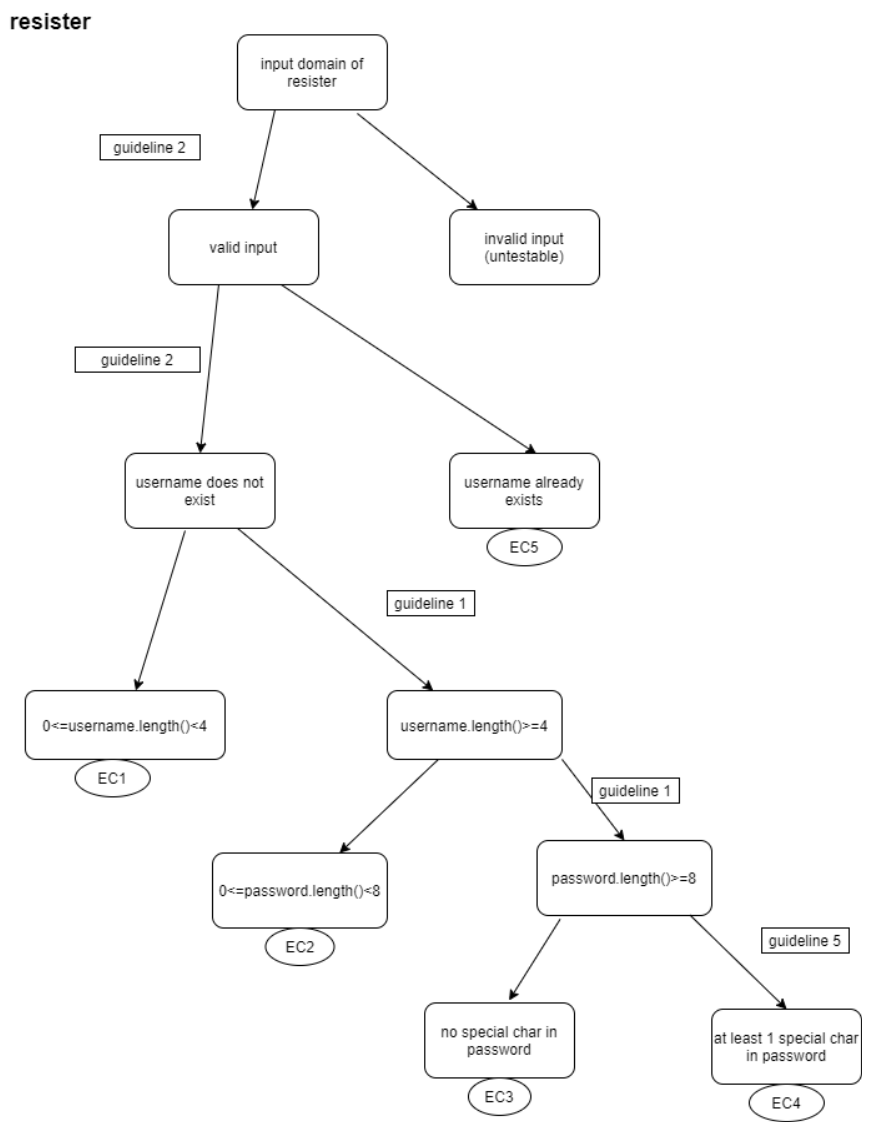
SWEN90006

Task 1



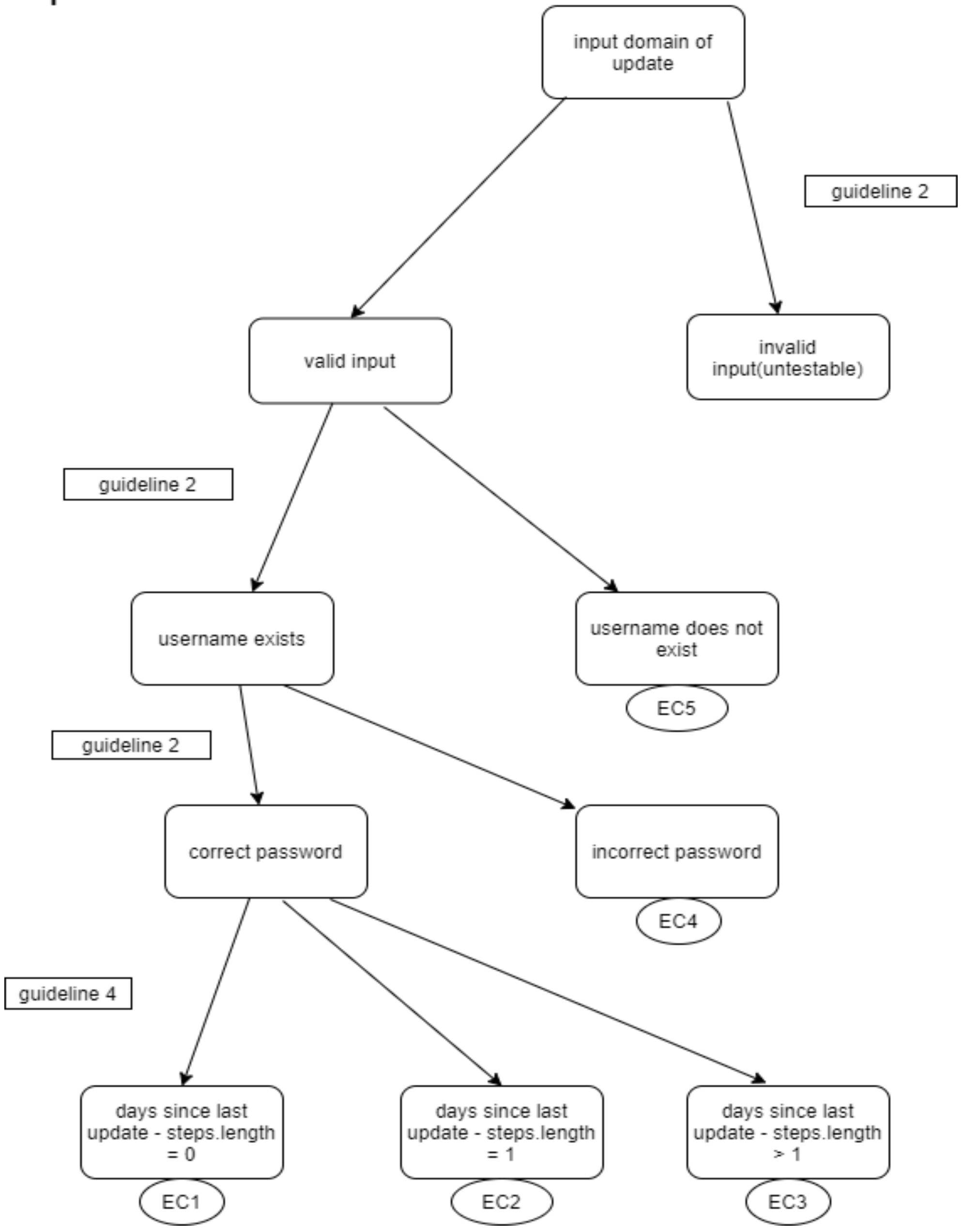
Resister:

Assumption: While we need instance of list of registered users to check if username exist, admin user should be put in the instance, thus we can assume that username is non-null.

The set of equivalent classes cover the input space as all leaves doesn't overlap each other and the leaf covers its parent nodes. I