Midterm Practice Paper

BigCo is a large company (more than 10 thousand employees) serving millions of customers. The customers interact with the company through their retail stores, call center and website. Management wants the Technology team to create a mobile app, both for Android and iOS, that leverages the existing services powering the applications used by retail stores and call center, and the website, to enable customers to access their account history, including orders in the stores and tickets in the call center; providing similar functionality to what is available today on the website.

The pain points today include slowness of the website, and somewhat frequent outages where both the employee applications and website are “temporarily out of order”.

The team has 10 backend developers, 2 Android and 2 iOS developers, plus 1 Software Development Engineer in Test (SDET) and 2 manual QA specialists. You were just hired as the SDET, and the team relies on your expertise to provide recommendations for improving quality and speed of the team delivery; plus performing test automation and coordinating all QA work.

1. **Test Planning**

   a) **Write the titles of the 5 test cases that you would consider higher priority to get all passing, continuously (BVT)**
      - User Authentication and authorization
      - View Account History
      - Place New Order
      - Create and Update Support Ticket
      - Service Availability and Performance

   b) **Indicate what would be the passing criteria for each of those 5 tests cases.**

      (a) User Authentication and Authorization
      1. The mobile app should successfully authenticate users using correct credentials.
      2. Unauthorized access attempts with incorrect credentials should be blocked.

      (b) View Account History
      1. Users should be able to view their entire account history, including past orders and support tickets.
      2. The displayed data should be accurate and up to date.

      (c) Place New Order
      1. Users should be able to place new orders successfully.
      2. The order details should be correctly reflected in the user’s account history.

      (d) Create and Update Support Ticket
      1. Users should be able to create new support tickets and update existing ones.
      2. Changes to the tickets should be accurately reflected in real-time in the user’s account history.

      (e) Service Availability and Performance
      1. The mobile app should be operational and accessible at all times, barring scheduled maintenance.
      2. The app should operate smoothly, without significant delays or timeouts.

   c) **As you run those test cases, you have a 60% pass rate. Do you change the passing criteria to reflect the current baseline, or do you keep the test cases? Why, and what is your next step?**
      I would keep the test cases as they are. Adjusting the passing criteria to match the current baseline would mean lowering the quality standards, which is not advisable. These test cases are essential for ensuring that the app works correctly and provides a good user experience.
      Why?

The test cases are designed to validate the core functionalities of the app. A 60% pass rate indicates that there are significant issues that need to be addressed. Lowering the standards would mean ignoring these issues, leading to a subpar product.

Next Steps:

Analyze the Failures:

Review the failed test cases to identify common issues or patterns.

Determine whether the failures are due to code defects, environment issues, or test case design.

Collaborate with the teams:

Communicate the findings with the development team for further analysis and bug fixing.

Work closely with developers to ensure that the identified issues are resolved promptly.

Improve Test Cases:

If some test cases are found to be flaky or not well-designed, improve them to be more reliable and effective.

Ensure that test cases cover different scenarios and edge cases.

Continuous Monitoring:

Continuously monitor the test results to track improvement and identify any new issues early.

Aim for a higher pass rate by iteratively improving the code and tests.

2. **Agile Testing:**

a) **Enumerate 3 good practices you can introduce to improve both quality and speed; for each of them, explain why they would help in this situation.**

i. **Automated Regression Testing:**
Automated regression testing ensures that the existing functionalities of the application are not broken by new code changes. It allows for quick identification of issues and ensures that every release maintains a consistent quality. Automating regression tests will enable the team to quickly validate the system's stability, freeing up time for other essential tasks like exploratory testing or focusing on new functionalities.

ii. **Continuous Integration/Continuous Deployment (CI/CD):**
Implementing a CI/CD pipeline helps in automating the deployment process, reducing manual errors, and speeding up the delivery process. Every code commit can be automatically built, tested, and deployed to a staging environment, ensuring that the code in the repository is always in a deployable state. This allows for quicker feedback and faster delivery of features and bug fixes to the customers.

iii. **Test-Driven Development (TDD)**
In TDD, developers write tests before writing the code that needs to be tested. This ensures that the code is built to be testable, reliable, and maintainable. It also helps in clarifying the requirements before the actual coding starts, reducing the back-and-forth and rework. TDD encourages developers to think about different scenarios and edge cases early in the development cycle, improving the overall quality of the code.

b) **Considering the different roles of the team members, what test activities would you assign to each role, and why?**

- **Backend Developers:**

Responsible for unit testing their code. They should write tests to cover different scenarios, edge cases, and integrations with external services or databases.

