T. Y. B.Sc. I.T.
Semester V

SOFTWARE TESTING

TEACHER's REFERENCE MANUAL
FOR PRACTICALS

2013 – 2014

Table of Contents

Practical No 1.

Title: Setting up a company that sells testing services to software houses

Problem Statement:

1. What is the testing process that will be followed in the company?
2. What is the focus of the testing services?
3. What kind of people are you going to hire as staff for the company?
4. How are you going to validate that a testing project carried out in the company has been beneficial to the customer?
5. What kind of automated tools will the company use?

Solution:

Answer 1:

The testing process that will be followed by the company is:

a) Prepare for testing a software system.
b) Plan the tests that will be conducted on the software system.
c) Execute the steps as defined in the test plan.
d) Conduct acceptance testing by the software system users. (Note: This testing may be assisted by the IT independent test group.)
e) Analyse test results and report them to the appropriate software system stakeholders.
f) Test the installation of the software into the operational environment, and test changes made to the software after it is placed into the operational environment.
g) Conduct a post-implementation analysis to evaluate the effectiveness and efficiency of the test process.

Answer 2:

The focus of testing services will be on:

a) **Software testing should reduce software development risk.** Risk is present in all software development projects, and testing is a control that reduces those risks.
b) **Testing should be performed effectively.** Testing should be performed in a manner in which the maximum benefits are achieved from the software testing efforts.
c) **Testing should uncover defects.** Ideally, at the conclusion of testing there should be no defects in the software.
d) **Testing should be performed using business logic.** Money should not be spent on testing unless it can be spent economically to reduce business risk. In other words, it does not make business sense to spend more money on testing than the losses that might occur from the business risk.
e) **Testing should occur throughout the development life cycle.** Testing is not a phase, but rather a process. It begins when development begins and ends when the software is no longer being used.
f) **Testing should test both structure and function.** Testing should test the functional requirements to ensure they are correct, and test the adequacy of the software structure to process those functional requirements in an effective and efficient manner.

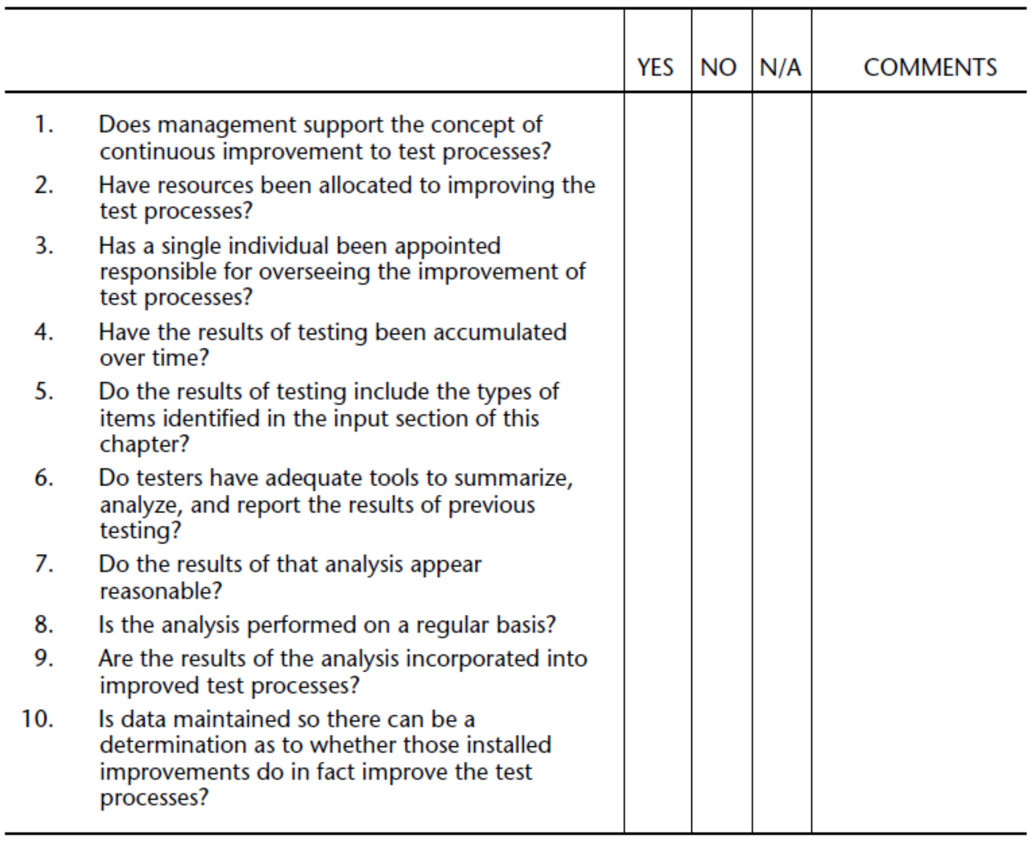
Answer 3:

The company will recruit people with the following qualities:

1. A **clear communicator**. A defect report is no good if we can't understand it.

2. **Patient.** Sometimes it takes a lot of back-and-forth to get to the root of a problem. And programmers have egos, they'll often try to push issues back to the tester.
3. **Passionate.** The best developers are the ones that really care about development and maybe even get a little excited about it sometimes. Testing isn't that much different.
4. **Creative.** Really exercising a system requires one to try non-intuitive ways of accomplishing tasks, to go *outside* the workflow that the program expects of them and do things that normal users wouldn't do. Task-oriented people who receive a set of instructions and do the exact same thing every time are *no good* for this job.
5. **Analytical**. Just finding a defect isn't enough - a tester has to be able to figure out how to reproduce it. If a report comes in as "intermittent" then there's about a 10% chance it'll get solved. Most developers won't even look at a case without a reasonably concise sequence of repro steps. Good testers have to be able to retrace their steps and narrow down the field of possibilities so as to come up with the simplest possible sequence of actions that trigger a bug.
6. **Not a programmer.** Programmers never want to do any *actual work* testing. They'll spend all their time trying to write automated tests and not do what really matters, which is to make sure the damn thing actually *works the way it is supposed to*. Although there are exceptions to every rule, most programmers simply find testing boring and will do the absolute minimum amount required.

Answer 4:



		YES	NO	N/A	COMMENTS
1.	Does management support the concept of continuous improvement to test processes?				
2.	Have resources been allocated to improving the test processes?				
3.	Has a single individual been appointed responsible for overseeing the improvement of test processes?				
4.	Have the results of testing been accumulated over time?				
5.	Do the results of testing include the types of items identified in the input section of this chapter?				
6.	Do testers have adequate tools to summarize, analyze, and report the results of previous testing?				
7.	Do the results of that analysis appear reasonable?				
8.	Is the analysis performed on a regular basis?				
9.	Are the results of the analysis incorporated into improved test processes?				
10.	Is data maintained so there can be a determination as to whether those installed improvements do in fact improve the test processes?				

Answer 5:
Write about any 5 automation tools:

a) Selenium
b) AutoIT
c) Bugzilla

d) QTP
e) WAPT
f) Win runner
g) VTest

Practical No 2

Title: Write a Test Plan

Problem Statement:
Prepare a small project and submit SRS, design, coding and test plan.

Solution:
The IEEE 829 format is to be prepared. Sample format is attached as PDF file.

Practical No 3.

Title: Black Box Testing – Equivalence Partitioning and Boundary value Analysis

Problem Statement:
The program reads an arbitrary number of temperatures (as integer numbers) within the range -60°C … +60°C and prints their mean value. Design test cases for testing the program with the black-box strategy.

Theory:
The first of the dynamic testing techniques are the specification-based testing techniques. These are also known as **'black-box'** or input/output-driven testing techniques because they view the software as a black-box with inputs and outputs, but they have no knowledge of how the system or component is structured inside the box. In essence, the tester is concentrating on what the software does, not how it does it.
Functional testing is concerned with what the system does, its features or functions. Non-functional testing is concerned with examining how well the system does something, rather than what it does. Non-functional aspects (also known as quality characteristics or quality attributes) include performance, usability, portability, maintainability, etc. Techniques to test these non-functional aspects are less procedural and less formalized than those of other categories as the actual tests are more dependent on the type of system, what it does and the resources available for the tests.
The four specification-based techniques are:

- Equivalence partitioning:
  Equivalence partitioning (EP) is a good all-round specification-based black-box technique. It can be applied at any level of testing and is often a good technique to use first. It is a common sense approach to testing, so much so that most testers practise it informally even though they may not realize it.
  The idea behind the technique is to divide (i.e. to partition) a set of test conditions into groups or sets that can be considered the same (i.e. the system should handle them equivalently), hence 'equivalence partitioning'. **Equivalence partitions** are also known as equivalence classes - the two terms mean exactly the same thing.

  The equivalence-partitioning technique then requires that we need test only one condition from each partition. This is because we are assuming that all the conditions in one partition will be treated in the same way by the software. If one condition in a partition works, we assume all of the conditions in that partition will work, and so there is little point in testing any of these others. Conversely, if one of the conditions in a partition does not work, then we assume that none of the conditions in that partition will work so again there is little point in testing any more in that partition. Of course these are simplifying assumptions that may not always be right but if we write them down, at least it gives other people the chance to challenge the assumptions we have made and hopefully help to identify better partitions.

- Boundary value analysis;
  Boundary value analysis (BVA) is based on testing at the boundaries between partitions. If you have ever done 'range checking', you were probably

using the boundary value analysis technique, even if you weren't aware of it. Note that we have both valid boundaries (in the valid partitions) and invalid boundaries (in the invalid partitions).

- Decision tables;
- State transition testing.

Solution:
The program accepts integers between -60 and +60 so we apply Equivalence partitioning and Boundary Value Analysis:

Equivalence partitions:

<-60	–60 to +60	>60
Invalid Partition	Valid Partition	Invalid Partition

Boundary Value Analysis:

-61	-60	-59
Invalid	Valid Boundary	Valid
59	60	61
Valid	Valid Boundary	Invalid

Test Cases:

Test Case ID	Input Value	Remark
1	-61	Invalid Boundary
2	-100	Invalid Partition
3	-60	Valid Boundary Value
4	0	Valid Partition
5	+61	Invalid Boundary
6	150	Invalid Partition

Conclusion:
As the program specifications are given. With the given specifications, the above test cases are generated and seem to work well.

Practical No 4.

Title: Black Box Testing – Equivalence Partitioning and Boundary value Analysis

Problem Statement:
When getting a person's weight and height as input, the program prints the person's body weight index. The weight is given in kilograms (as a real number, for instance: 82.0) and the height in meters (as a real number, for instance: 1.86). The body weight index equals weight divided by height squared: weight / (height * height). Design test cases for testing the program with the black-box strategy.

Program:
```
#include <stdio.h>
#include <conio.h>
void main() {
        float ht, wt, bmi;
        clrscr();
        ht = 0; wt = 0;
        while (ht<0.3 || ht>2.4) {
                printf("Enter valid height in meters");
                scanf("%f", &ht);
        }
        while (wt<2.0 || wt>350) {
                printf("\nEnter valid weight in kilograms");
                scanf("%f", &wt);
        }
        bmi=wt/(ht*ht);
        printf("\nBMI=%f kg/sq.m",bmi);
        getch();
}
```

Equivalence partitions (ht):

<0.3	0.3 to 2.4	>2.4
Invalid Partition	Valid Partition	Invalid Partition

Equivalence partitions (wt):

<2.0	2.0 to 350.0	>350.0
Invalid Partition	Valid Partition	Invalid Partition

Boundary Value Analysis (ht):

0.29	0.3	0.31
Invalid	Valid Boundary	Valid
2.39	2.40	2.41
Valid	Valid Boundary	Invalid

Boundary Value Analysis (wt):

1.99	2.0	2.01
Invalid	Valid Boundary	Valid
349.99	350	350.01

Valid	Valid Boundary	Invalid

Test Cases:

Test Case ID	ht (Input)	wt (Input)	Remark	Expected Output
1	0.29	*	(1)	Enter valid ht
2	*	1.99	(2)	Enter valid wt
3	0.30	2.00	(3)	22.22 kg/sq.m
4	2.39	349.99	(4)	
5	2.40	350	(5)	
6	2.41	*	(6)	Enter valid ht
7	*	350.01	(7)	Enter valid wt
8	1.86	86.00	(8)	

Conclusion:
The program was tested with the above test cases and worked fine.

Practical No 5

Title: Black Box Testing: Decision table and Cause Effect Graphing

Problem Statement:
An insurance agency has the following norms fixed to provide premium for its policy holders:

1. If age<=25 and no claim has been made, premium increase will be $50, else $25.
2. If age <=25 and number of claims made is one, premium increase will be $100, else $50.
3. If age <=25 and number of claims made is 2-4, premium increase will be $400, else $200.
4. If one or more claims are made, send warning letter. If the number of claims made is 5 or more, cancel policy.

