Lets Move Estate Agency

Project Goal

The project involves the development of web and mobile application for an Estate Agency to advertise and sell their properties. The success of the project is to make the Estate Agency on just a click away so that it is reachable anywhere and anytime.

Project Scope

The overall project scope is to develop two Applications for the Business – Web-based and Mobile-based apps.

- **Web Application**: Web application to let the agency advertise the properties for sale.
- **Mobile Application**: Mobile application allows users to browse and arrange viewings of the properties in their wish list.

Assumptions:

The Project proceeds with the below two major assumptions:

- The sellers don't use these applications to upload the properties directly. They contact the Estate Agency to upload or modify the details of the properties.
- The Buyers use the Mobile application to search, view properties and book appointments. The Mobile applications are especially designed for buyers.

Risk Phase:

The Major risk phase of the project is the '**Search Mechanism'** implementation in the Mobile application for buyers to refine the searches, provide filters and quickly provide the search results based on changing aspects. This involves consideration of performance management to filter complex data based on increasing number of users.

We are planning to introduce the '**Solr Search'** which works on indexing mechanism which can sort out this challenge.

Approach:

We are planning to follow '**Agile Methodology'** where we have separate teams working independently on Web Application frontend and backend working in parallel with minimum dependency, Mobile Application and Testing modules to be effective and meet deadlines.

Project planning:

The activity planning to complete the project with expected success rate is carried out brainstorming with the whole team and clients on requirement and the activities are identified as below in order:

1. **Activity A**: A brief Business Analysis brainstorming session is carried out with all the stakeholders involved in the project. The Discussion summary includes effective Front-end language to be used in the project based on available resources, Back-end tool and the deliverables and users at a primitive stage. This also involves creating specification document for the project.

2. **Activity B**: Client Approval for the Specification document to get started with the project. This requires brief sessions with client and development team if needed.

3. **Activity C**: Web Application Frontend team to design the web application to add new properties, remove or update properties based on sellers' requirement.

4. **Activity D**: Web Application Backend Team manages to upload the details to database on sellers' request.

5. **Activity E**: The Web Frontend team manages the application to handle schedules for the viewings and bookings.

6. **Activity F:** Now, the frontend application is been created with basic requirements. Testing team carries out initial level testing to ensure the entry level application meets requirements as expected.

7. **Activity G**: The DB Team should effectively maintain the database with proper security mechanism to ensure commit updates properly and rollback option. They should frequently update the database with buyers and properties states based on updates.

8. **Activity H:** The Mobile application team waits for the initial web application and database upload to complete and starts their activities. They start developing the mobile application for buyers.

9. **Activity I:** The Mobile application should be made accessible to users. The buyers should be directed to Login page on accessing it.

10. **Activity J:** The Mobile application should have the feature to perform background verification of users. The application requests the users' employment and income proof

to proceed further and trigger a credit score check internally to ensure users' information.

11. **Activity K:** Once the background verification goes successfully, the users are directed to their Home page to provide their preferences for properties.

12. **Activity L:** Agency adds new properties to database based on users' request.

13. **Activity M:** This is an interesting activity '**Notification Trigger'** which notifies the users based on their wish list.

    This activity records two dependencies:
    - Send Notification including newly added properties.
    - Users updated preferences in the mobile application to take into count.

14. **Activity N:** Ensure database is up to date with all newly added properties.

15. **Activity O**: This is the challenging activity '**Search Mechanism'** highlighted in Risk analysis phase.

    This activity also records two dependencies:
    - Database should be up to date with new indexes being added to indexing tables.
    - Search should refine and filter the properties based on users' preferences recorded in the previous stages. And, also provide recommended results based on their preferences on Home page with new updates.

16. **Activity P**: Testing team should test the Search mechanism and the complete mobile application.

    Dependency: The testing team can carry out this activity after they finish Web application testing.

17. **Activity Q:** The Web application and Mobile application development and testing phase is completed and provided initial level Demo to clients.
18. **Activity R:** The project should allow the development to edit the properties anytime and get updates on viewings. Also, the overall reliability and maintenance phase to be carried out in the long run.

Step 2: Estimate **KLOC** values

No	Activity	KLOC

1	A	0.30
2	B	0.25
3	C	1.50
4	D	0.90
5	E	1.20
6	F	0.80
7	G	0.30
8	H	1.45
9	I	0.25
10	J	0.95
11	K	0.25
12	L	0.35
13	M	1.25
14	N	0.40
15	O	1.65
16	P	0.85
17	Q	0.50
18	R	1.00

Step 3:

Using historical data of NASA to estimate the coefficients for **cost per KLOC** (variable a) and an **initial cost** (variable c).

