

Module 9: Testing JavaScript Code

Complete Presentation Materials

Course Information

- **Duration:** Week 15 (1 week intensive)
 - **Prerequisites:** Modules 1-8 completed
 - **Target:** Computer Science students ready for professional testing practices
 - **Focus:** Quality assurance through comprehensive testing strategies
-

Learning Objectives

By the end of this module, students will be able to:

1. Understand different types of testing and their purposes
 2. Write effective unit tests and integration tests
 3. Use testing frameworks and assertion libraries
 4. Implement mocking and stubbing for isolated testing
 5. Measure and analyze code coverage
 6. Apply Test-Driven Development (TDD) principles
 7. Design testable code architecture
 8. Debug and troubleshoot test failures
-

Module Structure

9.1 Testing Fundamentals (Sessions 1-2)

- Unit testing vs integration testing
- Test-driven development (TDD)
- Testing frameworks and assertions
- Test organization and structure
- Code coverage concepts

9.2 Advanced Testing Techniques (Sessions 3-4)

- Mocking and stubbing
 - Asynchronous testing
 - Test doubles and fixtures
 - Performance testing
 - Cross-browser testing considerations
-

9.1 Testing Fundamentals

Understanding Test Types

The Testing Pyramid

javascript

/**

* Testing Pyramid Structure:

*

* ^ E2E Tests (Few, Expensive, Slow)

* / \ - Full user workflows

* /____\ - Browser automation

* / \ - Real environments

* /_____\

*Integration Tests (Some, Moderate cost/speed)

* - Component interactions

* - API testing

* - Database integration

*

*Unit Tests (Many, Fast, Cheap)

* - Individual functions

* - Class methods

* - Pure logic testing

*/

// Example: Testing a student grade calculator

class GradeCalculator {

 calculateGPA(grades) {

 if (!grades || grades.length === 0) {

 return 0;

 }

 const total = grades.reduce((sum, grade) => sum + grade.points, 0);

 const credits = grades.reduce((sum, grade) => sum + grade.credits, 0);

 return credits > 0 ? (total / credits).toFixed(2) : 0;

 }

 getLetterGrade(gpa) {

 if (gpa >= 3.7) return 'A';

 if (gpa >= 3.3) return 'B+';

 if (gpa >= 3.0) return 'B';

 if (gpa >= 2.7) return 'B-';

 if (gpa >= 2.3) return 'C+';

 if (gpa >= 2.0) return 'C';

 if (gpa >= 1.7) return 'C-';

 if (gpa >= 1.3) return 'D+';

 if (gpa >= 1.0) return 'D';

 return 'F';

```
}  
  
isHonorStudent(gpa, creditHours) {  
  return parseFloat(gpa) >= 3.5 && creditHours >= 12;  
}  
}
```

