

Objectives

- Unit Testing

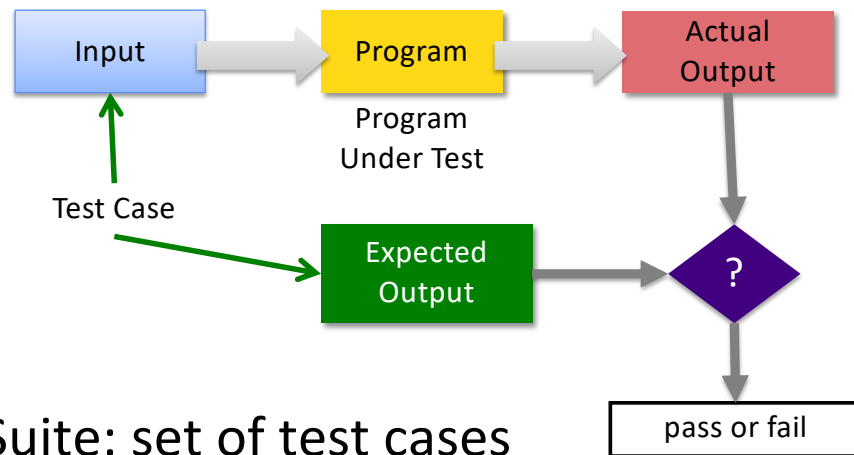
1

Review

1. Describe the general testing process
2. What is a set of test cases called?
3. What is *unit testing*?
4. What are the benefits of unit testing?
5. What are the characteristics of good unit tests?
6. What are the steps in a JUnit Test Case?
 - How do we implement those steps?
7. What is test-driven development?

2

Review: Software Testing Process



- Test Suite: set of test cases

Oct 25, 2021

Sprenkle - CSCI209

3

3

Review: Why Unit Test?

- Verify code works as intended in isolation
- Find defects **early** in development
 - Easier to test small pieces
 - Less cost than at later stages (e.g., when integrating)
- Suite of (small) test cases to run after code changes
 - As application evolves, new code is more likely to break existing code
 - Also called **regression** testing

Oct 25, 2021

Sprenkle - CSCI209

4

4

Review: Characteristics of Good Unit Testing

- **Automatic**
 - Since unit testing is done frequently, don't want humans slowing the process down
 - Automate executing test cases and evaluating results
 - Input: in test itself or from a file
- **Thorough**
 - Covers all code/functionality/cases
- **Repeatable**
 - Reproduce results (correct, failures)
- **Independent**
 - Test cases are independent from each other
 - Easier to trace fault to code

Oct 25, 2021

Sprenkle - CSCI209

5

5

Review: Structure of a JUnit Test

1. Set up the test case (optional)
 - Example: Creating objects
 - `@BeforeAll` (once per class), `@BeforeEach` (before each test)
2. Exercise the code under test
 - Within `@Test` method
3. Verify the correctness of the results
 - Within `@Test` method – use assert methods
4. Teardown (optional)
 - Example: reclaim created objects
 - `@AfterEach` (after each test), `@AfterAll` (once per class)

Oct 25, 2021

Sprenkle - CSCI209

6

6

Review: Assert Methods

- Defined in `org.junit.jupiter.api.Assertions`
 - Variety of assert methods available
- If fail, throw an error
- Otherwise, test keeps executing
- All are **static void**
- Example: `assertEquals(Object expected, Object actual)`

```
@Test
public void addTest() {
    ...
    assertEquals(4, calculator.add(3, 1));
}
```

Oct 25, 2021

7

7

Review: Example Testing the CD class

```
private CD testCD;

@BeforeEach
public void setUp() {
    testCD = new CD("CD title", "CD Artist", 100, 1997, 11, false);
}

@Test
public void testInCollection() {
    assertFalse( testCD.isInCollection() );
    testCD.setInCollection();
    assertTrue( testCD.isInCollection() );
}
```

Exercising the code and verifying its correctness

Oct 25, 2021

Sprenkle - CSC1209

8

8

Review: Expecting an Exception

- Sometimes an exception *is* the expected result

```
@Test
public void testIndexOutOfBoundsException() {
    List emptyList = new ArrayList();

    assertThrows(IndexOutOfBoundsException.class,
        () -> { Object o = emptyList.get(0); }
    );
}
```