Filter NASA Data based on 'stor' attribute and plotting in Linear Regression:

Stor attribute – Main memory constraint

I have chosen this attribute to filter because since I am using 'Solr Searches' in the project which access the data mostly from cache and access to main memory frequently is reduced. Though it increases the effort in terms of complexity, it is comparatively lesser than memory constraint.

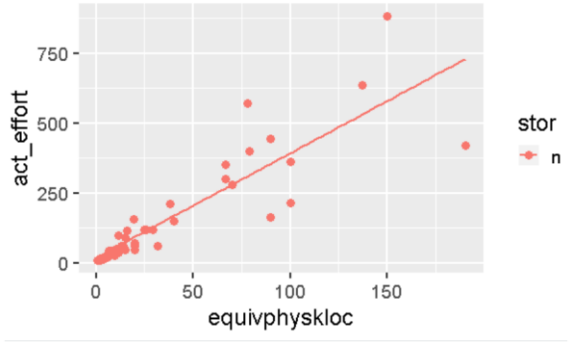
Also, I have filtered the projects from Nasa having low reliability and 200+ kloc since we are focusing more on Low-code development applications for this project.

According to Nasa dataset, my coefficients would be

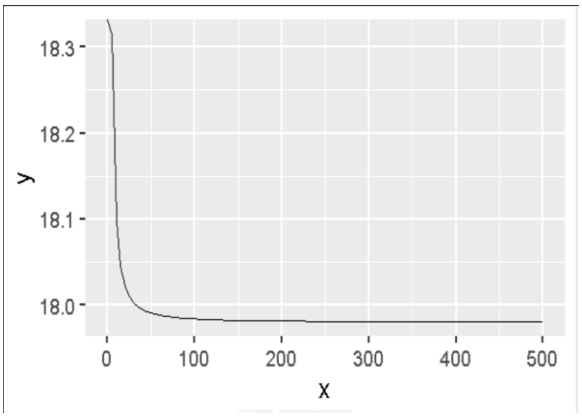
Cost per KLOC (a) – 3.74

Initial Cost (c) – 17.98

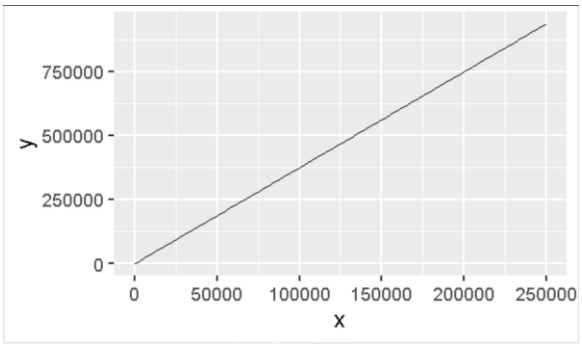
Computing Linear Regression model on this historical data provides the relationship between Effort and KLOC values.



(a) Linear Regression Model



(c) Effort Estimation with Scale factor -1.5



(b) Effort Estimation with a and c coefficients

Step 4: Calculate Scale Factor

There are two major factors which provide cost savings to the project:

- **Design Patterns** which provide code reusability which increases the project savings by **3%**
- **Agile Methodology** which is followed saves time and effort and in-turn increases the project savings by **1.5%**

There are two other factors which stand out as **highly challenging** and increase the project budget to some extent:

- **Reliability and Maintenance**: These two factors increase as the size of the project increases.
  Thus, the overall budget is dropped by **3%.**

So overall change in percentage is as follows:

1. **Increase by 3 % and 1.5 %** = [3+1.5+((3*1.5)/100)]% ≈ 4.5 % Increase

2. **Increase by 4.5 % and Decrease by 3 %** = [4.5-3-((4.5*3)/100)] ≈ 1.5 Increase

So, by this it means our **scale factor is -1.5** since the project budget is saved by 1.5%, it means the scale factor is b<1 and we have **b= - 1.5**