- **Android and iOS Developers:**

Responsible for unit and UI testing of the mobile applications. They should ensure that the app works correctly on different devices and screen sizes.

- **SDET (Software Development Engineer in Test):**

Focus on creating and maintaining automated regression test suites.

Coordinate with the development and QA teams to ensure that testing is integrated throughout the development process.

- **Manual QA Specialists:**

Focus on exploratory testing to identify issues that automated tests might miss.
Responsible for validating user experiences, usability, and ensuring that the application meets business requirements.

c) **Enumerate 3 test activities that you recommend shifting left, and 2 that you recommend performing at a later stage? For each, why?**
Shifting left: Moving the test to early stage in the testing pipeline.

- **Unit Testing**

Conducting unit tests early in the development process helps in identifying issues at the code level, making it easier and cheaper to fix.

- **Integration Testing**

Early integration testing ensures that different components or services work correctly together, helping in identifying integration issues sooner.

- **Automated Regression Testing**

Having automated regression tests run early and often ensures that new code changes do not break existing functionalities, maintaining the stability of the application.
Performing at a Later Stage:

i. **Exploratory Testing**
Exploratory testing is more effective when the application is more stable and mature. It helps in identifying fewer common issues and improving the overall user experience.

ii. **Performance and Load Testing**
These tests require a stable environment and application. Conducting them later in the cycle ensures that they reflect a realistic assessment of the system's performance underload.

3. Bug lifecycle

a) **Tasks and bugs are now tracked as issues, with states "Open" and "Closed". What would be your recommendation to handle tracking testing when a task is done, or a bug is closed, to make sure no task slips untested?**
To ensure that no task or bug slips through untested, I recommend implementing a more detailed issue (task/bug) lifecycle that includes states specifically related to testing. Here's a refined lifecycle with states and transitions that focus on testing:

1. **Open (New):** Initial state of the task or bug. It's identified but work has not started.
2. **In Progress:** Work has started on the task or bug, but it's not ready for testing yet.
3. **Ready for Testing:** The task or bug is developed and ready to be tested. It's moved to this state once development believes it's ready for QA.
4. **Testing:** The task or bug is currently being tested by the QA team.
5. **Passed Testing (Resolved):** The task or bug has passed all necessary tests and is considered resolved.
6. **Failed Testing (Reopened):** If the task or bug does not pass the testing, it is moved back to this state, signifying that further work is needed.
7. **Closed:** The task or bug is closed only after it passes all tests, and it's confirmed that it meets the required criteria. It could also be closed without fixing, based on product or business decisions.
   Process Flow:
   - Once a task is completed or a bug is fixed, it should be moved to the "Ready for Testing" state, not "Closed."
   - QA team picks up the task/bug from the "Ready for Testing" state, and it moves to "Testing."
   - If the task/bug passes all tests, it is marked as "Passed Testing (Resolved)." If it doesn't pass, it's

marked as "Failed Testing (Reopened)."
- A task/bug only reaches the "Closed" state after successful testing and any necessary rework.

Benefits:

Visibility: This process ensures that all tasks and bugs go through a testing phase, making the testing process transparent.
- Accountability: Clear states related to testing make it easy to identify who is responsible for each stage, improving accountability.

Quality: Ensuring that every task and bug is tested before being closed improves the overall quality of the product.

Tracking: It becomes easier to track the progress and state of each task and bug, ensuring that nothing is missed or slips through untested.

4. **Exploratory Testing:**

   a) **The first internal version of the app is just out and ready to be tested. Indicate 5 activities in your initial process to explore the app and find bugs right away.**

   i. **Familiarization with the App:**
   - Spend time navigating through the app to understand its functionality, flow, and user interface.
   - Identify the main features and components that need to be tested and interact with different parts of the application to get a feel for its behavior and usability.

   ii. **Performing Functional Testing**
   - Focus on the core functionalities of the app, such as user authentication, account history viewing, order placement, and support ticket creation and updating.
   - Ensure that these functionalities work as expected and meet the specified requirements.

   iii. **Checking User Interface (UI) and User Experience (UX)**
   - Evaluate the design, layout, and aesthetics of the app, ensuring that it is user-friendly and intuitive.
   - Check for inconsistencies, misalignments, or any other UI/UX issues that might hinder the user experience

   iv. **Testing on different Devices and Operating Systems**
   - Test the app on various devices, screen sizes, and operating systems to ensure that it functions correctly across different environments.
   - Identify any compatibility or responsiveness issues that might affect the usability of the app on different devices.

   v. **Performing Boundary and Edge Case Testing:**
   - Try to identify fewer common scenarios or edge cases that might not have been considered during the development process.
   - Experiment with unusual or unexpected user inputs and actions to see how the app handles them and to uncover potential issues or crashes.

