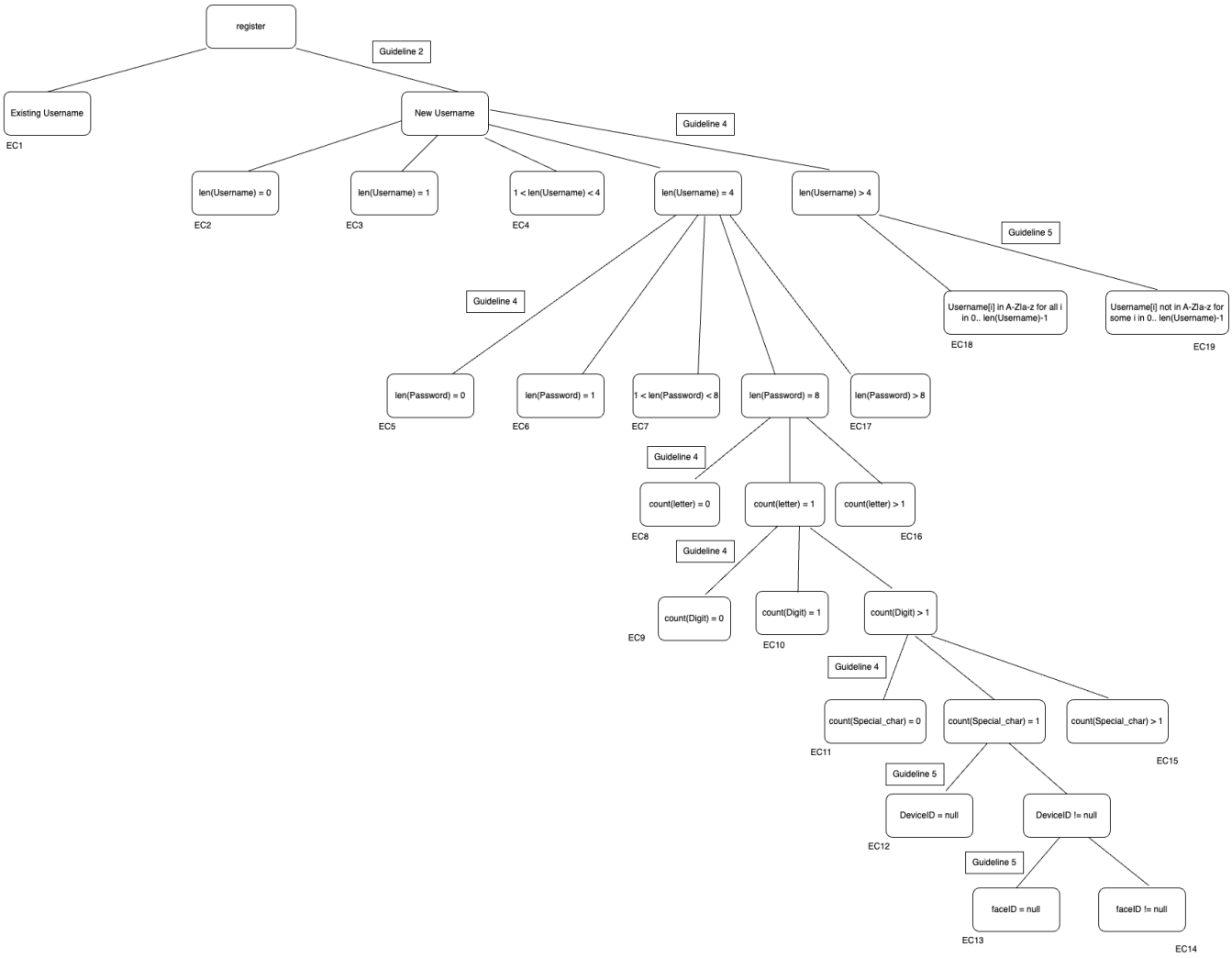
Assigment 1 report

Chengyi Huang 1173994

Task1 Equivalence partitioning

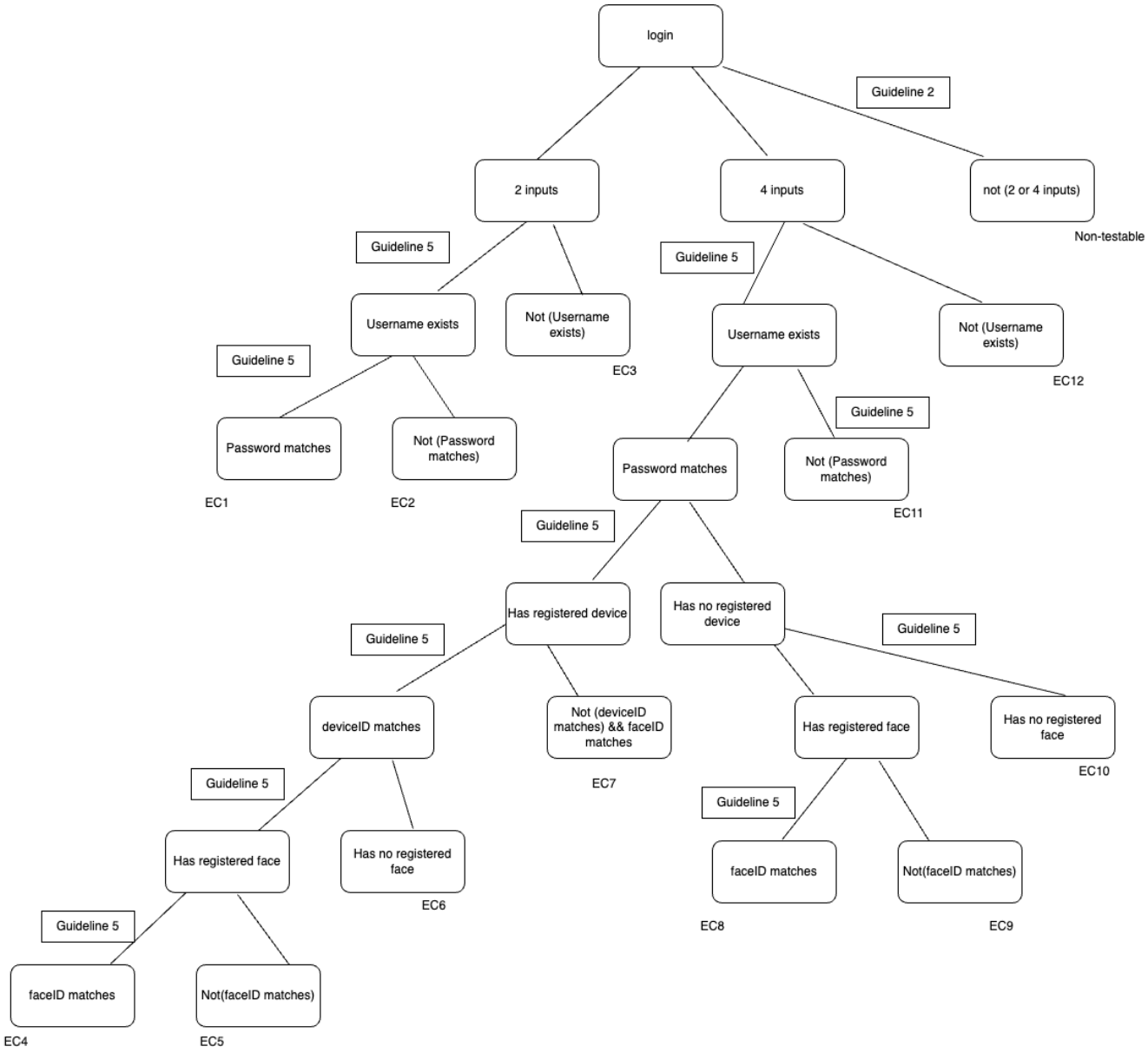
1.1 register



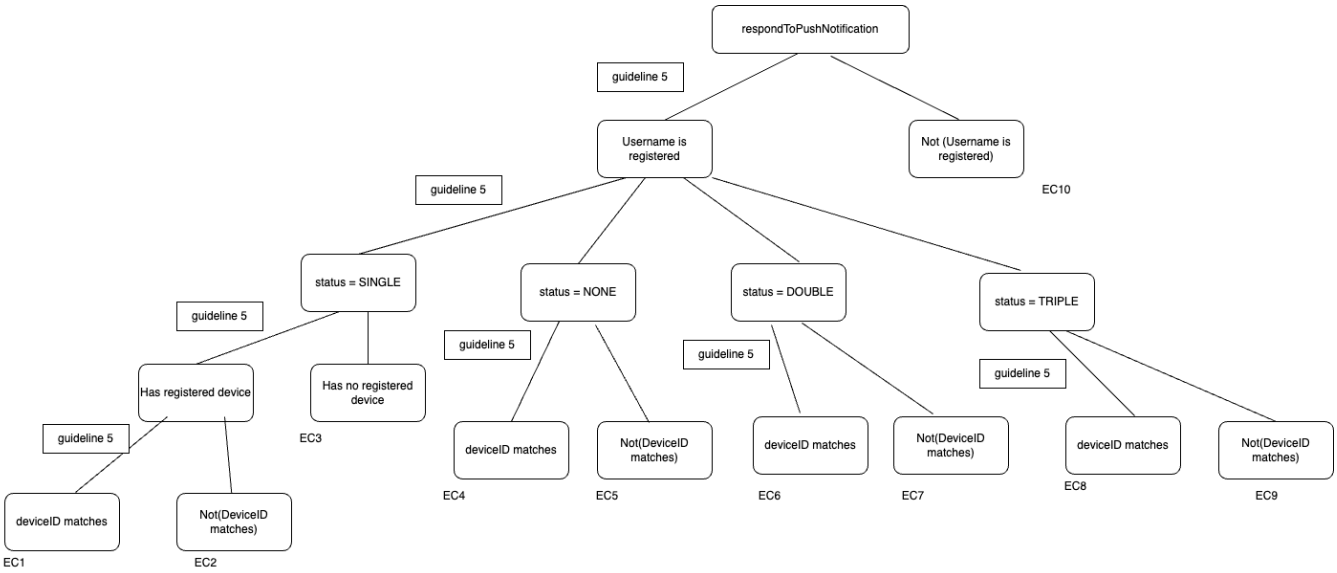
Assumption: To mitigate tree explosion, Each Choice Combination is used.

1. Since specification on Username is independent to password, equivalence classes for password is attached to an arbitrary leaf node, in this case len(Password) = 8
2. Likewise, number of letter, digits and special characters are all independent to each other, thus can be attached to arbitrary leaf node.

