SE-3002
SOFTWARE QUALITY ENGINEERING

RUBAB JAFFAR
RUBAB.JAFFAR@NU.EDU.PK

Part II-Software Testing

Structural Testing, Abstraction Level of Testing

Lecture # 28, 29, 30

8,9,10 Nov

TODAY'S OUTLINE

- Structural Testing
  - Mutation Testing

MUTATION TESTING

- Popular technique to assess the effectiveness of a test suite.
- A large number of test cases for any program can be generated but cant execute all of them due to time or resources.
- So the idea is to select a few test cases using any testing technique and prepare a test suite.
- But how to assess the effectiveness of a selected test suite? Is this test suite adequate for the program?
- If the test suite is not able to make the program fail, there may be one of the following reasons:
- The test suite is effective but hardly any errors are there in the program. How will a test suite detect errors when they are not there?
- The test suite is not effective and could not find any errors. Although there may be errors, they could not be detected due to poor selection of test suite. How will errors be detected when the test suite is not effective?

MUTATION AND MUTANTS

- Mutation testing may help us to assess the effectiveness of a test suite and may also enhance the test suite, if it is not adequate for a program.
- Mutation is the process of changing a program.
- This change may be limited to one, two or very few changes in the program.

MUTATION AND MUTANTS

- To mutate a program means to change a program. We generally make only one or two changes in order to assess the effectiveness of the selected test suite
- We may make many mutants of a program by making small changes in the program.
- Every mutant will have a different change in a program.
- Every change of a program may give a different output as compared to the original program.
- The original program and mutant are syntactically correct and should compile correctly.

EXAMPLE: FIND THE LARGEST NUMBER AMONGST THREE NUMBERS.

```
#include<stdio.h>
#include<conio.h>
1.  void main()
2.  {
3.  float A,B,C;
4.  clrscr();
5.  printf("Enter number 1:\n");
6.  scanf("%f", &A);
7.  printf("Enter number 2:\n");
8.  scanf("%f", &B);
9.  printf("Enter number 3:\n");
10. scanf("%f", &C);
    /*Check for greatest of three numbers*/
11. if(A>B) {
12. if(A>C) {
13.         printf("The largest number is: %f\n",A);
14.         }
15. else  {
16.         printf("The largest number is: %f\n",C);
17.         }
18. }
19. else {
20. if(C>B) {
21.         printf("The largest number is: %f\n",C);
22.         }
23. else  {
24.         printf("The largest number is: %f\n",B);
25.         }
26. }
27. getch();
28. }
```

FIRST ORDER MUTANT OF THE EXAMPLE PROGRAM

- Many changes can be made in the program.
- Mutant M_1 is obtained by replacing the operator '>' of line number 11 by the operator ' ='.
- Mutant M_2 is obtained by changing the operator '>' of line number 20 to operator '<'.
- These changes are simple changes. Only one change has been made in the original program to obtain mutant M_1 and mutant M_2.

```
/*Check for greatest of three numbers*/
11.   if(A>B){   ← if(A=B) { mutated statement ('>' is replaced by '=')
12.   if(A>C) {
```

M_1: First order mutant

```
19.   else {
20.   if(C>B) {   ← if(C<B) { mutated statement ('>' is replaced by '<')
21.         printf("The largest number is: %f\n",C);
```

M_2: First order mutant

HIGH ORDER MUTANT

- The mutants generated by making only one change are known as first order mutants.
- We may obtain second order mutants by making two simple changes in the program and third order mutants by making three simple changes, and so on.
- The second order mutant (M_3) of the example program can be obtained by making two changes in the program and thus changing operator '>' of line number 11 to operator '<' and operator '>' of line number 20 to ' >=' .
- The second order mutants and above are called higher order mutants.
- Generally, in practice, we prefer to use only first order mutants in order to simplify the process of mutation.