Effort Estimation: Effort is calculated using below formula:

$$E = a * s^b + c$$

No	Activity	KLOC	Effort
1	A	0.30	22.8
2	B	0.25	29.9
3	C	1.50	2.0
4	D	0.90	4.4
5	E	1.20	2.9
6	F	0.80	5.2
7	G	0.30	22.8
8	H	1.45	2.1
9	I	0.25	29.9
10	J	0.95	4.0
11	K	0.25	29.9
12	L	0.35	18.1
13	M	1.25	2.7
14	N	0.40	14.8
15	O	1.65	1.8
16	P	0.85	4.8
17	Q	0.50	10.6
18	R	1.00	3.7

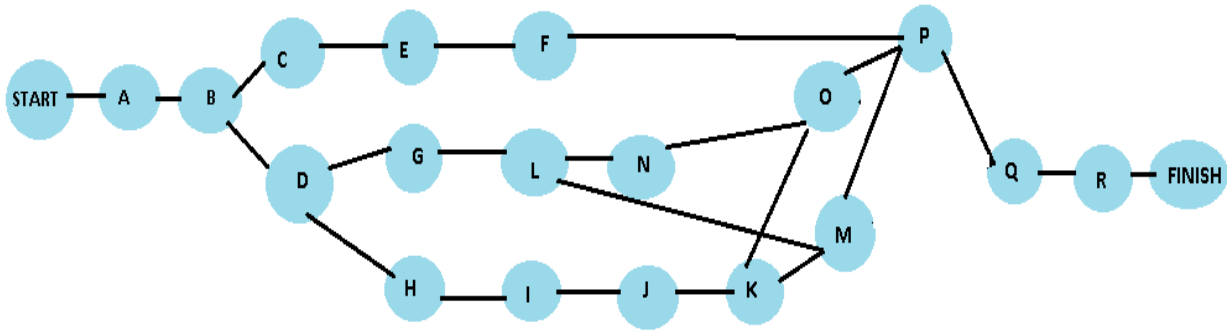
Overall Effort for the project as per above formula including initial cost = (22.8+29.9+2.0+4.4+2.9+5.2+22.8+2.1+29.9+4.0+29.9+4+29.9+18.1+2.7+14.8+1.8+4.8+10.6+3.7) + 17.98 = 230.30

Step 5:

Activity	Predecessor	Predecessor

		Optimistic	Normal	Pessimistic
A	-	20	22.8	35
B	A	25	29.9	45
C	B	1	2.0	5.0
D	B	2	4.4	8
E	C	2.3	2.9	6
F	E	3.8	5.2	9
G	D	20	22.8	35
H	D	1	2.1	4
I	H	25	29.9	45
J	I	3	4.0	7
K	J	25	29.9	45
L	G	15	18.1	24
M	L, K	1.6	2.7	5.8
N	L	13	14.8	20
O	N, K	1	1.8	4
P	O, F, M	4	4.8	6
Q	P	7	10.6	14
R	Q	3	3.7	4

Step 6: Dependency Network



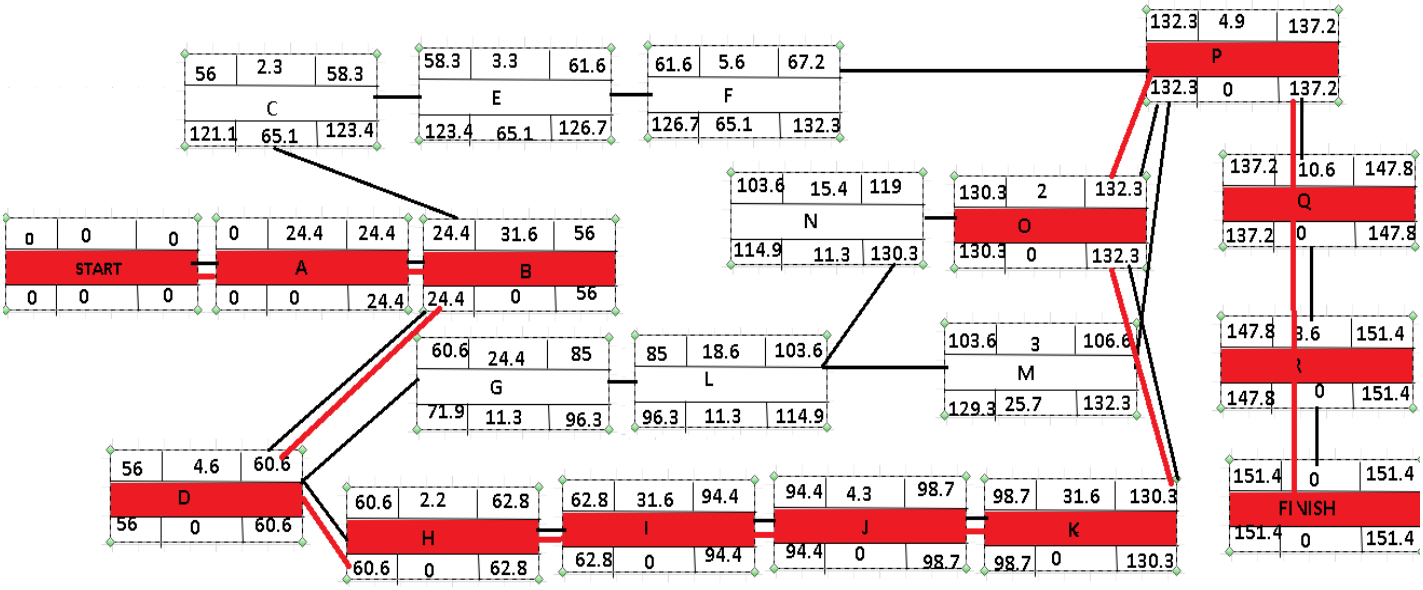
Step 7:

