Lecture Notes
October 29, 2020

“Assignment T2: Revised Project Proposal” due next week
https://courseworks2.columbia.edu/courses/104335/assignments/486956

You must meet with your primary IA mentor *before* submitting! Let’s try to have quick meetings right now.

I cannot automatically assign people to breakout rooms unless everyone logs in with their uni, so instead here is a list matching teams to breakout room, please try to go to the right room. The IAs will have to move between rooms.

Tentative Mentor Allocation

Working in a team: unit testing

What is a “unit”? The smallest piece of code that can be logically isolated.

Depends on the programming language, but generally a method, function, procedure, subroutine.

Small interconnected sets of such units, such as classes, modules, files, are typically tested during unit testing as well as individual units.

When does unit testing happen? Ideally all changed code is unit-tested before commit (e.g., git pre-commit hook), and the commit is aborted if there are any failures.

Unit testing is nearly always automated, where we create a set of unit tests, push a button, and wait.

The wait is usually negligible per unit, but can add up if a lot of units are tested at the same time. Many organizations conduct “nightly builds” where the entire codebase is rebuilt and all tests are executed.

Every major modern programming language has at least one widely-used unit testing framework.

C/C++: GoogleTest, CUTE

Java: JUnit, TestNG

Javascript: Jest, Mocha

Python: unittest, pytest

You should all have experience with JUnit from the individual project.

```java
/**
 * Test that it is possible to make a move to an unoccupied position
 * that is at a valid location on the gameboard.
 */
@Test
@DisplayName("Moves are allowed in unoccupied positions.")
void testIsValidMoveTrue() {
  char[][] boardState = { { 0, 0, 'O' }, { 0, 0, 'X' }, { 0, 0, 0 } };
  emptyTestBoard.setBoardState(boardState);
  Move attemptedMove = new Move(player1, 0, 1);
  assertEquals(true, emptyTestBoard.isValidMove(attemptedMove));
}
```

Unit testing frameworks provide facilities that help with writing and automating tests, such as assertions, test fixtures, and test runners.

Terminology may vary among tools. For example, “test case” sometimes means literally a single test, sometimes means a test class consisting of several related tests.

Assertion = notation for checking the results of executing a test. Usually supports checking for true/false result, checking whether an exception has been raised, and comparing an output to a known value.

```
  /**
   * Test that the played move is reflected in the gameboard
configuration.
   */
  @Test
  @DisplayName("Played move should reflect in gameboard.")
  void testPlayMove() {
    char[][] startingBoardState = { { 0, 0, 'O' }, { 0, 0, 'X' }, { 0, 0,
0 } };
    emptyTestBoard.setBoardState(startingBoardState);
    Move move = new Move(player1, 0, 1);
    emptyTestBoard.playMove(move);
    char[][] expectedBoardState = { { 0, 'X', 'O' }, { 0, 0, 'X' }, { 0,
0, 0 } };
    assertArrayEquals(expectedBoardState, emptyTestBoard.getBoardState());
  }
```

Test fixture = setup before running one or more tests, teardown after running those tests. Setup can create objects, open files or network/database connections, etc. shared by several tests, teardown cleans up afterwards.

```
/**
 * Setup an empty and active GameBoard for each test.
 */
@BeforeEach
void setGameboard() {
  this.emptyTestBoard = new GameBoard();
  this.activeTestBoard = new GameBoard(player1, player2, true, 1,
      emptyBoard, 0, false);
}
```

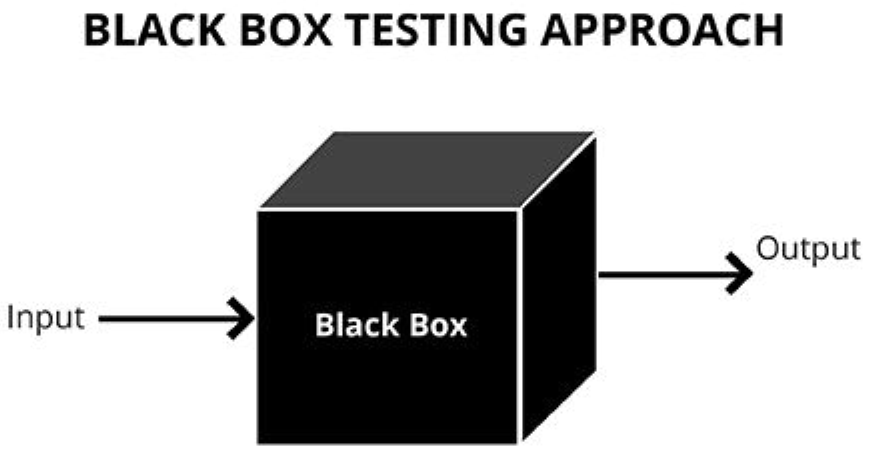
Test runner = executes a test suite in some order and reports results. Usually does not stop when a test fails - instead runs all the tests and reports all the results. Ordering might be built-in or specified by the tester.

How should we choose inputs for unit tests?

