Lecture 9: Testing and Quality Assurance

Ensuring Your Software Works Correctly

State University of Zanzibar (SUZA)
BSc Computer Science

Contents

1 Introduction to Software Testing

1.1 What is Software Testing?

Software testing is the process of evaluating software to find defects and verify that it meets specified requirements.

1.2 Why Testing Matters

- **Cost:** Finding bugs early is 100x cheaper than fixing them in production
- **Quality:** Ensures software meets user expectations
- **Reliability:** Builds confidence in the software
- **Security:** Identifies vulnerabilities before deployment
- **Documentation:** Tests serve as living documentation

1.3 Testing Pyramid

[fill=green!30] (0,0) – (6,0) – (3,1.5) – cycle; [fill=yellow!30] (0.75,1.5) – (5.25,1.5) – (3,3) – cycle; [fill=red!30] (1.5,3) – (4.5,3) – (3,4.5) – cycle;
at (3,0.5) Unit Tests (70%); at (3,2) Integration Tests (20%); at (3,3.5) E2E (10%);

2 Types of Testing

2.1 Unit Testing

What: Testing individual functions or components in isolation.
Characteristics:

- Tests smallest testable parts
- Fast execution
- Should be automated
- Run frequently (on every commit)

Example (JavaScript with Jest):

```
// Function to test
function add(a, b) {
    return a + b;
}

// Unit test
describe('add function', () => {
    test('adds two positive numbers', () => {
        expect(add(2, 3)).toBe(5);
    });
    test('adds negative numbers', () => {
        expect(add(-1, -1)).toBe(-2);
    });

    test('adds zero', () => {
        expect(add(5, 0)).toBe(5);
    });
});
```

Example (Python with pytest):

```
# Function to test
def calculate_discount(price, discount_percent):
    if discount_percent < 0 or discount_percent > 100:
        raise ValueError("Invalid␣discount")
    return price * (1 - discount_percent / 100)

# Unit tests
def test_calculate_discount_normal():
    assert calculate_discount(100, 20) == 80

def test_calculate_discount_zero():
    assert calculate_discount(100, 0) == 100

def test_calculate_discount_invalid():
    with pytest.raises(ValueError):
        calculate_discount(100, 150)
```

2.2 Integration Testing

What: Testing how components work together.

Examples:

- API endpoint tests
- Database operations
- External service integrations

Example (API Test):

```
describe('User API', () => {
    test('POST /users creates new user', async () => {
        const response = await request(app)
            .post('/api/users')
            .send({
                name: 'John Doe',
                email: 'john@example.com',
                password: 'password123'
            });

        expect(response.status).toBe(201);
        expect(response.body.data.name).toBe('John Doe');
    });

    test('GET /users/:id returns user', async () => {
        const response = await request(app)
            .get('/api/users/1');

        expect(response.status).toBe(200);
        expect(response.body.data.id).toBe(1);
    });
});
```

2.3 End-to-End (E2E) Testing

What: Testing complete user workflows.
Tools: Cypress, Selenium, Playwright
Example (Cypress):

```
describe('User Login', () => {
    it('should login successfully', () => {
        cy.visit('/login');
        cy.get('input[name="email"]').type('user@example.com');
        cy.get('input[name="password"]').type('password123');
        cy.get('button[type="submit"]').click();
        cy.url().should('include', '/dashboard');
        cy.contains('Welcome').should('be.visible');
    });

    it('should show error for invalid credentials', () => {
        cy.visit('/login');
        cy.get('input[name="email"]').type('wrong@example.com');
        cy.get('input[name="password"]').type('wrongpassword');
        cy.get('button[type="submit"]').click();
        cy.contains('Invalid credentials').should('be.visible');
    });
});
```

2.4 Other Testing Types

Type	**Purpose**
Smoke Testing	Quick test to ensure basic functionality works
Regression Testing	Verify changes don't break existing features
Performance Testing	Test speed, scalability, stability
Security Testing	Find vulnerabilities
Usability Testing	Evaluate user experience
UAT	User validates requirements are met

3 Test-Driven Development (TDD)

3.1 What is TDD?

Write tests BEFORE writing the code.

