<u>CartServiceImpl.java</u>

Several catch blocks for SQLExceptions

- Generally, there are several types of methods capable of throwing these exceptions, but none of them will activate unless the entire SQL server is offline, would make it more of a test of the server connection rather than the program itself
    - PreparedStatement.setString() / .setInt()
        - This is very unlikely to fail, as the queries written by the author contain the right number of parameters to indices, and never close the connection before running the setters.
    - PreparedStatement.executeQuery() / .executeUpdate() / Connection.prepareStatement()
        - For deletion, selection, and updating queries, execution does not throw SQLExceptions, even for invalid username/product ID. I tested this on the database itself directly, and in tests. The only case where this is not true is for insertion, which checks to see that the username/product ID exists.
        - Pairings of query types to these functions are almost certainly correct: this means that they are unlikely to throw a SQLException. Again, these are never run after closing a statement or connection.
    - ResultSet.next()
        - If next() throws an Exception, this means that a database access error has occurred. However, it can be said that since the value of the ResultSet depends on an execution function on a PreparedStatement, the execution will throw the Exception rather than this function. Again, as with the others, this is never run on a closed set.

addProductToCart: 61 (if flag)

- False branch:
    - For the path with flag calculation included, it must first be true that there exists an entry in the cart for a username and product ID, and a related product with a quantity less than the amount desired in the cart. Through this, we check that the username and product ID both exist, so there is no way this can cause an error later on.
    - The value of flag then depends on demand.addProduct(demandBean).
        - demand.addProduct then selects all demands by username and product ID, and if there is none found, then it inserts it and sets the flag to true if successful, and if a demand of the type is already found, then it sets the flag to true
    - The only times where flag can actually be false is if:
        - The server itself is not working and throws an Exception

        - This would cause an error earlier on and not reach the flag branch of addProductToCart
    - The demand does not yet exist in the server, and the insert call fails and throws an Exception
        - The call would only fail if a duplicate appeared, and the duplicate is handled already, and it sets flag to true
    - The insert call for the demand affects 0 rows in the table
        - Not possible in this case, as we established that this either will be successful, lead to a different branch where flag is true, or fail earlier on

getCartCount: 141

- As mentioned in the faults document, there is a misapplication of the ResultSet.wasNull() function. As a result, wasNull() will never return false as long as the current row exists (that is to say, next() is called before wasNull()).
- Additionally, next() will almost certainly return true unless the entire SQL database is offline. When there is a null column for quantity and when the username doesn't exist, which are the only reasonable errors for the getCartCount() function, it returns a null row/column to indicate the sum. However, this will pass the next() call since the row still exists.
- These make it so that the statement in the if condition is essentially always true && true, meaning branch coverage cannot be achieved

getProductCount: 350

- Same as getCartCount 141

getCartItemCount: 379

- Same as getCartCount 141

removeProductFromCart: 192

- The condition for this branch to activate is if k > 0, where k reflects the number of rows changed by the update query above it. However, note that all logic for the update query lies within a loop where rs.next() returns true, indicating that the first query for selecting rows in the cart by username and product ID returns at least one row. This means that when we update again by username and product ID, it will work on the same row(s) as before, meaning update will always succeed and find rows to update. The only case where this will not happen is if the SQL server shut down between the two queries, which is highly unlikely and is not a relevant whitebox or blackbox test case, but rather falls closer to server testing.

removeProductFromCart: 194

- This is an if-else if statement, however, note that the first condition is if productQuantity > 0. However, the else-if at line 194 is else if productQuantity <= 0. This really should just be an else statement, and since the two options are a complete partition of all possible cases for productQuantity, there is no way the false branch can be true.

removeProductFromCart: 204

- Same logic as removeProductFromCart 192

updateProductToCart: 292

- Same logic as removeProductFromCart 192

updateProductToCart: 303

- Same logic as removeProductFromCart 192

updateProductToCart: 318

- Here, we can see again that the branch depends on the value of k, which reflects the number of rows updated by an insert. However, note that whenever the insert fails (i.e. adding a nonexistent product code), this throws a SQLException, which in turn skips the if statement entirely, as seen in the tests for nonexistent product ID / username. In addition, note that duplicates are allowed in the database, so that is not a feasible failing case. Therefore, k must always return true whenever there is not a SQLException, meaning the false branch is not feasible to test.