Building a Simple Testing Framework

Custom Test Framework Implementation

javascript

// Simple Jest-like testing framework for educational purposes

```
class SimpleTestFramework {
  constructor() {
    this.tests = [];
    this.results = {
      passed: 0,
      failed: 0,
      total: 0,
      failures: []
    };
    this.currentSuite = null;
    this.beforeEachFn = null;
    this.afterEachFn = null;
    this.beforeAllFn = null;
    this.afterAllFn = null;
  }

  describe(description, testSuite) {
    console.group(` 📄 Test Suite: ${description}`);
    this.currentSuite = description;

    if (this.beforeAllFn) {
      try {
        this.beforeAllFn();
      } catch (error) {
        console.error('beforeAll failed:', error);
      }
    }

    testSuite();

    if (this.afterAllFn) {
      try {
        this.afterAllFn();
      } catch (error) {
        console.error('afterAll failed:', error);
      }
    }

    console.groupEnd();
    this.currentSuite = null;
  }
}
```

```
it(description, testFunction) {
  this.results.total++;

  try {
    if (this.beforeEachFn) {
      this.beforeEachFn();
    }

    testFunction();

    if (this.afterEachFn) {
      this.afterEachFn();
    }

    this.results.passed++;
    console.log(`✅ ${description}`);

  } catch (error) {
    this.results.failed++;
    this.results.failures.push({
      suite: this.currentSuite,
      test: description,
      error: error.message,
      stack: error.stack
    });
    console.error(`❌ ${description}`);
    console.error(`  ${error.message}`);
  }
}

beforeEach(fn) {
  this.beforeEachFn = fn;
}

afterEach(fn) {
  this.afterEachFn = fn;
}

beforeAll(fn) {
  this.beforeAllFn = fn;
}

afterAll(fn) {
  this.afterAllFn = fn;
}
```

```

}

expect(actual) {
    return new Expectation(actual);
}

runSummary() {
    console.log(`\n 📊 Test Results Summary:`);
    console.log(`Total tests: ${this.results.total}`);
    console.log(`Passed: ${this.results.passed}`);
    console.log(`Failed: ${this.results.failed}`);
    console.log(`Success rate: ${{{this.results.passed / this.results.total} * 100}.toFixed(1)}%`);

    if (this.results.failures.length > 0) {
        console.group(`\n ❌ Failed Tests:`);
        this.results.failures.forEach(failure => {
            console.log(`${failure.suite} > ${failure.test}`);
            console.log(`  Error: ${failure.error}`);
        });
        console.groupEnd();
    }

    return this.results;
}
}

// Expectation class for assertions
class Expectation {
    constructor(actual) {
        this.actual = actual;
    }

    toBe(expected) {
        if (this.actual !== expected) {
            throw new Error(`Expected ${expected}, but got ${this.actual}`);
        }
    }

    toEqual(expected) {
        if (JSON.stringify(this.actual) !== JSON.stringify(expected)) {
            throw new Error(`Expected ${JSON.stringify(expected)}, but got ${JSON.stringify(this.actual)}`);
        }
    }
}

```

```
toBeNull() {
  if (this.actual !== null) {
    throw new Error(`Expected null, but got ${this.actual}`);
  }
}

toBeUndefined() {
  if (this.actual !== undefined) {
    throw new Error(`Expected undefined, but got ${this.actual}`);
  }
}

toBeTruthy() {
  if (!this.actual) {
    throw new Error(`Expected truthy value, but got ${this.actual}`);
  }
}

toBeFalsy() {
  if (this.actual) {
    throw new Error(`Expected falsy value, but got ${this.actual}`);
  }
}

toContain(expected) {
  if (Array.isArray(this.actual)) {
    if (!this.actual.includes(expected)) {
      throw new Error(`Expected array to contain ${expected}`);
    }
  } else if (typeof this.actual === 'string') {
    if (!this.actual.includes(expected)) {
      throw new Error(`Expected string to contain "${expected}"`);
    }
  } else {
    throw new Error('toContain can only be used with arrays or strings');
  }
}

toThrow(expectedError) {
  if (typeof this.actual !== 'function') {
    throw new Error('Expected a function for toThrow assertion');
  }

  try {
```



```

    this.actual();
    throw new Error('Expected function to throw an error, but it did not');
} catch (error) {
    if (expectedError && !error.message.includes(expectedError)) {
        throw new Error(`Expected error to contain "${expectedError}", but got "${error.message}"`);
    }
}
}

toBeCloseTo(expected, precision = 2) {
    const actualRounded = Math.round(this.actual * Math.pow(10, precision)) / Math.pow(10, precision);
    const expectedRounded = Math.round(expected * Math.pow(10, precision)) / Math.pow(10, precision);

    if (actualRounded !== expectedRounded) {
        throw new Error(`Expected ${expected} (±${1/Math.pow(10, precision)}), but got ${this.actual}`);
    }
}

toHaveLength(expected) {
    if (!this.actual.hasOwnProperty('length')) {
        throw new Error('Expected object to have a length property');
    }

    if (this.actual.length !== expected) {
        throw new Error(`Expected length ${expected}, but got ${this.actual.length}`);
    }
}

toHaveProperty(property, value) {
    if (!this.actual.hasOwnProperty(property)) {
        throw new Error(`Expected object to have property "${property}"`);
    }

    if (value !== undefined && this.actual[property] !== value) {
        throw new Error(`Expected property "${property}" to have value ${value}, but got ${this.actual[property]}`);
    }
}

toBeInstanceOf(expectedClass) {
    if (!(this.actual instanceof expectedClass)) {
        throw new Error(`Expected instance of ${expectedClass.name}, but got ${this.actual.constructor.name}`);
    }
}