Draw the decision table and cause effect graph for Insurance renewal.

Theory:
The techniques of equivalence partitioning and boundary value analysis are often applied to specific situations or inputs. However, if different combinations of inputs result in different actions being taken, this can be more difficult to show using equivalence partitioning and boundary value analysis, which tend to be more focused on the user interface. The other two specification-based techniques, decision tables and state transition testing are more focused on business logic or business rules.

A **decision table** is a good way to deal with combinations of things (e.g. inputs). This technique is sometimes also referred to as a 'cause-effect' table. The reason for this is that there is an associated logic diagramming technique called 'cause-effect graphing' which was sometimes used to help derive the decision table.

If you begin using decision tables to explore what the business rules are that should be tested, you may find that the analysts and developers find the tables very helpful and want to begin using them too. Do encourage this, as it will make your job easier in the future. Decision tables provide a systematic way of stating complex business rules, which is useful for developers as well as for testers. Decision tables can be used in test design whether or not they are used in specifications, as they help testers explore the effects of combinations of different inputs and other software states that must correctly implement business rules. Helping the developers do a better job can also lead to better relationships with them.

Testing combinations can be a challenge, as the number of combinations can often be huge. Testing all combinations may be impractical if not impossible. We have to be satisfied with testing just a small subset of combinations but making the choice of which combinations to test and which to leave out is not trivial. If you do not have a systematic way of selecting combinations, an arbitrary subset will be used and this may well result in an ineffective test effort.

Decision tables aid the systematic selection of effective test cases and can have the beneficial side-effect of finding problems and ambiguities in the specification. It is a

technique that works well in conjunction with equivalence partitioning. The combination of conditions explored may be combinations of equivalence partitions.
In addition to decision tables, there are other techniques that deal with testing combinations of things: pairwise testing and orthogonal arrays.

Using decision tables for test design
The first task is to identify a suitable function or subsystem that has a behaviour which reacts according to a combination of inputs or events. The behaviour of interest must not be too extensive (i.e. should not contain too many inputs) otherwise the number of combinations will become cumbersome and difficult to manage. It is better to deal with large numbers of conditions by dividing them into subsets and dealing with the subsets one at a time.
Once you have identified the aspects that need to be combined, then you put them into a table listing all the combinations of true and false for each of the aspects.

Solution:

Conditions	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**	**R10**	**R11**	**R12**	**R13**	**R14**	**R15**	**R16**
Age<=25	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
No Claim	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
1 Claim	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F
2-4 Claims	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F
>=5 Claims	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F
Causes																
Pr Inc $25	X	X	X	X	X	X	X		X	X	X		X			X
Pr Inc $50	X	X	X	X	X	X	X	T	X	X	X		X			X
Pr Inc $100	X	X	X	X	X	X	X		X	X	X	T	X			X
Pr Inc $200	X	X	X	X	X	X	X		X	X	X		X			X
Pr Inc $400	X	X	X	X	X	X	X		X	X	X		X	T		X
Send Warning	X	X	X	X	X	X	X		X	X	X	T	X	T		X
Cancel Policy	X	X	X	X	X	X	X		X	X	X		X		T	X

Conditions	**R17**	**R18**	**R19**	**R20**	**R21**	**R22**	**R23**	**R24**	**R25**	**R26**	**R27**	**R28**	**R29**	**R30**	**R31**	**R32**
Age<=25	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
No Claim	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
1 Claim	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F
2-4 Claims	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F
>=5 Claims	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F
Causes																
Pr Inc $25	X	X	X	X	X	X	X	T	X	X	X		X			X
Pr Inc $50	X	X	X	X	X	X	X		X	X	X	T	X			X
Pr Inc $100	X	X	X	X	X	X	X		X	X	X		X			X
Pr Inc $200	X	X	X	X	X	X	X		X	X	X		X	T		X
Pr Inc $400	X	X	X	X	X	X	X		X	X	X		X			X
Send Warning	X	X	X	X	X	X	X		X	X	X	T	X	T		X
Cancel Policy	X	X	X	X	X	X	X		X	X	X		X		T	X

Condensed decision table:

No. of Claims	Insured Age	Premium Increase $	Send Warning	Cancel
0	≤25	50	No	No
0	>25	25	No	No
1	≤25	100	Yes	No
1	>25	50	Yes	No
2-4	≤25	400	Yes	No
2-4	>25	200	Yes	No
5 or more	*	0	No	Yes

Cause Effect Graph:

Cause-Effect Graphing-Black Box Software Testing Technique:

This is basically a hardware testing technique adapted to software testing. It considers only the desired external behaviour of a system. This is a testing technique that aids in selecting test cases that logically relate Causes (inputs) to Effects (outputs) to produce test cases.

A "Cause" represents a distinct input condition that brings about an internal change in the system. An "Effect" represents an output condition, a system transformation or a state resulting from a combination of causes.

According to Myer Cause & Effect Graphing is done through the following steps:

Step – 1: For a module, identify the input conditions (causes) and actions (effect).

Step – 2: Develop a cause-effect graph.

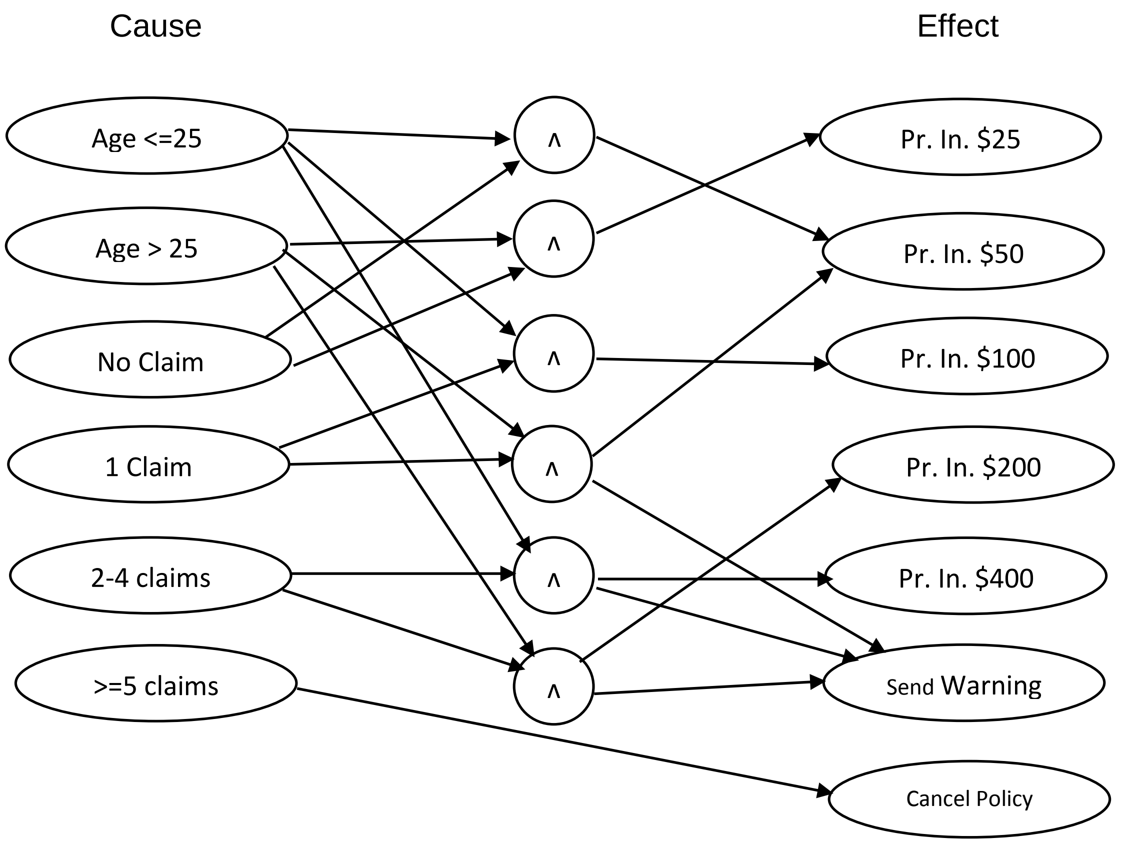
Step – 3: Transform cause-effect graph into a decision table.

Step – 4: Convert decision table rules to test cases. Each column of the decision table represents a test case.

Guidelines for cause-effect graphing:

1) If the variables refer to physical quantities, domain testing and equivalence class testing are indicated.
2) If the variables are independent, domain testing and equivalence class testing are indicated.
3) If the variables are dependent, decision table testing is indicated.

4) If the single-fault assumption is warranted, boundary value analysis (BVA) and robustness testing are indicated.
5) If the multiple-fault assumption is warranted, worst-case testing, robust worst-case testing and decision table testing are identical.
6) If the program contains significant exception handling, robustness testing and decision table testing are indicated.
7) If the variables refer to logical quantities, equivalence class testing and decision table testing are indicated.



Conclusion:
The decision table and cause effect graph for the given situation are drawn.

Practical No 6.

Title: Branch – Decision – Condition Coverage

Problem Statement:
For the following liability procedure, design test cases using branch, condition, decision and multiple decision coverage.

```
Procedure Liability(Age,Gender, Married,Premium)
Begin
        Premium:=500;
        If(Age<25) and (Gender=Male) and (not Married) Then
                Premium=Premium+1500;
        Else (if (Married or Gender=Female)) Then
                Premium=Premium-200;
        if (Age>45) and (Age<65) Then
                Premium=Premium-100;
End;
```

Theory:
Statement Coverage:
Statement coverage is achieved when the condition becomes either true or false. The objective of statement coverage is that all the statements in the program should be executed at least once. The statement coverage is necessary but not sufficient.

Branch or decision coverage:
Statement coverage does not address all the outcome of the decisions. Branches like if … else, while …, for, do … while are to be evaluated for both true and false.
Each branch direction must be traversed at least once.

Condition/Decision Coverage:
Branch coverage considers entire Boolean expression as one and tests for true and false. It ignores branches within Boolean expressions occurring due to relational and logical operators. All conditions should be executed at least once for true and false. Conditions using relational and logical operators should be checked for all possible outcomes. Condition coverage checks for true and false outcome of each Boolean sub expressions.

Multiple condition coverage:
Multiple condition coverage checks whether every possible combination of Boolean sub expression occurs at least once. Test cases required for full multiple condition coverage can be arrived at by using truth table of conditions.
A large number of test cases may be required for full multiple condition coverage.

Solution:
Branch or decision coverage:

Decision Coverage	Age	Gender	Married	Test Case
IF-1	<25	Male	False	(1) 23, M F
IF-1	<25	Female	False	(2) 23, F F
IF-2	*	Female	*	(2)
IF-2	>=25	Male	False	(3) 50 M F
IF-3	<=45	Female	*	(2)
IF-3	>45, <65	*	*	(3)

Condition Coverage:

Condition Coverage	Age	Gender	Married	Test Case
IF-1	<25	Female	False	(1) 23, F F
IF-1	>=25	Male	True	(2) 30 M T
IF-2	*	Male	True	(2)
IF-3	<=45	*	*	(1)
IF-3	>45	*	*	(3) 70 F F
IF-3	<65	*	*	(2)
IF-3	>=65	*	*	(3)

Decision/Condition Coverage

Decision/Condition Coverage	Age	Gender	Married	Test Case
IF-1 (Decision)	<25	Male	False	(1) 23 M F
IF-1 (Decision)	<25	Female	False	(2) 23 F F
IF-1 (Condition)	<25	Female	False	(2)
IF-1 (Condition)	>=25	Male	True	(3) 70 M T
IF-2 (Decision)	*	Female	*	(2)
IF-2 (Decision)	>=25	Male	False	(4) 50 M F
IF-2 (Condition)	*	Male	True	(3)
IF-2 (Condition)	*	Female	False	(2)
IF-3 (Decision)	<=45	*	*	(2)
IF-3 (Decision)	>45, <65	*	*	(4)
IF-3 (Condition)	<=45	*	*	(2)
IF-3 (Condition)	>45	*	*	(4)
IF-3 (Condition)	<65	*	*	(4)
IF-3 (Condition)	>=65	*	*	(3)

Multiple Condition Coverage:

Decision/Condition Coverage	Age	Gender	Married	Test Case
IF-1	<25	Male	True	(1)
IF-1	<25	Male	False	(2)
IF-1	<25	Female	True	(3)
IF-1	<25	Female	False	(4)
IF-1	>=25	Male	True	(5)
IF-1	>=25	Male	False	(6)
IF-1	>=25	Female	True	(7)
IF-1	>=25	Female	False	(8)
IF-2	*	Male	True	(5)
IF-2	*	Male	False	(6)
IF-2	*	Female	True	(7)
IF-2	*	Female	False	(8)
IF-3	<=45, >=65	*	*	Impossible
IF-3	<=45, <65	*	*	(8)
IF-3	>45,>65	*	*	(6)
IF-3	>45,<65	*	*	(7)

Conclusion:
All the types of test cases are designed for testing.