omitted the following invalid inputs (untestable): username or password is null as per spec, data type of username or password is not string.

We test that length of username and password that should be at least 4 and 8 respectively. As the length cannot be negative, the invalid inputs start from 0 to less than 4 for username and 0 to less than 8 for password.

update



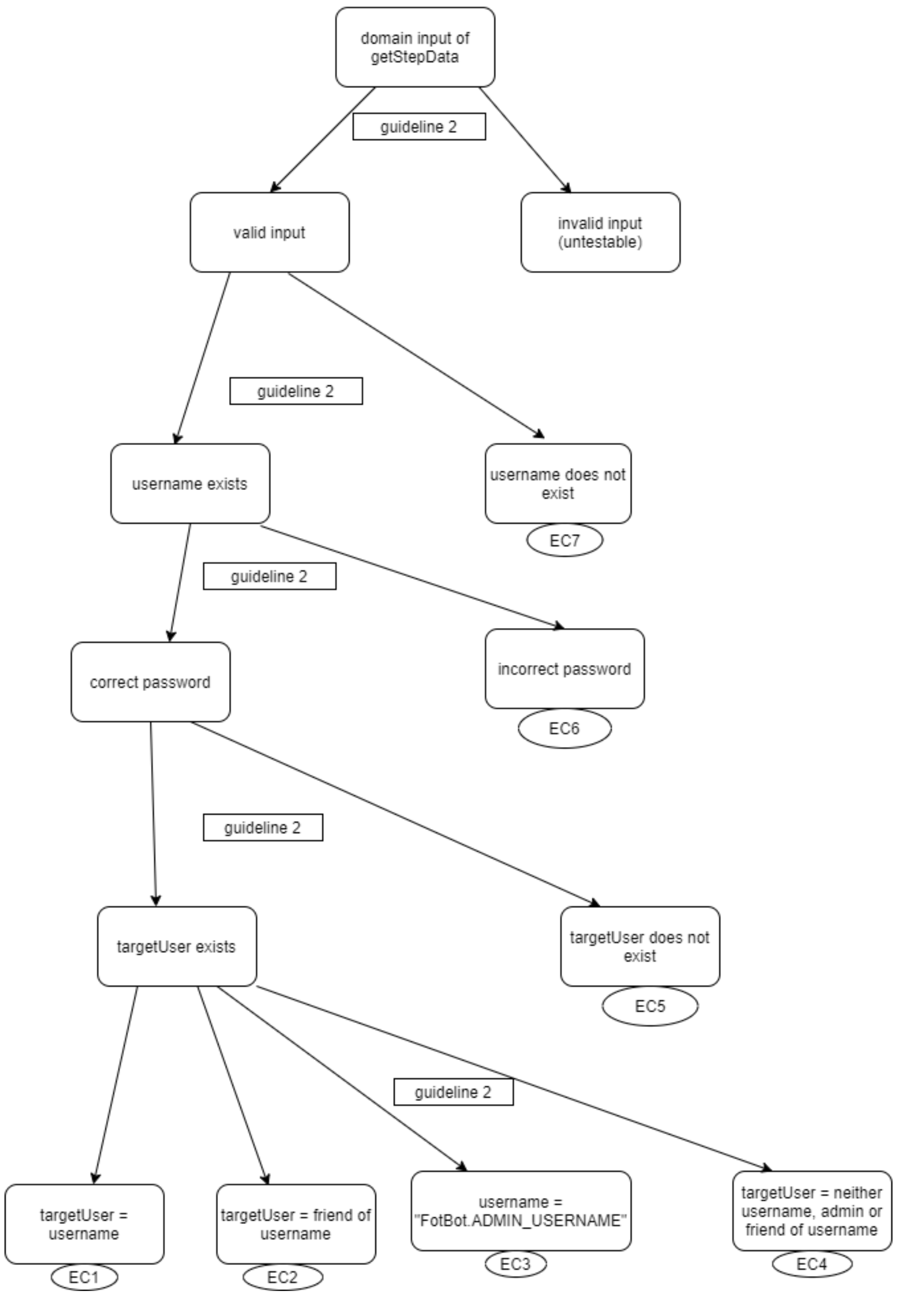
Update:

