um fluxo análogo ao 2.3.1 apenas retornando 2 caso o vencedor seja o O:

```
    // Espera-se que caso o vencedor do jogo seja o O(2), a funcao retorne 2.
    ASSERT_EQ(2, jogo1.getResultado());
    ASSERT_EQ(2, jogo2.getResultado());
    ASSERT_EQ(2, jogo3.getResultado());
    ASSERT_EQ(2, jogo4.getResultado());
```

2.3.3 Empate

Teste realizado para checar se o jogo está empatado e retornar 0 para função 2.3. A lógica é: se o jogo está cheio,o jogador X não venceu nem o jogador vencedor O, o jogo está empatado, a função de vitoriaX 2.3.1 e vitoriaO 2.3.2 já existiam, só foi necessário criar um método que checa se o jogo está cheio, isto é, todas posições da matriz são diferentes de 0.

Depois são passadas diversas matrizes com jogos empatados e é realizado o teste na função 2.3:

```
// Testes:
{
int jogoTeste1[3][3] = {
        {1, 1, 2},
        {2, 2, 1},
        {1, 1, 2}};
JogoDaVelha jogo1(jogoTeste1);
ASSERT_EQ(0, jogo1.getResultado());
...
}
  // Jogo empatado:
{
if (jogoCheio() && !vitoriaO() && !vitoriaX())
    {
        return true;
    } else {
        return false;
    }
}
```

2.3.4 Jogo Indefinido

Um jogo indefinido é semelhante ao teste do empate 2.3.3 entretanto, o jogo não deve estar cheio nem deve ter vencedores, retornando -1 ao método obter resultado 2.3:

```
// Testes:
{
int jogoTeste1[3][3] = {
        {0, 1, 0},
        {0, 0, 0},
        {0, 0, 0}};
    JogoDaVelha jogo1(jogoTeste1);
    ASSERT_EQ(-1, jogo1.getResultado());
    ...
}
// Jogo indefinido:
{
if (!jogoCheio() && !vitoriaO() && !vitoriaX())
    {
        return true;
    } esle {
        return false;
    }
}
```

2.3.5 Jogo Impossível

Um jogo é impossível caso siga algumas regras:

```
1. Possua valores diferentes de 0, 1 ou 2;
2. Quantidade de O > Quantidade de X (Significa que o O
comecou a partida, ou jogou duas vezes seguidas);
3. Quantidade de X > 1 + Quantidade de O (X jogou duas vezes
seguidas);
4. Vitoria do X e Vitoria do O;
```

Dessa forma, é realizado um *loop* pela matriz que conta a quantidade de X, quantidade de O e se há algum valor diferente de 0,1 e 2; Depois é realizado o teste para saber se as quantidades de X e O são válidas e por último é checado se o X ou O são vitoriosos simultaneamente:

```
// Testes:
{
 // Todas posicoes X:
    int jogoTeste1[3][3] = {
        {1, 1, 1},
```

```
        {1, 1, 1},
        {1, 1, 1}};
    JogoDaVelha jogo1(jogoTeste1);
    ASSERT_EQ(-2, jogo1.getResultado());
    // Dois vencedores:

    int jogoTeste2[3][3] = {
        {2, 1, 2},
        {2, 1, 1},
        {2, 1, 1}};
    JogoDaVelha jogo2(jogoTeste2);
    ASSERT_EQ(-2, jogo2.getResultado());

    // O comecou partida:

    int jogoTeste3[3][3] = {
        {0, 0, 2},
        {0, 0, 0},
        {0, 0, 0}};
    JogoDaVelha jogo3(jogoTeste3);
    ASSERT_EQ(-2, jogo3.getResultado());

    // Valores diferentes de 0, 1 ou 2
    int jogoTeste4[3][3] = {
        {0, 3, 2},
        {0, 0, 1},
        {0, 0, 0}};
    JogoDaVelha jogo4(jogoTeste4);
    ASSERT_EQ(-2, jogo4.getResultado());
}

// Jogo impossivel:
{
    int quantidadeX = 0;
    int quantidadeO = 0;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            switch (getValor(i, j))
            {
            case 1:
                quantidadeX++;
                break;
            case 2:
                quantidadeO++;
```

```
                break;
            case 0:
                break;
            default:
                return true;
                break;
            }
        }
    }
    if (quantidadeO > quantidadeX || quantidadeX > 1 + quantidadeO)
    {
        return true;
    }
    if (vitoriaX() && vitoriaO())
    {
        return true;
    }
    return false;
}
```

