**Scenaries of tests Controller**

|  |  |  |
| --- | --- | --- |
| **Name** | **Class** | **Scenario** |
| setUpStage1() | ControllerTest | Save a valid product and verify it is stored correctly. |
| setUpStage2() | ControllerTest | Attempt to save a product with an existing code. |
| setUpStage3() | ControllerTest | Save a product with an empty description. |
| setUpStage4() | ControllerTest | Save a valid customer and verify it is stored correctly. |
| setUpStage5() | ControllerTest | Attempt to save a customer with an existing ID. |
| setUpStage6() | ControllerTest | Save an order with an existing product and customer. |
| setUpStage7() | ControllerTest | Attempt to save an order with an existing order number. |
| setUpStage8() | ControllerTest | Attempt to save an order with an empty product list. |
| setUpStage9() | ControllerTest | Modify an existing product and verify it is updated correctly. |
| setUpStage10() | ControllerTest | Attempt to modify a non-existent product and catch the exception. |
| setUpStage11() | ControllerTest | Remove a valid order and verify it no longer exists. |
| setUpStage12() | ControllerTest | Attempt to remove an order that does not exist. |
| setUpStage13() | ControllerTest | Undo the last action of saving a product. |
| setUpStage14() | ControllerTest | Dispatch a product from an existing order and verify the product. |
| setUpStage15() | ControllerTest | Verify that a product exists in the system. |
| setUpStage16() | ControllerTest | Validate the password of an existing customer. |
| setUpStage17() | ControllerTest | Undo the last action of saving a customer. |
| setUpStage18() | ControllerTest | Undo the last action when no previous actions exist. |
| setUpStage19() | ControllerTest | Dispatch multiple products for a single user from an order. |
| setUpStage 20() | ControllerTest | Verify that a product does not exist in the system. |
| setUpStage 21() | ControllerTest | Calculate the total price of multiple products. |
| setUpStage 22() | ControllerTest | Calculate the total price when no products are present. |
| setUpStage 23() | ControllerTest | List products when products exist in the system. |
| setUpStage 24() | ControllerTest | List products when no products exist. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective: Validate the save products in a valid state, also with a duplicate code and with an empty description.** | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Expected Outcome** |
| Controller | saveProduct() | setUpStage1() | String code: "001", String name: "Mcqueen", int price: 16000, String description: "Cars3" | Returns “true”, saves the product and allows it to be searched correctly. |
| Controller | saveProduct() | setUpStage2() | String code: "001", String name: "Mcqueen", int price: 16000, String description: "Cars3"; String code: "001", String name: "Lightning", int price: 18000, String description: "Cars1" | Throws ExistentProductCodeException. |
| Controller | saveProduct() | setUpStage3() | String code: "002", String name: "Product2", int price: 2000, String description: "" | Returns “true”, saves the product with an empty description. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective: Validate the save Customers in a valid state, also with an id exceptions and saving multiples products** | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Expected Outcome** |
| Controller | saveCustomer() | setUpStage4() | String name: "Juan", String id: "Customer01", String address: "Icesi", String email: "juan@gmail.com", String password: "juan1234" | Returns “true”, saves the customer and allows it to be searched correctly. |
| Controller | saveCustomer() | setUpStage5() | String name: "Juan", String id: "Customer01", String address: "Icesi", String email: "juan@gmail.com", String password: "juan1234"; String name: "Pedro", String id: "Customer01", String address: "Other Address", String email: "pedro@gmail.com", String password: "pedro1234" | Throws ExistentCustomerIdException. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective: Validate the save Orders in a valid state, also with an numbers exceptions and saving an inexistent product.** | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Expected Outcome** |
| Controller | saveOrder() | setUpStage6() | List<Product> products: [Product("001", "Product1", 1000)], Customer customer: Customer("Customer01"), String orderNumber: "Order001" | Returns “true”, saves the order correctly. |
| Controller | saveOrder() | setUpStage7() | List<Product> products: [Product("001", "Product1", 1000)], Customer customer: Customer("Customer01"), String orderNumber: "Order001" | Throws ExistentOrderNumberException. |
| Controller | saveOrder() | setUpStage8() | List<Product> products: [], Customer customer: Customer("Customer01"), String orderNumber: "Order002". | Throws InexistentProductException. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective: Validate the modify product in a valid state, also with an inexistent product exception.** | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Expected Outcome** |
| Controller | modifyProduct() | setUpStage9() | String newCode: "002", String newName: "Lightning McQueen", int newPrice: 18000, String newDescription: "New Cars", String oldCode: "001" | Returns “true”, modifies the product correctly and removes the old code. |
| Controller | modifyProduct() | setUpStage10() | String newCode: "002", String newName: "Lightning McQueen", int newPrice: 18000, String newDescription: "New Cars", String oldCode: "001" | Throws InexistentProductException. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective: Validate the remove orders in a valid state, also with a non-existing orders.** | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Expected Outcome** |
| Controller | removeOrder() | setUpStage11() | String orderNumber: "Order002" | Returns “true”, removes the order and it can no longer be searched. |
| Controller | removeOrder() | setUpStage12() | String orderNumber: "NonExistentOrder" | Throws InexistentOrderException. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective: Validate the undo last action in a valid state, also with throw exception of no actions and also of undo last action for a customer.** | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Expected Outcome** |
| Controller | undoLastAction() | setUpStage12() | N/A | Returns “true”, undoes the action and the product no longer exists. |
| Controller | undoLastAction() | setUpStage13() | N/A | Returns “true”, undoes saving a customer and the customer no longer exists. |
| Controller | undoLastAction() | setUpStage17() | N/A | Throws an Exception because no previous actions exist. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective: Validate the dispatch products in a valid state and multiple products.** | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Expected Outcome** |
| Controller | dispatchProduct() | setUpStage14() | N/A | Returns the dispatched product, "Smartphone". |
| Controller | dispatchProduct() | setUpStage19() | N/A | Successfully dispatches multiple products in order: "Laptop" first, "Smartphone" second. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective: Validate the products exists in a valid state and invalid state.** | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Expected Outcome** |
| Controller | productExists() | setUpStage15() | String productCode: "P001" | Returns “true”, indicating the product exists. |
| Controller | productExists() | setUpStage20() | String productCode: "P999" | Returns “false”, indicating the product does not exist. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective: Validate the customer password exists in a valid state.** | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Expected Outcome** |
| Controller | validateCustomerPassword() | setUpStage16() | String customerID: "C001", String password: "password123" | Returns “true”, indicating the password is valid. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective: Validate the products price in multiple products and list products with no price.** | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Expected Outcome** |
| Controller | getProductsPrice() | setUpStage21() | List<Product> products: [Product("P001", "Toy Car", 1000), Product("P002", "Toy Train", 2000)] | Returns the total price of the products: 3000. |
| Controller | getProductsPrice() | setUpStage22() | List<Product> products: [] | Returns 0, since there are no products. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective: Validate the products price in multiple products and list products with no price.** | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Expected Outcome** |
| Controller | listProducts() | setUpStage23() | N/A | Returns a list containing both "Toy Car" and "Toy Train". |
| Controller | listProducts() | setUpStage24() | N/A | Returns an empty string since no products are available. |