5. **Testing adequacy:**

   a) **After a month, the team has created 5 E2E tests, 5 integration tests and 300-unit tests; this gives us 100% code coverage. What would be 2 questions to analyze where to improve the testing?**

   How do our tests reflect real-world user scenarios and behaviors?

   Even with 100% code coverage, it's possible that certain user scenarios or behaviors are not being tested. Ensuring that tests mirror real-world user activities can help uncover issues that might not be evident from a pure code coverage perspective.

   How often are our integration and E2E tests failing compared to unit tests, and what are the common reasons for those failures?

This question helps in understanding the stability of the system as a whole. While unit tests focus on individual components, integration and E2E tests consider the system's behavior as a whole. Identifying common reasons for failures in these tests can help in pinpointing systemic issues or areas that need more focused testing.

i. **Why the ratio is like that? Seems too few integration tests, doesn't match the common test pyramid pattern.**
ii. **What kind of tests are within the suites? Particularly, how many of those are functional and how many performance, which is a pain point?**
iii. **How coverage is distributed among the different tests types?**

b) **Test oracles: When analyzing an old test plan of the existing website, you find all test cases passing criteria are reaching the end step without a failure. What type of test oracle is then used, and what other type test oracles would you recommend adding first? Why?**

The existing test oracle is a process oracle since it determines the test's success based on reaching the end step without failure.

Other Type of Test Oracle to Recommend:

State Oracle: This involves comparing the application's actual state after a test with its expected state. For instance, after placing an order, the expected state might be that the order appears in the user's account history.

Why: Relying solely on a process oracle can miss important issues. For instance, a test might be completed without error (process oracle success), but the resulting state might be incorrect (state oracle failure). Introducing state oracles ensures that the system's output or state is as expected after a given input or action.

i. **The null test oracle is the one used in this situation.**
ii. **We can utilize specification based oracles, which would confirm the system behavior matches expectations. Mentioned first for the relative easy to implement them, and clearly a big company would have enough clarity about the processes required to have certainty on a good amount of the requirements.**
iii. **We can also use redundant computation to validate operations like adding the amount of items in an order**

c) **Application Lifecycle management. When looking at the outages concerns, you find that 75% of them occur after a deployment, which is generally done at midnight, when few customers are using the website or call center. No formal QA is performed by the engineers performing the deployment. What 3 practices would you prioritize to help reduce the outages? Why?**

Three practices to prioritize to help reduce the outages:

1. Implement a Staging Environment:

Before deploying to production, deploy the changes to a staging environment that mirrors the production setup as closely as possible. This helps in identifying potential issues before they affect the live system.

Why: Testing in a staging environment can catch deployment and integration issues that might lead to outages in the production environment.

2. Automate Deployment and Post-Deployment Testing:

Use automated deployment tools and scripts to ensure consistent and error-free deployments. Additionally, automated post-deployment tests that quickly verify the system's critical functionalities after a deployment.

Why: Automated deployments reduce human errors, and automated post-deployment tests provide immediate feedback on the system's health after a deployment.

3.Monitor System Health and Performance:

Implement monitoring tools that provide real-time insights into the system's health, performance, and error rates. Set up alerts to notify the team immediately if there are signs of potential outages or degraded performance.
Why: Real-time monitoring allows the team to quickly identify and address issues before they escalate into full-blown outages.

i. **Test automation, as it would allow deployment engineers to perform QA verification with simple steps without requiring them to acquire additional QA knowledge.**
ii. **Continuous integration, as a follow up of automation, automating the QA verifications**
iii. **Shifting left, for example by performing the deployment at business hours where developers, SDET and QA are generally available, and can help identify the problems faster; also, by them feeling the pain of the deployment, the team can identify areas to focus to test or improve robustness and reliability of the code.**

