پاسخ تکلیف دوم آزمون نرم افزار:

.1

الف)

The solution to this problem is TDD, changing legacy system often has 2 needs: refactoring existing code and changing the functionality of legacy code
Refactoring is a way to modify the structure of existing code without changing its behavior. The agile approach to refactoring legacy code is to provide test cases for just the section of code that is being refactored
In the case of changing functionality, either to introduce new behavior or to repair a fault, the process is slightly different. Some of these tests fail, of course, since either the new behavior is not yet implemented or the fault is not yet repaired. Once the tests are ready, the desired changes can be made. At the end of the process, all of the tests should pass, including those that failed earlier.
Eventually the software will have strong TDD tests.

قسمت اول ب)

This diagram illustrates how the cost of correcting a decision (or mistake) changes over time. The horizontal axis represents the time from the initial decision to the moment the decision is corrected, while the vertical axis shows the cost.

The purpose of the diagram is to demonstrate that the longer the time interval between the initial decision and its correction, the exponentially higher the correction cost becomes.

The delta cost (Δ) for corrections made immediately after the initial decision is small, but as the time interval increases, this cost rises significantly.

قسمت دوم ب)

a) Increase in the Volume of Work Dependent on the Initial Decision:

• Over time, more work is performed based on the initial decision. If the initial decision needs to be corrected, all the work that has been done based on that decision also needs to be revised, which increases the cost.

b) Difficulty in Finding the Root Cause of Problems:

•As the software grows, finding the root cause of errors becomes more challenging. This makes corrections more time-consuming and expensive.

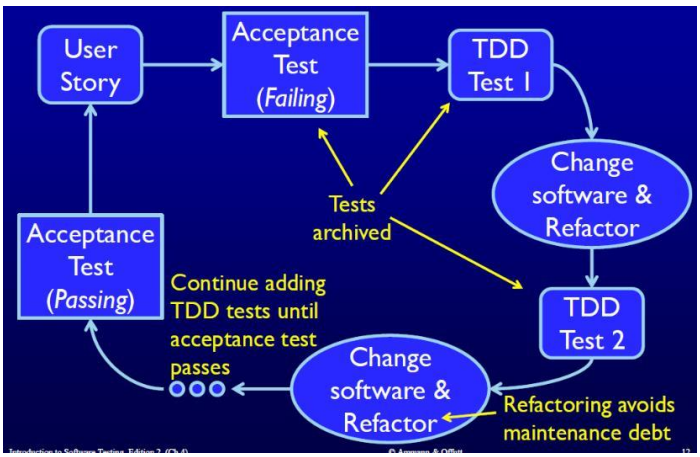
ج)

Agile methodology, while beneficial, presents challenges such as disruption to established teams, lack of traceability matrices, insufficient software evaluation, and a focus on happy paths. To manage these, teams should embrace change, use acceptance tests and TDD tests for traceability, improve test design for better software evaluation, and expand test coverage beyond happy paths.

Applying coverage criteria can help in designing high-quality tests.

د)

In Agile methodologies, user stories are central to test-driven development (TDD). Initially, a user story, which describes a user's needs or actions in simple terms, is written often with input from actual users.
This user story is then converted into one or more acceptance tests, which are expected to fail initially since the required functionality does not exist yet.
These failing acceptance tests guide the creation of a sequence of TDD tests. Each TDD test prompts the developer to implement more of the necessary functionality until the acceptance test passes. Once the acceptance test passes, indicating the user story's requirements are met, developers move on to the next user story, continuing the cycle.



و)

Controllability: **به معنای این است که تا چه حد میتوان ورودی¬ها و شرایط را در حین اجرای تست کنترل کرد. اگر قابلیت کنترل بالا باشد، میتوانیم تست¬های دقیق¬تری طراحی کنیم.**

Observability: **میزان امکان مشاهده و نظارت بر وضعیت داخلی سیستم را در حین اجرای تست نشان می¬دهد.**

Software Testability: **میزان سادگی در تست کردن یک سیستم را نشان می¬دهد.**

Prefix Values: **مقادیر اولیه یا اولین وضعیت سیستم قبل از اجرای هر تست است. برای مثال، وقتی میخواهیم لاگین به سیستم را تست کنیم، مقدار پیشوند میتواند وضعیت اولیه سیستم قبل از لاگین باشد.**

Suffix Values: **مقادیر نهایی وضعیت سیستم بعد از اجرای تست هستند. به معنی دیگر، مثل جواب نهایی تست¬های ما هستند.**

ه)

متد data-driven **یعنی جدا کردن داده‌های تست از منطق تست. این کار به ما اجازه می‌دهد که کد را با ورودی‌های مختلف (بدون تغییر کد)، تست کنیم.**

ابتدا، برای مثال، میتوانید یک فایل داده ایجاد کنید: یک فایل (login_test_data.txt) **ایجاد کنید که شامل چند تست است. هر خط در فایل نمایانگر یک مورد تست با نام کاربری، رمز عبور و نتیجه مورد انتظار (موفقیت یا شکست) می¬باشد**

```
1   user1 password1 success
2   user1 wrongpass failure
3   user2 password2 success
4   user3 wrongpass failure
5   user4 password4 failure
```

.