SECOND ORDER MUTANT

```
11.   if(A>B) {   ← if(A<B) { mutated statement (replacing '>' by '<')
12.   if(A>C) {
13.           printf("The largest number is: %f\n",A);
14.           }
15.   else  {
16.           printf("The largest number is: %f\n",C);
17.           }
18.   }
19.   else {
20.   if(C>B) {   ← if(C≥B) { mutated statement (replacing '>'by '≥')
21.           printf("The largest number is: %f\n",C);
```

MUTATION OPERATORS

- Mutants are produced by applying mutant operators. An operator is essentially a grammatical rule that changes a single expression to another expression. The changed expression should be grammatically correct as per the used language.
- If one or more mutant operators are applied to all expressions of a program, we may be able to generate a large set of mutants.
- We should measure the degree to which the program is changed. If the original expression is x + 1, and the mutant for that expression is x + 2, that is considered as a lesser change as compared to a mutant where the changed expression is (y * 2) by changing both operands and the operator.
- If x – y is changed to x – 5 to make a mutant, then we should not use the value of y to be equal to 5. If we do so, the fault will not be revealed.

MUTATION OPERATORS

- Some of the mutant operators for object oriented languages like Java, C++ are given as:
  - Static modifier change
  - Changing the access modifier, like public to private.
  - Argument order change
  - Super Keyword change
  - Operator change
  - Any operand change by a numeric value.

MUTATION SCORE

- When we execute a mutant using a test suite, we may have any of the following outcomes:
  - The results of the program are affected by the change and any test case of the test suite detects it. If this happens, then the mutant is called a killed mutant.
  - The results of the program are not affected by the change and any test case of the test suite does not detect the mutation. The mutant is called a live mutant.
- The mutation score associated with a test suite and its mutants is calculated as:

$$\text{Mutation Score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$

MUTATION SCORE

- The mutation score measures how sensitive the program is to the changes and how accurate the test suite is.
- A mutation score is always between 0 and 1.
- A higher value of mutation score indicates the effectiveness of the test suite although effectiveness also depends on the types of faults that the mutation operators are designed to represent.
- The live mutants are important for us and should be analyzed thoroughly.
- Why is it that any test case of the test suite not able to detect the changed behaviour of the program?
- One of the reasons may be that the changed statement was not executed by these test cases. If executed, then also it has no effect on the behaviour of the program.

EXAMPLE: PROGRAM TO FIND THE LARGEST OF THREE NUMBERS

- Generate five mutants (M_1 to M_5) and calculate the mutation score of this test suite.

S. No.	A	B	C	Expected Output
1.	6	10	2	10
2.	10	6	2	10
3.	6	2	10	10
4.	6	10	20	20

FIVE MUTANTS

Mutated statements

Mutant No.	Line no.	Original line	Modified Line
M_1	11	if(A>B)	if (A<B)
M_2	11	if(A>B)	if(A>(B+C))
M_3	12	if(A>C)	if(A<C)
M_4	20	if(C>B)	if(C=B)
M_5	16	printf("The Largest number is:%f\n",C);	printf("The Largest number is:%f\n",B);

PROGRAM RESULTS AFTER EXECUTING MUTANTS

Actual output of mutant M_1

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	6
2.	10	6	2	10	6
3.	6	2	10	10	10
4.	6	10	20	20	20

Actual output of mutant M_2

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	10
4.	6	10	20	20	20

PROGRAM RESULTS AFTER EXECUTING MUTANTS

Actual output of mutant M_3

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	2
3.	6	2	10	10	6
4.	6	10	20	20	20