```

```
toBeGreaterThan(expected) {  
  if (this.actual <= expected) {  
    throw new Error(`Expected ${this.actual} to be greater than ${expected}`);  
  }  
}  
  
toBeLessThan(expected) {  
  if (this.actual >= expected) {  
    throw new Error(`Expected ${this.actual} to be less than ${expected}`);  
  }  
}  
}
```

Test-Driven Development (TDD) Example

Red-Green-Refactor Cycle

javascript

// Initialize our testing framework

```
const test = new SimpleTestFramework();
```

// TDD Example: Building a Student Management System

// Step 1: RED - Write failing tests first

```
test.describe('Student Management System - TDD Approach', () => {
```

```
  let studentManager;
```

```
  test.beforeEach(() => {
```

```
    studentManager = new StudentManager();
```

```
  });
```

// RED: This test will fail because StudentManager doesn't exist yet

```
test.it('should create a new student manager', () => {
```

```
  test.expect(studentManager).toBeTruthy();
```

```
  test.expect(studentManager.students).toEqual([]);
```

```
});
```

```
test.it('should add a student successfully', () => {
```

```
  const student = {
```

```
    id: '001',
```

```
    name: 'Alice Johnson',
```

```
    email: 'alice@university.edu',
```

```
    major: 'Computer Science'
```

```
  };
```

```
  const result = studentManager.addStudent(student);
```

```
  test.expect(result.success).toBe(true);
```

```
  test.expect(studentManager.getStudentCount()).toBe(1);
```

```
  test.expect(studentManager.getStudent('001')).toEqual(student);
```

```
});
```

```
test.it('should reject duplicate student IDs', () => {
```

```
  const student1 = { id: '001', name: 'Alice', email: 'alice@test.com', major: 'CS' }; 
```

```
  const student2 = { id: '001', name: 'Bob', email: 'bob@test.com', major: 'IT' }; 
```

```
  studentManager.addStudent(student1);
```

```
  const result = studentManager.addStudent(student2);
```

```
  test.expect(result.success).toBe(false);
```

```
  test.expect(result.error).toContain('already exists');
```

```

    test.expect(studentManager.getStudentCount()).toBe(1);
  });

  test.it('should validate required fields', () => {
    const invalidStudent = { id: '002', name: '', email: 'invalid-email' };

    const result = studentManager.addStudent(invalidStudent);

    test.expect(result.success).toBe(false);
    test.expect(result.errors).toHaveLength(2); // name and email validation
  });

  test.it('should find students by major', () => {
    const students = [
      { id: '001', name: 'Alice', email: 'alice@test.com', major: 'CS' },
      { id: '002', name: 'Bob', email: 'bob@test.com', major: 'CS' },
      { id: '003', name: 'Charlie', email: 'charlie@test.com', major: 'IT' }
    ];

    students.forEach(student => studentManager.addStudent(student));

    const csStudents = studentManager.getStudentsByMajor('CS');

    test.expect(csStudents).toHaveLength(2);
    test.expect(csStudents[0].name).toBe('Alice');
    test.expect(csStudents[1].name).toBe('Bob');
  });
});

```

// Step 2: GREEN - Write minimal code to make tests pass

```

class StudentManager {
  constructor() {
    this.students = [];
  }

  addStudent(student) {
    // Validate input
    const validation = this.validateStudent(student);
    if (!validation.isValid) {
      return {
        success: false,
        errors: validation.errors,
        error: validation.errors.join(', ')
      };
    }
  }
}

```

```

    }

    // Check for duplicates
    if (this.students.find(s => s.id === student.id)) {
        return {
            success: false,
            error: `Student with ID ${student.id} already exists`
        };
    }

    // Add student
    this.students.push({ ...student });

    return { success: true };
}

validateStudent(student) {
    const errors = [];

    if (!student.name || student.name.trim().length === 0) {
        errors.push('Name is required');
    }

    if (!student.email || !this.isValidEmail(student.email)) {
        errors.push('Valid email is required');
    }

    return {
        isValid: errors.length === 0,
        errors
    };
}

isValidEmail(email) {
    return /^[^\\s@]+@[^\\s@]+\\.^[^\\s@]+$/i.test(email);
}

getStudent(id) {
    return this.students.find(s => s.id === id);
}

getStudentCount() {
    return this.students.length;
}

```

```

getStudentsByMajor(major) {
  return this.students.filter(s => s.major === major);
}
}