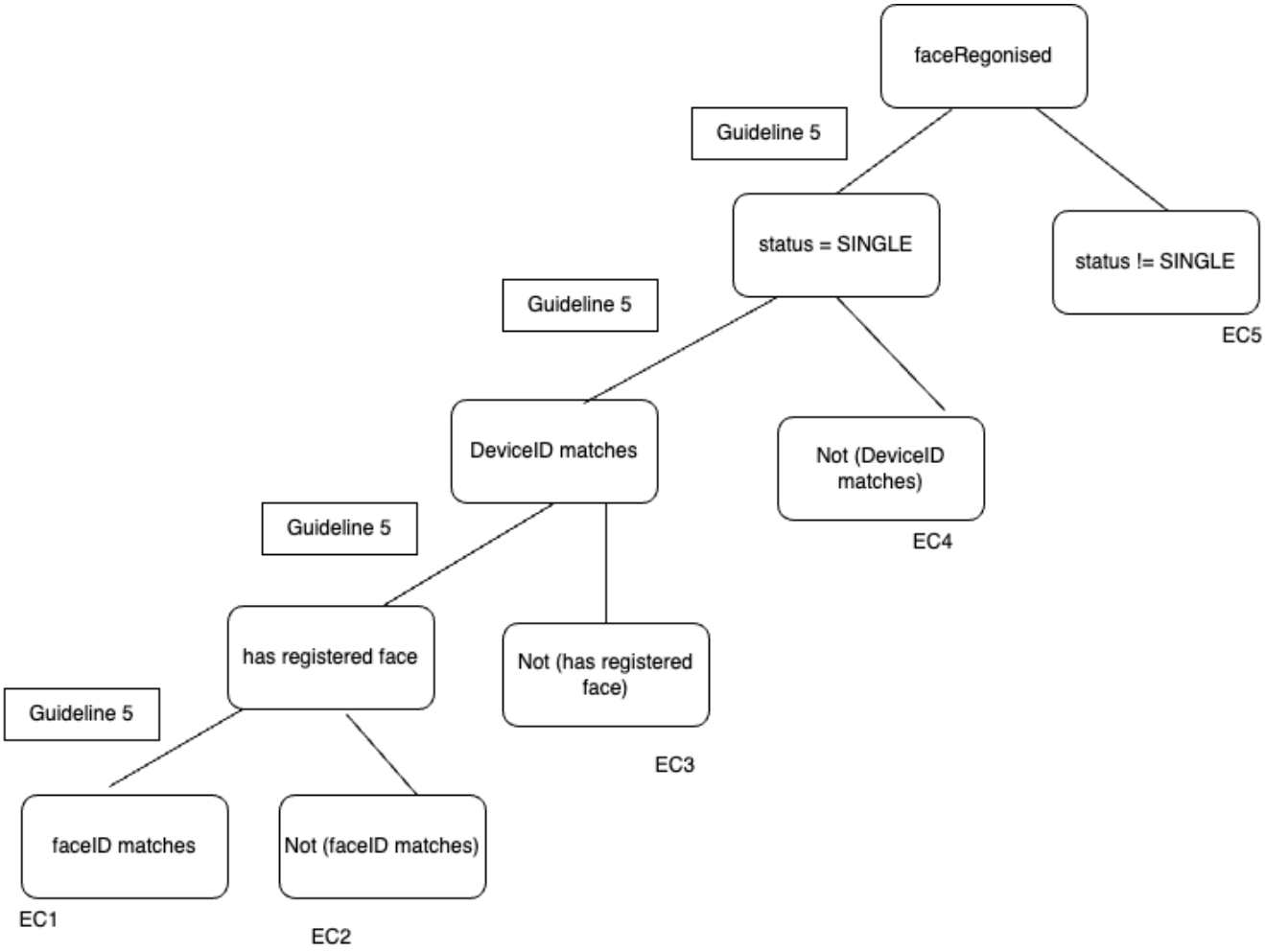
1.2 login



1.3 respondToPushNotification

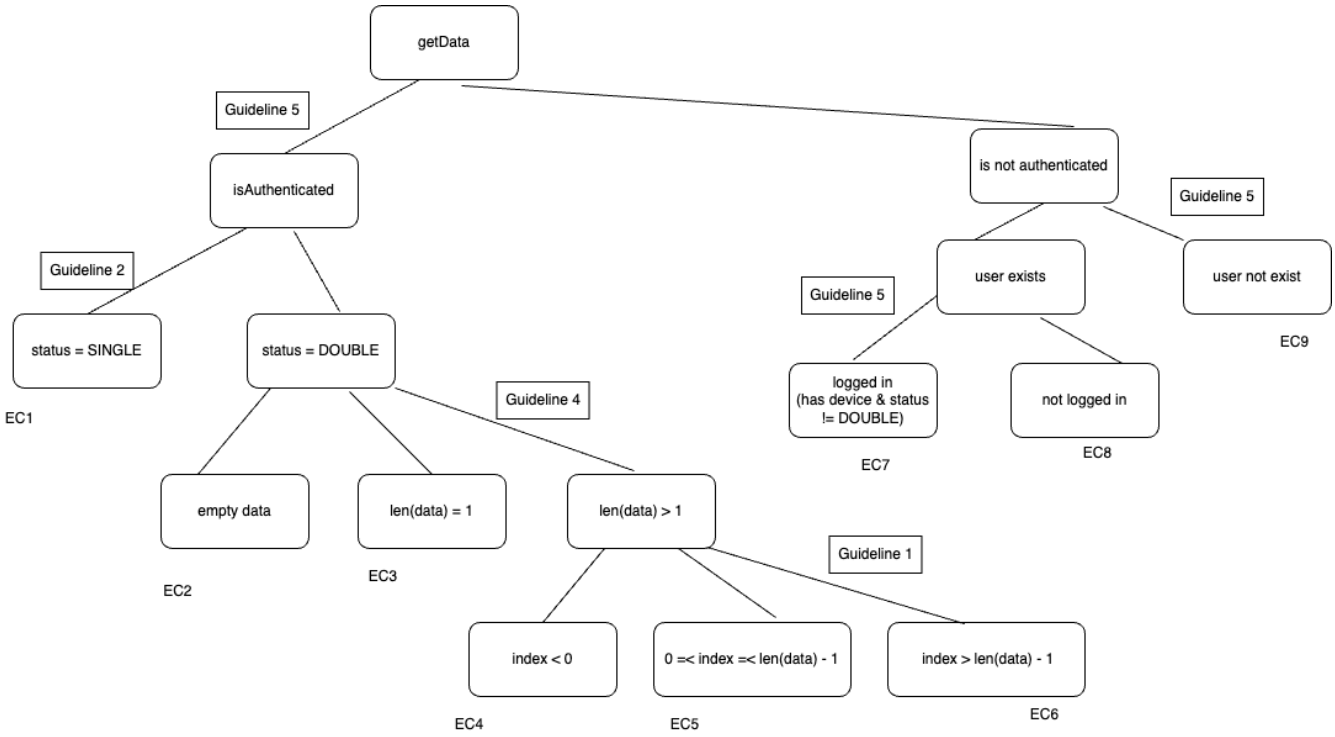


1.4 faceRegonised



Assumption: A large part of faceRegonised's functionality overlaps with that of respondToPushNotification, thus to avoid code repetition, for EC5 we are just going to use the test case (NONE, null, null) instead of breaking it into None, double etc.