6. **Unit Testing:**
   a) **What type of unit tests would you prioritize adding in this situation?**
   Performance-Centric Unit Tests:
   Purpose: To ensure that individual functions/methods execute within acceptable time frames.
   Details: These tests will focus on the performance of the code at a unit level, ensuring that each function or method runs efficiently and doesn’t cause unnecessary delays.
   Error Handling Unit Tests:
   Purpose: To ensure that functions/methods handle errors gracefully without causing system crashes.
   Details: These tests should validate that the code handles different types of errors or exceptions gracefully, ensuring that one issue doesn’t lead to a broader system outage.
   Boundary Case Unit Tests:
   Purpose: To ensure that functions/methods handle edge cases well.
   Details: These tests focus on the boundaries of input values, ensuring that the code can handle minimum, maximum, just below minimum, and just above maximum values, as well as null or empty values.
   Concurrency Unit Tests:
   Purpose: To test how functions/methods behave when accessed concurrently.
   Details: Given that the application is expected to handle many users, it’s essential to ensure that the code behaves correctly under concurrent access, helping to prevent potential outages or slowdowns due to concurrency issues.
   Database Interaction Unit Tests:
   Purpose: To ensure that functions/methods interacting with the database do so efficiently and correctly.
   Details: These tests should focus on code that interacts with the database, ensuring that queries are optimized, and the code handles database interactions correctly.

   i. **Functional for the components that support the BVT test suite, like Login and list orders**
   ii. **Performance for the reported areas of website slowness**

b) Indicate what types of tests correspond to the code below:

```
        [Fact]
        public void TestListOrders()
        {
            // arrange
            Customer testCustomer = new Customer(1, "Test customer");
            // act
            List<Order> output = Backend.ListOrders(testCustomer);
            // assert
            Assert.True(output.Count == 1 ); // there is exactly an order in the
test data
            Assert.Equal(testCustomer.Id, output[0].CustomerId); // the first order
customer Id is the test customer
        }
```

Functional, automated, apparently an integration test (assuming Backend is the real one and not a test double)

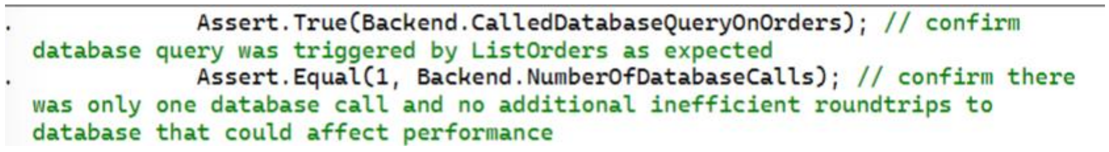
c) Indicate if this test is state based or behavior based: State Based

State Verification: The test verifies the state of the system after the method `Backend.ListOrders` has been executed. It checks whether the output list of orders is in the expected state, i.e., having one order and that order being associated with the correct customer ID.

No Behavior or Interaction Verification:
The test does not check the interactions between objects or the behavior of mocks/stubs, which would be characteristic of behavior-based testing. Instead, it focuses on the outcome or state resulting from the execution of the method.

In conclusion, because the test is focused on verifying the state of the output (list of orders) after executing the method, it is best classified as state based.

d) What changes would you apply to the Assert section in order to transform the test into behavior based or state based, opposite to your answer in C? Use pseudocode.



```
.            Assert.True(Backend.CalledDatabaseQueryOnOrders); // confirm
  database query was triggered by ListOrders as expected
.            Assert.Equal(1, Backend.NumberOfDatabaseCalls); // confirm there
  was only one database call and no additional inefficient roundtrips to
  database that could affect performance
```

e) If Backend were a Test double, what kind of Test double would you suggest, and why?

If `Backend` were to be used as a Test Double, I would suggest using a Stub as the type of Test Double.

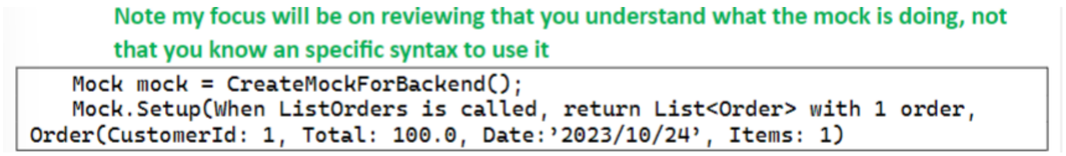
Interaction with External Systems: Since `Backend` likely interacts with external systems such as databases or APIs to fetch order details, using a stub allows simulating these external interactions. The stub can be programmed to return predefined responses, helping to isolate the unit of work being tested from external dependencies.