Assumption: while we need last update instance to compare the data to current one, we can assume that the default starting date should be already put.

I omitted the following invalid inputs (untestable): length(steps) > the number of days since last update, username, password or steps is null, “steps” does not record the order of the days from oldest to recent, data type of username or password is not string and steps is not list of integers.

The set of equivalent classes cover the input space as all leaves doesn’t overlap each other and the leaf covers its parent nodes. We again check username and password to make sure the user exists. As per spec, length(steps) > the number of days since last update is untestable, so we don’t test “the number of days since last update – steps.length < 0”.

getSteps


domain input of getStepData
guideline 2
valid input
invalid input (untestable)
guideline 2
username exists
username does not exist
EC7
guideline 2
correct password
incorrect password
EC6
guideline 2
targetUser exists
targetUser does not exist
EC5
guideline 2
targetUser = username
EC1
targetUser = friend of username
EC2
username = "FotBot.ADMIN_USERNAME"
EC3
targetUser = neither username, admin or friend of username
EC4

getSteps:

Assumption: while an instance of friends of users should be used, we can assume that the "friends" instance has empty input and is not null. Also, as discussed in register section, we can assume that username is non-null, so no null pointer exception will occur when checking if username exists.

I omitted the following invalid inputs (untestable): username, password or targetUser is null, data type of username, password or targetUser is not String.

The set of equivalent classes cover the input space as all leaves doesn't overlap each other and the leaf covers its parent nodes. We check username, password and targetUser to make sure the users exist.

Task 2

See Appendix 1.

Task 3

Boundary-analysis is to select test cases on the boundary conditions of a program. With the 4 guidelines in the lecture notes, we will conduct boundary-analysis. Based on the identified equivalent classes in the first task, the following tables are created for boundary-analysis.

Register:

Equivalent Class (EC)	Boundary	Boundary Type	Test case selection	Test case
1	0<=username.length()<4	Inequality, Open	Guideline 1: 1. On point: 4 2. Off point: 3 3. On point: 0 4. Off point: -1 (untestable)	~~1. valid input, user does not exist, username.length() = 4~~ (tested in other ECs) 2. valid input, username doesn’t exist, username.length() = 3 3. valid input, username doesn’t exist, username.length() = 0
2	0<=password.length()<8	Inequality, Open	Guideline 1: 1. On point: 8 2. Off point: 7 3. On point: 0 4. Off point: -1 (untestable)	1. ~~Valid input, username does not exist, username.length() = 4, password.length() = 8~~ (Tested in other ECs) 2. Valid input, username does not exist, username.length() = 4, password.length() = 7 3. Valid input, username does not exist, username.length() = 4, password.length() = 0

3	no special char in password	Unordered, Boolean	Guideline 3: 1. On point: no special char 2. Off point: special char (tested in EC4)	1. Valid input, username does not exist, username.length() = 4, password.length() = 8, no special char
4	at least 1 special char in password	Unordered, Boolean	Guideline 3: 1. On point: special char 2. Off point: no special char (tested in EC3)	1. Valid input, username does not exist, username.length() = 4, password.length() = 8, special char
5	username already exists	Unordered, Boolean	Guideline 3: 1. On point: username already exists 2. Off point: username doesn't exist	1. valid input, username exists 2. ~~valid input, username does not exist~~ (Tested in other ECs)