“Inputs” to units may include parameters, global variables (e.g., static fields in classes), data entered by the user, data received from network/database/files/devices, return values from API calls and their side-effects visible inside the code.

“Outputs” from units may include return values from the code, changes to global variables visible to other code after the code returns, data displayed to the user, data sent to network/database/files/devices, data supplied to APIs that becomes visible outside the code.

In most testing frameworks, assertions (which implement test oracles) can be parameterized by any of these kinds of inputs and outputs.



But as with system tests, the unit tests for a particular unit should include at least some simple “test to pass” cases.

```
  /**
   * Test that the played move is reflected in the gameboard
configuration.
   */
  @Test
  @DisplayName("Played move should reflect in gameboard.")
  void testPlayMove() {

    char[][] startingBoardState = { { 0, 0, 'O' }, { 0, 0, 'X' }, { 0, 0,
0 } };
    emptyTestBoard.setBoardState(startingBoardState);

    Move move = new Move(player1, 0, 1);
    emptyTestBoard.playMove(move);
    char[][] expectedBoardState = { { 0, 'X', 'O' }, { 0, 0, 'X' }, { 0,
0, 0 } };

    assertArrayEquals(expectedBoardState, emptyTestBoard.getBoardState());
  }
```

But focus should be on “test to fail” cases intended to reveal bugs in the functionality.

As with system tests, the “test to fail” cases should include some *invalid* inputs. The goal is not that these tests should actually fail, but instead that the invalid inputs are detected by the code and handled appropriately. The assertions should check that the expected handling indeed happens, so the test passes.

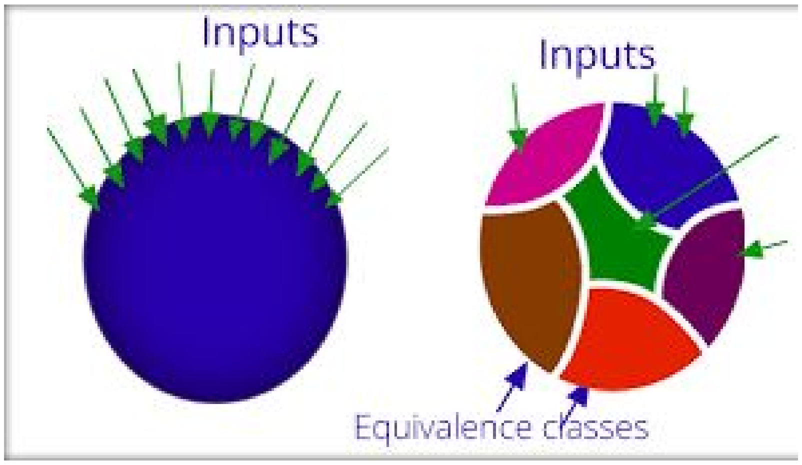
```
  /**
   * Test that it is not possible to make moves to positions off the
gameboard
   * (row out of range).
   */
  @Test
  @DisplayName("Moves are not allowed in positions off game board "
      + "(row doesn't exist #1).")
  void testIsValidMoveFalseRowOffBoard1() {

    char[][] boardState = { { 0, 0, 'O' }, { 0, 0, 'X' }, { 0, 0, 0 } };
    emptyTestBoard.setBoardState(boardState);

    Move attemptedMove = new Move(player1, 3, 0);
    assertEquals(false, emptyTestBoard.isValidMove(attemptedMove));
  }
```

Error handling typically involves checking the validity of each input, particularly external inputs. Then conditionals in the error handling code return a predefined status code and/or construct or select a string that describes the error.

Units may not correspond in a direct way to user stories or use cases, so may be hard to apply conditions of satisfaction or basic/alternative steps to choosing inputs.

But both valid and invalid inputs to units can be selected randomly or via equivalence partitions/boundary analysis.

Divide the input space into *equivalence partitions* (equivalence classes), typically based on data types at the unit testing level.

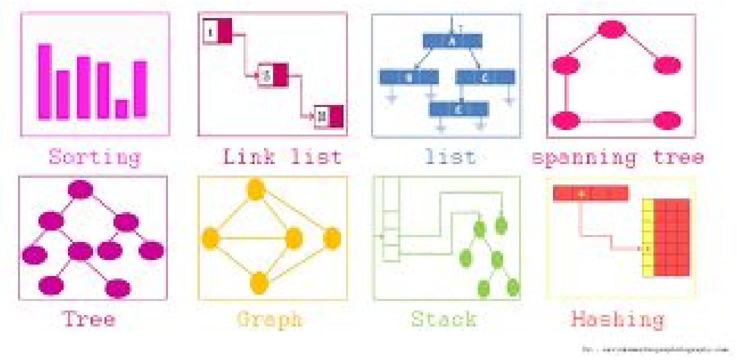


An equivalence partition is a set of alternative input values where the software can reasonably be expected to behave equivalently (similarly).

```java
/**
 * Test that it is possible to make a move to an unoccupied position
 * that is at a valid location on the gameboard.
 */
@Test
@DisplayName("Moves are allowed in unoccupied positions.")
void testIsValidMoveTrue() {

  char[][] boardState = { { 0, 0, 'O' }, { 0, 0, 'X' }, { 0, 0, 0 } };
  emptyTestBoard.setBoardState(boardState);

  Move attemptedMove = new Move(player1, 0, 1);
  assertEquals(true, emptyTestBoard.isValidMove(attemptedMove));
}
```

In the unit testing case, the members of an equivalence class typically follow the same leg of at least one branch (true or false), but they may not be the only inputs that take that path.