<u>DemandServiceImpl.java</u>

Several catch blocks for SQLExceptions

- Similar to the above file's unachievable coverage reasons
- SQLExceptions are only really thrown during failed insert statements, as seen in testing, since queries like update, delete, and select for their respective Java functions, will almost never throw exceptions.

addProduct: 53

- The false branch is infeasible for several reasons. Firstly, the value k reflects the number of entries affected by insert, which should be 1 if successful in this case, but generally, the number of rows affected should be a nonnegative number. However, note that if the insert is unsuccessful, then the database functions throw an exception instead of proceeding. This is seen in the test for duplicate demands, which indeed reaches the catch statement for a SQLException. Therefore, there is no way to reach the logic and have k be nonpositive, and therefore no way to reach this branch.

removeProduct: 101

- The false branch is again infeasible for similar reasons to ones mentioned before. Since the prerequisite to entering the branch to calculate k is for the select query to pass, and for the ResultSet returned to have at least one entry, this means that the subsequent delete can locate the exact same entry and delete it, and return a value of k greater than zero. Even if the database fails, for example, it would rather throw a SQLException and enter the catch block than proceed with a nonpositive value of k.

OrderServiceImpl.java

paymentSuccess: 52

- For this branch to be explored, CartServiceImpl.removeAProduct() must fail. However, note that the removeAProduct function only failed in the CartServiceImpl testing when removing a nonexistent item, whether by user or product ID. However, note that since the paymentSuccess function retrieves all cart items, it's clear that each product must already exist, so the removeAProduct cannot fail naturally in this context unless the database connection fails between the two functions.
- One initial solution I considered was to disable some checks artificially. For instance, I managed to get the previous addOrder function in paymentSuccess to fail, but that was by disabling the foreign key check. If I removed the null check on the column, added an invalid entry, and re added the null check, I could cover this branch by using a null value that would cause removeAProduct to fail later on. However, it seems quite difficult to ignore the null check, and at that point, I am testing for errors that are not feasible, and therefore tests that do not add anything to the program.

paymentSuccess: 63

- This branch is explored whenever OrderServiceImpl.addTransaction fails. For addTransaction to fail, some potential methods are to add a duplicate entry so insert throws an Exception, or to use something that violates the foreign key check. However, the first is not feasible or possible, as when addTransaction is called in paymentSuccess, it is initialized such that a new transaction ID is generated each time by the utility function. This means that each entry will be distinct from the others, since the only check for duplicates is the transaction ID. The second is infeasible, because when the foreign key check is disabled like in some other tests, it will cause a failure earlier on in the program in one of the other service functions called, because a foreign key check fails when a certain field does not exist in another table. Therefore, there is no feasible case where this is easily testable, and the logic itself is rather trivial, and perhaps does not need to be fully tested.

addOrder: 94

- Similar to addProduct 53, where insert simply succeeds or throws an exception, with little in between, meaning the number k must be greater than zero, or the branch will never be reached due to an exception. This is seen in the duplicate and invalid/null tests.

addTransaction: 124

- Similar to addOrder 94

countSoldItem: 152

- When sum is used in the query, even if no orders of the product ID are found, and even if the product ID is null or invalid, it always returns at least one row, even if it is null. This means that the rs.next() call is always valid, and is never false in this circumstance, and even if the database itself was down in that specific instance, it would throw an exception before reaching the branch.

countSoldItem: 155

- This is similar to other cases above where catch blocks for SQLExceptions are not feasible to test, since each function is paired correctly with the query type, and because sum will almost never throw an Exception if the query syntax is correct.

getAllOrders: 191

- Similar to other cases where catch blocks for SQLExceptions are not feasible to test. Even when the table is empty, the select query will not fail, and the rs.next() call will not fail, as seen in the test for that case. There is no naming, syntax, or any other error visible that might cause an exception to be thrown.

getOrdersByUserId: 217

- This is unreachable due to a fault in the function mentioned in the faults and weaknesses file

getAllOrderDetails: 263

- Similar to getAllOrders 191. The select query will not fail under most testable circumstances, naming and query/function pairings again are correct.

shipNow: 291

- Similar to getAllOrders 191. The update query will not fail under most testable circumstances, naming and query/function pairings again are correct.

<u>TransServiceImpl.java</u>

One catch blocks for SQLExceptions

- Similar to the above files' unachievable coverage reasons
- SQLExceptions are only really thrown during failed insert statements, as seen in testing, since queries like update, delete, and select for their respective Java functions, will almost never throw exceptions.

<u>UserServiceImpl.java</u>

Several catch blocks for SQLExceptions

- Similar to the above files' unachievable coverage reasons
- SQLExceptions are only really thrown during failed insert statements, as seen in testing, since queries like update, delete, and select for their respective Java functions, will almost never throw exceptions.

<u>ProductServiceImpl.java</u>

Several catch blocks for SQLExceptions

- Similar to the above files' unachievable coverage reasons
- SQLExceptions are only really thrown during failed insert statements, as seen in testing, since queries like update, delete, and select for their respective Java functions, will almost never throw exceptions.

addProduct: 38

- This cannot be reached because line 24 generates the prodId before calling this second addProduct function.

addProduct: 62

- This cannot be reached unless the SQL query fails and is difficult to reproduce.

updateProductWithoutImage: 421

- This cannot be reached unless the mailing service fails and is difficult to reproduce.