Update:

Equivalent Class (EC)	Boundary	Boundary Type	Test case selection	Test case
1	days since last update - steps.length = 0	Strict Equality	Guideline 2: 1. On point: 0 2. Off point: -1 (untestable) 3. Off point: 1 (tested in EC2)	1. Valid input, username exists, correct password, days since last update - steps.length = 0
2	days since last update - steps.length = 1	Strict Equality	Guideline 2: 1. On point: 1 2. Off point: 0 (tested in EC1) 3. Off point: 2 (tested in EC3)	1. Valid input, username exists, correct password, days since last update - steps.length = 1
3	days since last update -	Inequality, Open	Guideline 1:	2. Valid input, username

	steps.length > 1		1. On point: 1 (tested in EC2) 2. Off point: 2	exists, correct password, days since last update - steps.length = 2
4	incorrect password	Unordered, Boolean	Guideline 3: 1. On point: incorrect password 2. Off point: correct password	1. Valid input, username exists, incorrect password 2. ~~Valid input, username exists, correct password~~ (tested in other ECs)
5	username does not exist	Unordered, Boolean	Guideline 3: 1. On point: username does not exist 2. Off point: username exists	1. Valid input, username does not exist 2. ~~Valid input, username exists~~ (tested in other ECs)

getSteps:

Equivalent Class (EC)	Boundary	Boundary Type	Test case selection	Test case
1	targetUser = username	Unordered, String	Guideline 3: 1. On point: username 2. Off point: (tested in EC4)	1. valid input, username exists, correct password, targetUser exists, targetUser=username
2	targetUser = friend of username	Unordered, String	Guideline 3: 1. On point: friend of username 2. Off point: (tested in EC4)	1. valid input, username exists, correct password, targetUser exists, targetUser=friend of username
3	username = FotBot.ADMIN_USERNAME	Unordered, String	Guideline 3:	1. valid input, username exists, correct password,

			1. On point: FotBot.ADMIN_USERNAME 2. Off point: (tested in EC4)	targetUser exists, targetUser= FotBot.ADMIN_USERNAME
4	targetUser = neither username, admin nor friend of username	Unordered, String	Guideline 3: 1. On point: neither username, admin nor friend of username 2. Off point: (tested in one of EC1-3)	1. valid input, username exists, correct password, targetUser exists, targetUser= neither username, admin nor friend of username
5	targetUser does not exist	Unordered, Boolean	Guideline 3: 1. On point: TargetUser does not exist 2. Off point: targetUser exists	1. valid input, username exists, correct password, targetUser does not exist 2. ~~valid input, username exists, correct password, targetUser exists~~ (tested in other ECs)
6	incorrect password	Unordered, Boolean	Guideline 3: 3. On point: Incorrect password 4. Off point: correct password	1. valid input, username exists, incorrect password 2. ~~valid input, username exists, correct password~~ (tested in other ECs)
7	username does not exist	Unordered, Boolean	Guideline 3: 1. On point: username doesn't exist 2. Off point: username exists	1. valid input, username does not exist 2. ~~valid input, username exists~~ (tested in other ECs)

Task 4

See Appendix B.

Task 5

Register conditions:

Condition	Branch Code	Possible outcome
C1	if (passwords.containsKey(username))	{T, F}
C2	if (username.length() < MINIMUM_USERNAME_LENGTH)	{T, F}
C3	if (password.length() < MINIMUM_PASSWORD_LENGTH)	{T, F}
C4	for(char c : password.toCharArray())	{T, F}
C5 && C6 && C7	if (!('a' <= c && c <= 'z') && !('A' <= c && c <= 'Z') && !('0' <= c && c <= '9'))	{TTT, TFT, TTF, TFF, FTT, FFT, FTF, FFF}
C8	if (!special)	{T, F}

*For loop has a condition.

Update conditions:

Condition	Branch Code	Possible outcome
C1	if (!passwords.containsKey(username))	{T, F}
C2	if (!passwords.get(username).equals(password))	{T, F}
C3	while (!day.equals(userLastUpdate))	{T, F}
C4	if (i >= 0)	{T, F}

getSteps conditions:

Condition	Branch Code	Possible outcome
C1	if (!passwords.containsKey(username))	{T, F}
C2	if (!passwords.get(username).equals(password))	{T, F}
C3	if (!passwords.containsKey(targetUser))	{T, F}
C4 || C5 || C6	if (isFriend(targetUser, username) || username.equals(ADMIN_USERNAME) || username.equals(targetUser))	{TTT, TFT, TTF, TFF, FTT, FFT, FTF, FFF}

Equivalence partitioning:

Register objectives:

Equivalence partitioning Resister EC/C	C1	C2	C3	C4	C5 && C6 && C7	C8
EC1 test	F	T				
EC2 test	F	F	T			
EC3 test	F	F	F	T	TFF	T
EC4 test	F	F	F	T	FFT	F
EC5 test	T					
Observed	T, F	T, F	T, F	T	TFF, FFT	T, F

