IMPLANTAÇÃO DE TDD PARA PROGRAMAÇÃO COMPETITIVA

José Nivaldo da Silva Hypólito[1], Lucas Baggio Figueira[1], Débora Pelicano Diniz[1]

[1]Faculdade de Tecnologia de FATEC Ribeirão Preto (FATEC)

Ribeirão Preto, SP – Brasil

jose.hypolito@fatec.sp.gov.br,lucas.figueira@fatec.sp.gov.br, debora.diniz2@fatec.sp.gov.br

Resumo. *A maioria dos sistemas de avaliação de códigos utilizado por maratonas de programação não permitem uma fácil resolução dos problemas que possam haver no código e fazem com que o maratonista perca tempo decifrando o que está havendo no sistema ao invés de focar no desafio. Este trabalho apresentado neste artigo propõe demonstrar como implantar um sistema que utiliza TDD (Test Driven Development) para avaliar códigos e permitir um desenvolvimento e solução de exceções mais fácil ao usuário.*

Abstract. *Most code evaluation systems used by programming marathons do not allow an easy resolution of problems that may appear in the code and make the marathon runner waste time deciphering what is going on in the system instead of focusing on the challenge. This work proposes to demonstrate how to implement a system that uses TDD to evaluate codes and allow an easier development and solution of exceptions to the user.*

1. Introdução

Maratonas de programação são competições nas quais competidores ou grupos de competidores devem resolver diversos problemas de programação em um determinado período de tempo. Os desafios são geralmente propostos seguindo a metodologia TDD (*Test Driven Development*) com os desafios possuindo uma história e descrição sobre o problema que deve ser resolvido e exemplos de entrada e saída de dados esperada para que os competidores tenham um guia de como suas soluções devem ser desenvolvidas; as soluções são então avaliadas por um juiz e testadas com uma pilha de testes e, caso a solução passe em todos os testes, ela estará aprovada.

Problemas recorrentes neste tipo de competição são testes fixos e a falta de testes abertos. Muitas vezes poucos testes abertos são dados e, portanto, não cobrem todos os casos especiais que possam aparecer e que não são especificados na descrição do desafio, porém, estão nos casos de teste para validação da solução e, por muitas das vezes, por os testes de validação serem fixos, não se pode mostrar a exceção para os participantes, pois isso levaria à criação de exceções para a exceção, os chamados *hardcoded*, que são soluções fixas para aquela entrada de dados específica.

Considerando o que foi exposto, este artigo tem como objetivo demonstrar uso do TDD para desenvolver um sistema para validação de códigos para maratonas de programação.

Assim, o artigo está organizado da seguinte forma: Na Seção 2 estão apresentados os conceitos sobre TDD e o funcionamento das maratonas; na Seção 3 estão descritos métodos e exemplos de como implementar o TDD; na seção 4 estão as conclusões do trabalho e na seção 5 encontra-se as referências utilizadas.

2. Referencial Teórico

Nesta seção serão descritos os conceitos relacionados à TDD (*Test Driven Development*) e o funcionamento das maratonas de programação.

2.1. TDD (*Test Driven Development*)

De acordo com Beck (2002, p. 11) TDD (*Test Driven Development*) é:

> Desenvolvimento Guiado por testes é uma forma de administrar o medo durante a programação. Não quero falar de medo de uma forma ruim, mas medo no sentido legítimo de esse-é-um-problema-difícil-e-eu-não-consigo-ver-o-fim-a-partir-do-começo.

Ou seja, uma forma de se ter o começo e o fim de um problema para auxiliar o desenvolvimento do meio. Isto se resume em mostrar para o desenvolvedor o que deve ser recebido e como deve ser retornado e também mostrar como deve ser o retorno a cada caso de exceção e os requisitos de performance da aplicação. Porém se o desenvolvedor tivesse que, manualmente, colocar todos os dados de entrada e comparar as saídas da aplicação com as saídas esperadas não haveria ganho em performance de desenvolvimento, portanto, a melhor forma de se utilizar estas informações é criando testes automatizados que executam a aplicação, comparam os resultados e apresentam as informações de forma significante ao desenvolvedor.