Practical No 7

Title: State Transition Testing

Problem Statement:
Specifications:
The software responds to input requests to change the display mode for a time display device.
The display mode can be set to one of the four values:
Two corresponding to displaying either time or date.
The other two when altering either time or date.
Four possible input requests:
Change mode (CM)
Reset (R)
Time Set (TS)
Date Set (DS)

Change Mode (CM):
Activation of this shall cause the display mode to move between “Display Time (T)” and “Display Date (D)”
Reset (R):
If display mode is set to T or D, then a “reset” shall cause the display mode to be set to “Alter time (AT)” or “Alter Date (AD)” modes.
Time Set (TS):
Activation of this shall cause the display mode to return to T from AT.
Date Set (DS):
Activation of this shall cause the display mode to return to D from AD.
Draw the state transition diagram, possible transitions table, State table and write the test cases.

Theory:
State transition testing is used where some aspect of the system can be described in what is called a 'finite state machine'. This simply means that the system can be in a (finite) number of different states, and the transitions from one state to another are determined by the rules of the 'machine'. This is the model on which the system and the tests are based. Any system where you get a different output for the same input, depending on what has happened before, is a finite state system. A finite state system is often shown as a **state diagram.**
A state transition model has four basic parts:

- the states that the software may occupy;
- the transitions from one state to another;
- the events that cause a transition;
- the actions that result from a transition.

In any given state, one event can cause only one action, but that the same event - from a different state - may cause a different action and a different end state.
Test conditions can be derived from the state graph in various ways. Each state can be noted as a test condition, as can each transition. We can also consider transition pairs and triples and so on. Coverage of all individual transitions is also known as 0-switch coverage, coverage of transition pairs is l-switch coverage, coverage of transition triples is 2-switch coverage, etc. Deriving test cases from the state transition model is a black-box approach. Measuring how much you have tested (covered) is

getting close to a white-box perspective. However, state transition testing is regarded as a black-box technique.

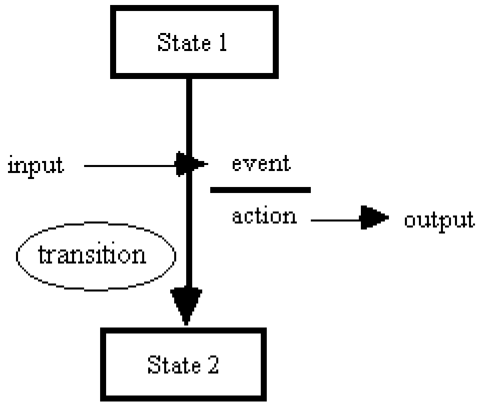
One of the advantages of the state transition technique is that the model can be as detailed or as abstract as you need it to be. Where a part of the system is more important (that is, requires more testing) a greater depth of detail can be modelled. Where the system is less important (requires less testing), the model can use a single state to signify what would otherwise be a series of different states.

Testing for invalid transitions

Deriving tests only from a state graph (also known as a state chart) is very good for seeing the valid transitions, but we may not easily see the negative tests, where we try to generate invalid transitions. In order to see the total number of combinations of states and transitions, both valid and invalid, a **state table** is useful.

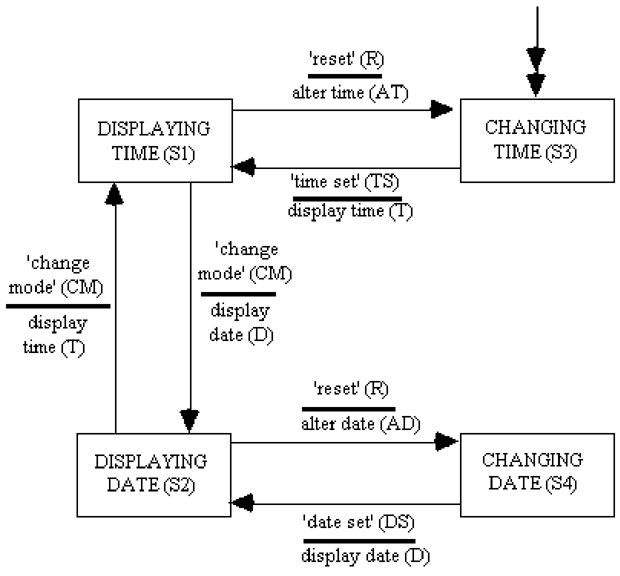
The state table lists all the states down one side of the table and all the events that cause transitions along the top (or vice versa). Each cell then represents a state-event pair. The content of each cell indicates which state the system will move to, when the corresponding event occurs while in the associated state. This will include possible erroneous events - events that are not expected to happen in certain states. These are negative test conditions.

State Transition Diagram:



Solution:

State Transition Diagram:



Test cases are initially derived from the state transition diagram to exercise each of the possible transitions (using the abbreviated STD labels):

Test Case	1	2	3	4	5	6
Start State	S1	S1	S3	S2	S2	S4
Input	CM	R	TS	CM	R	DS
Expected Output	D	AT	T	T	AD	D
Finish State	S2	S3	S1	S1	S4	S2

This indicates that for test case 1 the starting state is DISPLAYING TIME (S1), the input is 'change mode' (CM), the expected output is 'display date' (D), and the finish state is DISPLAYING DATE (S2).

This set of six test cases exercises each of the possible transitions and so achieves 0-switch coverage [Chow]. Tests written to achieve this level of coverage are limited in their ability to detect some types of faults because although they will detect the most obvious incorrect transitions and outputs, they will not detect more subtle faults that are only detectable through exercising sequences of transitions.

Tests written to achieve the next level of coverage, 1-switch, exercise all the possible sequential pairs of transitions, of which there are ten in the *manage_display_changes* component:

Test Case	1	2	3	4	5	6	7	8	9	10
Start State	S1	S1	S1	S3	S3	S2	S2	S2	S4	S4
Input	CM	CM	R	TS	TS	CM	CM	R	DS	DS
Exp.Output	D	D	AT	T	T	T	T	AD	D	D
Next State	S2	S2	S3	S1	S1	S1	S1	S4	S2	S2
Input	CM	R	TS	CM	R	CM	R	DS	CM	R
Exp.Output	T	AD	T	D	AT	D	AT	D	T	AD
FinishState	S1	S4	S1	S2	S3	S2	S3	S2	S1	S4

This indicates that test case 1 comprises two transitions. For the first transition the starting state is DISPLAYING TIME (S1), the initial input is 'change mode' (CM), the intermediate expected output is display date (D), and the next state is DISPLAYING DATE (S2). For the second transition, the second input is 'change mode' (CM), the final expected output is display time (T), and the finish state is DISPLAYING TIME (S1).

Note that intermediate states, and the inputs and outputs for each transition, are explicitly defined.

Longer sequences of transitions can be tested to achieve higher and higher levels of switch coverage, dependent on the level of test thoroughness required.

A limitation of the test cases derived to achieve switch coverage is that they are designed to exercise only the valid transitions in the component. A more thorough test of the component will *also* attempt to cause invalid transitions to occur. The STD only explicitly shows the valid transitions (all transitions not shown are considered invalid). A state model that explicitly shows both valid and invalid transitions is the state table. The notation used for state tables is briefly described below:

	Input 1	Input 2	etc.
Start State 1	Entry A	Entry B	etc.
Start State 2	Entry C	Entry D	etc.
etc.	etc.	etc.	etc.

where Entry X = Finish State / Output for the given start state and input.

The state table for the *manage_display_changes* component is shown below:

	CM	R	TS	DS
S1	S2/D	S3/AT	S1/N	S1/N
S2	S1/T	S4/AD	S2/N	S2/N
S3	S3/N	S3/N	S1/T	S3/N
S4	S4/N	S4/N	S4/N	S2/D

Any entry where the state remains the same and the output is shown as null (N) represents a null transition, where any *actual* transition that can be induced will represent a failure. It is the testing of these null transitions that is ignored by test sets designed just to achieve switch coverage. Thus a more complete test set will test both possible transitions and null transitions, which means testing the response of the component to all possible inputs in all possible states. The state table provides an ideal means of directly deriving this set of test cases.

There are 16 entries in the table above representing each of the four *possible* inputs that can occur in each of the four *possible* states, making 16 test cases which can be read from the state table as shown below:

	CM	R	TS	DS
S1	S2/D (Test Case 1)	S3/AT (Test Case 2)	S1/N (Test Case 3)	S1/N (Test Case 4)
S2	S1/T (Test Case 5)	S4/AD (Test Case 6)	S2/N (Test Case 7)	S2/N (Test Case 8)
S3	S3/N (Test Case 9)	S3/N (Test Case 10)	S1/T (Test Case 11)	S3/N (Test Case 12)
S4	S4/N (Test Case 13)	S4/N (Test Case 14)	S4/N (Test Case 15)	S2/D (Test Case 16)

which corresponds to:

Test Case	1	2	3	4	5	....	. .	....	12	13	14	15	16
Start State	S1	S1	S1	S1	S2	....	. .	....	S3	S4	S4	S4	S4
Input	CM	R	TS	DS	CM	....	. .	....	DS	CM	R	TS	DS
Exp.Output	D	AT	N	N	T	....	. .	....	N	N	N	N	D
Finish State	S2	S3	S1	S1	S1	....	. .	....	S3	S4	S4	S4	S2

If the above test cases are compared with those produced to achieve 0-switch coverage then it can be seen that by also testing the null transitions an extra 10 test cases are created (3,4,7,8,9,10,12,13,14 and 15).

Conclusion:
The state transition table is drawn and the test cases are developed.

Practical No 8

Title: Data Flow Testing

Problem Statement:

```
 0    void quad_eqn (float A,B,C, Boolean Is_Complex)
 1    {    float Discrim = B*B-4*A*C
 2         float R1,R2;
 3         {
 4            If Discrim<0.0
 5                Is_Complex=true;
 6            Else
 7                Is_Complex=false
 8            Endif;
 9            If not Is_Complex
 10               R1=(-B+Sqrt(Discrim))/(2.0*A);
 11               R2=(-B-Sqrt(Discrim))/(2.0*A);
 12           Endif;
 13          End quad_eqn;}
 14   }
```

For the above function, draw the following tables and write the test cases:

a. Occurrence of variables and their categories.
b. du pairs and their type
c. All c-uses
d. All p-uses

Theory:

Data flow testing is a structural testing technique that

- Aims to execute sub-paths from points where each variable is defined to points where it is referenced
- These sub-paths are called definition-use pairs or du-pairs (du-paths, du-chains)
- Data flow testing is centred on variables (data)

Most failures involve execution of an incorrect definition

- Incorrect assignment or input statement
- Incorrect path is taken, which leads to incorrect definition (predicate is faulty)
- Definition is missing (i.e. null definition)

Data flow testing follows the sequences of events related to a given data item with the objective to detect incorrect sequences. It explores the effect of using the value produced by each and every computation.

Terminology:

- **Variable definition**
  Occurrences of a variable where a variable is given a new value (assignment, input by the user, input from a file, etc.)
  Variable DECLARATION is NOT its definition !!!

- **Variable uses**
  Occurrences of a variable where a variable is not given a new value (variable DECLARATION is NOT its use)

- **p-uses (predicate uses)**
  Occur in the predicate portion of a decision statement such as if-then-else, while-do etc.