```

// Step 3: REFACTOR - Improve code while keeping tests green

```

class ImprovedStudentManager {
  constructor() {
    this.students = new Map(); // Better performance for lookups
    this.majorIndex = new Map(); // Index for efficient major searches
  }

```

```

  addStudent(student) {
    const validation = this.validateStudent(student);
    if (!validation.isValid) {
      return {
        success: false,
        errors: validation.errors,
        error: validation.errors.join(', ');
      };
    }

    if (this.students.has(student.id)) {
      return {
        success: false,
        error: `Student with ID ${student.id} already exists`
      };
    }

```

// Add to main storage

```

    this.students.set(student.id, { ...student });

```

// Update major index

```

    this.updateMajorIndex(student);

```

```

    return { success: true };
  }

```

```

  updateMajorIndex(student) {
    if (!this.majorIndex.has(student.major)) {
      this.majorIndex.set(student.major, []);
    }

    this.majorIndex.get(student.major).push(student.id);

```

```

    }

    validateStudent(student) {
        const errors = [];

        if (!student.name?.trim()) {
            errors.push('Name is required');
        }

        if (!student.email || !this.isValidEmail(student.email)) {
            errors.push('Valid email is required');
        }

        if (!student.major?.trim()) {
            errors.push('Major is required');
        }

        return {
            isValid: errors.length === 0,
            errors
        };
    }

    isValidEmail(email) {
        return /^[^\\s@]+@[^\\s@]+\\.\\.[^\\s@]+$/i.test(email);
    }

    getStudent(id) {
        return this.students.get(id);
    }

    getStudentCount() {
        return this.students.size;
    }

    getStudentsByMajor(major) {
        const studentIds = this.majorIndex.get(major) || [];
        return studentIds.map(id => this.students.get(id)).filter(Boolean);
    }
}

```

Comprehensive Test Suites

Complete GradeCalculator Test Suite

javascript


```
test.describe('GradeCalculator - Comprehensive Tests', () => {
  let calculator;

  test.beforeEach(() => {
    calculator = new GradeCalculator();
  });

  test.describe('calculateGPA', () => {
    test.it('should return 0 for empty grades array', () => {
      test.expect(calculator.calculateGPA([])).toBe('0');
      test.expect(calculator.calculateGPA(null)).toBe(0);
      test.expect(calculator.calculateGPA(undefined)).toBe(0);
    });

    test.it('should calculate GPA correctly for single course', () => {
      const grades = [{ points: 12, credits: 3 }]; // A grade (4.0) * 3 credits = 12 points
      test.expect(calculator.calculateGPA(grades)).toBe('4.00');
    });

    test.it('should calculate weighted GPA for multiple courses', () => {
      const grades = [
        { points: 12, credits: 3 }, // A (4.0) * 3 = 12
        { points: 9, credits: 3 }, // B (3.0) * 3 = 9
        { points: 8, credits: 2 } // B (4.0) * 2 = 8
      ];
      // Total: 29 points / 8 credits = 3.625
      test.expect(calculator.calculateGPA(grades)).toBe('3.63');
    });

    test.it('should handle zero credit courses', () => {
      const grades = [
        { points: 12, credits: 3 },
        { points: 0, credits: 0 } // Zero credit course
      ];
      test.expect(calculator.calculateGPA(grades)).toBe('4.00');
    });

    test.it('should handle edge case with all zero credits', () => {
      const grades = [
        { points: 0, credits: 0 },
        { points: 0, credits: 0 }
      ];
      test.expect(calculator.calculateGPA(grades)).toBe('0');
    });
  });
});
```

```

});

test.it('should round to 2 decimal places', () => {
  const grades = [{ points: 10, credits: 3 }]; // 3.333...
  test.expect(calculator.calculateGPA(grades)).toBe('3.33');
});

});

test.describe('getLetterGrade', () => {
  test.it('should return correct letter grades for all ranges', () => {
    test.expect(calculator.getLetterGrade(4.0)).toBe('A');
    test.expect(calculator.getLetterGrade(3.7)).toBe('A');
    test.expect(calculator.getLetterGrade(3.69)).toBe('B+');
    test.expect(calculator.getLetterGrade(3.3)).toBe('B+');
    test.expect(calculator.getLetterGrade(3.29)).toBe('B');
    test.expect(calculator.getLetterGrade(3.0)).toBe('B');
    test.expect(calculator.getLetterGrade(2.99)).toBe('B-');
    test.expect(calculator.getLetterGrade(1.0)).toBe('D');
    test.expect(calculator.getLetterGrade(0.99)).toBe('F');
    test.expect(calculator.getLetterGrade(0)).toBe('F');
  });

  test.it('should handle string inputs', () => {
    test.expect(calculator.getLetterGrade('3.8')).toBe('A');
    test.expect(calculator.getLetterGrade('2.5')).toBe('C+');
  });

  test.it('should handle boundary conditions', () => {
    // Test exact boundary values
    test.expect(calculator.getLetterGrade(3.7)).toBe('A');
    test.expect(calculator.getLetterGrade(3.699999)).toBe('B+');
  });

});

test.describe('isHonorStudent', () => {
  test.it('should identify honor students correctly', () => {
    test.expect(calculator.isHonorStudent(3.6, 15)).toBe(true);
    test.expect(calculator.isHonorStudent('3.8', 12)).toBe(true);
  });

  test.it('should reject students with low GPA', () => {
    test.expect(calculator.isHonorStudent(3.4, 15)).toBe(false);
  });
});