Activity	Predecessor	Predecessor			Expected
		Optimistic	Normal	Pessimistic	
A	-	20	22.8	35	24.4

B	A	25	29.9	45	31.6
C	B	1	2.0	5.0	2.3
D	B	2	4.4	8	4.6
E	C	2.3	2.9	6	3.3
F	E	3.8	5.2	9	5.6
G	D	20	22.8	35	24.4
H	D	1	2.1	4	2.2
I	H	25	29.9	45	31.6
J	I	3	4.0	7	4.3
K	J	25	29.9	45	31.6
L	G	15	18.1	24	18.6
M	L,K	1.6	2.7	5.8	3.0
N	L	13	14.8	20	15.4
O	N,K	1	1.8	4	2.0
P	O,F,M	4	4.8	6	4.9
Q	P	7	10.6	14	10.6
R	Q	3	3.7	4	3.6

Step 7: PERT Chart

Based on the dependency network for the project, the PERT chart is depicted as below:



Step 8:

Critical Path:

The critical path for the project according to the PERT chart is

Start -> A -> B -> D -> H -> I -> J -> K -> O-> P -> Q -> R -> FINISH

This path is critical as we have identified earlier in the **Risk analysis phase of Project Scope** that **'Search Mechanism**' is a complex implementation part using '**Solr indexes'** approach.

Also, this path involves the major activities such as

- Web application database upload which is a major activity.
- Mobile Application Development initial phase
- Also the Mobile application background verification procedure which is complex.
- Also, the Testing of all the major activities such as 'Notification Trigger', 'Search Mechanism' and the entire application as a whole and final Demo to clients for approval.

We take this critical path and increase resources and efforts compensating from the other teams support and keeps tasks on track.

TASK 2: BLACK BOX TESTING

1. **MC/DC Usecase:**

   Considering individual case statements in the program and evaluating the MC/DC testcase sets.

   - Mastercard

Length !=16 (Line no: 11)	Substring(0,2) <51 (Line no: 12)	Substring(0,2) >56 (Line no: 13)	Decision 11||12||13	Output Line number
T	F	F	**T**	**16**
F	**T**	F	**T**	**16**
F	F	**T**	**T**	**16**
F	**F**	**F**	**F**	**14**

   - VISA

Length! =13 (Line no: 19)	Length! =16 (Line no:19)	Substring (0,1)!=4 (Line no: 20)	Decision (19&&19)||20	Output Line number
T	**T**	F	**T**	**23**
F	T	**T**	**T**	**23**
T	**F**	F	**F**	**21**

F	T	F	F	21

- AMEX

Length! =15 (Line no: 26)	Substring (0,2)! =34(Line no: 27)	Substring (0,2)! =37 (Line no: 28)	Decision (26||(27&&28))	Output Line number
T	T	F	T	31
F	T	F	F	29
F	T	T	T	31
F	F	T	F	29

- DISCOVER:

Length! =16 (Line no: 94)	Substring (0,5)! =6011 (Line no: 95)	Decision 94||95	Output Line number
T	F	T	38
F	T	T	38
F	F	F	36

- **DINERS**

Length () !=14 (Line no. 41)	Substring (0,2)! = 36 (Line 42)	Substring (0,2)! = 38 (Line 43)	Substring (0,3) < 300 (Line 44)	Substring (0,3) > 305 (Line 45)	Outcome (41 || ((42 && 43) && 44 || 45))	Output Line number
F	T	F	T	F	F	46
F	T	T	T	F	T	48
F	T	T	F	F	F	46
T	F	T	T	F	T	48
F	F	T	T	F	F	46
F	F	T	T	T	T	48

Brief Overall Idea from MC/DC:

Thus, we are giving the type of card and its number as input and validating the input card number by its length and substring where every given input Length and substring individually affected the overall outcome of the functionality.