Missed				F	TTT, TFT, TTF, FFF, FTT, FTF	

Register coverage score = (objectives met) / (total objectives) = 11/18 = 61%

Update objectives:

Equivalence partitioning Update EC/C	C1	C2	C3	C4
EC1 test	F	F	F	
EC2 test	F	F	T	T, F
EC3 test	F	F	T	T, F
EC4 test	F	T		
EC5 test	T			
Observed	T, F	T, F	T, F	T, F
Missed				

Update coverage score = 8/8 = 100%

getSteps objectives:

Equivalence partitioning getSteps EC/C	C1	C2	C3	C4 || C5 || C6
EC1 test	F	F	F	FFT
EC2 test	F	F	F	TFF
EC3 test	F	F	F	FTF
EC4 test	F	F	F	FFF
EC5 test	F	F	T	
EC6 test	F	T		
EC7 test	T			
Observed	T, F	T, F	T, F	FFT, TFF, FTF, FFF
Missed				TTT, TFT, TTF, FTT

getSteps coverage score = 10/14 = 71%

Boundary-value analysis:

Register objectives:

Boundary-value Analysis Resister EC/C	C1	C2	C3	C4	C5 && C6 && C7	C8

EC1 test EC1 test_2	F	T				
EC2 test EC2 test_2	F	F	T			
EC3 test	F	F	F	T	TFF	T
EC4 test	F	F	F	T	FFT	F
EC5 test	T					
Observed	T, F	T, F	T, F	T	FFF, FFT	T, F
Missed				F	TTT, TFT, TTF, FFF, FTT, FTF	

Register coverage score = (objectives met) / (total objectives) = 11/18 = 61%

Update objectives:

Boundary-value Analysis Update EC/C	C1	C2	C3	C4
EC1 test	F	F	F	
EC2 test	F	F	T	T, F
EC3 test	F	F	T	T, F
EC4 test	F	T		
EC5 test	T			
Observed	T, F	T, F	T, F	T, F
Missed				

Update coverage score = 8/8 = 100%

getSteps objectives:

Boundary-value Analysis getSteps EC/C	C1	C2	C3	C4 || C5 || C6
EC1 test	F	F	F	FFT
EC2 test	F	F	F	TFF
EC3 test	F	F	F	FTF
EC4 test	F	F	F	FFF
EC5 test	F	F	T	
EC6 test	F	T		
EC7 test	T			
Observed	T, F	T, F	T, F	FFT, TFF, FTF, FFF
Missed				TTT, TFT, TTF, FTT

getSteps coverage score = 10/14 = 71%

Task 6

See the codes.

Task 7

Input/output domain coverage: both equivalence partitioning and boundary value analysis are derived from the same ECs. As explained using the test template trees in Task 1, the input spaces are covered since the ECs don't overlap each other and each breakdown covers its parent node.

Multiple condition coverage: the results of multiple condition coverage show the two sets of test cases have the same coverage score on three methods: register 61%, update 100%, getSteps 71%. This is because both sets of test cases are derived from the same ECs and similar test cases were chosen.

Mutants: in the five created mutants for Task 6, the boundary value analysis could kill all five cases, while the equivalence partitioning could kill only four cases. This indicates that boundary value analysis is more sensitive to the small changes made by mutants, which programmers could commonly make, than equivalence partitioning. Boundary value analysis checks the boundaries of ECs while equivalence partitioning treat values in an EC equally. For example, if programmer made a mistake ">=" to ">", boundary analysis is more likely to detect this fault than equivalence partitioning.

Overall, boundary value analysis slightly outperforms equivalence partitioning based on these results.

Appendix:

