TESTE UNITÁRIO

https://replit.com/@engsoftmoderna/ExemploTesteUnidade-JavaScript#stack.test.js

stack.js

A classe tem métodos básicos para manipulação de uma pilha, como push (adicionar elemento no topo), pop (remover elemento do topo), isEmpty (verificar se a pilha está vazia) e size (retornar o tamanho da pilha).

Construtor stack:

O construtor inicializa a pilha (this.items) como um array vazio.

Método size:

Retorna o comprimento (número de elementos) da pilha.

Método isEmpty:

Retorna true se a pilha estiver vazia, ou seja, se o comprimento do array items for igual a zero.

Método push:

Adiciona um elemento ao topo da pilha. Utiliza o método push do array para inserir o elemento no final.

Método pop:

Remove e retorna o elemento no topo da pilha.

Verifica se a pilha está vazia antes de tentar fazer o pop. Se estiver vazia, lança uma exceção indicando que a pilha está vazia.

Erro no Construtor:

O construtor está usando function stack() em vez de function Stack(). Geralmente, por convenção, os nomes de classes em JavaScript começam com letra maiúscula.

Correção: Altere function stack() para function Stack().

Erro na exceção de pilha vazia no método pop:

A exceção é lançada como uma string (throw "pilha vazia"). Seria mais apropriado lançar uma instância de Error.

Correção: Modifique throw "pilha vazia" para throw new Error("pilha vazia").

Erro de escopo no beforeEach do arquivo stack.test.js:

Falta declarar a variável s antes de usá-la nos testes. Isso pode causar comportamento inesperado em alguns ambientes.

Correção: Adicione let s; antes do beforeEach(() => {...}).

stack.test.js

Teste 'pilha vazia':

Verifica se a pilha está vazia usando a função isEmpty().

Espera-se que isEmpty() retorne verdadeiro.

Teste 'pilha não vazia':

Adiciona um elemento à pilha e verifica se a função isEmpty() retorna falso.

Espera-se que isEmpty() retorne falso.

Teste 'pilha com 3 elementos':

Adiciona três elementos à pilha e verifica se a função size() retorna 3.

Espera-se que size() retorne 3.

Teste 'pilha após desempilhar dois elementos':

Adiciona três elementos à pilha, desempilha um elemento e verifica se o próximo desempilhado é o esperado.

Espera-se que o segundo pop() retorne 20.

Teste 'pilha com exceção de pilha vazia ao desempilhar':

Adiciona um elemento à pilha, desempilha esse elemento e tenta desempilhar novamente.

Espera-se que a segunda tentativa de pop() lance uma exceção.

Teste 'pilha vazia':

Acerto: O teste parece correto, verificando se a pilha está vazia.

Observação: O ponto e vírgula extra no final do expect(s.isEmpty()).toBeTruthy(); pode ser removido.

Teste 'pilha não vazia':

Acerto: O teste verifica se a pilha não está vazia após adicionar um elemento.

Observação: O ponto e vírgula extra no final do expect(s.isEmpty()).toBeFalsy(); pode ser removido.

Teste 'pilha com 3 elementos':

Acerto: Este teste verifica se a função size() retorna corretamente o número de elementos na pilha.

Observação: Não há observações adicionais.

Teste 'pilha após desempilhar dois elementos':

Acerto: O teste verifica se a pilha se comporta corretamente ao desempilhar dois elementos.

Observação: Não há observações adicionais.

Teste 'pilha com exceção de pilha vazia ao desempilhar':

Acerto: Este teste verifica se a exceção é lançada corretamente ao tentar desempilhar de uma pilha vazia.

Observação: Não há observações adicionais.

Conceitos sobre testes de unidade

BookSearchTest.java

Acertos:

Comentários: O código inclui comentários que explicam o propósito do arquivo e a referência ao livro/texto do capítulo 8 sobre testes de Engenharia de Software Moderna. Comentários são sempre úteis para entender o contexto e a finalidade do código.

Mocking com Mockito: O uso do Mockito para criar mocks do BookService é apropriado para isolar a unidade de teste e garantir que os testes se concentrem apenas na lógica da classe BookSearch.

Testes Unitários: Existem três métodos de teste (testGetBook, testGetAnotherBook, testGetNullBook) que verificam diferentes cenários, proporcionando uma cobertura básica de testes.

Configuração de Mocks: A configuração dos mocks no método init usando o when e thenReturn está correta e parece abranger os casos relevantes para testar a classe BookSearch.

Asserts: O uso de assertEquals é adequado para verificar se os resultados esperados são iguais aos resultados reais.

Possíveis Melhorias:

Testes Independentes: Embora os testes estejam bem escritos, é importante garantir que cada teste seja independente dos outros. Isso significa que a execução de um teste não deve depender do resultado de outro. Certifique-se de que cada teste não está afetando o estado compartilhado.

Nomes de Testes Descritivos: Os nomes dos métodos de teste são informativos, mas poderiam ser mais descritivos. Em vez de testGetBook, seria mais claro se fosse algo como testGetBookWithValidId. Isso tornaria mais fácil entender o que o teste está validando.

Teste de Exceções: Pode ser útil adicionar testes para casos em que exceções podem ser lançadas. Por exemplo, um teste para verificar o comportamento quando service.search retorna uma exceção em vez de um livro esperado.

Comentários no Código dos Testes: Adicionar alguns comentários explicativos no código dos testes pode ser útil, especialmente para futuras referências ou para outros desenvolvedores que possam trabalhar no código.