1.5 getData



Task3 Boundary-value analysis

Overview: A lot of the Boundary-value analysis test cases overlaps with test cases from Equivalence partitioning, as guideline 1, guideline 4 and 5 have a similar idea of

choosing classes "on" and "off" the boundary. Previously covered test cases will be labelled with "EC" + number, and they will not be explicitly written in BoundaryTest.java

since all ECs are inherited. Boundary tests that has not been covered will be labeled with "BVA" + number and is highlighted in red, these cases will be written in BoundaryTest.java.

Register

1. Domain constraint: len(username) >= 4

    a. On point: len(username) = 4 (EC5-17)

    b.Off point: len(username) = 3 (BVA1)

1. Domain constraint: len(password) >= 8
    a. on point: len(password) = 8 (EC8-16)
    b. off point: len(password) = 7 (BVA2)
2. Domain constraint: len(letter) in password >= 1
    a. on point: len(letter) = 1 (EC9-15)
    b. off point: len(letter) = 0 (EC8)

3. Domain constraint: len(digit) in password >= 1
    a. on point: len(digit) = 1 (EC10)
    b. off point: len(digit) = 0 (EC9)
4. Domain constraint: len(special_char) in password >= 1
    a. on point: len(special_char) = 1 (EC12-14)
    b. off point: len(special_char) = 0 (EC11)
5. Equivalence class: username[i] not in A-Z|a-z for some i in 0..len(username)-1
    a. on point: count(not upper or lower case letter) = 0 (EC18)
    b. off point: count(not upper or lower case letter) = 1 (BVA3)
6. Equivalence class: username[i] in A-Z|a-z for all i in 0..len(username)-1

for ASCII character A-Z:

1. off point: '@' (EC19)
2. off point: '[' (BVA4)

for ASCII character a-z:

1. off point: '`' (BVA5)
2. off point: '{' (BVA6)

And all the boolean boundaries partitioned using guideline 5 in Equivalence Partitioning (see Task1 and 2)

Login

1. number of arguments for the "login" method is either 2 or 4:
    a. on point: count(arguments) = 4 & count(arguments) = 2
    b. off point: count(arguments)t = 1, 3, 5

However, this input constraint is non-testable as it will lead to erroneous code

All other boundaries are boolean boundaries partitioned using guideline 5 in Equivalence Partitioning (see Task1 and 2)

RespondToPushNotification

All boundaries are boundaries partitioned using guideline 5 in Equivalence Partitioning (see Task1 and 2)

FaceRecognised

All boundaries are boolean boundaries partitioned using guideline 5 in Equivalence Partitioning (see Task1 and 2)

getData

1. Len(data) >= 1 (can't be empty):
    a. on point: Len(data) = 1 (EC3)
    b. off point: Len(data) = 0 (EC2)
2. 0 <= Index <= len(data) - 1:
    a. on point: index = 0 (EC3)
    b. on point: index = len(data) - 1 (EC5)
    c. off point: index = -1 (EC4)
    d. off point: index = len(data) (EC6)

All other boundaries are boolean boundaries partitioned using guideline 5 in Equivalence Partitioning (see Task1 and 2)

Task5 Multiple-condition Coverage

overview: When certain predicates in the function are met, exception will be thrown and the code will not go on to run other conditions in the rest of the function, e.g. in register, if "passwords.containsKey(username)" = true,

the function throws exception and ends it there, therefore there is no need to enumerate combinations that come after it. In such, the conditions will be crossed using "-", or strike through, like the first row in register table.