Appendix A

```
    //add a default user and friend
    @Before public void setUp()
    throws DuplicateUserException, InvalidUsernameException, InvalidPasswordEx
ception, NoSuchUserException, IncorrectPasswordException
    {
        fotbot = new FotBot();
        fotbot.register("userName1", "password1!");
        //resister a friend of userName1
        fotbot.register("friendUsername1", "fpassword1!");
        //add a friend of userName1
        fotbot.addFriend("userName1", "password1!", "friendUsername1");
        //add friendUsername1 as friend of userName1
        fotbot.addFriend("friendUsername1", "fpassword1!", "userName1");
        //resister another non friend user
        fotbot.register("userName2", "password2!");
    }
```

```java
    // register EC1 username.length()<4
    @Test(expected = InvalidUsernameException.class)
    public void resister_EQ1()
    throws DuplicateUserException, InvalidUsernameException, InvalidPasswordEx
ception
    {
        fotbot.register("R1", "AZaz129@");
    }

    // register EC2 password.length()<8
    @Test(expected = InvalidPasswordException.class)
    public void resister_EQ2()
    throws DuplicateUserException, InvalidUsernameException, InvalidPasswordEx
ception
    {
        fotbot.register("Register2", "Abc1@");
    }

    // register EC3 no special char in password
    @Test(expected = InvalidPasswordException.class)
    public void resister_EQ3()
    throws DuplicateUserException, InvalidUsernameException, InvalidPasswordEx
ception
    {
        fotbot.register("Register3", "ABYZ1289");
    }

    // register EC4 at least 1 special char in password so no exception
    @Test public void resister_EQ4()
    throws DuplicateUserException, InvalidUsernameException, InvalidPasswordEx
ception
    {
        fotbot.register("Register4", "AZabcxyz@");
        assertTrue(fotbot.isUser("Register4"));
    }

    // register EC5 username already exists
    @Test(expected = DuplicateUserException.class)
    public void resister_EQ5()
    throws DuplicateUserException, InvalidUsernameException, InvalidPasswordEx
ception
    {
        fotbot.register("Register5", "ABYZ1289@");
        fotbot.register("Register5", "ABYZ1289@");
    }
    //update EC1 days since last update - steps.length = 0
    @Test public void update_EQ1()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(2);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username1 is created in the setUp() method, which is run before ever
y test
        fotbot.update("userName1", "password1!", newSteps);

        List<Integer> steps = fotbot.getStepData("userName1", "password1!", "u
serName1");
        List<Integer> expected = list(new Integer [] {1000, 2000});
        assertEquals(expected, steps);
    }

    //update EC2 days since last update - steps.length = 1
    @Test public void update_EQ2()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username1 is created in the setUp() method, which is run before ever
y test
        fotbot.update("userName1", "password1!", newSteps);

        List<Integer> steps = fotbot.getStepData("userName1", "password1!", "u
serName1");
        List<Integer> expected = list(new Integer [] {0, 1000, 2000});
        assertEquals(expected, steps);
    }

    //update EC3 days since last update - steps.length > 1
    @Test public void update_EQ3()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(7);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username1 is created in the setUp() method, which is run before ever
y test
        fotbot.update("userName1", "password1!", newSteps);
        List<Integer> steps = fotbot.getStepData("userName1", "password1!", "u
serName1");
        List<Integer> expected = list(new Integer [] {0, 0, 0, 0, 0, 1000, 200
0});
        assertEquals(expected, steps);
    }

    // update EC4 incorrect password
    @Test(expected = IncorrectPasswordException.class)
    public void update_EQ4()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username1 is created in the setUp() method, which is run before ever
y test
        fotbot.update("userName1", "Incorrect12@", newSteps);

        //List<Integer> steps = fotbot.getStepData("userName1", "password1!",
"userName1");
        //List<Integer> expected = list(new Integer [] {0, 1000, 2000});
        //assertEquals(expected, steps);
    }

    // update EC5 username does not exist
    @Test(expected = NoSuchUserException.class)
    public void update_EQ5()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username1 is created in the setUp() method, which is run before ever
y test
        fotbot.update("nonExist", "password1!", newSteps);

        //List<Integer> steps = fotbot.getStepData("userName1", "password1!",
"userName1");
        //List<Integer> expected = list(new Integer [] {0, 1000, 2000});
        //assertEquals(expected, steps);
    }



    // getSteps EC1 targetUser = username
    @Test public void getSteps_EQ1()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username1 is created in the setUp() method, which is run before ever
y test
        fotbot.update("userName1", "password1!", newSteps);

        List<Integer> steps = fotbot.getStepData("userName1", "password1!", "u
serName1");
        List<Integer> expected = list(new Integer [] {0, 1000, 2000});
        assertEquals(expected, steps);
    }

    // getSteps EC2 targetUser = friend of username
    @Test public void getSteps_EQ2()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //friendUsername1 is created in the setUp() method, which is run befor
e every test
        fotbot.update("friendUsername1", "fpassword1!", newSteps);

        List<Integer> steps = fotbot.getStepData("userName1", "password1!", "f
riendUsername1");
        List<Integer> expected = list(new Integer [] {0, 1000, 2000});
        assertEquals(expected, steps);
    }

    // getSteps EC3 username = "FotBot.ADMIN_USERNAME"
    @Test public void getSteps_EQ3()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);

        //as admin, it should return empty list
        List<Integer> steps = fotbot.getStepData("userName1", "password1!", Fo
tBot.ADMIN_USERNAME);
        List<Integer> expected = list(new Integer [] {});
        assertEquals(expected, steps);
    }
    // getSteps EC4 targetUser = neither username, admin or friend of username
    @Test public void getSteps_EQ4()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username2 is created in the setUp() method, which is run before ever
y test
        fotbot.update("userName2", "password2!", newSteps);

        //as userName2 is not related to userName1 it should return empty list
        List<Integer> steps = fotbot.getStepData("userName1", "password1!", "u
serName2");
        List<Integer> expected = list(new Integer [] {});
        assertEquals(expected, steps);

        //List<Integer> expected = list(new Integer [] {0, 1000, 2000});
        //assertEquals("This failed because user cannot view non-
related user info.", expected, steps);
    }

    // getSteps EC5 targetUser does not exist
    @Test(expected = NoSuchUserException.class)
    public void getSteps_EQ5()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //nonExist is non existed user
        fotbot.update("nonExist", "password1!", newSteps);

        //List<Integer> steps = fotbot.getStepData("userName1", "password1!",
"nonExist");
        //List<Integer> expected = list(new Integer [] {1000, 2000});
        //assertEquals(expected, steps);
    }

    // getSteps EC6 incorrect password
    @Test(expected = IncorrectPasswordException.class)
    public void getSteps_EQ6()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);
        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username2 is created in the setUp() method, which is run before ever
y test
        fotbot.update("userName1", "incorrect1!", newSteps);

        //as userName2 is not related to userName1 it should return empty list
        //List<Integer> steps = fotbot.getStepData("userName1", "password1!",
"userName1");
        //List<Integer> expected = list(new Integer [] {});
        //assertEquals(expected, steps);
    }

    // getSteps EC7 username does not exist
    @Test(expected = NoSuchUserException.class)
    public void getSteps_EQ7()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username2 is created in the setUp() method, which is run before ever
y test
        fotbot.update("userName123", "password1!", newSteps);

        //as userName2 is not related to userName1 it should return empty list
        //List<Integer> steps = fotbot.getStepData("userName1", "password1!",
"userName1");
        //List<Integer> expected = list(new Integer [] {});
        //assertEquals(expected, steps);
    }
```