Actual output of mutant M_4

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	10
4.	6	10	20	20	10

Actual output of mutant M_5

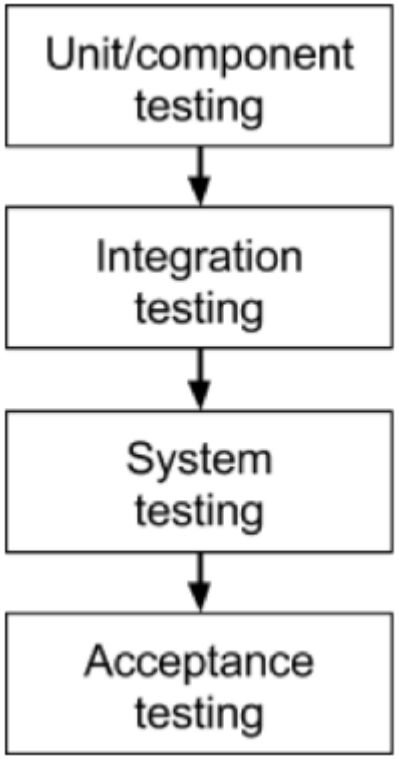
Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	2
4.	6	10	20	20	20

MUTATION SCORE

$$\text{Mutation Score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$
$$= \frac{4}{5}$$
$$= 0.8$$

TODAY'S OUTLINE

- Abstraction Level of Testing
- Unit testing
- Integration testing
- System testing
- Acceptance testing



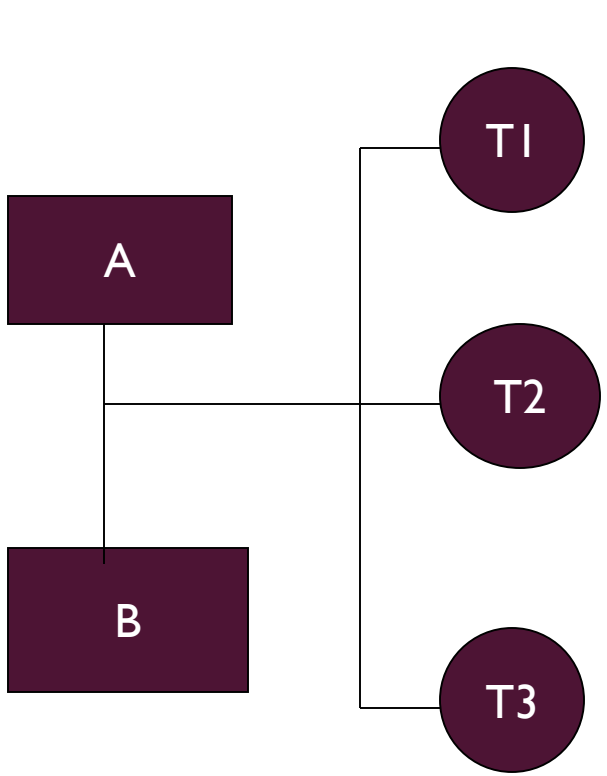
UNIT TESTING

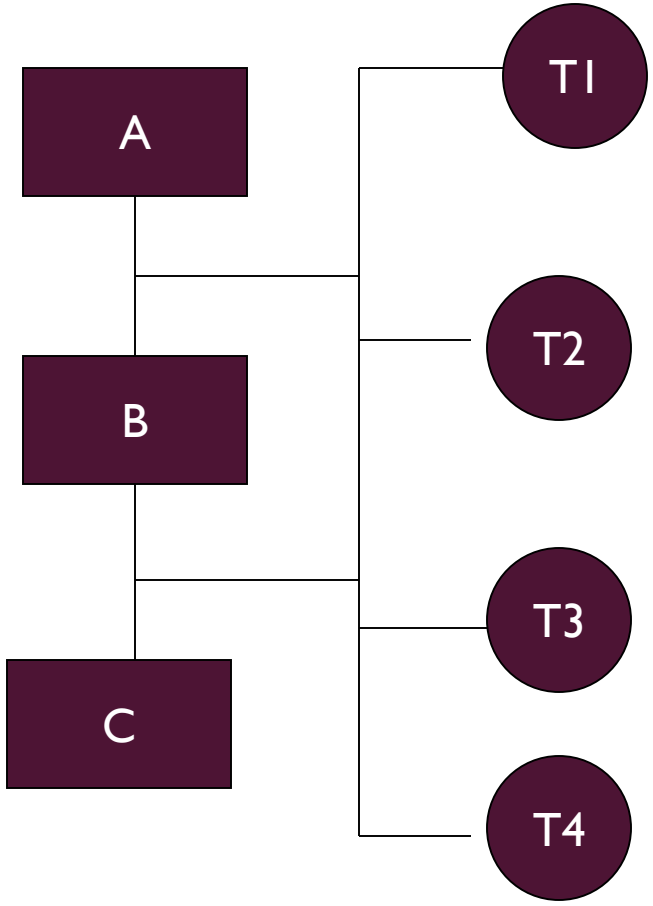
- The most basic type of testing
- Unit testing aims to verify each part of the software by isolating it and then perform tests to demonstrate that each individual component is correct in terms of fulfilling requirements and the desired functionality.
- White box testing
- This type of testing is performed at the earliest stages of the development process, and in many cases it is executed by the developers themselves before handing the software over to the testing team.
- The advantage of detecting any errors in the software early in the day is that by doing so the team minimizes software development risks, as well as time and money wasted in having to go back and undo fundamental problems in the program once it is nearly completed.