**Scenario Configuration Hash Table**

|  |  |  |
| --- | --- | --- |
| **Name** | **Class** | **Scenario** |
| setUpStage1() | HashTableTest | Insert a valid product and verify it is stored correctly. |
| setUpStage2() | HashTableTest | Handle multiple collisions for the same key. |
| setUpStage3() | HashTableTest | Insert a duplicate key and verify the most recent entry is stored. |
| setUpStage4() | HashTableTest | Delete a valid key and verify the entry is removed. |
| setUpStage5() | HashTableTest | Attempt to delete a non-existent key and ensure no exception is thrown. |
| setUpStage6() | HashTableTest | Search for a non-existent key and verify it returns null. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Validate the hash table’s functionality for handling insertions, collisions, deletions, and searches, ensuring correct behavior in all scenarios. | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Expected Outcome** |
| HashTable | insert () | setUpStage1() | String key: "004", Product product: Product("004", "Macbook pro", 500, "MODEL2024") | Returns “true”, the product is inserted, and can be found using the key. |
| HashTable | insert() | setUpStage2() | String key: "001", Product product1: Product("001", "Macbook pro", 500, "MODEL2024"); Product product2: Product("001", "iPhone15", 800, "MODEL2023"); Product product3: Product("001", "Apple Watch", 400, "MODEL2022") | Handles collisions: Product "Apple Watch" is found first, followed by "iPhone15" and "Macbook pro". |
| HashTable | insert() | setUpStage3() | String key: "004", Product product1: Product("004", "Macbook pro", 500, "MODEL2024"); Product product2: Product("004", "iPhone15", 800, "MODEL2023") | The most recent product ("iPhone15") is found first, followed by "Macbook pro". |
| HashTable | delete() | setUpStage4() | String key: "004", Product product1: Product("004", "Macbook pro", 500, "MODEL2024"); String key: "105", Product product2: Product("105", "iPhone 15", 800, "MODEL2023") | Product "Macbook pro" is deleted, while "iPhone 15" remains in the table. |
| HashTable | delete() | setUpStage5() | String key: "004", Product product: Product("004", "Macbook pro", 500, "MODEL2024"); String nonExistentKey: "999" | No exception is thrown when deleting a non-existent key. "Macbook pro" remains in the table. |
| HashTable | search() | setUpStage6() | String key: "999" | Searching for a non-existent key returns null. |