3.2 TDD Cycle: Red-Green-Refactor

1. **RED:** Write a failing test
2. **GREEN:** Write minimal code to pass the test
3. **REFACTOR:** Improve code while keeping tests green

3.3 TDD Example

Step 1: Write failing test (RED)

```
test('should validate email format', () => {
    expect(isValidEmail('test@example.com')).toBe(true);
    expect(isValidEmail('invalid-email')).toBe(false);
});
// Test fails - function doesn't exist yet!
```

Step 2: Write minimal code (GREEN)

```
function isValidEmail(email) {
    const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    return regex.test(email);
}
// Test passes!
```

Step 3: Refactor if needed

```
// Add more robust validation if needed
function isValidEmail(email) {
    if (!email || typeof email !== 'string') return false;
    const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    return regex.test(email.trim());
}
```

4 Writing Good Test Cases

4.1 Test Case Structure: AAA Pattern

- **Arrange:** Set up test data and conditions
- **Act:** Execute the function/action being tested
- **Assert:** Verify the expected outcome

```
test('should calculate total with discount', () => {
    // Arrange
    const cart = {
        items: [
            { price: 100, quantity: 2 },
            { price: 50, quantity: 1 }
        ],
        discountPercent: 10
    };

    // Act
    const total = calculateTotal(cart);

    // Assert
    expect(total).toBe(225); // (200 + 50) * 0.9
});
```

4.2 Test Case Template

Test Case ID	TC-001
Title	User can login with valid credentials
Preconditions	User account exists in database
Test Steps	1. Navigate to login page
2. Enter valid email	
3. Enter valid password	
4. Click login button	
Expected Result	User is redirected to dashboard
Actual Result	[Fill during execution]
Status	Pass / Fail

4.3 What to Test

- **Happy path:** Normal, expected usage
- **Edge cases:** Boundary values, empty inputs
- **Error cases:** Invalid input, exceptions
- **Security:** Authentication, authorization

5 Testing Tools

5.1 JavaScript/Node.js

Tool	**Purpose**
Jest	Unit testing, mocking
Mocha + Chai	Flexible testing framework
Supertest	API testing
Cypress	E2E testing

5.2 Python

Tool	**Purpose**
pytest	Unit testing
unittest	Built-in testing
Selenium	Browser automation

5.3 Java

Tool	**Purpose**
JUnit	Unit testing
Mockito	Mocking
Selenium	Browser automation

6 Bug Reporting

6.1 Bug Report Template

Bug ID	BUG-001
Title	Login fails with special characters in password
Severity	High
Environment	Chrome 120, Windows 11
Steps to Reproduce 2. Enter email: test@example.com 3. Enter password: P@ss#word! 4. Click login	1. Go to login page
Expected	User logs in successfully
Actual	Error: "Invalid characters"
Screenshot	[Attach image]

6.2 Bug Severity Levels

- **Critical:** System crash, data loss, security breach
- **High:** Major feature broken, no workaround
- **Medium:** Feature partially broken, workaround exists

- **Low:** Minor issue, cosmetic problems

7 Code Coverage

7.1 What is Code Coverage?

Percentage of code executed during testing.

7.2 Coverage Types

- **Line coverage:** Lines of code executed
- **Branch coverage:** Decision branches taken
- **Function coverage:** Functions called

7.3 Coverage Goals

- Aim for 80%+ coverage
- 100% coverage doesn't mean bug-free
- Focus on critical paths

Running coverage (Jest):

```
npm test -- --coverage
```

8 Continuous Testing

8.1 CI/CD Integration

Run tests automatically on:

- Every commit
- Pull requests
- Before deployment

GitHub Actions Example:

```
name: Tests
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
      - run: npm install
      - run: npm test
```

9 Practical Exercise

For your project, implement:

1. At least 10 unit tests for core functions
2. At least 3 integration tests for API endpoints
3. Test coverage report
4. Bug report for any issues found

10 Summary

- Testing is essential for quality software
- Follow the testing pyramid: many unit tests, fewer E2E tests
- Use TDD to write better code
- Automate tests in CI/CD pipeline
- Document bugs clearly with steps to reproduce
- Aim for meaningful coverage, not just numbers