- **c-uses (computation uses)**
  All others, including variable occurrences in the right hand side of an assignment statement, or an output statement

- **du-path**
  A sub-path from a variable definition to its use

Data Flow testing

- Checks the correctness of the du-paths in a Control
  - Flow Graph of a program
  -
- Test case definitions based on four groups of coverage
  - All definitions
  - All c-uses
  - All p-uses
  - All du-paths

Data Flow testing: key steps

Given a code (program or pseudo-code)

1. Number the lines
2. List the variables
3. List occurrences & assign a category to each variable
4. Identify du-pairs and their use (p- or c-)
5. Define test cases, depending on the required coverage

Solution:
Step 1: Number the lines:

```
0    void quad_eqn (float A,B,C, Boolean Is_Complex)
1    {    float Discrim = B*B-4*A*C
2         float R1,R2;
3         {
4            If Discrim<0.0
5                Is_Complex=true;
6            Else
7                Is_Complex=false
8            Endif;
9            If not Is_Complex
10               R1=(-B+Sqrt(Discrim))/(2.0*A);
11               R2=(-B-Sqrt(Discrim))/(2.0*A);
12           Endif;
13         End quad_eqn;}
14   }
```

Step 2: List the variables:

- A, B, C
- DISCRIM
- Is_Complex
- R1, R2

Step 3: List occurrences and assign a category to each variable

Line No	Category		
	Definition	**c-use**	**p-use**
0	A,B,C		
1	DISCRIM	A,B,C	
2			
3			
4			DISCRIM
5	Is_Complex		
6			
7	Is_Complex		
8			
9			Is_Complex
10	R1	A,B,DISCRIM	
11	R2	A,B,DISCRIM	
12			
13			
14			

Step 4: Identify du-pairs and their use (p- or c-)

Definition – use pair Start line → end line	Variables	
	c-use	**p-use**
0 → 1	A,B,C	
0 → 10	A,B	
0 → 11	A,B	
1 → 4		DISCRIM
1 → 10	DISCRIM	
1 → 11	DISCRIM	
5 → 9		Is_Complex
7 → 9		Is_Complex
10 → 14	R1	
11 → 14	R2	

Step 5: Define Test Cases:

D c-use pairs	Variables	Subpaths	Test Case	Inputs			Is_Complex	R1	R2
				A	B	C			
0→1	A,B,C	0-1	1	1	1	1	T	.	.
0→10	A,B	0-1-2-3-4-6-7-8-9-10	2	1	2	1	F	-1	-1
0-11	A,B	0-1-2-3-4-6-7-8-9-10-11	3	1	2	1	F	-1	-1
1-10	DISCRIM	1-2-3-4-6-7-8-9-10	4	1	2	1	F	-1	-1
1-11	DISCRIM	1-2-3-4-6-7-8-9-10-11	5	1	2	1	F	-1	-1

5-14	Is_Complex	5-8-9-12-13-14	6	1	1	1	T	.	.
7-14	Is_Complex	7-8-9-10-11-12-13-14	7	1	2	1	F	-1	-1
10-14	R1	10-11-12-13-14	8	1	2	1	F	-1	-1
11-14	R2	11-12-13-14	9	1	2	1	F	-1	-1

d p-use pairs	**Variables**	**Subpaths**	**Test Case**	**Inputs**			**Is_Complex**	**R1**	**R2**
				A	B	C			
1-(4-5)	Discrim	1-2-3-4-5	10	1	1	1	T	.	.
1-(4-6)	Discrim	1-2-3-4-6	11	1	2	1	F	-1	-1
5-(9-12)	Is_Complex	5-8-9-12	12	1	1	1	T	.	.
7-(9-10)	Is_Complex	7-8-9-10	13	1	2	1	F	-1	-1

Conclusion:
13 test cases were designed for data flow testing.

Practical No 9

Title: Structured Testing – Loop Coverage, Call coverage and Path Coverage.

Problem Statement:
For the following program, draw the path coverage diagram, determine cyclomatic complexity write the basis paths to be tested and the test cases.

```
Euclid(int m, int n)
     int r;
     if (n > m){
          r = m;
          m = n;
          n = r;
     }
     r = m % n;
     while( r! = 0){
          m = n;
          n = r;
          r = m % n;
     }
     return n;
}
```

Theory:
Control flow graph:
Control flow graphs describe the logic structure of software modules. A module corresponds to a single function or subroutine in typical languages, has a single entry and exit point, and is able to be used as a design component via a call/return mechanism. This document uses C as the language for examples, and in C a module is a function. Each flow graph consists of nodes and edges. The nodes represent computational statements or expressions, and the edges represent transfer of control between nodes.
Each possible execution path of a software module has a corresponding path from the entry to the exit node of the module's control flow graph. This correspondence is the foundation for the structured testing methodology.

Definition of cyclomatic complexity, v(G)
Cyclomatic complexity is defined for each module to be e - n + 2, where e and n are the number of edges and nodes in the control flow graph, respectively. Cyclomatic complexity is also known as v(G), where v refers to the cyclomatic number in graph theory and G indicates that the complexity is a function of the graph. The word "cyclomatic" comes from the number of fundamental (or basic) cycles in connected, undirected graphs. More importantly, it also gives the number of independent paths through strongly connected directed graphs. A strongly connected graph is one in which each node can be reached from any other node by following directed edges in the graph. The cyclomatic number in graph theory is defined as e - n + 1. Program control flow graphs are not strongly connected, but they become strongly connected when a "virtual edge" is added connecting the exit node to the entry node. The cyclomatic complexity definition for program control flow graphs is derived from the cyclomatic number formula by simply adding one to represent the contribution of the virtual edge. This definition makes the cyclomatic complexity equal the number of

independent paths through the standard control flow graph model, and avoids explicit mention of the virtual edge.

Characterization of v(G) using a basis set of control flow paths

Cyclomatic complexity can be characterized as the number of elements of a basis set of control flow paths through the module. Some familiarity with linear algebra is required to follow the details, but the point is that cyclomatic complexity is precisely the minimum number of paths that can, in (linear) combination, generate all possible paths through the module. To see this, consider the following mathematical model, which gives a vector space corresponding to each flow graph.

Each path has an associated row vector, with the elements corresponding to edges in the flow graph. The value of each element is the number of times the edge is traversed by the path.

Considering a set of several paths gives a matrix in which the columns correspond to edges and the rows correspond to paths. From linear algebra, it is known that each matrix has a unique rank (number of linearly independent rows) that is less than or equal to the number of columns. This means that no matter how many of the (potentially infinite) number of possible paths are added to the matrix, the rank can never exceed the number of edges in the graph. In fact, the maximum value of this rank is exactly the cyclomatic complexity of the graph. A minimal set of vectors (paths) with maximum rank is known as a basis, and a basis can also be described as a linearly independent set of vectors that generate all vectors in the space by linear combination. This means that the cyclomatic complexity is the number of paths in any independent set of paths that generate all possible paths by linear combination.

Given any set of paths, it is possible to determine the rank by doing Gaussian Elimination on the associated matrix. The rank is the number of non-zero rows once elimination is complete.

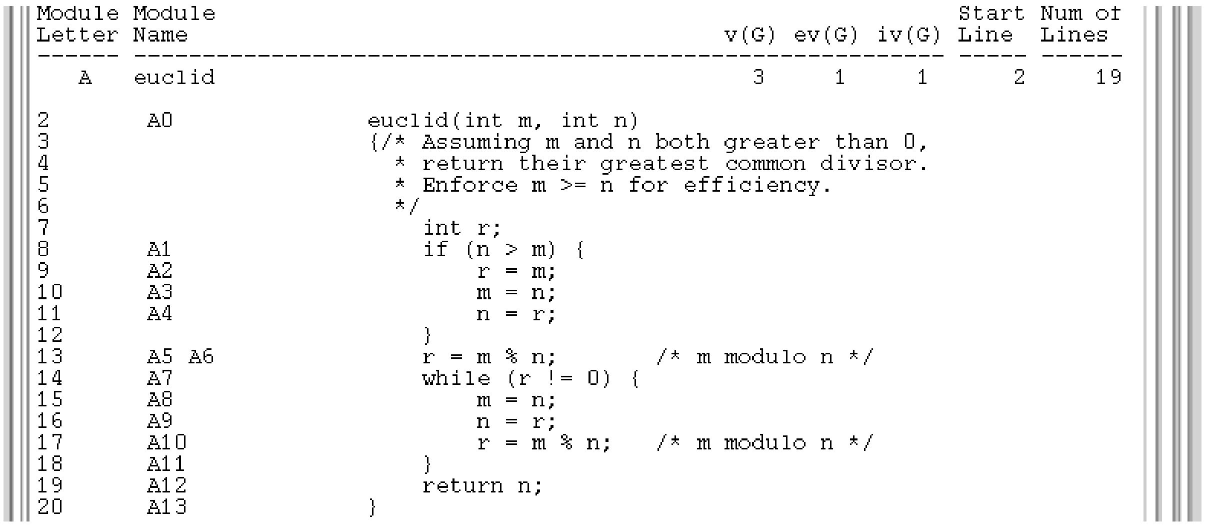
If no rows are driven to zero during elimination, the original paths are linearly independent. If the rank equals the cyclomatic complexity, the original set of paths generate all paths by linear combination. If both conditions hold, the original set of paths are a basis for the flow graph. There are a few important points to note about the linear algebra of control flow paths. First, although every path has a corresponding vector, not every vector has a corresponding path.

This is obvious, for example, for a vector that has a zero value for all elements corresponding to edges out of the module entry node but has a nonzero value for any other element cannot correspond to any path. Slightly less obvious, but also true, is that linear combinations of vectors that correspond to actual paths may be vectors that do not correspond to actual paths.

This follows from the non-obvious fact (shown in section 6) that it is always possible to construct a basis consisting of vectors that correspond to actual paths, so any vector can be generated from vectors corresponding to actual paths. This means that one can not just find a basis set of vectors by algebraic methods and expect them to correspond to paths—one must use a path-oriented technique such as that of section 6 to get a basis set of paths. Finally, there are a potentially infinite number of basis sets of paths for a given module. Each basis set has the same number of paths in it (the cyclomatic complexity), but there is no limit to the number of different sets of basis paths. For example, it is possible to start with any path and construct a basis that contains it.

The details of the theory behind cyclomatic complexity are too mathematically complicated to be used directly during software development. However, the good news is that this mathematical insight yields an effective operational testing method in which a concrete basis set of paths is tested: structured testing.

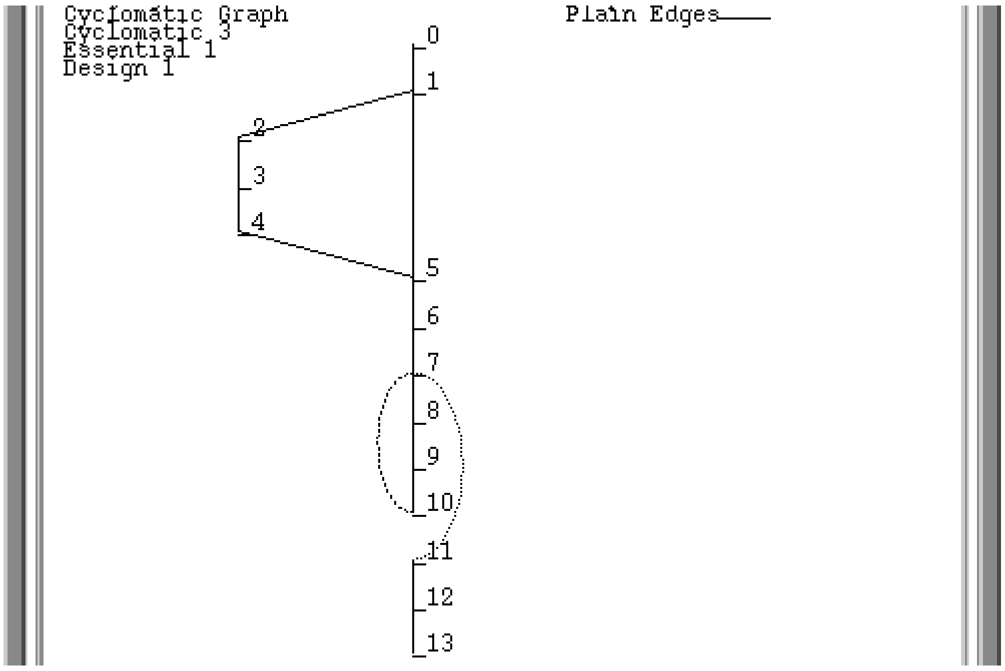
Solution:



```
Module Module                                                    Start Num of
Letter Name                                    v(G) ev(G) iv(G) Line  Lines
------ -------------------------------------------------------- ----- ------
   A   euclid                                     3     1     1      2     19

2       A0              euclid(int m, int n)
3                       {/* Assuming m and n both greater than 0,
4                         * return their greatest common divisor.
5                         * Enforce m >= n for efficiency.
6                         */
7                           int r;
8       A1                  if (n > m) {
9       A2                      r = m;
10      A3                      m = n;
11      A4                      n = r;
12                          }
13      A5 A6               r = m % n;        /* m modulo n */
14      A7                  while (r != 0) {
15      A8                      m = n;
16      A9                      n = r;
17      A10                     r = m % n;   /* m modulo n */
18      A11                 }
19      A12                 return n;
20      A13             }
```

Flow graph:



Cyclomatic Complexity:

V(G) = E – N + 2
= 15 – 14 + 2
= 3

Basis Test Path:

Basis Test Paths: 3 Paths

Test Path B1: 0 1 5 6 7 11 12 13

8(1): n>m ==> FALSE

14(7): r!=0 ==> FALSE

Test Path B2: 0 1 2 3 4 5 6 7 11 12 13

8(1): n>m ==> TRUE

14(7): r!=0 ==> FALSE

Test Path B3: 0 1 5 6 7 8 9 10 7 11 12 13

8(1): n>m ==> FALSE

14(7): r!=0 ==> TRUE

14(7): r!=0 ==> FALSE

Conclusion:
The control flow graph is drawn and the cyclomatic complexity is determined to be 3. Three basis paths are to be tested.