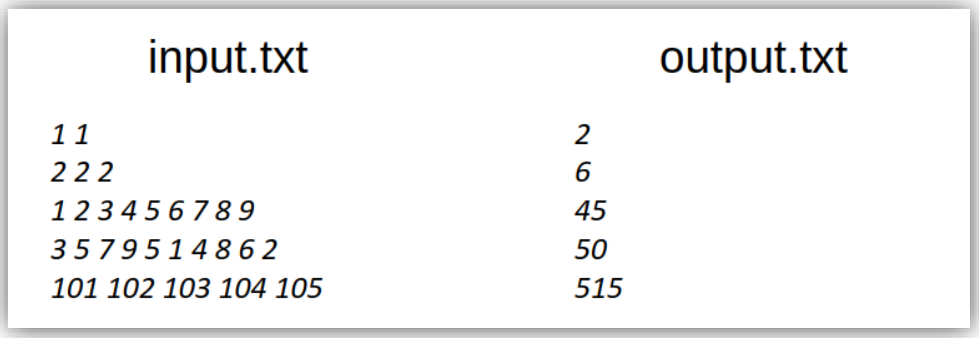
Os benefícios do TDD incluem: Eficiência. O ciclo de codificar e testar permite ao desenvolvedor encontrar problemas e falhas em seu código nos primeiros estágios do desenvolvimento, eliminando logo no início tais problemas; redução de introdução de erros. Quando é preciso alterar o código para corrigir problemas ou acrescentar funcionalidades há uma alta chance de serem introduzidos erros no código, com o TDD, caso alguma mudança introduza algum erro, um teste que antes passava parará de passar, indicando o problema. (WILLIAMS; MAXIMILIEN; VOUK, 2003)

2.2. Funcionamento das maratonas

Em muitas das competições de programação atuais devido ao modo que os testes para validação do código são feitos não há como dar uma resposta útil ao competidor, como em qual linha houve um erro ou qual foi a diferença entre o resultado retornado e o esperado.

A técnica usada na maioria das competições consiste em:

- Criar um arquivo de texto de entrada que será passado para o código do competidor e um arquivo de texto de saída que será comparado com o arquivo de saída do código do competidor para saber se o código retornou a solução esperada. (Exemplo Figura 2.1).



input.txt	output.txt
1 1	2
2 2 2	6
1 2 3 4 5 6 7 8 9	45
3 5 7 9 5 1 4 8 6 2	50
101 102 103 104 105	515

Figura 2.1 Exemplo de entrada e saída de um desafio de programação
Fonte: Autoria própria

- O usuário desenvolve seu código, com base nesse formato de entrada e saída. Na Figura 2.2 está apresentado um exemplo de código, no qual:
  - Nas linhas 1, 14 e 15 é feito a captura e o tratamento das exceções que possam ocorrer;
  - Nas linhas 2 a 7 são recebidos e transformados os dados para que sejam do tipo necessário para a solução;
  - A solução do desafio consiste somente das linhas 9 a 13.

```python
try:
    while True:
        entrada = input()

        lista = []
        for i in entrada.split():
            lista.append(int(i))

        x = 0
        for i in lista:
            x += i

        print(x)
except EOFError:
    pass
```

Figura 2.2 Exemplo de solução para o problema apresentado na Figura 1.1
Fonte: Autoria própria

- Usando um *script* de terminal o arquivo de entradas é passado para o código e a saída do código é salva em um arquivo resultados, então o arquivo de saídas e resultados são comparados para verificar se existem diferenças. (Exemplo Figura 2.3).

```
#!/bin/bash

python3 solution.py < inputs.txt > results.txt

diff outputs.txt results.txt
```

Figura 2.3 Passagem das entradas para a solução criada e comparação das saídas
Fonte: Autoria própria

Devido aos testes serem fixos não há como mostrar as entradas e saídas para o maratonista pois isto possibilitaria a criação de soluções específicas para aquele caso, conhecidas como *hardcoded*. (Exemplo Figura 2.4).

Figura 2.4 Exemplo de solução *hardcoded*
Fonte: Autoria própria

```python
if entrada == "2 2":
    print(4)
```

3. Metodologia

Nesta seção serão apresentadas diferentes formas de se implementar o TDD com ênfase em testes aleatórios.