INTEGRATION TESTING

- Integration testing aims to test different parts of the system in combination in order to assess if they work correctly together.
- By testing the units in groups, any faults in the way they interact together can be identified.
- There are many ways to test how different components of the system function at their interface; testers can adopt either a bottom-up or a top-down integration method.
- In bottom-up integration testing, testing builds on the results of unit testing by testing higher-level combination of units (called modules) in successively more complex scenarios.
- It is recommended that testers start with this approach first, before applying the top-down approach which tests higher-level modules first and studies simpler ones later.

Top-down Testing





STUBS AND DRIVERS

- When testing incomplete portions of software, developers and testers often need extra software components, sometimes called scaffolding.
- The two most common types of scaffolding are known as test stubs and test drivers.
- A test stub is a skeletal or special-purpose implementation of a software module, used to develop or test a component that calls the stub or otherwise depends on it. It replaces a called component.
- For OO programs, the XP community has developed a version of the stub called the mock. A mock is a special-purpose replacement class that includes behavior verification to check that a class under test made the correct calls to the mock.
- The simplest action for a stub is to assign constant values to the outputs. (top-down integration)

STUBS AND DRIVERS

- A test driver is a software component or test tool that replaces a component that takes care of the control and/or the calling of a software component.
- The simplest form of driver is a main() method for a class.
- Effective programmers often include a main() for every class, containing statements that carry out simple testing of the class. (Bottom-up integration)
- Test drivers can include hard-coded values or retrieve the values from an external source like the tester or a file. Tools exist to generate test drivers automatically.
- Both test driver and test stub generators are included in other test tools.

SYSTEM TESTING

- As the name implies, all the components of the software are tested as a whole in order to ensure that the overall product meets the requirements specified.
- System testing is a very important step as the software is almost ready to ship and it can be tested in an environment which is very close to that which the user will experience once it is deployed.
- System testing enables testers to ensure that the product meets business requirements, as well as determine that it runs smoothly within its operating environment. This type of testing is typically performed by a specialized testing team.
- Black box testing

ACCEPTANCE TESTING

- Finally, acceptance testing is the level in the software testing process where a product is given the green light or not. The aim of this type of testing is to evaluate whether the system complies with the end-user requirements and if it is ready for deployment.
- The testing team will utilize a variety of methods, such as pre-written scenarios and test cases to test the software and use the results obtained from these tools to find ways in which the system can be improved.
- By performing acceptance tests, the testing team can find out how the product will perform when it is installed on the user's system. There are also various legal and contractual reasons why acceptance testing has to be carried out.

WHO PERFORMS THE TESTS?