Focus on State Verification: Given that the initial focus of the test is state verification (ensuring that the output is as expected), a stub is suitable because it allows control over the indirect inputs to the system under test (SUT), enabling the testing of the SUT's response.

Simplicity and Control: Stubs provide a simple and controlled way to manage the indirect inputs of the SUT, making them a good choice when the goal is to focus on the state-based testing of the SUT's functionality without actual interactions with external dependencies.

Conclusion: A stub as a Test Double for `Backend` enables the testing of the SUT's functionality and state based on predefined indirect inputs, ensuring that the test is focused, deterministic, and not affected by actual interactions with external dependencies.

f) Let's substitute Backend with a Mock. Indicate with pseudocode what would be the steps to be added in the Arrange section to setup that mock.



Note my focus will be on reviewing that you understand what the mock is doing, not that you know an specific syntax to use it

```
    Mock mock = CreateMockForBackend();
    Mock.Setup(When ListOrders is called, return List<Order> with 1 order,
Order(CustomerId: 1, Total: 100.0, Date:'2023/10/24', Items: 1)
```

g) **Continuous integration: You found that there are already a good number of tests for the website; but they are run sporadically, because they are configured to only run on a developer's machine that was configured with specific steps. Explain 2 to 5 steps to implement a strategy that allows those tests to be run regularly.**

Implementing a strategy that allows tests to run regularly in a Continuous Integration (CI) environment involves several steps to ensure that the tests are automated, reliable, and executed consistently. Here are steps to achieve this:

1. Centralize Test Execution:

Setup a Continuous Integration (CI) Server: Utilize a CI server like Jenkins, CircleCI, or GitHub Actions. The CI server should be configured to pull the latest code from the version control system and execute the test suite.

Automate Test Execution: Configure the CI server to automatically run the test suite whenever changes are pushed to the version control system (e.g., Git). This ensures that tests are executed consistently and not reliant on a developer's machine.

2. Containerize the Test Environment:

Use Docker or Similar Technologies: Containerize the application and its dependencies, ensuring that the tests run in a consistent environment. Containers encapsulate the application and its environment, reducing inconsistencies due to different configurations.

3. Schedule Regular Test Runs:

Nightly Builds: Schedule the test suite to run at least once daily, regardless of whether new code has been pushed. This helps identify any issues introduced due to external factors, such as changes in external services or data.

4. Monitor and Notify:

Automated Notifications: Configure the CI server to notify the team of the test results after each run. Notifications can be sent via email, messaging platforms (e.g., Slack), or integrated project management tools.

Immediate Feedback: Ensure that the team receives immediate feedback on the test results, enabling quick responses to any failures or issues detected.

5.Maintain and Review Tests:

Regular Maintenance: Regularly review and maintain the test suite, ensuring that it remains relevant, effective, and up to date with the application's evolution.

Continuous Improvement: Continuously look for opportunities to improve the test suite, such as adding new tests, enhancing existing ones, or removing obsolete or redundant tests.

Conclusion:

Implementing these steps will help in transitioning from sporadic, developer-machine-based test execution to a more automated, consistent, and reliable testing process integrated within a Continuous Integration environment. This approach enhances the regularity, relevance, and effectiveness of the test execution, contributing to the overall quality and reliability of the application.

7. **Testing Object Oriented Software:**

a) **For the mobile app, developers do not know anything about TDD; there is no test harness yet, much less we know how to face the differences between Android and iOS. What test tools would you look first to introduce the mobile developers to TDD?**

Test Tools to Introduce Mobile Developers to TDD

For mobile app development, there are various tools available for both Android and iOS platforms that can facilitate Test-Driven Development (TDD). Here are some tools that I would recommend looking into:

For Android:

JUnit:

A foundational testing framework that is widely used in Java projects, including Android.
It's essential for writing unit tests and is well integrated into Android development environments.

Mockito:

A mocking framework that is compatible with JUnit.
Useful for creating mocks and stubs, enabling isolated unit testing of components.

Espresso:

A UI testing framework specifically designed for Android.
Helps in writing automated UI tests, ensuring that the user interface works as expected.

For iOS:

XCTest:

Apple's native testing framework integrated into Xcode.
It allows for unit and UI testing and is tightly integrated into the iOS development workflow.