3.1. *Framework* de testes

Utilizar um *framework* (biblioteca ou aplicação para auxiliar no desenvolvimento de uma tarefa) de testes proporciona grande vantagens ao desenvolvimento TDD, como: maior facilidade para desenvolver os testes; criação de testes complexos que testam outras características além do retorno; fácil manutenção; mensagens de erro informativas; portabilidade para outras plataformas e uma aproximação maior de como as aplicações são comumente desenvolvidas, entre outras vantagens.

Mas existe uma desvantagem: os *frameworks* são específicos às linguagens, então é preciso escrever novos testes para cada linguagem presente na competição.

Demonstração de uma implementação de um *framework* de testes utilizando Unittest (PYTHON, 2023):

- O competidor desenvolve uma função que receberá todos os dados como argumentos, removendo a necessidade de ler e transformar cada entrada de dados, e retorna o resultado de seu algoritmo. (Exemplo Figura 3.1).

```python
def soma(lista: List[int]) -> int:
    x = 0
    for i in lista:
        x += i
    return x
```

Figura 3.1 Exemplo de função
Fonte: Autoria própria

- São implementados testes fixos para realizar a validação inicial do código e testar todos os possíveis casos que possam existir no problema, os testes também exibem uma mensagem de erro com a saída esperada e a saída do usuário caso o teste falhe. (Exemplo Figura 3.2).

```python
class TestSolution(unittest.TestCase):
    def test_small_list(self):
        actual = soma([2, 2])
        self.assertEqual(4, actual, f"esperava 4, recebeu {actual}")

    def test_long_list(self):
        actual = soma([3, 5, 7, 9, 5, 1, 4, 8, 6, 2])
        self.assertEqual(50, actual, f"esperava 50, recebeu {actual}")

    def test_big_values(self):
        actual = soma([101, 102, 103, 104, 105])
        self.assertEqual(515, actual, f"esperava 515, recebeu {actual}")
```

Figura 3.2 Exemplo de testes fixos para validação inicial
Fonte: Autoria própria

- Para realizar a validação final são implementados uma solução ideal para a resolução do problema e um gerador de entradas aleatórias que cria os dados de

acordo como são definidos no desafio. (Exemplo Figura 3.3).

```
def check_soma(lista):
    return sum(lista)

from random import randint
def random_list():
    return [randint(0, 500) for _ in range(randint(100, 10**4))]
```

Figura 3.3 Exemplo de validação final
Fonte: Autoria própria

- E é criado um caso de teste que gera uma entrada aleatória, passa a para a solução ideal e para a solução do competidor e compara os resultados, provendo uma mensagem de erro caso algum teste falhe e evitando soluções *hardcoded* pois a cada teste as entradas serão diferentes. (Exemplo Figura 3.4).

```
def test_random(self):
    for i in range(20):
        with self.subtest(i=i):
            lista = random_list()
            expected = check_soma(lista)
            actual = soma(lista)
            mensagem = f"entradas {lista} \nesperava {expected}, recebeu {actual}"
            self.asserEqual(expected, actual, mensagem)
```

Figura 3.4 Caso de testes gerando uma entrada aleatória
Fonte: Autoria própria

3.2. Sem um *framework*

Com o crescimento do TDD e a visibilidade do quão importantes os testes são muitas linguagens estão implementando utilidades de testes nelas mesmas descartando a necessidade de *frameworks*. Outras pessoas preferem não utilizar *frameworks* pois eles estão sempre mudando e podem vir a ser descontinuados, deixando vulnerabilidades no sistema.

Implementar TDD sem o uso de um framework tem como vantagem o fato de não ser necessário aprender a trabalhar com um framework. Mas é possível citar algumas desvantagens como: dependendo de quantas ferramentas para criação de testes a linguagem possui, haverá uma maior dificuldade para criar testes apropriados em linguagens mais antigas; não há um padrão de desenvolvimento, cada desenvolvedor cria os testes da forma que achar melhor, o que pode dificultar a manutenção do código; as mensagens de erro dificilmente serão tão úteis quanto as de um *framework* de testes.