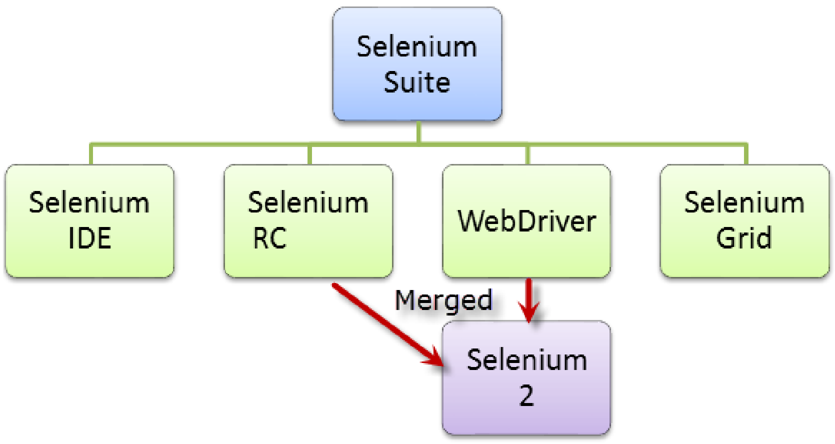
Practical No 10

Title: Test Automation using Selenium IDE

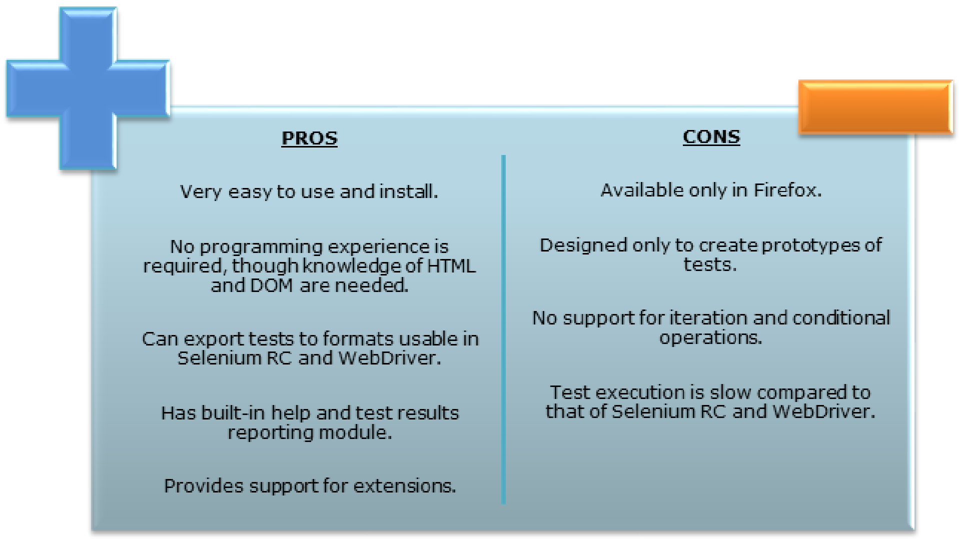
Problem Statement:
Automate web test using Selenium IDE and Firebug.

Theory:

- **Selenium is a free (open source) automated testing suite for web applications across different browsers and platforms.**
- Selenium is not just a single tool but a suite of software, each catering to different testing needs of an organization. **It has four components.**
    - Selenium Integrated Development Environment (IDE)
    - Selenium Remote Control (RC)
    - WebDriver
    - Selenium Grid

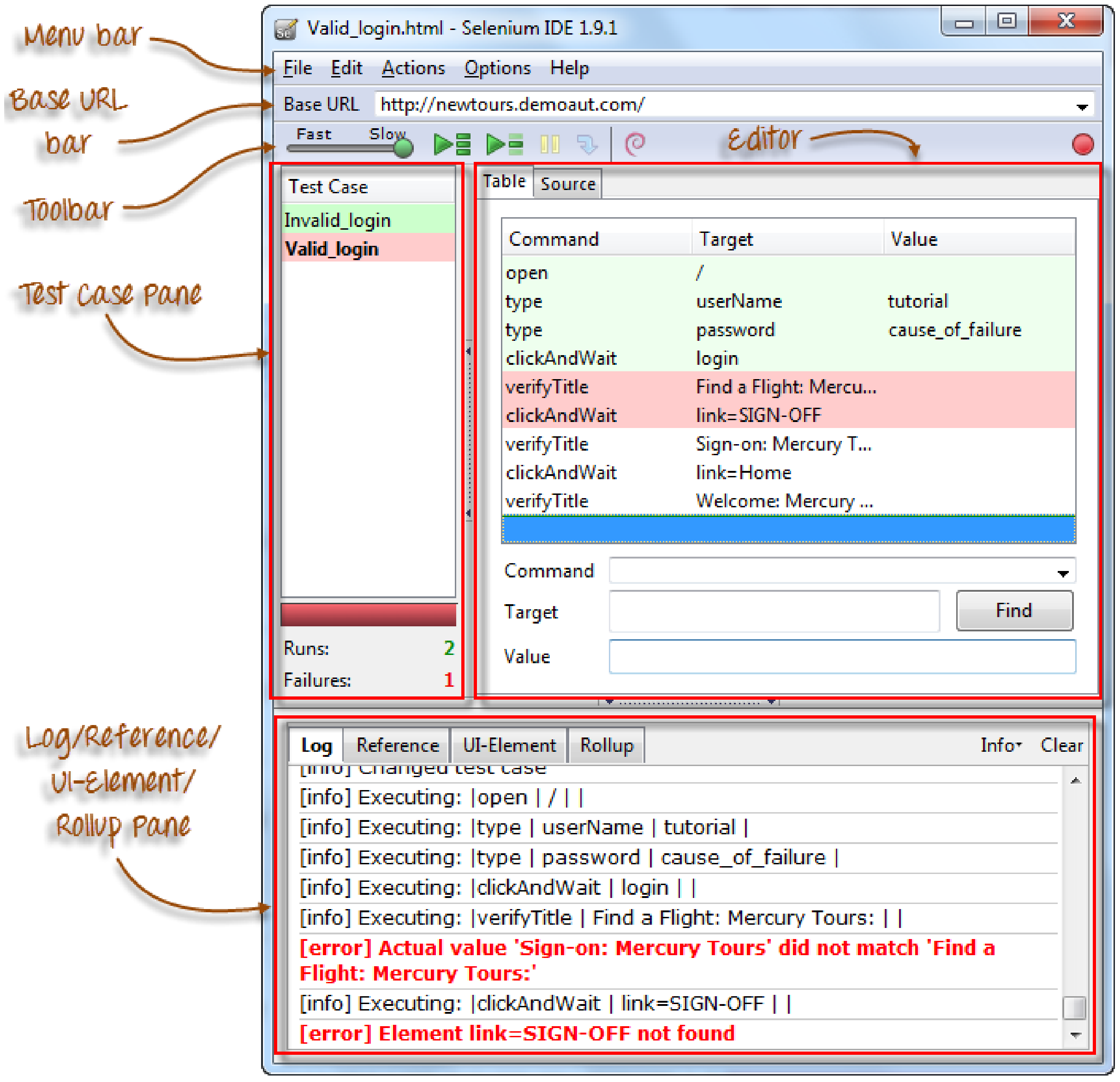


- Selenium Integrated Development Environment (IDE) is the **simplest framework** in the Selenium suite and is **the easiest one to learn**. It is a **Firefox plugin** that you can install as easily as you can with other plugins. However, because of its simplicity, Selenium IDE should only be used as a **prototyping tool**.
- If you want to create more advanced test cases, you will need to use either Selenium RC or WebDriver.



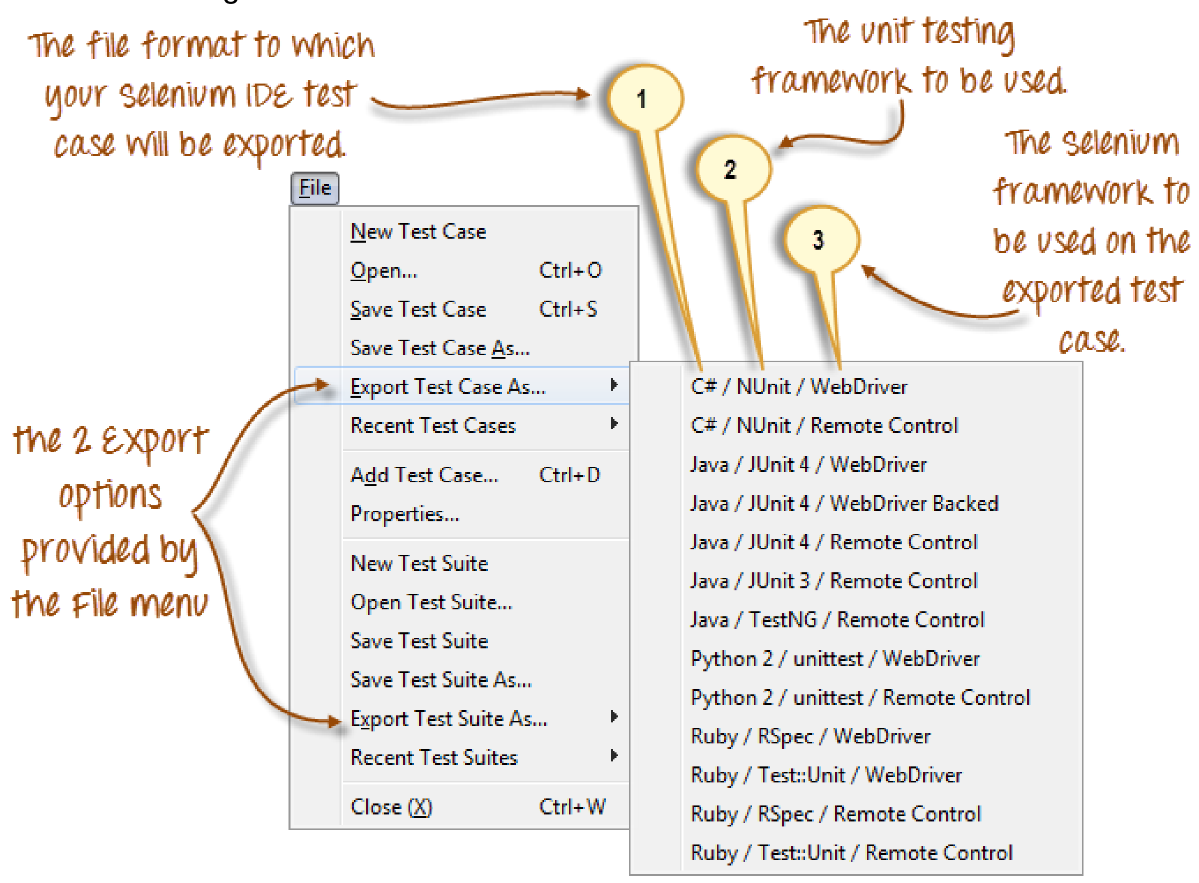
PROS	CONS
Very easy to use and install.	Available only in Firefox.
No programming experience is required, though knowledge of HTML and DOM are needed.	Designed only to create prototypes of tests.
Can export tests to formats usable in Selenium RC and WebDriver.	No support for iteration and conditional operations.
Has built-in help and test results reporting module.	Test execution is slow compared to that of Selenium RC and WebDriver.
Provides support for extensions.	