Suppose my input is 5506 9005 1000 0234 – which is of type Mastercard, then the length is 16 or substring (0,2) is 55 both of which individually satisfies the criteria confirming it is a valid Mastercard.

2. Category Partition Method:

1) The **categories** that I am choosing are,

- **Type** - It identifies to find the card to match the properties based on below two properties length and substring. It is kind of output which matches the given two inputs.
- **Length** - I take it as a category because the length differs for different cards.
- **Substring** -It is taken as a category because each card starts with unique numbers.

I ended up choosing this category set because given a number and substring, we can validate the type of card.

2) Constraints:

I have considered the unique conditions for the card as constraints.

- **Mastercard:**
  Mastercard begins with 51 through 55
- **Visa:** Visa card begins with 4 and has length 13.
- **Amex:**
  Amex card begins with 34 or 37 and its length is 15
- **Discover:**
  Discover card begins with 6011
- **Diners:**
  Diners card begins with 36 or 38 or 300 through 305 and of length 14.
- **Possible Error constraints**:
  Type of card, which is invalid, length which is 1 or 40 as invalid by any case a card length cannot be 40 or 0. Also a card cannot begin with 0

3) Test Frames

The possible test frames are below

- 432 testcases generated when error and constraints
- 224 frames generated when there is only error constraint
- 22 frames generated when adding error and constraints.

Impact of constraints:

- Initially we had 432 test frames (6*6*12)
- After adding error constraints, we will get (5*4*11) +4 – excluding the errors and adding them at last.
- After adding constraints, we will get 22 frames as a result.

4) Sample Testcase

Inputs: Length is 14 and substring 305

Output matching type can be Diners as per testcase.

Below is the sample test frame having Length as 14 and begins with 305 which is identified as Diners card.

```
Test Case 22              (Key = 5.5.12.)
   Type       :  Diners
   Length     :  14
   substring  :  305
```

TASK 3: WHITE BOX TESTING

1. **Junit Testcases in Java file:**

Attaching the junit test case file which is successfully compiled and executed.

StringStackTest.java

StringStackTest.java

2. The identified testcases cover the below branches and code in the class.

TEST CASE	LINE NUMBER	BRANCH COVERED
1. IsEmptyTest ()	9	Public Boolean isEmpty () - (isEmpty method successful case)
2. popExceptionTest ()	20	Public String pop () { If (pointer<= 0) - (else-branch of pop)
3. pushExceptionTest ()	25	Public void push (String o) { If(pointer>=capacity) - (else-branch of push)
4. pushTest ()	36	objects[pointer++] = o (if-branch of push)
5. popTest ()	42	Objects [--pointer] (if-branch of pop)
6. IsNotEmptyTest ()	14	Public Boolean isEmpty () (isEmpty method unsuccessful case)

3. MUTATION TESTING

- **Coverage Testing is not completely trustworthy:**

Coverage testing, the Branch testing we performed above is not fully desirable because we cannot make sure all the criteria are tested simply because all the lines of code or branches are executed in the test suite.

Code coverage does not always ensure the quality of the testcases written. The 100 lines of code executed in a program does not make sure the 100 lines are correct or tested.

- **Mutation testing:**

Mutation testing is the process of introducing the mutants to the code which makes the testcase to fail. Create mutants by simulating previous mistakes in experience if we can get rid of those errors. If one or more test cases fail for the mutant code, then we can ensure the quality of the testcase is good enough to kill the mutants. But if the mutants survive, then we need to improve the quality of testcases to be fault tolerant.

- **Possible Mutants that can be introduced:**

The possible mutants in the code that can be introduced are:

1) Operand replacement
2) Operator replacement
3) Statement modification
4) Datatype modifications
5) Removing else part, etc.