- **Unit Testing**
  - done by the programmer and/or development team.
- **Integration Testing**
  - can be the development team or a testing unit.
- **System Testing**
  - usually done by an independent testing team (internal or external (consultants) team.
- For small companies, another testing team from another development team can be used and swapped.
- Can always outsource testing too.

WHERE TO PERFORM THE TESTS?

- Typically at the software **developer's** site..
- For system tests, test at **developer's** or **customer's** site (target site).
- If outsourced, testing can be done at **consultant's** site.

TEST PLANS

- The goal of test planning is to establish the list of tasks which, if performed, will identify all of the requirements that have not been met in the software. The main work product is the *test plan*.
  - The test plan documents the overall approach to the test. The test plan serves as a summary of the test activities that will be performed.
  - It shows how the tests will be organized, and outlines all of the testers' needs which must be met in order to properly carry out the test.
  - The test plan should be inspected by members of the engineering team and senior managers.

TEST PLAN OUTLINE

Purpose

A description of the purpose of the application under test.

Features to be tested

A list of the features in the software that will be tested. It is a catalog of all of the test cases (including a test case number and title) that will be conducted, as well as all of the base states.

Features not to be tested

A list of any areas of the software that will be excluded from the test, as well as any test cases that were written but will not be run.

Approach

A description of the strategies that will be used to perform the test.

Suspension criteria and resumption requirements

Suspension criteria are the conditions that, if satisfied, require that the test be halted. Resumption requirements are the conditions that are required in order to restart a suspended test.

Environmental Needs

A complete description of the test environment or environments. This should include a description of hardware, networking, databases, software, operating systems, and any other attribute of the environment that could affect the test.

Schedule

An estimated schedule for performing the test. This should include milestones with specific dates.

Acceptance criteria

Any objective quality standards that the software must meet, in order to be considered ready for release. This may include things like stakeholder sign-off and consensus, requirements that the software must have been tested under certain environments, minimum defect counts at various priority and severity levels, minimum test coverage numbers, etc.

Roles and responsibilities

A list of the specific roles that will be required for people in the organization, in order to carry out the test. This list can indicate specific people who will be testing the software and what they are responsible for.

TEST CASES

- A *test case* is a description of a specific interaction that a tester will have in order to test a single behavior of the software. Test cases are very similar to use cases, in that they are step-by-step narratives which define a specific interaction between the user and the software.
  - A typical test case is laid out in a table, and includes:
    - A unique *name* and *number*
    - A *requirement* which this test case is exercising
    - *Preconditions* which describe the state of the software before the test case (which is often a previous test case that must always be run before the current test case)
    - *Steps* that describe the specific steps which make up the interaction
    - *Expected Results* which describe the expected state of the software after the test case is executed
- Test cases must be repeatable.
  - Good test cases are data-specific, and describe each interaction necessary to repeat the test exactly.

TEST CASES – GOOD EXAMPLE

Name	TC-47: Verify that lowercase data entry results in lowercase insert
Requirement	FR-4 (Case sensitivity in search-and-replace), bullet 2
Preconditions	The test document TESTDOC.DOC is loaded (base state BS-12).
Steps	1. Click on the "Search and Replace" button. 2. Click in the "Search Term" field. 3. Enter *This is the Search Term.* 4. Click in the "Replacement Text" field. 5. Enter *This IS THE Replacement TeRM.* 6. Verify that the "Case Sensitivity" checkbox is unchecked. 7. Click the OK button.
Expected results	1. The search-and-replace window is dismissed. 2. Verify that in line 38 of the document, the text *this is the search term* has been replaced by *this is the replacement term.* 3. Return to base state BS-12.

TEST CASES – BAD EXAMPLE

Steps	1. Bring up search-and-replace. 2. Enter a lowercase word from the document in the search term field. 3. Enter a mixed-case word in the replacement field. 4. Verify that case sensitivity is not turned on and execute the search.
Expected results	1. Verify that the lowercase word has been replaced with the mixed-case term in lowercase.

