CIS640 Mid term Exam

03/16/2017 @ 1:05PM

Maximum points: 60 + 9

Duration: 75 minutes

Instructions:

A. On the first page of your solution, please write your name, KSU ID, and the total number of pages in your solution.
B. Number each page used in your solution.
C. Make sure you sign the roster both before starting the exam and after completing the exam.
D. The exam is closed resource. So, access to any sort of resources (including mobile phones/devices) is not permitted during the exam. Except pen, pencil, and eraser, all resources should be placed under the table.
E. Use of mobile phones during the exam is not allowed. So, please turn them off.
F. Please return the exam with your solution.

Questions:

1. With code example(s), illustrate the concepts of fault, error, failure, and the possibilities in terms of their activation and propagation. **(8 points) (4 points** if the illustration is based on an example not similar to the one discussed in class.**)**

```
def add2(x):  # 1 point for code
    y = x * 2 # fault: should be x + 2
    if y == 6: # line A
        y -= 1 # line B
    return y
```

When x is 2, fault is desensitized as 2*2 = 2+2. [1 point]

When x is 3 [1 point], fault is sensitized (as 3*2 != 3+2) [1 point] but error is masked (as the expected value 5 is returned) due to lines A and B [1 point].

When x is 1 [1 point], fault is sensitized (as 1*2 != 1+2) [1 point] and error is propagated (as 2 is returned instead of 3) leading to a failure [1 point].

2. Define fault by commission and fault by omission with an example. **(4 points)**

Fault by commission is a fault stemming from incorrect implementation, e.g., using x*2 instead of x+2. **Fault by omission** is a fault stemming from absence of correct implementation, e.g., using x%3 instead of (x*2)%3.

3. What are the differences between black-box testing and white-box testing? **(4 points)**

Black-box testing is based on specification (independent of implementation) and it cannot deal with all exhibited (observable) behaviors of UUT.

White-box testing is based on implementation (independent of specification) and it cannot deal with unsupported behaviors. [It also can lead to brittle tests.]

4. List four aspects of testing that can be automated to make testing better/easy. **(4 points)**
   - Test data generation
   - Test case generation
   - Test execution
   - Test result collection and compilation

5. What kind of actions are (typically) performed in test fixture methods? **(3 points)**
   - Put the UUT in the state required for testing,
   - Perform actions common to a set of test cases, and
   - Perform clean up actions.
6. Each test double pattern is well-suited for certain situations. What are such situations for Fake, Stub, and Spy? **(3 points)**
   Fake is well suited when DOC cannot be used or we want to avoid slow tests.
   Stub is well suited when we want to control indirect input from DOC to SUT.
   Spy is well suited when we want to capture (and check) indirect output from SUT to DOC.
7. What is the typical structure of parameterized unit tests? **(5 points)**
   - Setup,
   - Check assumptions about parameters,
   - Execute,
   - Verify, and
   - Teardown
8. Write example-based unit tests in Python to test the function `search(ints, element)` that searches for `element` in `ints`. The function expects `ints` to be a list of integers and `element` to be an integer. It should return `True` when `element` is in `ints`; `False`, otherwise. It should raise RuntimeError exception if the inputs are not as expected. **(13 points)**

```
def test_ints_is_not_list():
  with pytest.raises(RuntimeError):  # 1 point
    search("as", 3)  # element should be valid / 1 + 1 points

def test_ints_is_not_list_of_integers():
  with pytest.raises(RuntimeError):  # 1 point
    search(['a', 'b'], 3)  # element should be valid / 1 + 1 points

def test_element_is_not_integer():
  with pytest.raises(RuntimeError): # 1 point
    search([1,3,4], 'a')  # ints should be valid / 1 + 1 points

def test_element_in_ints():
  assert search([1,5,3], 3)  # 2 points

def test_element_not_in_ints():
  assert not search([1,5,3], 4)  # 2 points
```

9. Consider a variant of `search` function from Q8 that implements search via binary search. Binary search works only on sorted arrays and this variant of `search` does not sort `ints`. Modify your test suite for Q8 to test this variant. **(2 points)**
   We will use the first three tests as is along with the following tests.

```
def test_element_in_sorted_ints():
  assert search([1,3,5], 3)  # 1 point
```

```
def test_element_not_in_sorted_ints():
  assert not search([1,3,5], 4)  # 1 point

def test_existing_element_not_found_in_unsorted_ints():
  assert not search([1,2,5,4,3], 4)  # 1 point
```

10. Write a parameterized unit test in Python (along with parameter decorators) to test the `search` function in Q8 returns `True` when `element` is in `ints`. **(5 points) (2 points** if the PUT uses least number of parameters (> 0).**)**

```
@pytest.mark.parametrize("ints", [[1,3,4,7], [7,3,10]])  # 1 points
@pytest.mark.parametrize("element", [3,7])  # 1 points
def test_element_in_ints(ints, element):  # 1 point
    assert search(ints, element) # 2 points

@pytest.mark.parametrize("ints", [[1,3,4], [-7,5,10]])  # 2 points
def test_element_in_ints(ints):  # 1 point
  for i in ints:  # 2 points
    assert search(ints, i) # 2 points
```

11. Write a parameterized unit test in Python to test the function `permute(ints)` that returns a permutation/reordering [with repetition] of the given integers. `values` is a list of integers. The function should not modify the input list. **(12 points)**

```
import collections
@pytest.mark.parametrize("values", [[1,3,4], [-7,5,10]])  # 2 points
def test_permute(values):
  tmp1 = list(values)  # 2 point
  result = permute(values)  # 1 point
  assert tmp1 == values  # 2 point
  tmp1 = collections.Counter(result)  # 3 points
  tmp2 = collections.Counter(values)
  assert tmp1 == tmp2  # 2 point
```