OCMock:

A mocking framework for Objective-C, helpful for creating mock objects in unit tests.

Quick and Nimble:

Quick is a testing framework, and Nimble is a matching framework; both are inspired by RSpec (a Ruby testing tool).
These tools make the tests more descriptive and readable.

Multi-platform (Both Android and iOS):

Appium:

A cross-platform testing tool that allows for writing tests once and running them on multiple platforms.
Useful for end-to-end testing and automated UI testing.

b) When mobile developers start with the first module, the app login, you review their code. You find it hard to test because everything is bundled in the same code source file. What would be your first recommendation on how to split the code, to allow best practices of testability? Which principles are implemented in this first recommendation?

Recommendation on How to Split the Code for Testability

Recommendation: Apply the Single Responsibility Principle (SRP) and Modular Design

Split the Code into Logical Modules:

Separate the login functionality into different classes or modules, each handling a specific aspect such as UI, business logic, and data access.

Separate UI from Business Logic:

Ensure that the UI code is separate from the business logic. This separation makes it easier to write unit tests for the business logic without depending on the UI.

Use Dependency Injection:

Implement dependency injection to make the code more modular and testable. Dependencies such as databases or network services should be injectable, allowing them to be mocked or stubbed in tests.

Principles Implemented:

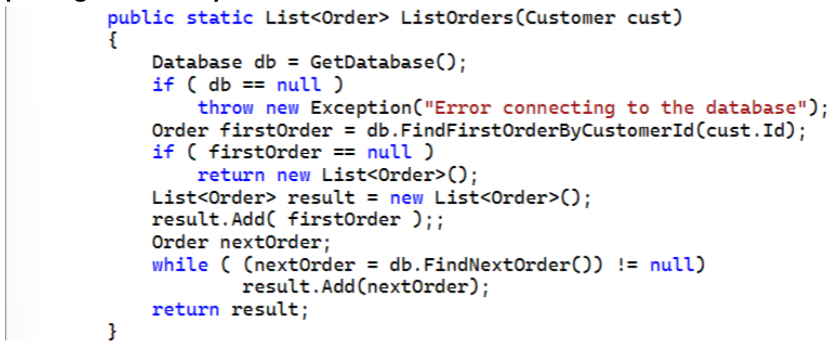
Single Responsibility Principle (SRP):

By ensuring each class or module does one thing, the code becomes more manageable and testable.

Separation of Concerns (SoC):
Keeping different aspects such as UI and business logic separate improves code maintainability and testability.
Dependency Inversion Principle (DIP):
Through dependency injection, the code depends on abstractions rather than concrete implementations, enhancing modularity and testability.
Conclusion:
Choosing appropriate tools and applying solid design principles like SRP, SoC, and DIP will set a strong foundation for implementing TDD practices in mobile app development, improving the testability and overall quality of the code.

8. **Program Analysis:**

a) **Calculate Code coverage for this method, using the test in question #6, assuming the test is passing successfully.**



```
public static List<Order> ListOrders(Customer cust)
{
    Database db = GetDatabase();
    if ( db == null )
        throw new Exception("Error connecting to the database");
    Order firstOrder = db.FindFirstOrderByCustomerId(cust.Id);
    if ( firstOrder == null )
        return new List<Order>();
    List<Order> result = new List<Order>();
    result.Add( firstOrder );;
    Order nextOrder;
    while ( (nextOrder = db.FindNextOrder()) != null)
            result.Add(nextOrder);
    return result;
}
```

Assuming line coverage: We have 12 executable lines (doesn’t include the function declaration and opening/closing brackets). 3 or them are not executed:
1. The 3rd line “throw”, as the test would not have passed if the exception had occurred
2.The 6th line that returns an empty list of orders, as the test would not have passed if the output had 0 orders and not one.
3. The 11th line, as that would have increased one order to 2 (or more).
Conclusion: the test executes 9 lines out of 12 possibles. That is 9/12 (you can leave the operation indicated), or 75% code coverage (9/12=0.75).

b) **Now, calculate the Cyclomatic complexity of the method.**
There are 4 possible paths in this code. Thus, cyclomatic complexity is 4.
i. It runs to the end without any if or while being true.
ii. The first if is true, and the code ends with an exception.
iii. The second if is true, and the code returns with an empty list.
iv. The while is true, and iterates until it is false, then runs to the end.