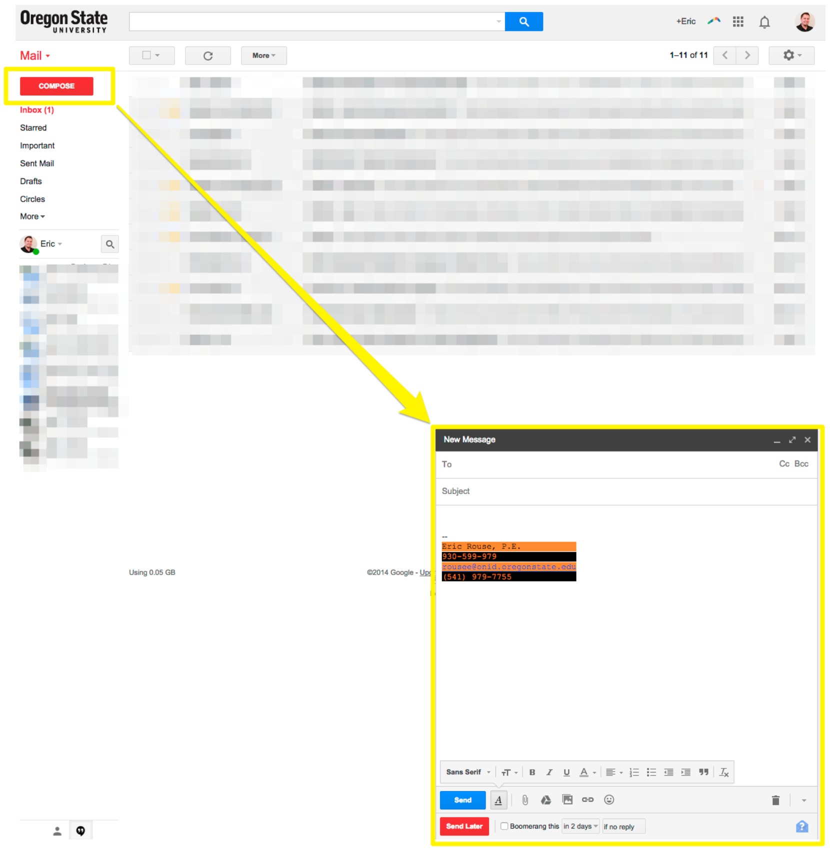
Eric Rouse
Arpit Christi; christia@onid.oregonstate.edu
Hongyan Yi; yih@onid.oregonstate.edu
December 7, 2014

CS362 Final Exam

If a system contains a defect, and the customer is never affected by the defect, then does the defect exist? Does the system have "quality?"
http://www.yitsplace.com/Programming/sw_test_phil.htm

Question 1: Manual Testing

I suppose if I were asked to manually test the compose feature of google mail, I'd first log in to google mail and click the compose button just to see what happens.