A função obter resultado finalizada possui esse escopo:

```
// obter resultado:
{
    if (jogoImpossivel())
    {
        return -2;
    }
    else if (jogoIndefinido())
    {
        return -1;
    }
    else if (jogoEmpatado())
    {
        return 0;
    }
    else if (vitoriaX())
    {
        return 1;
    }
    else if (vitoriaO())
    {
        return 2;
    }
}
```

3 Conclusão

Por fim todos os testes são compilados com um arquivo *makefile* utilizando o framework *gtest* e passam:

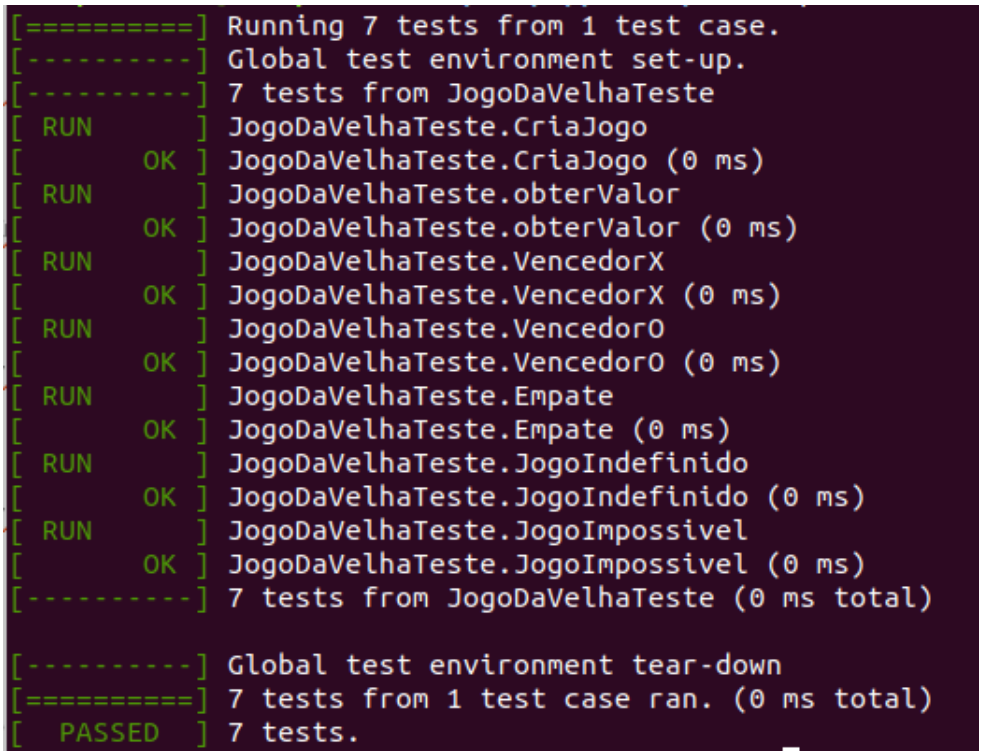

Figura 1: Execução do teste.

Na página do *GitHub* é possível encontrar um arquivo *readme* com as especificações de compilação, além de todo fluo de programação pelo meio do histórico de *commits*.

Bibliografia

Anotações Sala de aula.

Projeto no GitHub. Link GitHub do código do trabalho, 2019.

Framework de testes Google Test. Link GitHub, 2019.

Framework Valgrind. Link Página Principal, 2019.

CppCheck Link Página Principal, 2019.

Implementações na sala de aula.