TEST EXECUTION

- The software testers begin executing the test plan after the programmers deliver the *alpha build*, or a build that they feel is feature complete.
  - The alpha should be of high quality—the programmers should feel that it is ready for release, and as good as they can get it.
- There are typically several iterations of test execution.
  - The first iteration focuses on new functionality that has been added since the last round of testing.
  - A *regression test* is a test designed to make sure that a change to one area of the software has not caused any other part of the software which had previously passed its tests to stop working.
  - Regression testing usually involves executing all test cases which have previously been executed.
  - There are typically at least two regression tests for any software project.

TEST EXECUTION

- When is testing complete?
  - No defects found
  - Or defects meet acceptance criteria outlined in test plan

TABLE 8-6. Acceptance criteria from a test plan

1. Successful completion of all tasks as documented in the test schedule.
2. Quantity of medium- and low-level defects must be at an acceptable level as determined by the software testing project team lead.
3. User interfaces for all features are functionally complete.
4. Installation documentation and scripts are complete and tested.
5. Development code reviews are complete and all issues addressed. All high-priority issues have been resolved.
6. All outstanding issues pertinent to this release are resolved and closed.
7. All current code must be under source control, must build cleanly, the build process must be automated, and the software components must be labeled with correct version numbers in the version control system.
8. All high-priority defects are corrected and fully tested prior to release.
9. All defects that have not been fixed before release have been reviewed by project stakeholders to confirm that they are acceptable.
10. The end user experience is at an agreed acceptable level.
11. Operational procedures have been written for installation, set up, error recovery, and escalation.
12. There must be no adverse effects on already deployed systems.

TEST SUITE

- A set of test scripts or test procedures to be executed in a specific test run.
- When you have hundreds / thousands of test cases, a test suite allows you to categorize them in a way that matches your planning or analysis needs.
- For example, you could have a test suite for each of the core features of the software or you could have a separate test suite for a particular type of testing.
- An example of a test suite for purchasing a product could comprise of the following test cases:
- Test Case 1: Login
- Test Case 2: Add Products
- Test Case 3: Checkout
- Test Case 4: Logout
- Each of the test cases above are dependent on the success of the previous test cases.

TEST LOG REPORT

- The test case log documents different test cases for a particular test type to be executed during testing.
- It also records the results of the tests, which provides the detailed evidence for the test log summary report and enables us to reconstruct the test, if necessary.
- Test Case ID
- Requirement ID
- Test Description
- Test Data
- Expected Result
- Actual result
- Pass/Fail

REGRESSION TESTING

- Regression testing is the process of re-testing software that has been modified.
- Large components or systems tend to have large regression test suites. Small changes to one part of a system often cause problems in distant parts of the system.
- Regression testing is used to find this kind of problem.
- Changes to software are often classified as corrective, perfective, adaptive, and preventive. All of these changes require regression testing.
- If one or more regression tests fail, the first step is to determine if the change to the software is faulty, or if the regression test set itself is broken.
- In either case, additional work is required.

REGRESSION TESTING

- It is worth emphasizing that regression tests must be automated.
- Indeed, it could be said that unautomated regression testing is equivalent to no regression testing.
- A wide variety of commercially available tools are available.
- Capture/replay tools automate testing of programs that use graphical user interfaces.
- Version control software, already in use to manage different versions of a given system, effectively manages the test sets associated with each version.
- Scripting software manages the process of obtaining test inputs, executing the software, marshaling the outputs, comparing the actual and expected outputs, and generating test reports.

SOFTWARE TESTING

- Testing software applications is mandatory and inevitable because errors often become the part of software unintentionally during the design, code and implementation phase.
- Software is becoming more complex nowadays, having many lines of code, and as a result more iterations of testing are needed to be performed.
- In case the design issues go undetected, at that point it'll become more troublesome to follow back bugs and repair it. It'll end up in a high budget to settle it. Sometimes, whereas repairing one bug may instigate another bug in other modules unintentionally. Discovering bugs in the initial phases of developing software takes much lesser costs to fix them. Because of this it is imperative to discover errors within the early stages of the computer program advancement life cycle.
- Software testing is time taking and a costly process; hence, it is necessary to lessen human efforts in testing.