Test case passes only if exception is thrown

Expecting an Exception: Breaking It Down

[assertThrows\(Class<T> expectedType, Executable executable\)](#)

```
@Test
public void testIndexOutOfBoundsException() {
    List emptyList = new ArrayList();

    assertThrows(IndexOutOfBoundsException.class,
        () -> { Object o = emptyList.get(0); }
    );
}
```

How to read assertThrows:
Execute the highlighted code (in {})
and check if it throws that exception type

A lot more can be said about lambda expressions... but not in CSCI209

Expecting an Exception

- Can also check characteristics of the thrown exception

```
@Test
public void testIndexOutOfBoundsException() {
    List myList = new ArrayList();
    IndexOutOfBoundsException ioobExc =
        assertThrows(IndexOutOfBoundsException.class, () -> {
            myList.get(0);
        });
    System.out.println(ioobExc.getMessage());
    assertEquals("Index 0 out of bounds for length 0",
        ioobExc.getMessage());
}
```

Test case passes only if exception is thrown
and message matches

Oct 25, 2021

Sprenkle - CSCI209

11

11

Review: Some Approaches to Testing Methods

- Typical case
 - Test typical values of input/parameters
- Boundary conditions
 - Test at boundaries of input/parameters
 - Many faults live “in corners”
- Parameter validation
 - Verify that parameter and object bounds are documented and checked
 - Example: pre-condition that parameter isn't null

➔ All black-box testing approaches

Oct 25, 2021

12

12

EVALUATING TEST SUITES

Oct 25, 2021

Sprenkle - CSCI209

13

13

Evaluating Test Suites

- Software testing research question:
Is my approach to generating a test suite better than the state-of-the-art test suite generation?
- One approach to answer question:
Fault-based Evaluation
 - Given known faults (a.k.a. mutants)
 - How many faults/mutants does my test suite kill/unveil?
 - *Kill* a fault by creating a test case that fails when exercising that fault

Oct 25, 2021

Sprenkle - CSCI209

14

14

Lab: Catching the Mutants

- Set Up
 - Use of jar file (contains mutant class files)
 - Classpath – tell compiler/JVM to use JUnit and mutants.jar
- Objective: Practice writing JUnit test cases
- Goal: reveal all the bugs/mutants!

Oct 25, 2021

Sprenkle - CSCI209

15

15

Lab: Catching the Mutants

- Objective: Practice writing JUnit test cases
- Goal: reveal all the bugs/mutants!
- Why designed this way:
 - You get feedback on if you've tested "enough"
 - Practice testing – knowing how much more you need to do
 - Not typically known in the real world!

Oct 25, 2021

Sprenkle - CSCI209

16

16

Catching the Mutants: Post-Mortem

- What are the benefits of unit testing/using JUnit?
 - Consider if you were developing/maintaining the method
 - How would your testing/development process change?
- Why did the output come out in strange orders sometimes?
- Is it okay that some mutants passed some of the test cases?
- Recall the characteristics of good unit tests
 - How did you achieve them in your testing?

Oct 25, 2021

Sprenkle - CSCI209

17

17

Are These Effective Tests?

```
@Test
public void testThirdShortest() {
    String[] words = { "a", "ab", "abc" };
    String actual = mutant.thirdShortest(words);
    assertEquals(3, actual.length());
}
```

```
@Test
public void testExceptionThrown() {
    String[] words = { "a" };
    assertThrows(Exception.class, () -> {
        mutant.thirdShortest(words);
    });
}
```

Oct 25, 2021

Sprenkle - CSCI209

18

18

Test Discussion

- They are correct tests
 - They will reveal bugs
- However, they are *weak* tests
 - Cover necessary invariants, but they are not sufficient to expose failures

```
@Test
public void testThirdShortest() {
    String[] words = { "a", "ab", "abc" };
    String actual =
        mutant.thirdShortest(words);
    assertEquals("abc", actual);
}
    Check the actual result
```

```
@Test
public void testExceptionThrown() {
    String[] words = { "a" };
    assertThrows(IllegalArgumentException.class,
        () -> {
            mutant.thirdShortest(words);
        });
}
    Expect the exact exception
```