**Scenario Configuration Priority Queue**

|  |  |  |
| --- | --- | --- |
| **Name** | **Class** | **Scenario** |
| setUpStage1() | PriorityQueueTest | Insert elements and verify the maximum element is correct. |
| setUpStage2() | PriorityQueueTest | Attempt to find the maximum element in an empty queue and catch the exception. |
| setUpStage3() | PriorityQueueTest | Extract the maximum element and verify the next maximum is correct. |
| setUpStage4() | PriorityQueueTest | Attempt to extract the maximum element from an empty queue and catch the exception. |
| setUpStage5() | PriorityQueueTest | Insert elements into a full queue and verify the exception is thrown. |
| setUpStage6() | PriorityQueueTest | Increase a key value and verify the maximum is updated correctly. |
| setUpStage7() | PriorityQueueTest | Attempt to increase a key with a smaller value and catch the exception. |
| setUpStage8() | PriorityQueueTest | Verify the size of the queue after multiple insertions. |
| setUpStage9() | PriorityQueueTest | Insert elements and verify the queue maintains the correct order after extractions. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective:** Validate the Priority queue | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Expected Outcome** |
| PriorityQueue | insert()maximum() | setUpStage1() | Insert: int 10, int 20, int 5 | Returns 20 as the maximum element. |
| PriorityQueue | maximum() | setUpStage2() | No elements inserted | Throws NoElementFoundException when calling maximum(). |
| PriorityQueue | extractMaximum(), maximum() () | setUpStage3() | Insert: int 30, int 15, int 40 | Extracts 40 as the maximum, then 30 as the next maximum. |
| PriorityQueue | extractMaximum() | setUpStage4() | No elements inserted | Throws NoElementFoundException when calling extractMaximum(). |
| PriorityQueue | |  | | --- | |  |   insert() | setUpStage5() | Insert 20 elements, then attempt to insert one more | Throws PriorityFullException when trying to insert into a full queue. |
| PriorityQueue | increaseKey(), maximum() | setUpStage6() | Insert: int 10, int 15, int 5, Increase key at index 2 to 20 | Returns 20 as the new maximum. |
| PriorityQueue | increaseKey() | setUpStage7() | Insert: int 25, int 15, int 35, Attempt to increase key at index 1 to 10 | Throws SmallerKeyException when increasing to a smaller value. |
| PriorityQueue | size() | setUpStage8() | Insert: int 10, int 20 | Returns 2 as the size of the queue. |
| PriorityQueue | insert(), extractMaximum(), maximum() | setUpStage9() | Insert: int 50, int 40, int 60, Extract maximum, then check next maximum | Returns 60 as the maximum, then 50 after extraction. |

**Scenario Configuration Stack**

|  |  |  |
| --- | --- | --- |
| **Name** | **Class** | **Scenario** |
| setUpStage1() | StackTest | Push an item to the top of the stack and verify the size. |
| setUpStage2() | StackTest | Pop the top item from the stack and verify the item and size. |
| setUpStage3() | StackTest | Attempt to pop from an empty stack and verify the exception. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test objective**: Validate the stack’s ability to handle push, pop, and size operations, and ensure proper exception handling when popping from an empty stack**.** | | | | |
| **Class** | **Method** | **Scenario** | **Input Values** | **Expected Outcome** |
| Stack | itShouldPushItemToTheTopOfTheStack() | setUpStage1() | Push: int 10 | Size increases to 1 after pushing an element. |
| Stack | itShouldPopTheTopItemFromTheStack() | setUpStage2() | Push: int 10, int 20; Pop 2 times | Pops 20 first, then 10; size decreases to 0 and stack becomes empty. |
| Stack | itShouldThrowExceptionWhenPoppingFromEmptyStack() | setUpStage3() | No items in the stack | Throws StackException with the message "Stack is empty". |