At the unit testing level, inputs/outputs can be arbitrary data structures, so equivalence classes can be more complex - and inputs more complicated to construct



It is possible to construct sample inputs at unit level that could never actually be supplied when the unit is executed within the full program

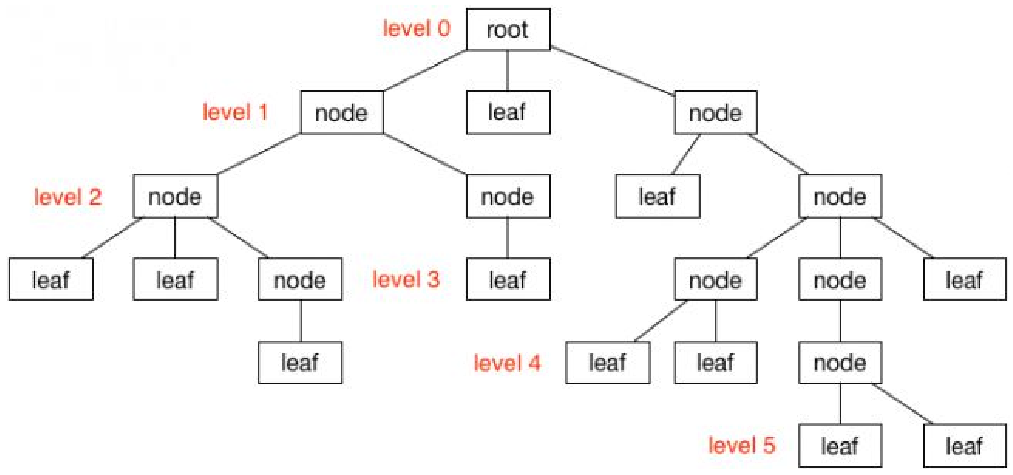
If those inputs find bugs, are these false positives or true positives?

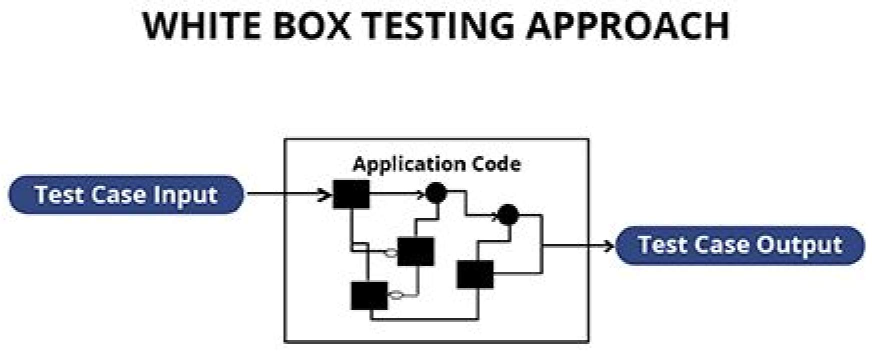
Container data structures with content elements:

- In or not in container
- Empty or full container (or min/max number of elements)
- Duplicate elements may or may not be allowed
- Ordered or organized correctly, or not

Specific kinds of containers, e.g., tree - root node, interior node, leaf node, null tree, tree with exactly one node, “full” tree, balanced vs. unbalanced, various specialty trees - and graphs

How would test cases construct root, node and leaf objects to use as inputs?





Since the developer who wrote the unit is often the same person who writes the unit tests, white box testing comes naturally and white box tools are often integrated with unit testing tools.

It’s usually much easier to force specific branches with inputs to the enclosing unit than with inputs to the entire application.

When covering all branches, beware the missing else!

```
if (condition) { do something }
else { do something else }
// some other code

if (condition) { do something }
// some other code
```

```
  /**
   * Plays the Move submitted, adding it to the board and checking to see
if it
   * was a winning move; if so, update the board to reflect the change in
board
   * state.
   *
   * @param move Instance of Move object representing player and position
to play
   */
  public void playMove(Move move) {
    int x = move.getMoveX();
    int y = move.getMoveY();
    char type = move.getPlayer().getType();

    this.boardState[x][y] = type;

    if (isWinningMove(x, y, type)) {
      int playerId = move.getPlayer().getId();
      this.setWinner(playerId);
    }
```

```
  }
```

“Assignment T2: Revised Project Proposal” due next week
https://courseworks2.columbia.edu/courses/104335/assignments/486956

You must meet with your primary IA mentor *before* submitting! The entire team should try to attend, but it’s better for a subset of your team to meet with your mentor asap than to wait for more convenient scheduling.