Appendix B

```
// register EC1 valid input, username doesn’t exist, username.length() = 3
```

```
    @Test(expected = InvalidUsernameException.class)
    public void resister_EQ1()
    throws DuplicateUserException, InvalidUsernameException, InvalidPasswordEx
ception
    {
        fotbot.register("Re1", "AZaz129@");
    }

    // register EC1 valid input, username doesn’t exist, username.length() = 0
    @Test(expected = InvalidUsernameException.class)
    public void resister_EQ1_2()
    throws DuplicateUserException, InvalidUsernameException, InvalidPasswordEx
ception
    {
        fotbot.register("", "AZaz129@");
    }

    // register EC2 Valid input, username does not exist, username.length() =
4, password.length() = 7
    @Test(expected = InvalidPasswordException.class)
    public void resister_EQ2()
    throws DuplicateUserException, InvalidUsernameException, InvalidPasswordEx
ception
    {
        fotbot.register("Reg2", "Abcd12@");
    }

    // register EC2 Valid input, username does not exist, username.length() =
4, password.length() = 0
    @Test(expected = InvalidPasswordException.class)
    public void resister_EQ2_2()
    throws DuplicateUserException, InvalidUsernameException, InvalidPasswordEx
ception
    {
        fotbot.register("Reg2", "");
    }

    // register EC3 Valid input, username does not exist, username.length() =
4, password.length() = 8, no special char
    @Test(expected = InvalidPasswordException.class)
    public void resister_EQ3()
    throws DuplicateUserException, InvalidUsernameException, InvalidPasswordEx
ception
    {
        fotbot.register("Reg3", "ABYZ1289");
    }
```

```java
    // register EC4 Valid input, username does not exist, username.length() =
4, password.length() = 8, special char
    @Test public void resister_EQ4()
    throws DuplicateUserException, InvalidUsernameException, InvalidPasswordEx
ception
    {
        fotbot.register("Register4", "AZabcxyz@");
        assertTrue(fotbot.isUser("Register4"));
    }

    // register EC5 valid input, username exists
    @Test(expected = DuplicateUserException.class)
    public void resister_EQ5()
    throws DuplicateUserException, InvalidUsernameException, InvalidPasswordEx
ception
    {
        fotbot.register("Reg5", "ABYZ1289@");
        fotbot.register("Reg5", "ABYZ1289@");
    }



    //update EC1 Valid input, username exists, correct password, days since la
st update - steps.length = 0
    @Test public void update_EQ1()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(2);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username1 is created in the setUp() method, which is run before ever
y test
        fotbot.update("userName1", "password1!", newSteps);

        List<Integer> steps = fotbot.getStepData("userName1", "password1!", "u
serName1");
        List<Integer> expected = list(new Integer [] {1000, 2000});
        assertEquals(expected, steps);
    }

    //update EC2 Valid input, username exists, correct password, days since la
st update - steps.length = 1
    @Test public void update_EQ2()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);
        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username1 is created in the setUp() method, which is run before ever
y test
        fotbot.update("userName1", "password1!", newSteps);

        List<Integer> steps = fotbot.getStepData("userName1", "password1!", "u
serName1");
        List<Integer> expected = list(new Integer [] {0, 1000, 2000});
        assertEquals(expected, steps);
    }

    //update EC3 Valid input, username exists, correct password, days since la
st update - steps.length = 2
    @Test public void update_EQ3()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(4);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username1 is created in the setUp() method, which is run before ever
y test
        fotbot.update("userName1", "password1!", newSteps);

        List<Integer> steps = fotbot.getStepData("userName1", "password1!", "u
serName1");
        List<Integer> expected = list(new Integer [] {0, 0, 1000, 2000});
        assertEquals(expected, steps);
    }

    // update EC4 Valid input, username exists, incorrect password
    @Test(expected = IncorrectPasswordException.class)
    public void update_EQ4()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username1 is created in the setUp() method, which is run before ever
y test
        fotbot.update("userName1", "Incor12@", newSteps);

        //List<Integer> steps = fotbot.getStepData("userName1", "password1!",
"userName1");
        //List<Integer> expected = list(new Integer [] {0, 1000, 2000});
        //assertEquals(expected, steps);
    }

    // update EC5 Valid input, username does not exist
    @Test(expected = NoSuchUserException.class)