Oct 25, 2021

Sprenkle - CSCI209

19

19

Testing More Than One Possible Answer

- `thirdShortest` only returns one answer (a String) but there could be multiple different correct answers
 - We can discuss if this is the best design but ...
- Example test

```
@Test
public void testMoreInArray2() {
    String[] words = { "a", "b", "bc", "ab", "bye", "and" };
    String result = mutant.thirdShortest(words);
    assertTrue(result.equals("bye") || result.equals("and"));
}
    20
```

20

Is This An Effective Test?

```

@Test
public void testAll() {
    String[][] tests = { { "a", "ab", "abc" },
        { "1", "12", "12345", "12345345", "234oi34iuwer" },
        { "cba", "abc", "bca", "a", "a", "a", "ab", "ab", "ab" } };
    assertEquals(mutant.thirdShortest(tests[0]), "abc");
    assertEquals(mutant.thirdShortest(tests[1]), "12345");
    assertTrue(mutant.thirdShortest(tests[2]).equals("cba") ||
        mutant.thirdShortest(tests[2]).equals("abc") ||
        mutant.thirdShortest(tests[2]).equals("bca"));
    assertThrows(IllegalArgumentException.class, () -> {
        mutant.thirdShortest(null) });
    assertThrows(IllegalArgumentException.class, () -> {
        mutant.thirdShortest(new String[]{}); });
    assertThrows(IllegalArgumentException.class, () -> {
        mutant.thirdShortest(new String[]{"hey"}); });
    assertThrows(IllegalArgumentException.class, () -> {
        mutant.thirdShortest(new String[]{"hey", "there"}); });
    String[] words = { "abcde", "b", "bc", "ab", "bye", "and" };
    String[] original = { "abcde", "b", "bc", "ab", "bye", "and" };
    result = mutant.thirdShortest(words);
    assertTrue(result.equals("bye") || result.equals("and"));
    assertEquals(Arrays.asList(words), Arrays.asList(original));
    ...
}

```

21

Is This An Effective Test?

```

@Test
public void testAll() {
    String[][] tests = { { "a", "ab", "abc" },
        { "1", "12", "12345", "12345345", "234oi34iuwer" },
        { "cba", "abc", "bca", "a", "a", "a", "ab", "ab", "ab" } };
    assertEquals(mutant.thirdShortest(tests[0]), "abc");
    assertEquals(mutant.thirdShortest(tests[1]), "12345");
    assertTrue(mutant.thirdShortest(tests[2]).equals("cba") ||
        mutant.thirdShortest(tests[2]).equals("abc") ||
        mutant.thirdShortest(tests[2]).equals("bca"));
    assertThrows(IllegalArgumentException.class, () -> {
        mutant.thirdShortest(null) });
    assertThrows(IllegalArgumentException.class, () -> {
        mutant.thirdShortest(new String[]{}); });
    assertThrows(IllegalArgumentException.class, () -> {
        mutant.thirdShortest(new String[]{"hey"}); });
    assertThrows(IllegalArgumentException.class, () -> {
        mutant.thirdShortest(new String[]{"hey", "there"}); });
    String[] words = { "abcde", "b", "bc", "ab", "bye", "and" };
    String[] original = { "abcde", "b", "bc", "ab", "bye", "and" };
    result = mutant.thirdShortest(words);
    assertTrue(result.equals("bye") || result.equals("and"));
    assertEquals(Arrays.asList(words), Arrays.asList(original));
    ...
}

```

May be effective but hard to use
 Tests are not independent
 Will be hard to pinpoint bugs

22

Guidance for Writing Tests

- Group tests in methods, classes
 - Class could be by behavior, by error conditions, ...
- Test methods should focus on one behavior
 - If test case fails, should be helpful in narrowing down where the problem is
- See examples on course schedule

Oct 25, 2021

Sprenkle - CSCI209

23

23

Review: Test-Driven Development

- A development style, evolved from Extreme Programming
- Idea: write tests first *without code bias*
- The Process:
 1. Write tests that code/new functionality should pass
 - Like a specification for the code (pre/post conditions)
 - All tests will initially *fail*
 2. Write the code and verify that it passes test cases
 - Know you're done coding when you pass **all** tests

How do you know you're "done" in traditional development?

What assumption does this make?