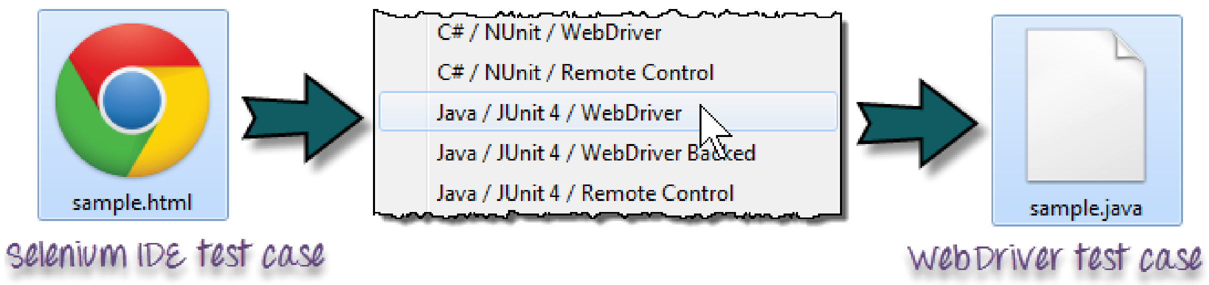
Selenium IDE:

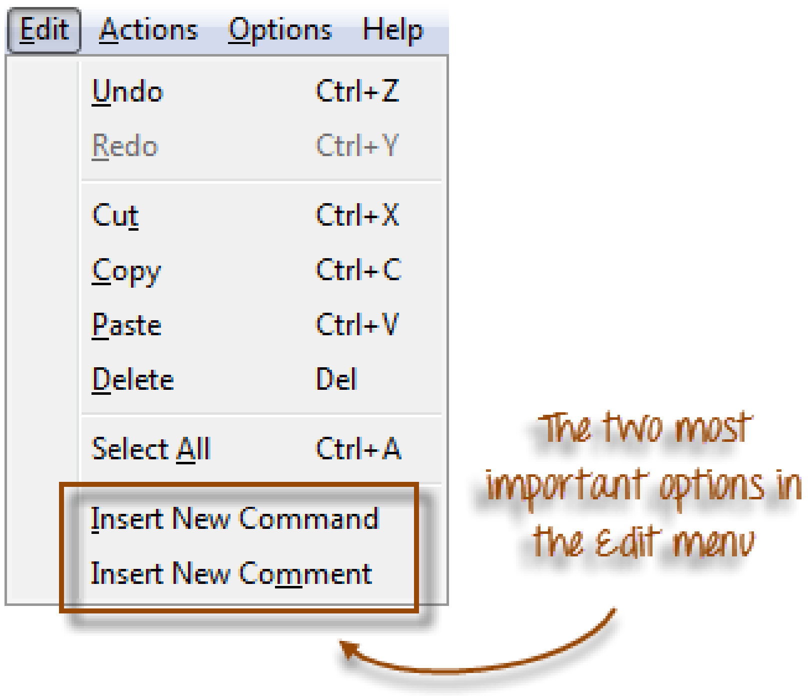

Menu bar
Base URL bar
Toolbar
Test Case Pane
Editor
Log/Reference/UI-Element/Rollup Pane
Valid_login.html - Selenium IDE 1.9.1
File Edit Actions Options Help
Base URL http://newtours.demoaut.com/
Fast Slow
Test Case
Invalid_login
Valid_login
Runs: 2
Failures: 1
Table Source
Command
Target
Value
Find
Log Reference UI-Element Rollup
Info Clear

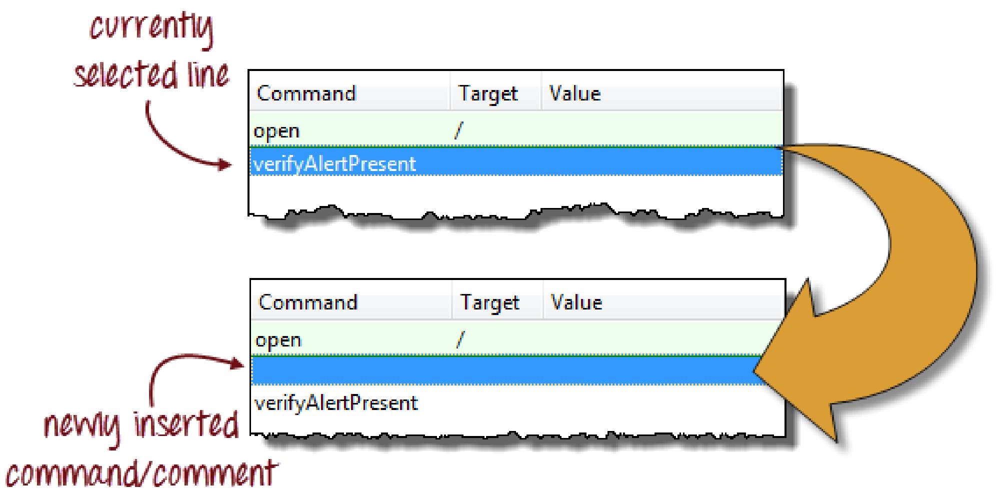
Command	Target	Value
open	/	
type	userName	tutorial
type	password	cause_of_failure
clickAndWait	login	
verifyTitle	Find a Flight: Mercu...	
clickAndWait	link=SIGN-OFF	
verifyTitle	Sign-on: Mercury T...	
clickAndWait	link=Home	
verifyTitle	Welcome: Mercury ...	

```
[info] Changed test case
[info] Executing: |open | / | |
[info] Executing: |type | userName | tutorial |
[info] Executing: |type | password | cause_of_failure |
[info] Executing: |clickAndWait | login | |
[info] Executing: |verifyTitle | Find a Flight: Mercury Tours: | |
[error] Actual value 'Sign-on: Mercury Tours' did not match 'Find a
Flight: Mercury Tours:'
[info] Executing: |clickAndWait | link=SIGN-OFF | |
[error] Element link=SIGN-OFF not found
```


The file format to which your selenium IDE test case will be exported.
The unit testing framework to be used.
The selenium framework to be used on the exported test case.
the 2 export options provided by the File menu
1
2
3
File
New Test Case
Open... Ctrl+O
Save Test Case Ctrl+S
Save Test Case As...
Export Test Case As...
Recent Test Cases
Add Test Case... Ctrl+D
Properties...
New Test Suite
Open Test Suite...
Save Test Suite
Save Test Suite As...
Export Test Suite As...
Recent Test Suites
Close (X) Ctrl+W
C# / NUnit / WebDriver
C# / NUnit / Remote Control
Java / JUnit 4 / WebDriver
Java / JUnit 4 / WebDriver Backed
Java / JUnit 4 / Remote Control
Java / JUnit 3 / Remote Control
Java / TestNG / Remote Control
Python 2 / unittest / WebDriver
Python 2 / unittest / Remote Control
Ruby / RSpec / WebDriver
Ruby / Test::Unit / WebDriver
Ruby / RSpec / Remote Control
Ruby / Test::Unit / Remote Control

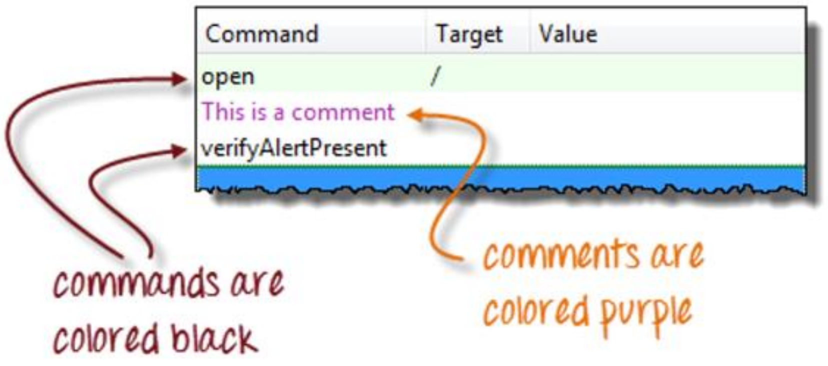

sample.html
C# / NUnit / WebDriver
C# / NUnit / Remote Control
Java / JUnit 4 / WebDriver
Java / JUnit 4 / WebDriver Backed
Java / JUnit 4 / Remote Control
sample.java
selenium IDE test case
WebDriver test case


Edit Actions Options Help
Undo Ctrl+Z
Redo Ctrl+Y
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Delete Del
Select All Ctrl+A
Insert New Command
Insert New Comment
The two most important options in the Edit menu

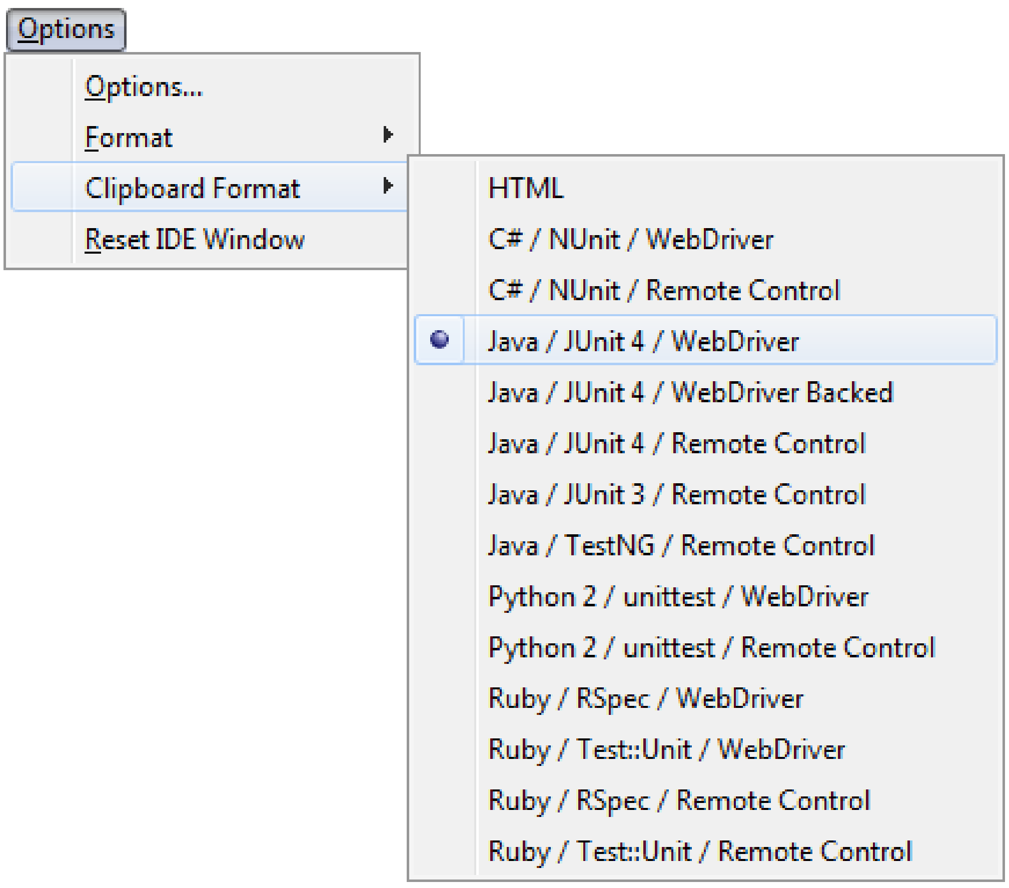


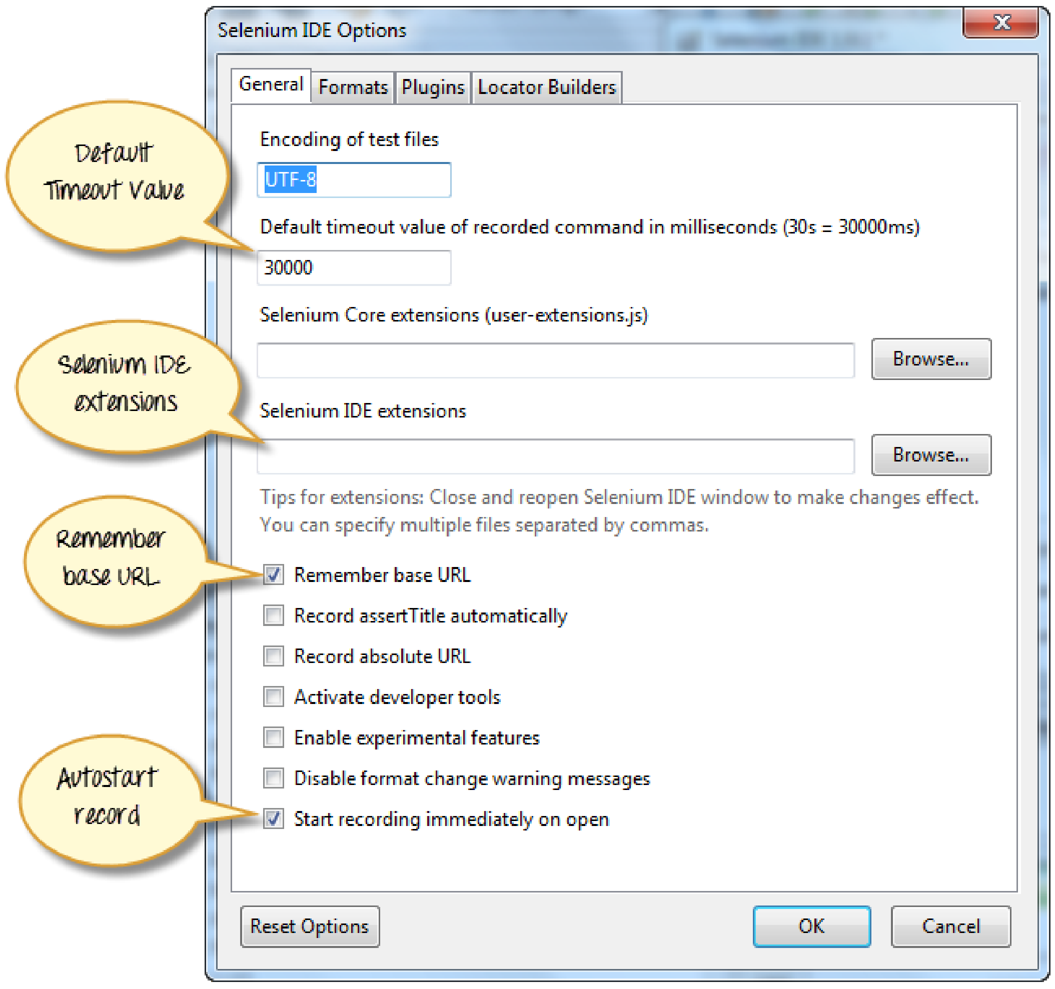
Command	Target	Value
open	/	
verifyAlertPresent		

Command	Target	Value
open	/	
verifyAlertPresent		



Command	Target	Value
open	/	
This is a comment		
verifyAlertPresent		


Options
Options...
Format
Clipboard Format
Reset IDE Window
HTML
C# / NUnit / WebDriver
C# / NUnit / Remote Control
● Java / JUnit 4 / WebDriver
Java / JUnit 4 / WebDriver Backed
Java / JUnit 4 / Remote Control
Java / JUnit 3 / Remote Control
Java / TestNG / Remote Control
Python 2 / unittest / WebDriver
Python 2 / unittest / Remote Control
Ruby / RSpec / WebDriver
Ruby / Test::Unit / WebDriver
Ruby / RSpec / Remote Control
Ruby / Test::Unit / Remote Control



Selenium IDE Options

General | Formats | Plugins | Locator Builders

Encoding of test files

UTF-8

Default timeout value of recorded command in milliseconds (30s = 30000ms)

30000

Selenium Core extensions (user-extensions.js)

Browse...