    public void update_EQ5()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username1 is created in the setUp() method, which is run before ever
y test
        fotbot.update("nonExist", "password1!", newSteps);

        //List<Integer> steps = fotbot.getStepData("userName1", "password1!",
"userName1");
        //List<Integer> expected = list(new Integer [] {0, 1000, 2000});
        //assertEquals(expected, steps);
    }



    // getSteps EC1 valid input, username exists, correct password, targetUser
 exists, targetUser=username
    @Test public void getSteps_EQ1()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username1 is created in the setUp() method, which is run before ever
y test
        fotbot.update("userName1", "password1!", newSteps);

        List<Integer> steps = fotbot.getStepData("userName1", "password1!", "u
serName1");
        List<Integer> expected = list(new Integer [] {0, 1000, 2000});
        assertEquals(expected, steps);
    }

    // getSteps EC2 valid input, username exists, correct password, targetUser
 exists, targetUser=friend of username
    @Test public void getSteps_EQ2()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);
        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //friendUsername1 is created in the setUp() method, which is run befor
e every test
        fotbot.update("friendUsername1", "fpassword1!", newSteps);

        List<Integer> steps = fotbot.getStepData("userName1", "password1!", "f
riendUsername1");
        List<Integer> expected = list(new Integer [] {0, 1000, 2000});
        assertEquals(expected, steps);
    }

    // getSteps EC3 valid input, username exists, correct password, targetUser
 exists, targetUser= FotBot.ADMIN_USERNAME
    @Test public void getSteps_EQ3()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);

        //as admin, it should return empty list
        List<Integer> steps = fotbot.getStepData("userName1", "password1!", Fo
tBot.ADMIN_USERNAME);
        List<Integer> expected = list(new Integer [] {});
        assertEquals(expected, steps);
    }

    // getSteps EC4 valid input, username exists, correct password, targetUser
 exists, targetUser= neither username, admin nor friend of username
    @Test public void getSteps_EQ4()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username2 is created in the setUp() method, which is run before ever
y test
        fotbot.update("userName2", "password2!", newSteps);

        //as userName2 is not related to userName1 it should return empty list
        List<Integer> steps = fotbot.getStepData("userName1", "password1!", "u
serName2");
        List<Integer> expected = list(new Integer [] {});
        assertEquals(expected, steps);

        //List<Integer> expected = list(new Integer [] {0, 1000, 2000});
        //assertEquals("This failed because user cannot view non-
related user info.", expected, steps);
```

```
    }

    // valid input, username exists, correct password, targetUser does not exi
st
    @Test(expected = NoSuchUserException.class)
    public void getSteps_EQ5()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //nonExist is non existed user
        fotbot.update("nonExist", "password1!", newSteps);

        //List<Integer> steps = fotbot.getStepData("userName1", "password1!",
"nonExist");
        //List<Integer> expected = list(new Integer [] {1000, 2000});
        //assertEquals(expected, steps);
    }

    // getSteps EC6 valid input, username exists, incorrect password
    @Test(expected = IncorrectPasswordException.class)
    public void getSteps_EQ6()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);

        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username2 is created in the setUp() method, which is run before ever
y test
        fotbot.update("userName1", "incorr1!", newSteps);

        //as userName2 is not related to userName1 it should return empty list
        //List<Integer> steps = fotbot.getStepData("userName1", "password1!",
"userName1");
        //List<Integer> expected = list(new Integer [] {});
        //assertEquals(expected, steps);
    }

    // getSteps EC7 valid input, username does not exist
    @Test(expected = NoSuchUserException.class)
    public void getSteps_EQ7()
    throws NoSuchUserException, IncorrectPasswordException
    {
        fotbot.incrementCurrentDay(3);
        List<Integer> newSteps = list(new Integer [] {1000, 2000});

        //username2 is created in the setUp() method, which is run before ever
y test
        fotbot.update("userName123", "password1!", newSteps);

        //as userName2 is not related to userName1 it should return empty list
        //List<Integer> steps = fotbot.getStepData("userName1", "password1!",
"userName1");
        //List<Integer> expected = list(new Integer [] {});
        //assertEquals(expected, steps);
    }
```