SHORTCOMINGS OF MANUAL TESTING

- Time-consuming process
- Involves more human efforts
- Not as much of precise outcomes
- Testing several features concurrently is not possible
- Deficiency of reusability of testing trials
- Deficiency of test comprehensiveness.

AUTOMATED SOFTWARE TESTING

- Benefits of Automation Testing
- Drawbacks of Automation Testing
- Factors to go for Automation Testing
- Factors to Choose Automation Tool
- Tools Available for Testing Software

AUTOMATED SOFTWARE TESTING

- The automated software testing is a method during which software testing tools run the already drafted test cases on a software program.
- These testing tools manage the carrying out of test cases and then do the evaluation of actual results by comparing it with the expected results. It can automate
  - Software Business Application Critical test cases,
  - test cases that need to be run again and again
  - test Cases that are very laborious to perform manually or
  - test Cases that take a lot of time for execution
- in a systematize testing process.
- The foremost objective of automated testing is to ease the efforts involved in the testing process as much as possible by writing a minimal set of testing scripts.

COMMON AUTOMATION TOOLS

- There are many test automation tools available in the market.
- Each automation tool has some strong points and some shortcomings and serves an alternate need.
- A thorough investigation of those different automation tools ought to be performed prior to choosing any tool.
- Many factors are thought of for the evaluation of these tools such as cost, software type, testing prerequisites, abilities needed to utilize the tool etc.
- This evaluation process needs a great deal of exertion, time, and planning.
- There follows a clarification of the absolute most usually utilized automation tools alongside their benefits and detriments.

AUTOMATION TOOLS COMPARISON

	Price	Platform	Supported Language	Tested app	Coding Skills required	Learning curve
Selenium	Free	Windows/Mac/Linux	Java, Python, C#, PHP, Javascript, Ruby, Perl	Web, mobiles	Advanced skills	Steep
Test complete	License and maintenance fees	Windows	VB, Jscript, Python, C++, C#, PHP, Javascript, Ruby,	Web, mobiles, desktop	Minimum skills, Advanced skills for pro scripting	Mild
Katalon Studio	Free	Windows, Mac	Java, Groovy	Web, mobile	Minimum skills, Advanced skills for pro scripting	Mild
UFT	License and maintenance fees	Windows	VB Script	Web, mobiles, desktop	Minimum skills, Advanced skills for pro scripting	Moderate
Renorex	License and maintenance fees	Windows	C#, VB.Net, Python	Web, mobiles, desktop	Minimum skills, Advanced skills for pro scripting	Moderate

SELECTION OF AUTOMATION TOOL

- Any automation tool can't be good or bad for all software projects.
- The decision of an automation tool generally relies upon the idea of the product to be tried. For instance, Selenium might be the most well-known automation tool, however in the case that the product to be tested is desktop based then this automation tool has no utilization here. Thus the testing needs for the software system should be clearly understood before selecting the automation tool for the testing process.
- Depending on the testing requirements, a combination of automation tools can also be used.

CHALLENGES IN TEST AUTOMATION

- Automating the testing process is an important aspect of software testing.
- By utilizing automated testing, we can facilitate and enhance the software validation process and increment testing inclusion. Nonetheless, there are a ton of difficulties in applying test automation for applications under test (AUT).
- Without conquering these difficulties, testers may confront innumerable bad effects that can cause automated software testing failure.
- The main four difficulties that highestly affect the general automation test exertion and task achievement are as per the following.
  - Viable Communicating and Collaborating in team
  - Selecting a Right Tool
  - Selecting a Proper Testing Approach
  - High Upfront Investment Cost

That is all