Demonstração de implantação em Python e C++:

- Python possui boas ferramentas para criação de testes, a principal delas é o *assert* que permite criar testes semelhantes aos do *framework* Unittest, porém funciona somente para valores booleanos, então para criar testes mais complexos e para testar outras condições de execução a estrutura do código pode se tornar mais

complicada. (Exemplo Figura 3.5).

```python
def test_equal():
    lista = random_list()
    expected = check_soma()
    actual = soma()
    mensagem = f"entradas {lista} \nesperava {expected}, recebeu {actual}"
    assert expected == actual, mensagem
```

Figura 3.5 Exemplo de testes em Python
Fonte: Autoria própria

- Por C++ não possuir utilidades para auxiliar o desenvolvimento de testes a criação deles torna-se exponencialmente mais complexa de acordo com a complexidade do que deve ser testado. (Exemplo Figura 3.6).

```cpp
void test_equal() {
    vector<int> lista = random_list();
    int expected = check_soma();
    int actual = soma();

    if (actual != expected) {
        std::cerr << "entrada";
        for (auto const& c : lista) std::cerr << ' ' << c;
        std::cerr << std::endl;

        std::cerr << "esperava " << expected << ", recebeu " << actual << std::endl;
    }
}
```

Figura 3.6 Exemplo de testes em C++
Fonte: Autoria própria

3.3. Shell script

Criar testes com *Shell script* permite testar implementações independentemente da linguagem em que foram implementadas.

O objetivo deste método é ser implementado ao método que foi mostrado na Seção 2.1de forma a ter poucas alterações no sistema original da maratona, mas ainda implementar TDD com testes aleatórios para permitir dar o retorno dos resultados aos competidores.

Trabalhar desta forma permite criar somente um teste para validar soluções implementadas em qualquer linguagem, mas não proporciona boas mensagens de erro e é preciso fazer um tratamento especial da entrada e da saída no desenvolvimento da solução para que ambas sejam de acordo com o formato esperado pelo modelo de testes, diminuindo assim a semelhança com o desenvolvimento de aplicações reais e reduzindo a eficiência dos competidores.

Os passos para se criar testes com *Shell script* são:

- Escreve-se o código para gerar as entradas aleatórias. O *script* ou aplicação pode ser escrito em qualquer linguagem. (Exemplo Figura 3.7).

```
#!/bin/bash

> inputs.txt
for ((i=$RANDOM%10000; i>0; i--)); do
    list=()
    for ((j=$RANDOM%50+51; j>0; j--)); do
        list+=($RANDOM)
    done
    echo ${list[@]} >> inputs.txt
done
```

Figura 3.7 Código para gerar entradas aleatórias
Fonte: Autoria própria

Então antes da execução dos testes, executa-se o *script* para gerar as entradas, passa-se as entradas geradas para a solução ideal e para a solução do competidor e ambas as saídas são geradas e comparadas. Também são capturadas quaisquer exceções que a saída do usuário apresente e as diferenças entre as duas saídas. (Exemplo Figura 3.8).

```
#!/bin/bash

./gerar_entradas
python check_solution.py < inputs.txt > check_results.txt
python solution.py < inputs.txt > results.txt 2> errors.txt
diff results.txt check_results.txt > diffs.txt
```

Figura 3.8 Script para validar a solução do competidor
Fonte: Autoria própria

- Caso haja diferenças entre as saídas todos os arquivos gerados (inputs.txt, check_results.txt, results.txt, errors.txt e diffs.txt) são enviadas ao competidor para que ele possa saber onde está o problema na sua solução.

3.4. Sites de desafios de programação

Existem muitos sites que propõem desafios de programação para os usuários resolverem na própria plataforma, dando ao usuário tudo o que ele necessita para desenvolver sua solução. Uma boa solução seria utilizar a plataforma do site que muitas das vezes é aberta para a comunidade contribuir e colocar seus próprios desafios ou criar uma plataforma semelhante para facilidade do desenvolvedor.

Alguns desses sites são: Codewars (CODEWARS, 2023), CodinGame (CODINGAME, 2023), LeetCode (LEETCODE, 2023), entre vários outros.

Demonstração do Codewars (Vide Figura 3.9):

1. Área para o desenvolvedor escrever o seu código;
2. Caixas de opção para o usuário escolher a linguagem na qual deseja desenvolver e a versão dela;
3. Área com testes de exemplo para o usuário conferir que tipos de dados ele receberá e o que deve retornar;
4. Aba de instruções para o usuário conferir a descrição do desafio
5. Área para o usuário conferir os resultados da execução de seu código, com mensagens de sucesso e erro de forma explicativa e intuitiva;
6. Botões testar e validar, para o usuário testar sua solução contra os testes de exemplo e os testes de validação com os testes aleatórios.

