UNIVERSIDADE FEDERAL DO ESTADO DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS ELETRÔNICOS
PROGRAMAÇÃO DE SISTEMAS ELETRÔNICOS
WUERIKE HENRIQUE DA SILVA CAVALHEIRO

TESTES UNITÁRIOS

Neste documento relata-se os resultados da aplicação de testes unitários na resolução do Bar dos Filósofos [1], software previamente desenvolvido nesta mesma disciplina.

A filosofia dos testes unitários prega a execução de testes nas menores unidades possíveis de um determinado código, garantindo o correto funcionamento e tratamento de erro das partes pequenas pode-se então esperar uma maior confiabilidade na execução do software como um todo, porém, não eliminado outros possíveis testes como os testes funcionais [2].

O código utilizado para o desenvolvimento desta atividade foi desenvolvido através do paradigma de orientação a objetos, desta forma as menores partes a serem testadas são os métodos das classes envolvidas.

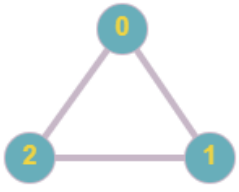
Elaborou-se testes para verificar as pré e pós condições para cada método passível a ser testado, isto é, não foram testados métodos que não alteram o estado da aplicação, como por exemplo os métodos que apenas realizam um print no terminal.

O código a ser testado possui duas classes chamadas de Philosopher e Table, a classe Philosopher representa cada filosofo e suas possíveis ações enquanto a classe Table lida com as ações que envolvem a interação entre os filósofos, de forma que, sendo essa uma aplicação que é executada em múltiplas threads, a classe Table é a responsável por lidar com as condições de disputa e garantir um controle justo dos recursos compartilhados.

Para a realização dos testes utilizou-se o framework Google Test [3], uma biblioteca para execução de testes unitários aplicáveis às linguagens C/C++. De forma mais específica foram utilizadas as Fixtures Classes, classes que são executadas pelo Google Test a cada teste, realizando as configurações necessárias e evitando assim linhas repetitivas entre testes que necessitem uma mesma configuração inicial.

Na Fixture Class criada para testar a classe Philosopher foram definidas as condições iniciais mínimas para a execução dos testes, onde foram instanciados 3 filósofos e definidas as relações de vizinhança entre eles, formando o grafo abaixo, sendo os filósofos representados pelos vértices enquanto as arestas determinam as relações de vizinhança.

Figura 1 – Relacionamento entre filósofos para os testes



Fonte: O Autor.

A Figura 2 abaixo apresenta um dos métodos da classe Philosopher que foi testado, este método é responsável por receber um vetor de instâncias de Philosopher que representam todos os filósofos que são vizinhos de um determinado filósofo.

Figura 2 – Método set neighbors.

```
/*
 * Set which are the this philosopher's neighbors
 */
void Philosopher::set_neighbors (vector<Philosopher *> neighbors)
{
    this->neighbors = neighbors;
    this->num_bottle_max = neighbors.size();
}
```

Fonte: O Autor.

O método da Figura 3 que executa os testes necessários nesta classe, porém, devido a utilização da Fixture Class a relação de vizinhança é definida mesmo antes deste teste executar, desta forma, inicialmente cria-se um vetor de Philosopher vazio que é então definido como sendo os vizinhos do filósofo p0.

Como visível na figura 2, ao receber esse vetor o método o salva em um atributo e também altera o atributo de número máximo de garrafas compartilhadas, se um filósofo possui dois vizinhos, possuirá também duas garrafas compartilhadas, uma com cada vizinho. Desta forma o próximo passa na rotina de testes é confirmar que o numero máximo de garrafas que p0 compartilha é igual a zero.

