CMPS 356

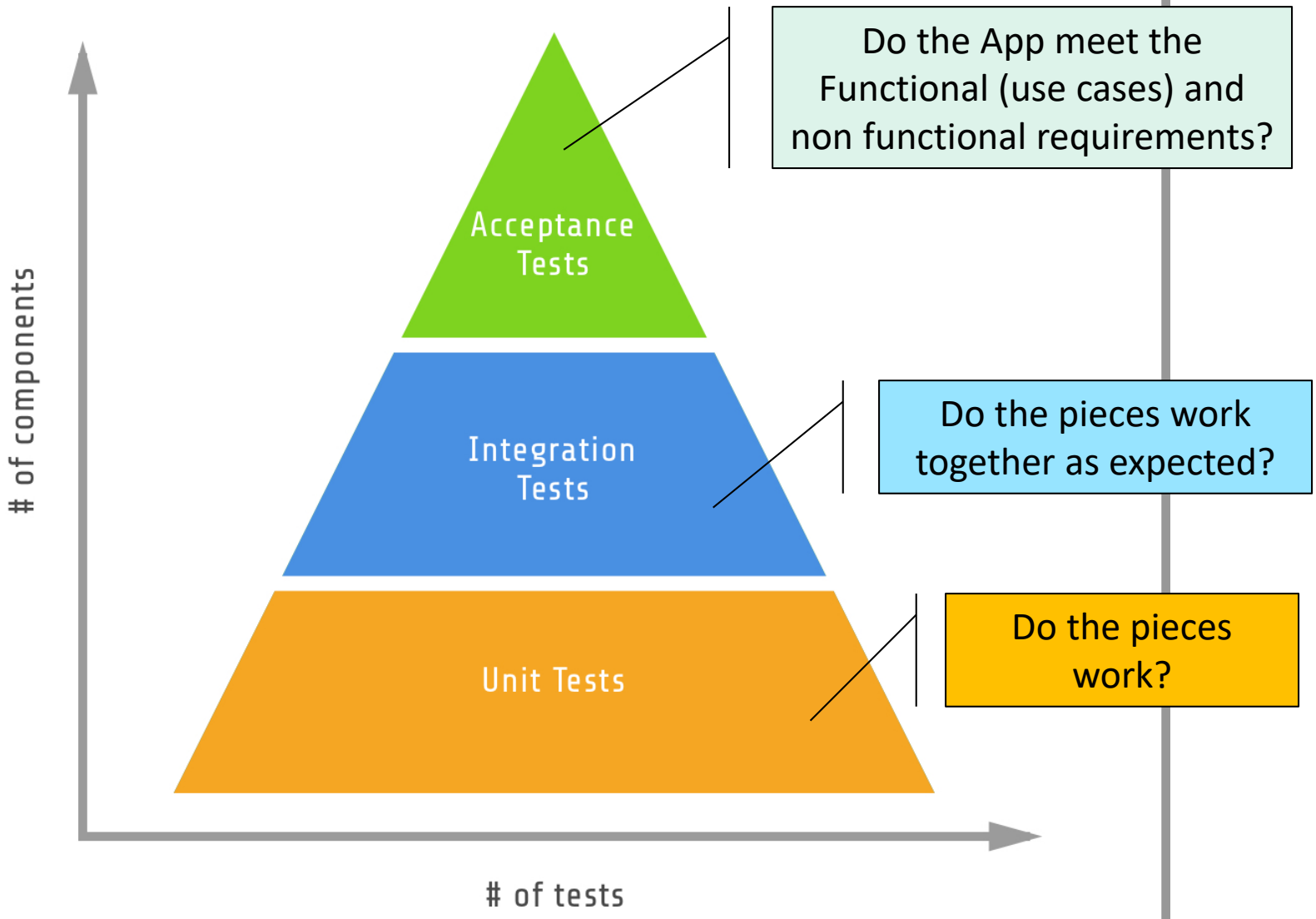
# Unit Testing in JavaScript

**Dr. Abdelkarim Erradi**
Dept. of Computer Science & Engineering
**QU**

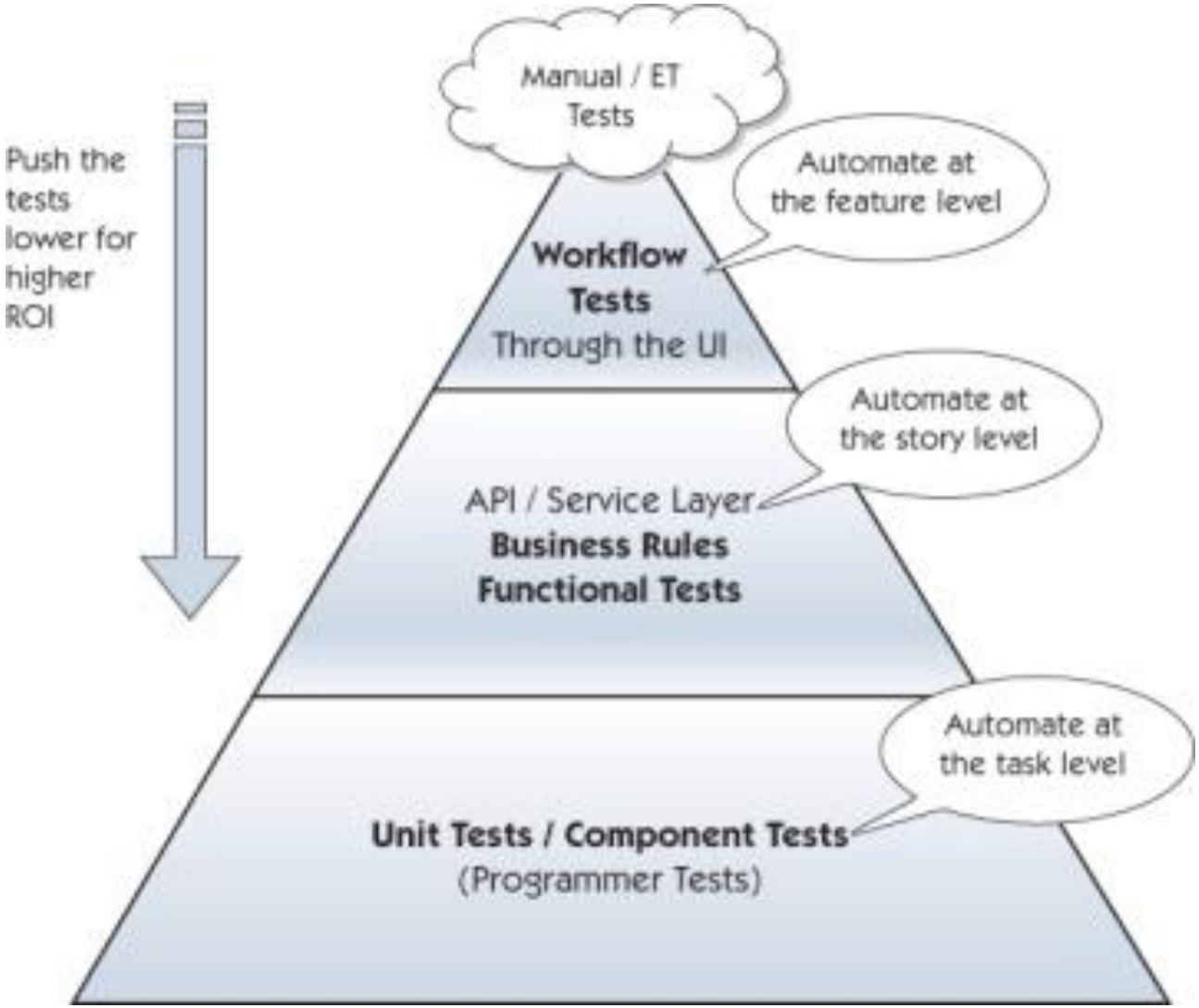
# Outline

1. Unit Testing Overview
2. Mocha & Chai
3. Creating a test suites and specs

# THE PYRAMID OF TESTS



# Levels of Testing



# Types of Testing

- Black/White Box
- Unit Testing
- Integration Testing
- Functional Tests
- System Tests
- End to End Tests
- Regression Test
- Acceptance Tests
- Load Testing
- Stress Test
- Performance Tests
- Usability Tests
- + More

# What is Unit Testing?

- A unit test **tests one unit of work** to verify that it works **as expected**
- A unit test should be:
  - Isolated and Independent from other tests
  - Repeatable
  - Predictable
- Unit tests are released into the code repository along with the code they test
- Unit testing framework is needed
  - e.g., QUnit, Jasmine, **Mocha**
  - We'll use Mocha https://mochajs.org/

# Manual Testing

- You may have already done unit testing by without using a Unit Testing framework
- **Manual tests** are less efficient
  - Not structured
  - Not repeatable
  - Not all code covered
  - Not easy to run automatically
- A Unit Testing framework enables better structure of the testing code