سپس کدی بنویسید که فایل را بخواند و کد را اجرا کند و نتیجه¬ها را مطابقت دهد.

```
struct TestCase {
    string username;
    string password;
    string expectedResult;
};

vector<TestCase> readTestCases(const string& filename) {
    vector<TestCase> testCases;
    ifstream file(filename);
    string line;

    while (getline(file, line)) {
        istringstream iss(line);
        TestCase testCase;
        iss >> testCase.username >> testCase.password >> testCase.expectedResult;
        testCases.push_back(testCase);
    }

    return testCases;
}

int main() {
    LoginSystem system;
    vector<TestCase> testCases = readTestCases("login_test_data.txt");

    for (const auto& testCase : testCases) {
        bool loginResult = system.login(testCase.username, testCase.password);
        string result = loginResult ? "success" : "failure";

        cout << "Test case: Username=" << testCase.username << ", Password=" << testCase.password << "\n";
        cout << "Expected Result: " << testCase.expectedResult << ", Actual Result: " << result << "\n";

        if (result == testCase.expectedResult) {
            cout << "Test Passed!\n\n";
        } else {
            cout << "Test Failed!\n\n";
        }
    }

    return 0;
}
```

با استفاده از این روش، می‌توان به راحتی تست بیشتری را با تغییر فایل اضافه کرد بدون اینکه نیاز باشد منطق را در برنامه تغییر دهیم.

د)

Controllability and **Observability.**

Controllability: 2>3>1

Observability: 2>1>3

(2

```python
import unittest

class ShoppingCart:
    def __init__(self):
        self.items = {}

    def add_item(self, item, quantity=1):
        if quantity < 0:
            raise ValueError("Quantity must be a non-negative
integer")

        if item in self.items:
            self.items[item] += quantity
        else:
            self.items[item] = quantity

    def remove_item(self, item, quantity=1):
        if item not in self.items:
            raise ValueError(f"{item} is not in the cart")

        if quantity >= self.items[item]:
            del self.items[item]
        else:
            self.items[item] -= quantity

    def get_total_items(self):
        return sum(self.items.values())

    def get_item_quantity(self, item):
        return self.items.get(item, 0)

class TestShoppingCart(unittest.TestCase):
    def setUp(self):
        self.cart = ShoppingCart()

    def test_add_items_with_different_quantities(self):
        self.cart.add_item('apple', 1)
        self.assertEqual(self.cart.get_item_quantity('apple'), 1)

        self.cart.add_item('banana', 3)
        self.assertEqual(self.cart.get_item_quantity('banana'), 3)

    def test_remove_items_with_different_quantities(self):
        self.cart.add_item('apple', 5)
        self.cart.remove_item('apple', 2)
        self.assertEqual(self.cart.get_item_quantity('apple'), 3)

    def test_remove_items_not_in_cart(self):
        with self.assertRaises(ValueError):
            self.cart.remove_item('banana')

    def test_get_total_items_after_adding_and_removing(self):
        self.cart.add_item('apple', 3)
        self.cart.add_item('banana', 2)
        self.cart.remove_item('apple', 2)
        self.assertEqual(self.cart.get_total_items(), 3)

    def test_add_negative_quantity(self):
        with self.assertRaises(ValueError):
            self.cart.add_item('apple', -2)

    def test_remove_more_items_than_available(self):
        self.cart.add_item('apple', 3)
        self.cart.remove_item('apple', 5)
        self.assertEqual(self.cart.get_total_items(), 0)
```

(3

```python
class TestLibrarySystem(unittest.TestCase):
    def setUp(self):
        self.library = LibrarySystem()

    def test_add_book(self):
        self.library.add_book(1, "Book Title", "Author Name", 5)
        self.assertIn(1, self.library.books)
        self.assertEqual(self.library.books[1]['title'], "Book
Title")
        self.assertEqual(self.library.books[1]['author'], "Author
Name")
        self.assertEqual(self.library.books[1]['quantity'], 5)
        with self.assertRaises(ValueError):
            self.library.add_book(1, "Another Title", "Another
Author", 3)

    def test_borrow_book(self):
        self.library.add_book(1, "Book Title", "Author Name", 5)
        self.assertTrue(self.library.borrow_book(1))
        self.assertEqual(self.library.books[1]['quantity'], 4)
        self.library.books[1]['quantity'] = 0
        self.assertFalse(self.library.borrow_book(1))
        self.assertFalse(self.library.borrow_book(2))

    def test_return_book(self):
        self.library.add_book(1, "Book Title", "Author Name", 5)
        self.library.borrow_book(1)
        self.assertTrue(self.library.return_book(1))
        self.assertEqual(self.library.books[1]['quantity'], 5)
        self.assertFalse(self.library.return_book(2))

    def test_get_book_quantity(self):
        self.library.add_book(1, "Book Title", "Author Name", 5)
        self.assertEqual(self.library.get_book_quantity(1), 5)
        self.assertIsNone(self.library.get_book_quantity(2))
```

(4

Prefix: 1
Test values: 1 2 3 4 5 6 7 8 9 10
Expected value: 5.5
Postfix: -1 2