Sprenkle - CSCI209

24

24

Project: Test-Driven Development

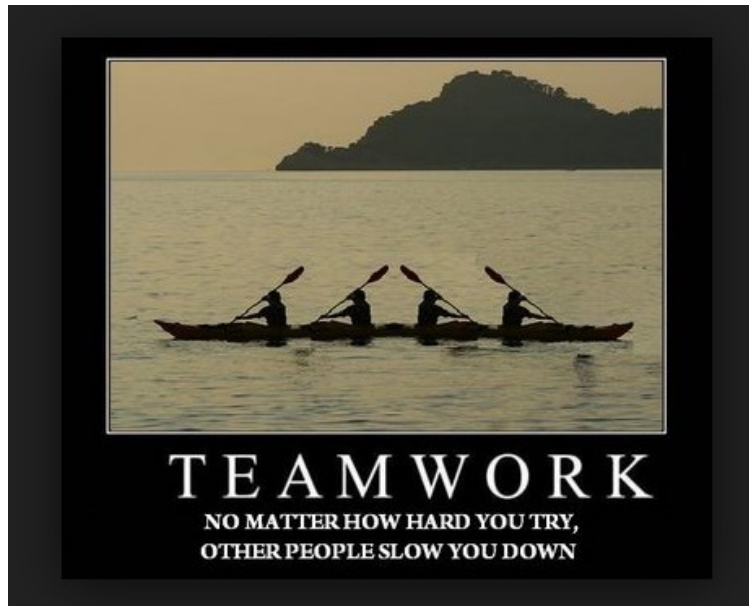
- Given: a `Car` class that only has enough code to compile
- Your job: Create a **good** set of test cases that **thoroughly/effectively** test `Car` class
 - Find faults in my faulty version of `Car` class
 - Start: look at code, think about how to test, set up JUnit tests
 - Written analysis of process
- First team project: teams of **3**
 - Practice collaboration
 - Every student must commit code to the repository
- Due before 5 p.m. today
 - First step: create teams (and *team names!*) today

Oct 25, 2021

Sprenkle - CSCI209

25

25



Oct 25, 2021

Sprenkle - CSCI209

26

26

Think about Team (Group) Projects

- Why did some work well?
- Why were some disasters?

Oct 25, 2021

Sprenkle - CSCI209

27

27

Teams Work Best When They are **Interdependent**

- In code terms, we want *loose coupling*
 - Depend on each other but don't depend on their details
- Consider
 - Are you allowing your team to truly be interdependent?
 - Who might be you be ignoring?
 - Who might be allowing themselves to feel inadequate?
 - How do you show appreciation for each other and yourself?

Oct 25, 2021

Sprenkle - CSCI209

28

28

Collaboration: Team Project

- Need to talk about the solution
- Discuss your plan, e.g.,
 - Your system for testing to make sure that you test everything
 - Your assumptions about the Car class
 - Organization of test cases
 - Naming
 - Division of labor
- Maintain planning documents too
 - in GitHub or elsewhere

Oct 25, 2021

Sprenkle - CSCI209

29

29

Collaboration: Team Project

- Version Control does not eliminate need for communication
 - Process becomes much more difficult if developers do not communicate
- Keep the version to be graded in **main** branch
- Before picking up again, **pull** the repository
 - Get others' changes

Oct 25, 2021

Sprenkle - CSCI209

30

30

Collaboration: Workflow – Seeking Feedback

1. Create a branch for your work
 - Commit periodically
 - Write descriptive comments so your team members know what you did and why
2. Push your branch
3. Open a **Pull Request** on your branch
 - Discuss and review potential changes – can still update
 - You can tag your teammates to let them know that you've completed your work
4. Merge pull request into main branch
5. Pull the main branch to get the latest code

Oct 25, 2021

Sprenkle - CSCI209

31

31

Collaboration: Workflow

1. Create a branch for your work
 - Commit periodically
 - Write descriptive comments so your team members know what you did and why
2. Switch to main
3. Pull main branch
4. Merge your branch into the main branch
 - Handle merge conflicts
 - Commit
5. Push main branch

Oct 25, 2021

Sprenkle - CSCI209

32

32

Looking Ahead

- Testing Project due next Tuesday at 11:59 p.m.
 1. THINK
 2. DISCUSS as a team
 3. Then write the tests
- Teams finalized today
- Lab was an in-class exercise
 - Practice JUnit testing before project