# Why Unit Tests?

- Help to detect bugs in early stages of the project => **improve code quality**
- Can expose high coupling => encourage refactoring to produce testable code
- Serve as live documentation
- Reduce the cost of change
- Allow refactoring with confidence (refactoring = change the code structure without changing its functionality)
- Decrease the defect-injection rate due to refactoring / changes
- Change & refactor with confidence

# Mocha Overview

- Mocha is a feature-rich framework that helps us write and run unit tests
  - Run in both the browser and on Node.js
  - Can test async code
  - Often used with Chai.js for writing **test assertions** http://chaijs.com/

```
describe('#sum', () => {
  it('when empty array, expect to return 0', () => {
    let actual = sum([])
    expect(actual).to.equal(0)
  })
  it('when with single number, expect the number', () => {
    let number = 6;
    let actual = sum([number]);
    let expected = number;
    expect(actual).to.equal(expected);
  })
})
```

# Mocha

- Test suites describe the functionality
  - Using a **describe**() function
  - Can be nested
  - Typically a single suite should be written for each JavaScript file
  - **xdescribe** to disable a test suite
- Specs test the functionality
  - Using an **it**() function to tell the test what **it** should **expect** from running a unit of code
  - Contain one or more expectations (compare **actual** with **expected** results)
  - **xit** to disable a spec

# Mocha Test Suites

- Mocha uses test suites to order the tests
  - Tests suites are created with the **describe**(name, callback) function
    - Provide a name of the test suite and a callback function

```
describe('Test Suite Name', () => {
  //here are the tests
})
```

- Test suites can be nested in one another

```
describe('Person Test Suite', () => {
  describe('when initializing', …
  describe('when changing name', …
});
```

# Mocha Specs

- Specs (tests) are contained in a test suites
  - Tests suites are created with the **it(name, callback)**
    - Has a name and a callback:

```
describe('Person Test Suite', () => {
  describe('when initializing', () => {
    it('with valid names, expect ok', () => {
      let person = new Person('Ali', 'Faleh')
      expect(person.firstname()).to.equal('Ali')
      expect(person.lastname()).to.equal('Faleh')
    })
  })
})
```

# Expectations

- Expectations are assertions that can be either true or false
- Use the **expect** function within a spec to declare an expectation
  - Receives the actual value as parameter
  - A Matcher is a comparison between the actual and the expected values
- Chai.js is a assertion framework to write expectations. It has three styles

➢ Assert style

```
assert.equal( person.getName(), 'Ali')
```

➢ Expect style

```
expect( person.getName() ).to.equal('Ali')
```

➢ Should style

```
person.getName().should.equal('Ali')
```

# Chai Expect Assertion Style

- Chai.js expect assertion style has a fluent and expressive syntax:

```
expect(person.getName()).to.equal('Ali')
expect(person).to.be.a('Person')
expect(person).to.not.be.undefined
expect(person).not.to.be.null
expect(person).to.be.ok
expect(person).not.to.be.ok
```

# Matchers

- Used to verify expectations
  - `expect(1 === 1).to.be.true`
  - `expect('b' + 'a' + 'r' ).to.not.equal('bar')`
  - `expect( 10 / 0 ).to.throw(Error)`
  - `expect( { foo: 'bar' })`
    `.to.have.property('foo') .and. equal('bar')`
  - `expect( ( ) => x.y.z ).to.throw()`

- More examples @ http://chaijs.com/api/bdd/

# Setup and Teardown

- **before** – runs before each test suite
- **after** – runs before each test suite
- **beforeEach** – runs before each test
- **afterEach** – runs after each test

```
before( async () => {
    await mongoose.connect('mongodb://localhost/books')
    await mongoose.connection.db.dropDatabase()
})

beforeEach(() => { ... })

afterEach( () => { ... })

after( async () => {
    await mongoose.disconnect()
})
```

# Good Unit Tests

- Test one thing at a time (One focus per test)
- Test results, not internals
- Test your code for different scenarios
- You want to write positive tests **and** negative tests
- Negative tests involve values that are outside acceptable ranges
  - They should fail
    - You're testing to make sure that they do
- Positive tests are ones that should pass

# Summary

- Unit Tests are an important part of any development process:
- Mocha library can help you test your JavaScript code
- Mocha uses test suites to order the tests
  - Tests suites are created with the **describe**
- Specs (tests) are contained in a test suites
  - Tests suites are created with the **it**
- Guidelines for Testable JS
  - **Modular code**: Simple, single-purpose functions
  - Don’t intermingle responsibilities

# Resources

- Mocha

https://mochajs.org/

- Chai

http://chaijs.com/