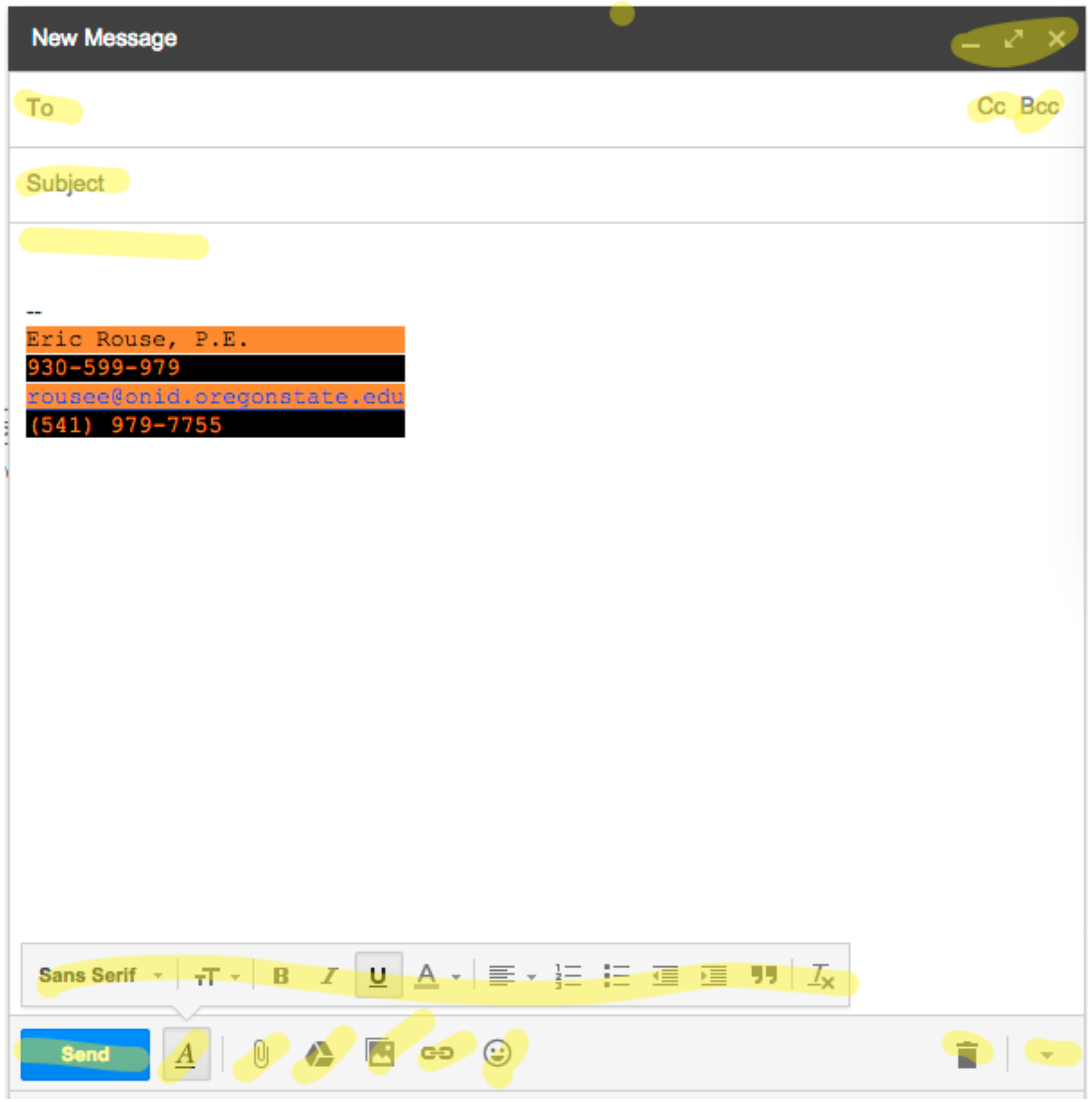


--
Eric Rouse, P.E.
930-599-979
rousee@onid.oregonstate.edu
(541) 979-7755

Since the only discernible action associated with the clicking compose button is the opening of a composition window I'd assume that, at least in part, I was being asked to verify the functionality of the window popping open when the compose button was clicked.

Thus, in the first scenario, I would click the compose button several more times, to verify a window opens at every click.

Satisfied the compose button works, I'd turn my attention to the editor window itself. It is made up a few fields (To:, Subject:, Composition Panel) and has several buttons. The pertinent parts are highlighted here.



--
Eric Rouse, P.E.
930-599-979
rousee@onid.oregonstate.edu
(541) 979-7755

The next scenario is much more difficult than the first, it involves checking every field and button for proper function. This means that first the function of each field and button must be understood, then applied. For instance, text highlight: to test this feature one must first input some text into the composition panel and then try to highlight it with the text highlight function.

Also, each button has a tool tip, so besides "clicking" every button, every button must be "hovered over" and the tool tip verified for the action. This would be a painstaking process that would take hours.

Question 2: Owesome Merge

The first thing to remember is that the customer must be treated with respect and concern, after all, they are paying our salary! So, we treat them and their problem seriously and courteously. I'd first ask what exactly is failing with the merge, what is it specifically that it is doing wrong? What is exactly is incorrect with the output?

Then I'd ask if I could have the offending array A and array B to see if they are anomalous in some way, and also to test the customer's exact input to see if I can recreate the problem. Then I'd run the merge method on their data and verify the result.

If I got the same result — great! Verify the output is not as it is designed to be (i.e. it has duplicates or it isn't in order). If so — on to finding the bug!

If I didn't get the same result I'd have the customer walk me through how they're applying the method to see if there is an error in the application or use of our method. I'd ask questions about the development environment, the target environment, type of machine etc, trying to find differences that could lead to errors.

If there are no errors in the output this would have to be gently and carefully explained to the customer, maybe with an online demo. Some way to show the output is correct, that the merge is working without casting undue blame on the customer. It's our job to make them understand to give them to tools to use our software.

Question 3: Chess

There are four possible outcomes when using the method RookMove, two of which are valid and two of which are faulty:

1. Chessboard is changed and RookMove is true. This is a valid outcome.
2. Chessboard is not changed and RookMove is false. This is a valid outcome.
3. Chessboard is changed and RookMove is false. This is a faulty outcome.
4. Chessboard is not changed and RookMove is true. This is a faulty outcome.

So, for any unit test, we must check for the proper status of of chessboard (whether changed or not) the proper status of the discard pile (whether changed or not) and the proper status of the RookMove method (whether true or false).