O recebimento de um vetor vazio, apesar de funcionar como o esperado, pode se considerar como uma condição que não deveria ocorrer, já que um filósofo deve ter vizinhos. Esse erro é tratado fora do método, antes de ser chamado, onde garante-se que uma matriz que represente um grafo, como na Figura 1, foi recebida e corretamente interpretada. Entretanto, para fins do correto funcionamento deste método como uma unidade, poderia verificar se o vetor é vazio, caso confirmado lançar uma Exception que represente um argumento invalido e então lidar com a exceção externamente ao método.

Figura 3 – Teste do método set neighbors.

```
/*
*   Tests the set_neighbors method
*/
TEST_F(PhilosopherTest, SetNeighborsTest) {
    // create a empty vector of neighbors
    vector<Philosopher *> neighbors;
    // Sets zero neighbors to p0
    p0->set_neighbors(neighbors);
    // Check if p0 has a bottle for each neighbor
    ASSERT_EQ(p0->num_bottle_max, 0);

    // Add p1 and p2 as p0 neighbors
    neighbors.push_back(p1);
    neighbors.push_back(p2);
    p0->set_neighbors(neighbors);
    // Check if p0 has a bottle for each neighbor
    ASSERT_EQ(p0->num_bottle_max, neighbors.size());
    // Check if p0 has access to its neighbors
    ASSERT_EQ(p0->neighbors[0]->philosopher_id, p1->philosopher_id);
    ASSERT_EQ(p0->neighbors[1]->philosopher_id, p2->philosopher_id);
}
```

Fonte: O Autor.

Na continuidade da execução deste se da pela adição dos filósofos p1 e p2 ao vetor de vizinhos e posteriormente este vetor é adicionado ao p0. Neste momento as asserções são para confirmar que agora o filósofo p0 tem dois vizinhos e que inclusive pode acessar informações de deles, fator primordial para a execução do código.

Demais métodos da classe Philosopher seguiram a mesma filosofia de preparação do setup, chamada do método a ser testado e então asserções confirmando o correto impacto do método sobre os estados dos atributos.

Enquanto os testes da classe Philosopher foram focados em testar uma instancia da classe por vez, de forma sequencial, na classe Table testou-se o comportamento de diversas instancias de filósofos sendo executados de forma concorrente, um em cada thread.

Como existem muitas pré e pós condições nas execuções dos métodos, o método Request Handler da classe Table não pode ser executado em múltiplas threads por si só, desta forma criou-se uma função auxiliar Figura 4, que executa os métodos Thirsty e Drinking da classe Philosopher, de forma que Request Handler é executado dentro de Thirsty enquanto Drinking é necessário para a sequência da execução concorrente.

Figura 4 – Função auxiliar para teste multithreads.

```
/*
*   Auxiliary function to test the fair concurrency in multithreading
*/
void ThirstyDrinkAux(Philosopher* philosopher){
    philosopher->thirsty();
    philosopher->drinking();
}
```

Fonte: O Autor.

A Figura 5 apresenta os testes aplicados no método Request Handler da classe Table, sendo esse o único método a ser testado nesta classe já que os outros métodos apenas realizam prints no terminal.