```

```

test.it('should reject students with insufficient credit hours', () => {
  test.expect(calculator.isHonorStudent(3.8, 11)).toBe(false);
});

test.it('should handle edge cases', () => {
  test.expect(calculator.isHonorStudent(3.5, 12)).toBe(true);
  test.expect(calculator.isHonorStudent(3.49999, 12)).toBe(false);
});

test.it('should handle string inputs', () => {
  test.expect(calculator.isHonorStudent('3.6', '15')).toBe(true);
  test.expect(calculator.isHonorStudent('3.6', 'invalid')).toBe(false);
});
});

test.describe('Integration Tests', () => {
  test.it('should work with complete student workflow', () => {
    const studentGrades = [
      { points: 15, credits: 3 }, // A+ course
      { points: 12, credits: 4 }, // A course
      { points: 9, credits: 3 }  // B course
    ];

    const gpa = calculator.calculateGPA(studentGrades);
    const letterGrade = calculator.getLetterGrade(gpa);
    const isHonor = calculator.isHonorStudent(gpa, 10);

    test.expect(parseFloat(gpa)).toBeCloseTo(3.6, 1);
    test.expect(letterGrade).toBe('B+');
    test.expect(isHonor).toBe(true);
  });
});

test.describe('Error Handling', () => {
  test.it('should handle invalid grade objects', () => {
    const invalidGrades = [
      { points: 'invalid', credits: 3 },
      { points: null, credits: 3 },
      { credits: 3 }, // Missing points
      { points: 12 } // Missing credits
    ];

    // Should not throw errors, but handle gracefully
    test.expect(() => calculator.calculateGPA(invalidGrades)).not.toThrow();
  });
});

```

```
});

test.it('should handle negative values', () => {
  const grades = [{ points: -5, credits: 3 }];
  const result = calculator.calculateGPA(grades);
  test.expect(parseFloat(result)).toBeLessThan(0);
});
});
});
```