Register

#	passwords.containsKey (username)	username.length() < MINIMUM_USERNAME_LENGTH	password.length() < MINIMUM_PASSWORD_LENGTH	username Chars are Letters	password Char has Letter (A)	password Char has Digit (B)	password Char has Special Char (C)	Valid Password (A & B & C)
1	T	-	-	-	-	-	-	-
2	F	T	-	-	-	-	-	-
3	F	F	T	-	-	-	-	-
4	F	F	F	F	-	-	-	-
5	F	F	F	T	T	T	T	T
6	F	F	F	T	T	T	F	F
7	F	F	F	T	T	F	T	F
8	F	F	F	T	F	T	T	F
9	F	F	F	T	T	F	F	F
10	F	F	F	T	F	T	F	F
11	F	F	F	T	F	F	T	F
12	F	F	F	T	F	F	F	F

Coverage score: 8/12 * 100% = 66.67%

Login Two factors

#	checkUsernamePassword(username, password)	deviceIDs.get(username) != null	Test Case covered (Equivalence Partitioning)
1	T	F	EC1
2	T	T	
3	F	T	
4	F	F	EC2

Coverage score: 2/4 * 100% = 50%

Login three factors

#	checkUsernamePassword(username, password)	deviceIDs.get(username) != null	respondToPushNotification() = double	faceRecognised() = tripele	Test Case covered (Equivalence partitioning)
1	T	T	T	T	EC4
2	T	T	T	F	EC5
3	T	T	F	T	
4	T	T	F	F	
5	T	F	T	T	EC8
6	T	F	T	F	EC10
7	T	F	F	T	EC7
8	T	F	F	F	EC11
9	~~F~~	~~T~~	~~T~~	~~T~~	
10	~~F~~	~~T~~	~~T~~	~~F~~	
11	~~F~~	~~T~~	~~F~~	~~T~~	

12	~~F~~	~~T~~	~~F~~	~~F~~	
13	~~F~~	~~F~~	~~T~~	~~T~~	
14	~~F~~	~~F~~	~~T~~	~~F~~	
15	~~F~~	~~F~~	~~F~~	~~T~~	
16	F	F	F	F	EC12

*Strike through means when "checkUsernamePassword(username, password)" = false, all other conditions will not be evaluated, thus only one such case needed to be tested

Coverage score: 7/9 * 100% = 77.78%

Respond to push

#	isUser (username)	authenticationStatus = SINGLE	deviceIDs.get(username) != null	deviceIDs.get(username) != deviceID	Test case covered (Equivalence)
1	T	T	T	T	EC2
2	T	T	T	F	EC1
3	T	T	F	T	EC3
4	T	T	F	F	
5	T	F	T	T	EC4, EC6, EC8
6	T	F	T	F	EC5, EC7, EC9
7	T	F	F	T	
8	T	F	F	F	
9	~~F~~	~~T~~	~~T~~	~~T~~	
10	~~F~~	~~T~~	~~T~~	~~F~~	
11	~~F~~	~~T~~	~~F~~	~~T~~	
12	~~F~~	~~T~~	~~F~~	~~F~~	
13	~~F~~	~~F~~	~~T~~	~~T~~	
14	~~F~~	~~F~~	~~T~~	~~F~~	
15	~~F~~	~~F~~	~~F~~	~~T~~	
16	F	F	F	F	EC10

Coverage score: 6/9 * 100% = 66.67%

faceRecognised

#	isUser (username)	authenticationStatus = SINGLE	deviceIDs. get (username) ! = null	deviceIDs.get (username) ! = deviceID	authenticationStatus = DOUBLE	faceRecognised. containsKey (username)	faceRecognised. get(username). equals(facialId)	test case covered (Equivalence partitioning)
1	F	-	-	-	-	-	-	
2	T	F	-	-	-	-	-	EC5
3	T	T	F	-	-	-	-	EC4
4	T	T	T	T	-	-	-	EC3
5	T	T	T	F	T	T	T	
6	T	T	T	F	T	T	F	EC1
7	T	T	T	F	T	F	-	EC2
8	T	T	T	F	F	-	-	

Coverage score: 5/8 * 100% = 62.5%

getData

#	isAuthenticated(username)	test case covered (Equivalence partitioning)
1	F	EC9
2	T	EC5

Coverage score: 5/8 * 100% = 100%

Task7 Comparison

In general, Equivalence partitioning covers more test cases when input domain has more variety, e.g. when the input have zero-one-many and discrete set sort of input, those would not

be covered as effectively by BVA. However, BVA can be more effective when there are a lot of numerical predicates, e.g. range of variables, maximum of input size = ..., and BVA is especially

advantageous in catching mutants, because Equivalence partitioning can only catch computational errors, whereas BVA can catch both computational errors and boundary shift, e.g. in mutant 4

where ">" were changed into ">=".