Figura 5 – Teste do método request handler.

```
/*
*   Tests the request_handler method in multi thread
*/
TEST_F(TableTest, RequestHandlerMultiThreadTest) {
    // Assert that philosophers arent holding bottles
    ASSERT_EQ(p0->holded_bottle.size(), 0);
    ASSERT_EQ(p1->holded_bottle.size(), 0);

    // Starts the multi threading with p0 and p1
    thread t0 (ThirstyDrinkAux, p0);
    thread t1 (ThirstyDrinkAux, p1);
    t0.join(); t1.join();

    // Assert only p0 and p1 drinkend so far
    ASSERT_EQ(table->drink_session.size(), 2);
    ASSERT_EQ(table->drink_session[0], p0->philosopher_id);
    ASSERT_EQ(table->drink_session[1], p1->philosopher_id);
    // After get the bottles, they're released in drinking method
    ASSERT_EQ(p0->holded_bottle.size(), 0);
    ASSERT_EQ(p1->holded_bottle.size(), 0);

    // Starts the multi threading with all philosophers
    thread t10 (ThirstyDrinkAux, p0);
    thread t11 (ThirstyDrinkAux, p1);
    thread t12 (ThirstyDrinkAux, p2);
    t10.join(); t11.join(); t12.join();
    // p0 and p1 have drinked before, so the first to drink now is p2
    // when p2 drinks a drinking session in complete and drink_session is cleared
    // this way drink_session now should record p0 and p1 id's like before
    ASSERT_EQ(table->drink_session.size(), 2);

    // Assert the 2 itens in drink session are p0 and p1 no matter the order
    vector<int>::iterator position;
    position = find(table->drink_session.begin(), table->drink_session.end(), p0->philosopher_id);
    ASSERT_NE(position, table->drink_session.end());
    position = find(table->drink_session.begin(), table->drink_session.end(), p1->philosopher_id);
    ASSERT_NE(position, table->drink_session.end());

}
```

Fonte: O Autor.

Neste teste manteve-se o numero de filósofos e a relação entre eles definida na Figura 1, executou-se então a função da Figura 3 passando cada filósofo como argumento e em uma thread diferente, buscando verificar o correto funcionamento de Request Handler, e por sua vez, o correto compartilhamento de recursos.

As asserções mais importantes neste teste são as três últimas, onde o atributo que registra quantos filósofos participaram da sessão de bebedeira atual deve ser igual a dois, bem como os

dois filósofos que participaram dessa sessão devem ser os filósofos p0 e p1, não importando a ordem de quem bebeu primeiro.

Estes resultados devem ser assim, pois, inicialmente executam-se apenas os filósofos p0 e p1 de forma concorrente, então ambos são registrados na sessão de bebedeira e são feitas as asserções que confirmam o funcionamento. Na sequência, são executados todos os 3 filósofos, e mesmo que as threads de p0 e p1 sejam executadas primeiro, obrigatoriamente o filósofo p2 deve ser o primeiro a ter acesso aos recursos, já que ele é o único que ainda não esta presente na sessão de bebedeira numero um, ao finalizar sua execução p0 e p1 são então executados, iniciando assim a segunda sessão de bebedeira, e resultando nos estados descritos anteriormente.

Para as duas classes envolvidas, foram escritos um total de 9 testes e conforme os testes apresentados, cada teste executa diversas asserções buscando confirmar todos os estados alterados por determinado método. Devido a forte interação entre métodos, alguns testes envolveram chamadas de funções adicionais, conforme demonstrado na Figura 4 e 5, não sendo completamente fiel ao intuito dos testes unitários de sempre testar as unidades mínimas.

Como visto, alguns métodos recebem parâmetros que não fazem sentido para a aplicação sem acusar erro, como um vetor de vizinhos vazio, comportamento que se repete em alguns dos outros métodos testados. Durante o desenvolvimento do software testado as pré condições foram tratadas antes da chamada destes métodos, garantindo sempre a correta utilização, entretanto, a solução mencionada anteriormente poderia ser aplicada a todos estes casos, onde em cada método poderia se verificar o atributo recebido e em caso de inválido, lançar uma exceção, avisando assim o método de origem que algo não estava dentro dos conformes.

Referências

[1] K. M. CHANDY and J. MISRA. The Drinking Philosophers Problem. ACM Transactions on Programming Languages and Systems, Vol. 6, No. 4, October 1984, Pages 632-646. Disponível em: https://www.cs.utexas.edu/users/misra/scannedPdf.dir/DrinkingPhil.pdf.

[2] https://medium.com/@dayvsonlima/entenda-de-uma-vez-por-todas-o-que-s%C3%A3o-testes-unit%C3%A1rios-para-que-servem-e-como-faz%C3%AA-los-2a6f645bab3

[3] https://github.com/google/googletest/blob/master/googletest/docs/primer.md