Test Organization and Best Practices

Test Structure and Naming Conventions

javascript

```
// Test organization following AAA pattern (Arrange, Act, Assert)
test.describe('Course Registration System', () => {
  let registrationSystem;
  let mockDatabase;
  let testStudent;
  let testCourse;

  test.beforeEach(() => {
    // Arrange - Set up test data and mocks
    registrationSystem = new CourseRegistrationSystem();
    mockDatabase = new MockDatabase();

    testStudent = {
      id: 'STU001',
      name: 'Alice Johnson',
      email: 'alice@university.edu',
      major: 'Computer Science',
      creditHours: 0
    };

    testCourse = {
      id: 'CS101',
      name: 'Introduction to Programming',
      credits: 3,
      capacity: 30,
      enrolled: []
    };

    registrationSystem.setDatabase(mockDatabase);
  });

  test.describe('Student Registration', () => {
    test.it('should successfully register eligible student', () => {
      // Arrange
      mockDatabase.setStudent(testStudent);
      mockDatabase.setCourse(testCourse);

      // Act
      const result = registrationSystem.registerStudent(testStudent.id, testCourse.id);

      // Assert
      test.expect(result.success).toBe(true);
      test.expect(result.message).toContain('successfully registered');
```

```
test.expect(mockDatabase.getCourse(testCourse.id).enrolled).toContain(testStudent.id);
});
```

```
test.it('should reject registration when course is full', () => {
  // Arrange
  const fullCourse = {
    ...testCourse,
    enrolled: new Array(30).fill().map((_, i) => `STU${i.toString().padStart(3, '0')}`)
  };
  mockDatabase.setStudent(testStudent);
  mockDatabase.setCourse(fullCourse);

  // Act
  const result = registrationSystem.registerStudent(testStudent.id, fullCourse.id);

  // Assert
  test.expect(result.success).toBe(false);
  test.expect(result.error).toContain('course is full');
  test.expect(fullCourse.enrolled).not.toContain(testStudent.id);
});
```

```
test.it('should prevent duplicate registration', () => {
  // Arrange
  const courseWithStudent = {
    ...testCourse,
    enrolled: [testStudent.id]
  };
  mockDatabase.setStudent(testStudent);
  mockDatabase.setCourse(courseWithStudent);

  // Act
  const result = registrationSystem.registerStudent(testStudent.id, testCourse.id);

  // Assert
  test.expect(result.success).toBe(false);
  test.expect(result.error).toContain('already registered');
});
});
```

```
test.describe('Prerequisite Checking', () => {
  test.it('should enforce course prerequisites', () => {
    // Arrange
    const advancedCourse = {
      id: 'CS201',
```

```
    name: 'Advanced Programming',
    credits: 3,
    capacity: 25,
    prerequisites: ['CS101'],
    enrolled: []
  };
```

```
mockDatabase.setStudent(testStudent);
mockDatabase.setCourse(advancedCourse);
```

```
// Act
```

```
const result = registrationSystem.registerStudent(testStudent.id, advancedCourse.id);
```

```
// Assert
```

```
test.expect(result.success).toBe(false);
test.expect(result.error).toContain('prerequisites not met');
```

```
});
```

```
test.it('should allow registration when prerequisites are satisfied', () => {
```

```
  // Arrange
```

```
  const studentWithPrereqs = {
    ...testStudent,
    completedCourses: ['CS101']
  };

```

```
  const advancedCourse = {
    id: 'CS201',
    name: 'Advanced Programming',
    credits: 3,
    capacity: 25,
    prerequisites: ['CS101'],
    enrolled: []
  };

```

```
  mockDatabase.setStudent(studentWithPrereqs);
  mockDatabase.setCourse(advancedCourse);
```

```
  // Act
```

```
  const result = registrationSystem.registerStudent(studentWithPrereqs.id, advancedCourse.id);
```

```
  // Assert
```

```
  test.expect(result.success).toBe(true);
```

```
});
```

```
});
```

```
test.describe('Credit Hour Validation', () => {
  test.it('should prevent over-enrollment (> 18 credit hours)', () => {
    // Arrange
    const studentNearLimit = {
      ...testStudent,
      creditHours: 16
    };

    const highCreditCourse = {
      ...testCourse,
      credits: 4
    };

    mockDatabase.setStudent(studentNearLimit);
    mockDatabase.setCourse(highCreditCourse);

    // Act
    const result = registrationSystem.registerStudent(studentNearLimit.id, highCreditCourse.id);

    // Assert
    test.expect(result.success).toBe(false);
    test.expect(result.error).toContain('exceeds credit hour limit');
  });

  test.it('should allow registration within credit limits', () => {
    // Arrange
    const studentWithRoom = {
      ...testStudent,
      creditHours: 12
    };

    mockDatabase.setStudent(studentWithRoom);
    mockDatabase.setCourse(testCourse);

    // Act
    const result = registrationSystem.registerStudent(studentWithRoom.id, testCourse.id);

    // Assert
    test.expect(result.success).toBe(true);
  });
});
```

9.2 Advanced Testing Techniques

Mocking and Test Doubles

Creating Mock Objects

javascript

```
// Mock Database for testing
class MockDatabase {
  constructor() {
    this.students = new Map();
    this.courses = new Map();
    this.enrollments = new Map();
    this.callHistory = [];
  }

  // Spy functionality - track method calls
  recordCall(method, args) {
    this.callHistory.push({
      method,
      args: [...args],
      timestamp: Date.now()
    });
  }

  async getStudent(id) {
    this.recordCall('getStudent', [id]);

    // Simulate network delay
    await this.simulateDelay(50);

    const student = this
```