Inclusão de Asserts Adicionais: Pode ser benéfico adicionar asserts adicionais para garantir que outros aspectos dos objetos retornados, além dos títulos, estejam corretos.

Limpeza de Recursos: Considere adicionar um método @After para limpar/reciclar recursos após a execução dos testes.

TestRunner.java

Acertos:

Comentários: Os comentários no início do arquivo fornecem informações úteis sobre o propósito do código e sua relação com o capítulo 8 sobre testes em Engenharia de Software Moderna.

Uso do JUnitCore: O uso da classe JUnitCore para executar os testes é apropriado, e a obtenção do resultado (Result) permite que você avalie o desempenho dos testes.

Mensagens de Saída: As mensagens de saída fornecem informações sobre o tempo de execução, a quantidade de testes executados e se foram bem-sucedidos ou não. Isso pode ser útil para uma rápida análise do resultado dos testes.

Tratamento de Falhas: O código inclui uma seção que itera sobre as falhas, se houver alguma, e imprime informações sobre cada falha. Isso facilita a identificação dos problemas encontrados durante a execução dos testes.

Instruções de Não Alterar o Código: A inclusão da mensagem "IMPORTANTE: não altere o código abaixo" fornece uma diretriz clara para os usuários de que esse código não deve ser modificado.

Possíveis Melhorias:

Método main() Corrigido: O método main() está ausente do argumento String[] args, que é padrão para um método main() em Java. Deve ser corrigido para public static void main(String[] args).

Uso de System.exit(): Em ambientes de teste e algumas situações, pode ser melhor usar System.exit() para indicar o status de execução, por exemplo, System.exit(0) para sucesso e System.exit(1) para falha.

Mensagens mais Descritivas: As mensagens de resultado podem ser mais descritivas para fornecer mais detalhes sobre os resultados dos testes.

Tempo de Execução: O tempo de execução está sendo impresso em milissegundos. Dependendo da situação, pode ser mais útil converter isso para segundos para melhor compreensão.

Manuseio de Exceções: Considere adicionar um bloco try-catch ao redor da execução dos testes para lidar com exceções, caso ocorram.

TDD

ShoppingCartTest.java

Acertos:

Comentários Descritivos: Os comentários fornecem informações úteis sobre o propósito do código, mencionando que é um exemplo de TDD na "fase de Estado Vermelho".

Método de Teste Adequado: O método de teste testAddGetTotal parece adequado para testar a funcionalidade de adicionar livros ao carrinho e obter o total.

Uso de assertEquals: O uso do método assertEquals é apropriado para verificar se o valor retornado por cart.getTotal() é igual ao valor esperado.

Possíveis Melhorias:

Mais Cenários de Teste: O teste atual cobre um cenário básico, mas seria benéfico adicionar mais cenários de teste para garantir uma cobertura mais completa, como testar o comportamento do carrinho quando nenhum livro é adicionado.

Mensagens de Falha Descritivas: As mensagens de falha podem ser mais descritivas para indicar exatamente o que falhou. Por exemplo, "Falha ao calcular o total do carrinho" seria mais informativo do que a mensagem padrão do assertEquals.

Inclusão de Comentários no Código: Adicionar alguns comentários explicativos no código do teste pode ser útil, especialmente para futuras referências ou para outros desenvolvedores que possam trabalhar no código.

Testes de Limites: Considere adicionar testes para situações de limite, como adicionar livros com preços zero ou negativos, ou testar o comportamento do carrinho quando um livro é removido.

Testes de Exceções: Pode ser útil adicionar testes para cenários em que exceções podem ser lançadas, por exemplo, ao tentar obter o total de um carrinho vazio.

TestRunner.java

Acertos:

Comentários Descritivos: Os comentários fornecem informações úteis sobre o propósito do código e a orientação para não alterar o código abaixo. Esses comentários são claros e úteis para outros desenvolvedores que possam interagir com o código.

Uso do JUnitCore: O uso da classe JUnitCore para executar os testes é apropriado, e a obtenção do resultado (Result) permite que você avalie o desempenho dos testes.

Mensagens de Saída: As mensagens de saída fornecem informações sobre o tempo de execução, a quantidade de testes executados e se foram bem-sucedidos ou não. Isso pode ser útil para uma rápida análise do resultado dos testes.

Tratamento de Falhas: O código inclui uma seção que itera sobre as falhas, se houver alguma, e imprime informações sobre cada falha. Isso facilita a identificação dos problemas encontrados durante a execução dos testes.

Instruções de Não Alterar o Código: A inclusão da mensagem "IMPORTANTE: não altere o código abaixo" fornece uma diretriz clara para os usuários de que esse código não deve ser modificado.

Possíveis Melhorias:

Método main() Corrigido: O método main() está ausente do argumento String[] args, que é padrão para um método main() em Java. Deve ser corrigido para public static void main(String[] args).

Uso de System.exit(): Em ambientes de teste e algumas situações, pode ser melhor usar System.exit() para indicar o status de execução, por exemplo, System.exit(0) para sucesso e System.exit(1) para falha.

Mensagens mais Descritivas: As mensagens de resultado podem ser mais descritivas para indicar exatamente o que falhou. Por exemplo, "Falha nos testes da classe ShoppingCartTest" seria mais informativo do que a mensagem padrão do assertEquals.

Tempo de Execução: O tempo de execução está sendo impresso em milissegundos. Dependendo da situação, pode ser mais útil converter isso para segundos para melhor compreensão.

Manuseio de Exceções: Considere adicionar um bloco try-catch ao redor da execução dos testes para lidar com exceções, caso ocorram.