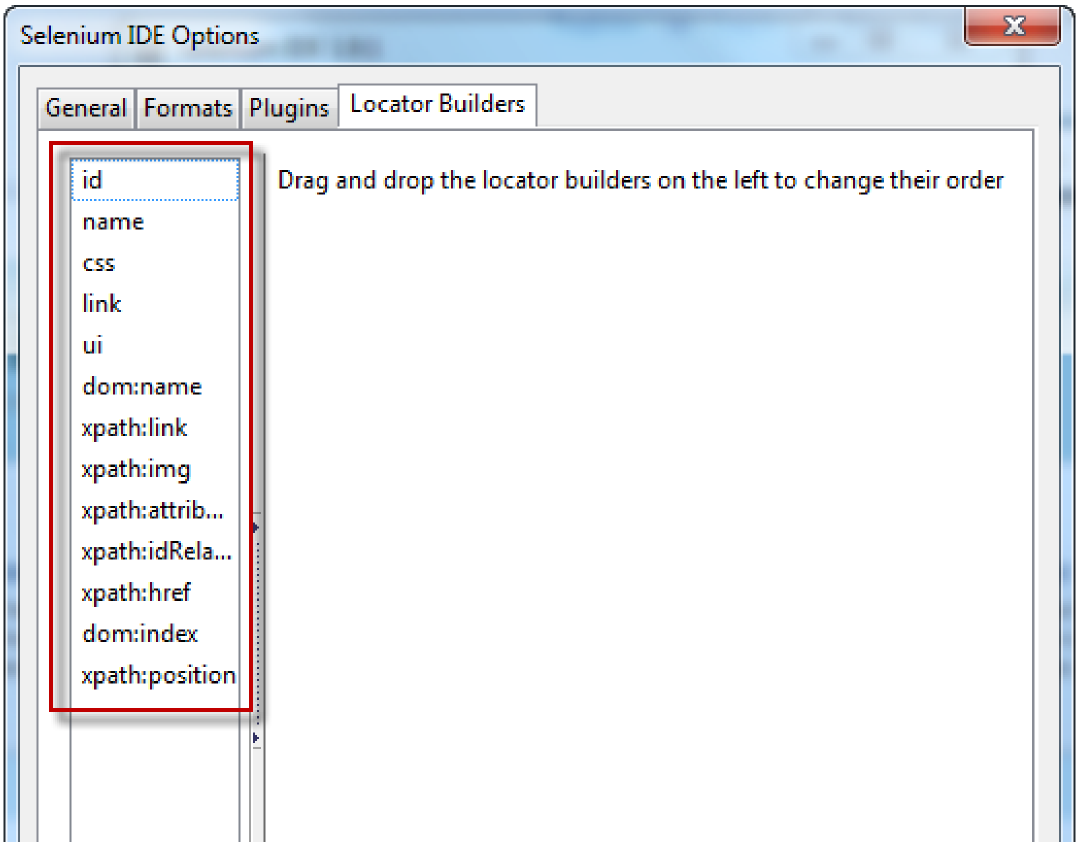
Selenium IDE extensions

Browse...

Tips for extensions: Close and reopen Selenium IDE window to make changes effect.
You can specify multiple files separated by commas.

- [✓] Remember base URL
- [ ] Record assertTitle automatically
- [ ] Record absolute URL
- [ ] Activate developer tools
- [ ] Enable experimental features
- [ ] Disable format change warning messages
- [✓] Start recording immediately on open

Reset Options | OK | Cancel



Selenium IDE Options

General | Formats | Plugins | Locator Builders

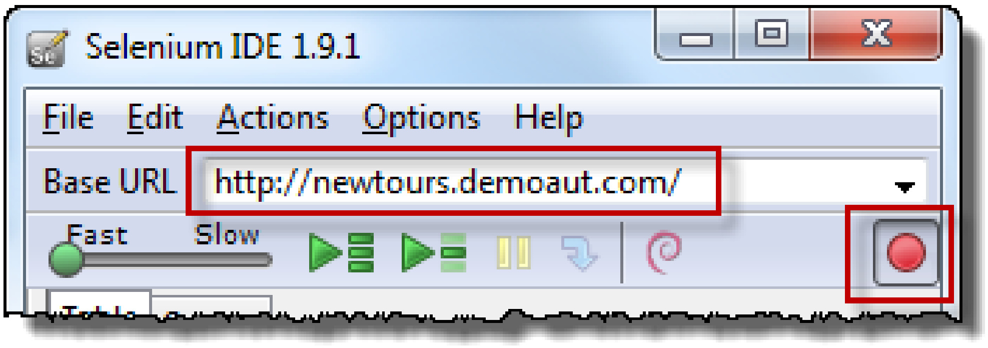
id
name
css
link
ui
dom:name
xpath:link
xpath:img
xpath:attrib...
xpath:idRela...
xpath:href
dom:index
xpath:position

Drag and drop the locator builders on the left to change their order

Create a Script by Recording:

Step 1:

- Launch Firefox and Selenium IDE.
- Type the value for our Base URL: http://newtours.demoaut.com/.
- Toggle the Record button on (if it is not yet toggled on by default).

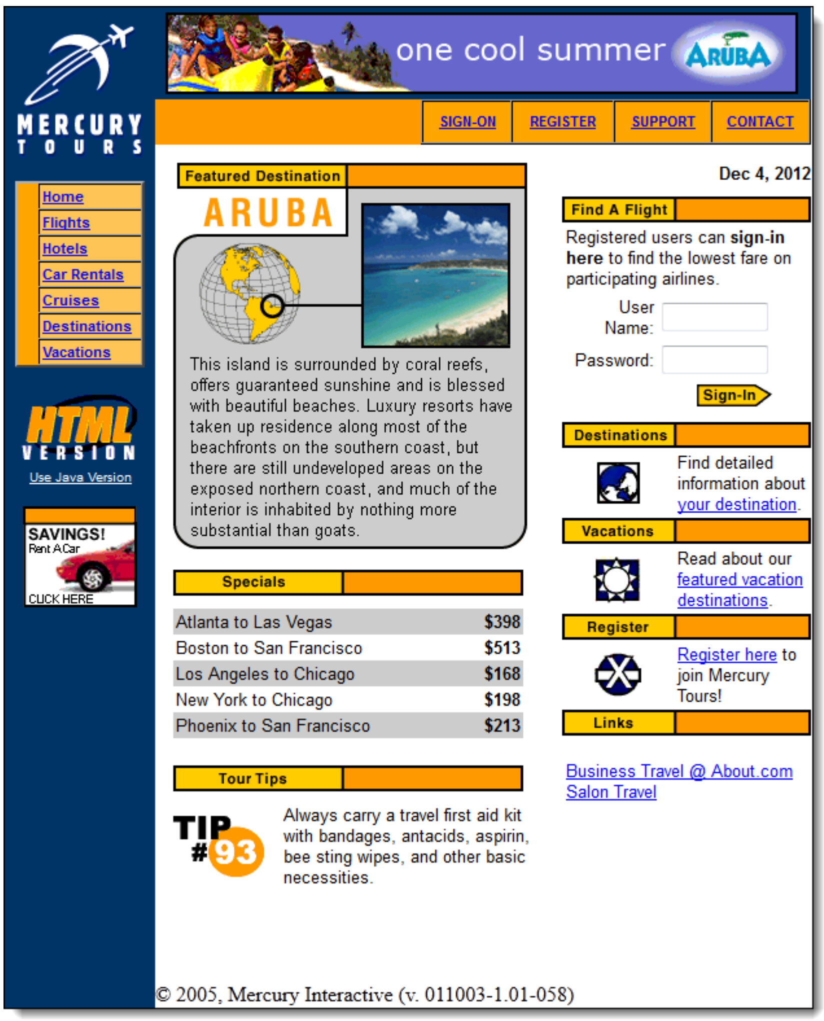


Selenium IDE 1.9.1

Base URL: http://newtours.demoaut.com/

Step 2:

- In Firefox, navigate to http://newtours.demoaut.com/. Firefox should take you to the page similar to the one shown below.



one cool summer ARUBA

MERCURY TOURS

SIGN-ON | REGISTER | SUPPORT | CONTACT

Home
Flights
Hotels
Car Rentals
Cruises
Destinations
Vacations

HTML VERSION
Use Java Version

SAVINGS! Rent A Car CLICK HERE

Featured Destination

ARUBA

This island is surrounded by coral reefs, offers guaranteed sunshine and is blessed with beautiful beaches. Luxury resorts have taken up residence along most of the beachfronts on the southern coast, but there are still undeveloped areas on the exposed northern coast, and much of the interior is inhabited by nothing more substantial than goats.

Specials

Atlanta to Las Vegas	$398
Boston to San Francisco	$513
Los Angeles to Chicago	$168
New York to Chicago	$198
Phoenix to San Francisco	$213

Tour Tips

TIP #93

Always carry a travel first aid kit with bandages, antacids, aspirin, bee sting wipes, and other basic necessities.

Dec 4, 2012

Find A Flight

Registered users can **sign-in here** to find the lowest fare on participating airlines.

User Name:

Password:

Sign-In

Destinations

Find detailed information about your destination.

Vacations

Read about our featured vacation destinations.

Register

Register here to join Mercury Tours!

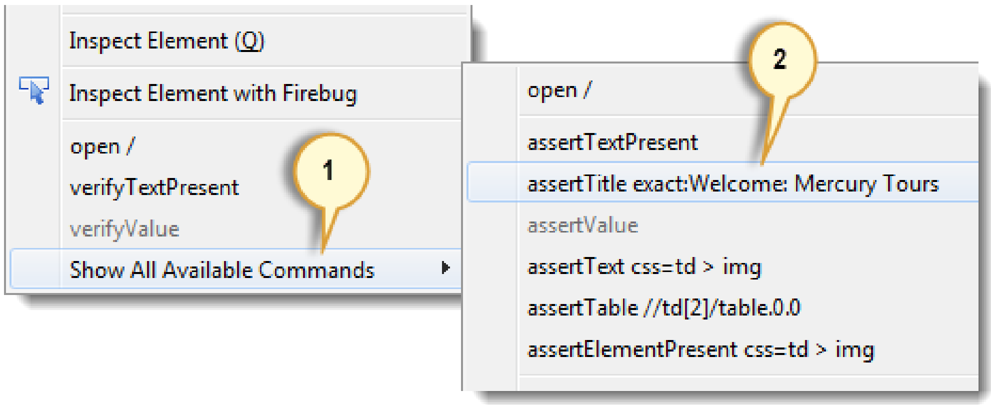
Links

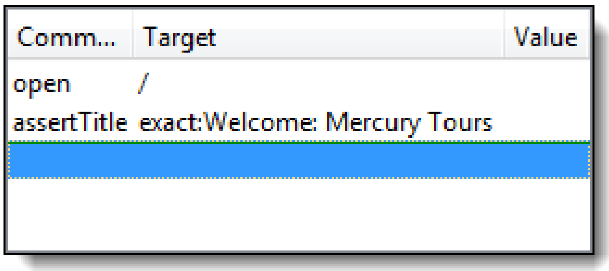
Business Travel @ About.com
Salon Travel

© 2005, Mercury Interactive (v. 011003-1.01-058)

Step 3:

- Right-click on any blank space within the page, like on the Mercury Tours logo on the upper left corner. This will bring up the Selenium IDE context menu. Note: Do not click on any hyperlinked objects or images
- Select the "Show Available Commands" option.
- Then, select "assertTitle exact:Welcome: Mercury Tours". This is a command that makes sure that the page title is correct.