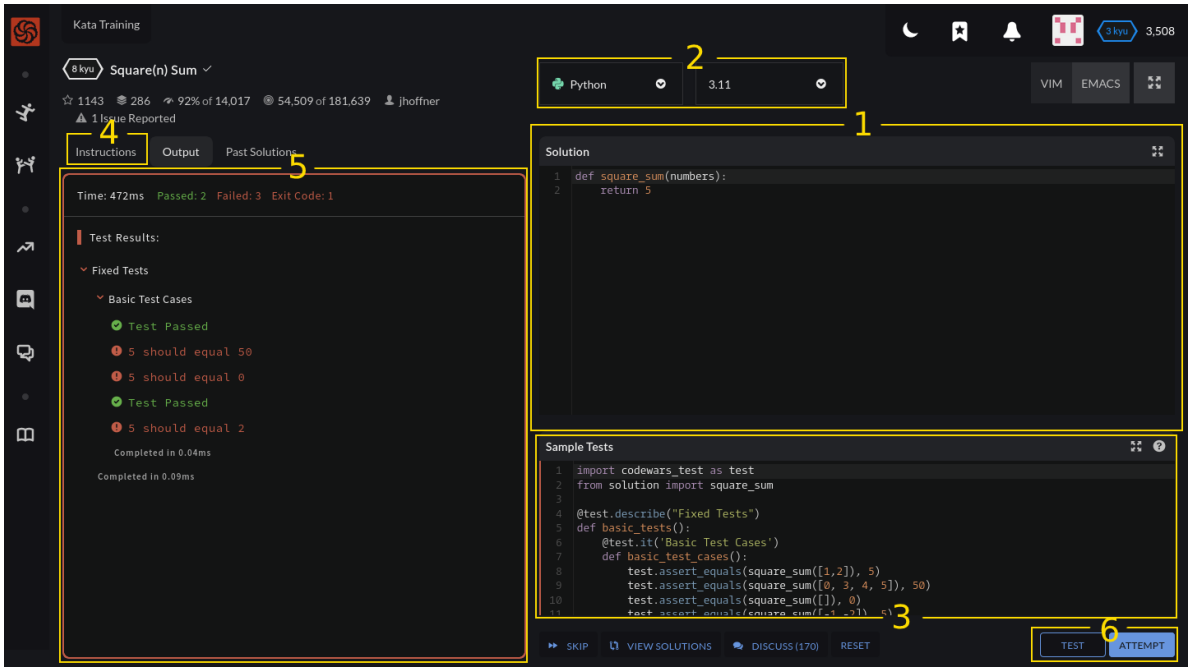


Figura 3.9 Tela do Codewars
Fonte: Autoria própria

Todas as entradas e saídas são visíveis ao usuário, ele pode ver a entrada simplesmente fazendo um print e a saída esperada e a de sua solução sempre será mostrada nas mensagens de erro, o que garante que o usuário não crie soluções *hardcoded* são os testes aleatórios, que estão presentes em todos os desafios.

4. Conclusões

Conclui-se que para validar as soluções dos desafios em maratonas de programação é necessário haver testes, porém, quando não feitos corretamente, podem apresentar vulnerabilidades, como soluções hardcoded e não retornar mensagens de erro úteis ao competidor.

Com os exemplos apresentados é possível ver e entender a praticidade e importância do TDD para o desenvolvimento de códigos, principalmente no meio da programação competitiva onde ele proporciona mensagens de erro mais explicativas ao

usuário e quando utilizado com testes aleatórios remove a necessidade de testes ocultos e impossibilita soluções *hardcoded*.

5. Referências

BECK, K. (2002). Test Driven Development: By Example, Addison-Wesley Professional.

WILLIAMS, L., MAXIMILIEN E. M. AND VOUK M. (2003) Test-Driven Development as a Defect-Reduction Practice. IEEE.

CODEWARS. (2023). Disponível em: <https://www.codewars.com>. Acesso em: 14/05/2023

CODINGAME. (2023). Disponível em: <https://www.codingame.com>. Acesso em: 14/05/2023

LEETCODE. (2023). Disponível em: <https://leetcode.com/>. Acesso em: 14/05/2023

PYTHON. (2023). Unittest. Disponível em: <https://docs.python.org/3.11/library/unittest.html>. Acesso em: 14/05/2023