Unit Test 1: Clear to Move

For the first unit test we want to make sure we are able to move when we should. So we set up a chessboard with a rook at 2,7 and no other pieces on the board. Thus we should be able to move to 2, 3 un hindered, so RookMove should return true and chessboard should change. Also, since we captured no pieces, we expect the discard pile to not change.

```
/*UNIT TEST 1*/
/*legal move (no hopping), without capturing*/
#instantiate objects
Chessobject chessboard[][]
Chessobject oracleboard[][]
Discardobject discardpile[32]
Discardobject oraclepile[32]
#clear the board
for x in range(8):
      for y in range(8):
            chessboard[x][y] = null
#set whiterook1 on appropriate spot
chessboard[2][7] = whiterook1
#copy the game state into the oracle
memcopy(oracleboard, chessboard)
memcopy(oraclepile, discardpile)
#call the move, must return true
assertTrue("valid move", RookMove(chessboard, discardpile, whiterook1, 2, 7,
2, 3))
#verify chessboard and discard states:
assertTrue("board must change", chessboard != oracleboard)
assertTrue("discardpile must not change", discardpile == oraclepile)
#verify the move completed accurately
```

```
assertTrue("rook must move to 2, 3", chessboard[2][3] == whiterook1)
#verify the discard pile
assertTrue("discard pile should be empty", discard[0] == null)
```

Unit Test 2: Clear to Capture

For the second unit test we want to make sure we can move *and* capture a piece, so we set up the board in a similar way as before, but we also add a piece to the target move position. Thus we expect the discard pile to change, the chessboard to change and the RookMove method to return true.

```
/*UNIT TEST 2*/
/*legal move (no hopping), with capturing*/
#instantiate objects
Chessobject chessboard[][]
Chessobject oracleboard[][]
Discardobject discardpile[32]
Discardobject oraclepile[32]
#clear the board
for x in range(8):
      for y in range(8):
            chessboard[x][y] = null
#set whiterook1 on appropriate spot, give it a target to capture
chessboard[2][7] = whiterook1
chessboard[2][3] = blackpawn1
#copy the game state into the oracle
memcopy(oracleboard, chessboard)
memcopy(oraclepile, discardpile)
#call the move, must return true
assertTrue("valid move", RookMove(chessboard, discardpile, whiterook1, 2, 7,
2, 3))
#verify chessboard and discard states:
assertTrue("board must change", chessboard != oracleboard)
assertTrue("discardpile must change", discardpile != oraclepile)
#verify the move completed accurately
assertTrue("rook must move to 2, 3", chessboard[2][3] == whiterook1)
#verify the discard pile
assertTrue("discard pile should have a pawn", discard[0] == blackpawn1)
```

Unit Test 3: Illegal Move(s)

For the third unit test we want to verify that a piece on the board will stop us from moving. Thus we insert an opposing teams piece between us and the target position and a piece from our own team in the same spot. We expect the discard pile and chessboard to not change and the RookMove to return false in both cases.

```
/*UNIT TEST 3*/
/*illegal move (piece hopping), with capturing*/
#instantiate objects
Chessobject chessboard[][]
Chessobject oracleboard[][]
Discardobject discardpile[32]
Discardobject oraclepile[32]
#clear the board
for x in range(8):
      for y in range(8):
            chessboard[x][y] = null
#set whiterook1 on appropriate spot, give it a target to capture
chessboard[2][7] = whiterook1
chessboard[2][3] = blackpawn1
chessboard[2][4] = whitepawn1
#copy the game state into the oracle
memcopy(oracleboard, chessboard)
memcopy(oraclepile, discardpile)
#call the move, must return false
assertFalse("invalid move", RookMove(chessboard, discardpile, whiterook1, 2,
7, 2, 3))
#verify chessboard and discard states:
assertTrue("board must not change", chessboard == oracleboard)
assertTrue("discardpile must not change", discardpile == oraclepile)
#verify the move did not execute
assertTrue("rook must be left on 2, 7", chessboard[2][7] == whiterook1)
assertTrue("blackpawn1 should be on 2, 3", chessboard[2][3] == blackpawn1)
assertTrue("whitepawn1 should be on 2, 4", chessboard[2][4] == whitepawn1)
#verify the discard pile
assertTrue("discard pile should be empty", discard[0] == null)
#change the blocking piece
chessboard[2][4] = blackpawn2
#call the move, must return false
assertFalse("invalid move", RookMove(chessboard, discardpile, whiterook1, 2,
7, 2, 3))
#verify chessboard and discard states:
assertTrue("board must not change", chessboard == oracleboard)
assertTrue("discardpile must not change", discardpile == oraclepile)
#verify the move did not execute
assertTrue("rook must be left on 2, 7", chessboard[2][7] == whiterook1)
assertTrue("blackpawn1 should be on 2, 3", chessboard[2][3] == blackpawn1)
assertTrue("blackpawn2 should be on 2, 4", chessboard[2][4] == blackpawn2)
#verify the discard pile
assertTrue("discard pile should be empty", discard[0] == null)
```