- **Three types of Mutation testing**:

From the above-mentioned kinds of mutants, we can summarize the types of mutation testing as:

1) Statement Mutation: We can modify (add/remove) part of code
2) Values Mutation: Primary input values are modified
3) Decision Mutation: Control statements like conditions can be altered.

- **Mutation Score:**

We can calculate the Mutation score as follows:

Mutation Score = (Killed Mutants/ Total number of Mutants) * 100

- **Mutation Testing Vs Code Coverage with our Usecase:**

Code Coverage fails to evaluate certain border conditions in the code.

In this example, we have all simple values getting evaluated. So, there are not many cases which may get failed in Junit.

But one case according to my understanding is that, the input String which is to be pushed is not evaluated if it is the allowed string to be pushed into the Stack. We have no reference in the code that validates the input String by the user. User may sometimes give any value like “[a]”, “[1]”. This must be evaluated as well for a complex program because input string is the primary parameter for a program.

"**Mutation testing gives the freedom to test this use case by modifying the data type and check if that mutant is killed by the testcase**"

- **Benefits of Mutation Testing**:

Mutation testing enhances the quality of testcases and thereby making sure almost the entire code works as expected.

It moves out if the metrics satisfaction approach which is followed by coverage testing and goes much beyond with more efforts.

4. MUTANT IDENTIFICATION

Attaching the Mutant source code here:


StringStackMutant.java


StringStackMutant.java

1) TESTCASE: The testcase being evaluated is **isEmpty ()**

MUTANT (SOURCE CODE): The identified Mutant in the source code is as below where the condition is modified.

```
public boolean isEmpty() {
   // return pointer <= 0;
    return pointer >0; // killed by isEmptyTestMutant testcase
}
```

2) TESTCASE: The testcase being evaluated is **popExceptionTest ()**

MUTANT (SOURCE CODE): The identified Mutant in the source code is as below where I have pushed a value to stack to mutate the stack empty argument. Now the pointer value increases to 1 and the condition fails eventually.

```
public String pop() {
   // if (pointer <= 0)
    pointer++;  //killed by popExceptioinMutantTest testcase
    if (pointer <= 0)
        throw new IllegalArgumentException("Stack empty");
```

JUNIT FAILURE FOR MUTANT TESTCASE: The mutant created is being killed by the testcase written for the branch and identifies that the exception is expected but not thrown.

3) TESTCASE: The testcase being evaluated is **pushExceptionTest ()**

MUTANT (SOURCE CODE): The identified Mutant in the source code is as below where I have popped out a value after adding everytime from the Stack and the expected exception 'Stack exceeded' never occurs.

```
 if (pointer >= capacity)
 {

    throw new IllegalArgumentException("Stack ex
 }

objects[pointer++] = o;
pointer--;  //killed by pushExceptionMutantTest.
}
```

JUNIT FAILURE FOR MUTANT TESTCASE: The mutant created is being killed by the testcase written for the branch and identifies that the exception is expected but not thrown.

4) **TESTCASE:** The testcase being evaluated is **popTest ()**

MUTANT (SOURCE CODE): The identified Mutant in the source code is as below where in the pop method, I have modified the pointer index from 'pointer - -' to 'pointer' so that it will point to null value.

```
public String pop() {
   // if (pointer <= 0)
    pointer++;  //killed by popExceptioinMutantTest testcase
    if (pointer <= 0)
        throw new IllegalArgumentException("Stack empty");
   // return objects[--pointer];
    return objects[pointer]; //killed by popMutantTest testcase

}
```

JUNIT FAILURE FOR MUTANT TESTCASE: The mutant created is being killed by the testcase since the expected value is 'b' and the actual value is 'NULL'.

5) **TESTCASE:** The testcase being evaluated is **pushTest ()**

MUTANT (SOURCE CODE) : The identified Mutant in the source code is as below where in the push method, I have set the pointer value to 15 which goes to exception path and it never satisfies the push method condition.

```
public void push(String o) {
// pointer=12;
/*killed by pushMutantTest. Comment this to get pushExceptionMutant :
    and uncomment to get pushMutant successful.
/* Both pushmutant and pushexception mutant will not
execute at the same time due to opposite conditions. so comment one
 if (pointer >= capacity)
 {

    throw new IllegalArgumentException("Stack exceeded capacity!");
 }
```

JUNIT FAILURE FOR MUTANT TESTCASE: The mutant created is being killed by the testcase since the expected value is 'b' and the actual value is 'NULL'.