Inspect Element (Q)
Inspect Element with Firebug
open /
verifyTextPresent
verifyValue
Show All Available Commands
1
2
open /
assertTextPresent
assertTitle exact:Welcome: Mercury Tours
assertValue
assertText css=td > img
assertTable //td[2]/table.0.0
assertElementPresent css=td > img

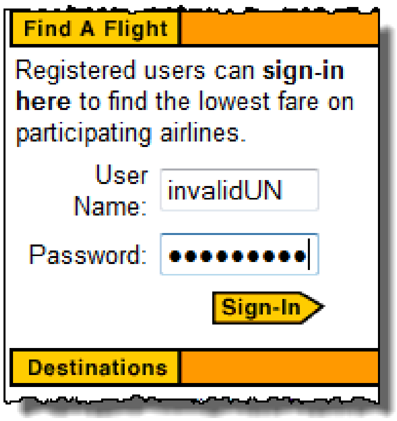


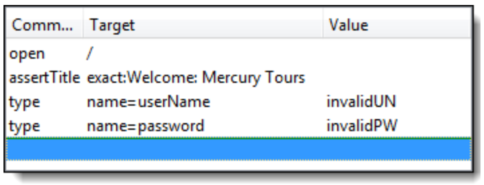
Comm...	Target	Value
open	/	
assertTitle	exact:Welcome: Mercury Tours	

After clicking on the assertTitle context menu option, your Selenium IDE Editor pane should now contain the following commands

Step 4:

- In the "User Name" text box of Mercury Tours, type an invalid username, "invalidUN".
- In the "Password" text box, type an invalid password, "invalidPW".

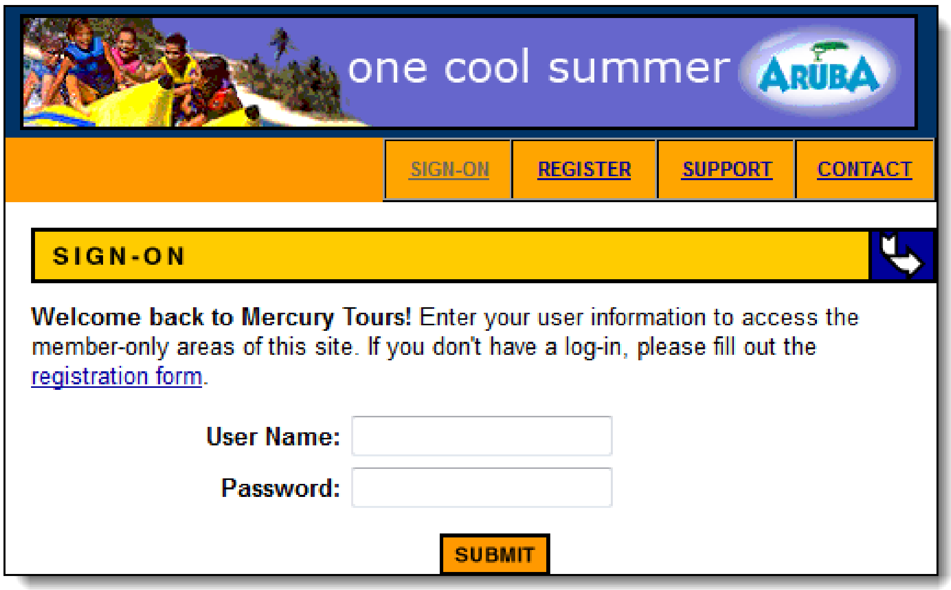

Find A Flight
Registered users can sign-in here to find the lowest fare on participating airlines.
User Name: invalidUN
Password: ●●●●●●●●●
Sign-In
Destinations



Comm...	Target	Value
open	/	
assertTitle	exact:Welcome: Mercury Tours	
type	name=userName	invalidUN
type	name=password	invalidPW

Your Editor should now look like this

Step 5:



Welcome back to Mercury Tours! Enter your user information to access the member-only areas of this site. If you don't have a log-in, please fill out the registration form.

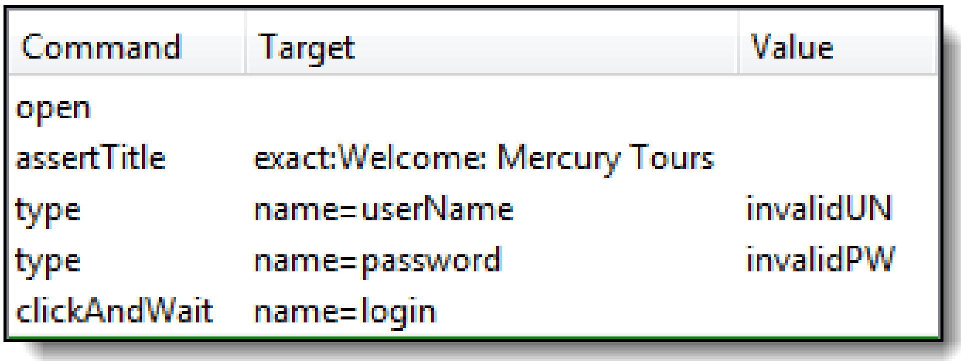
User Name:

Password:

SUBMIT

Step 6:

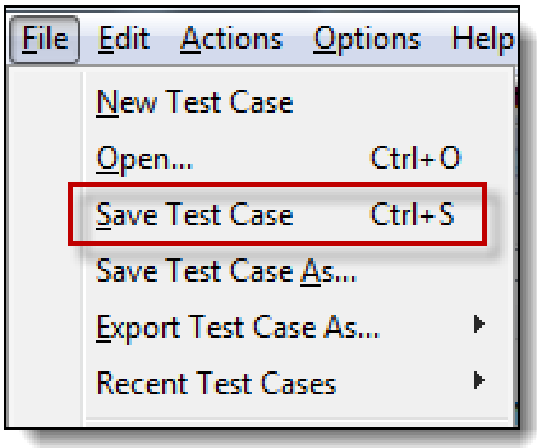
- Toggle the record button off to stop recording. Your script should now look like the one shown below.



Command	Target	Value
open		
assertTitle	exact:Welcome: Mercury Tours	
type	name=userName	invalidUN
type	name=password	invalidPW
clickAndWait	name=login	

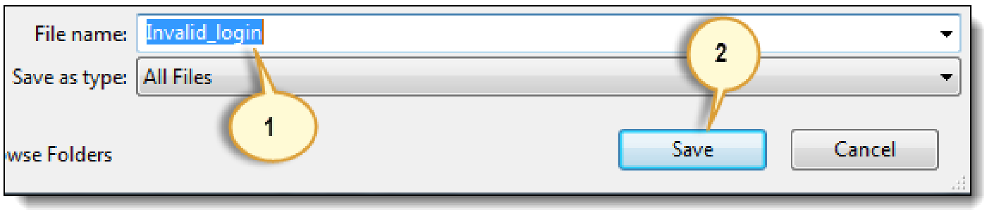
Step 7:

- Now that we are done with our test script, we shall save it in a test case. In the File menu, select “Save Test Case”. Alternatively, you can simply press Ctrl+S.



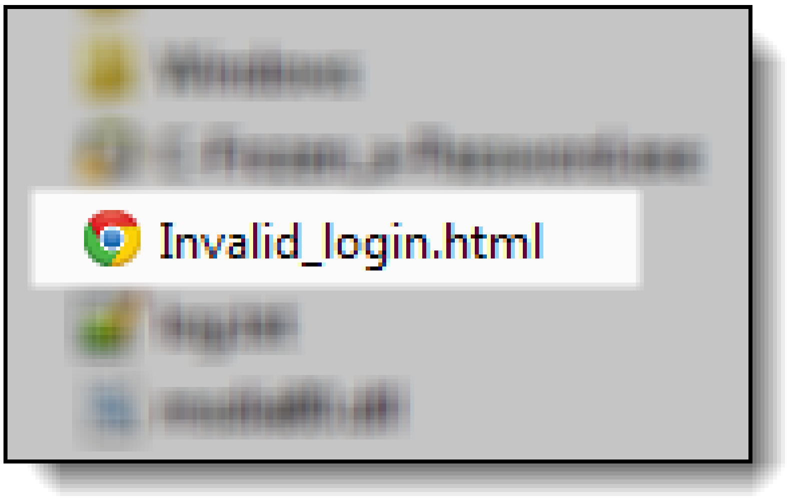
Step 8:

- Choose your desired location, and then name the test case as "Invalid_login".
- Click the "Save" button.



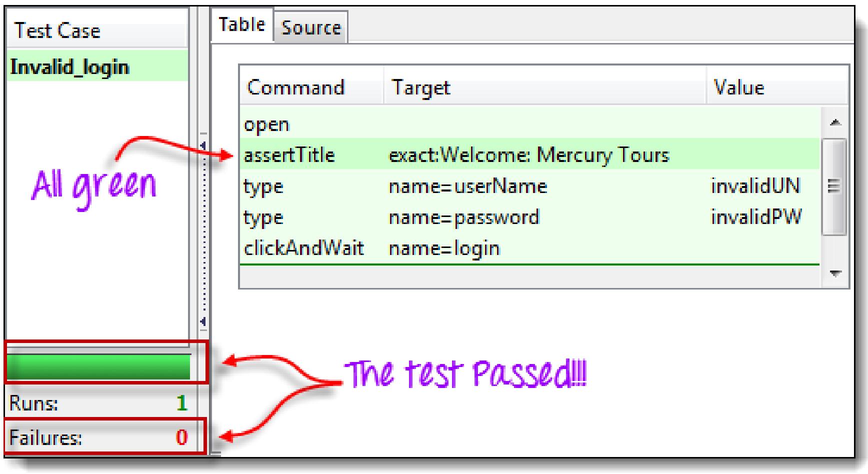
Step 9:

- Notice that the file was saved as HTML.



Step 10:

- Go back to Selenium IDE and click the Playback button to execute the whole script. Selenium IDE should be able to replicate everything flawlessly.



Command	Target	Value
open		
assertTitle	exact:Welcome: Mercury Tours	
type	name=userName	invalidUN
type	name=password	invalidPW
clickAndWait	name=login	

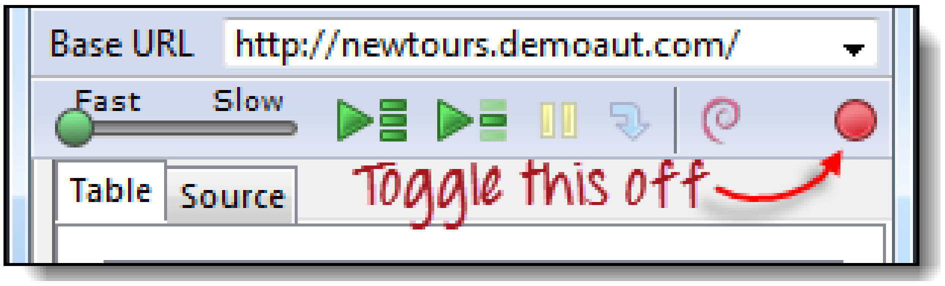
Common Commands:

Command	No. of Parameters	Description
open	0 - 2	Opens a page using a URL.
click/clickAndWait	1	Clicks on a specified element.
type/typeKeys	2	Types a sequence of characters.
verifyTitle/assertTitle	1	Compares the actual page title with an expected

verifyTextPresent	1	Checks if a certain text is found within the page.
verifyElementPresent	1	Checks the presence of a certain element.
verifyTable	2	Compares the contents of a table with expected
waitForPageToLoad	1	Pauses execution until the page is loaded
waitForElementPresent	1	Pauses execution until the specified element

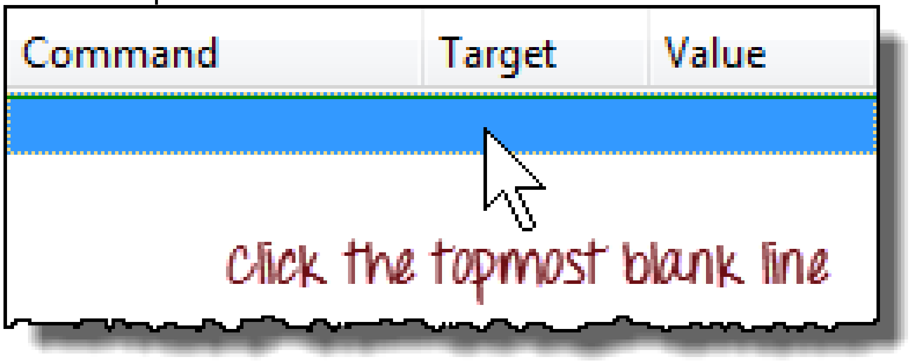
Create a Script Manually with Firebug:
Step 1:

- Open Firefox and Selenium IDE.
- Type the base URL (http://newtours.demoaut.com/).
- The record button should be OFF.

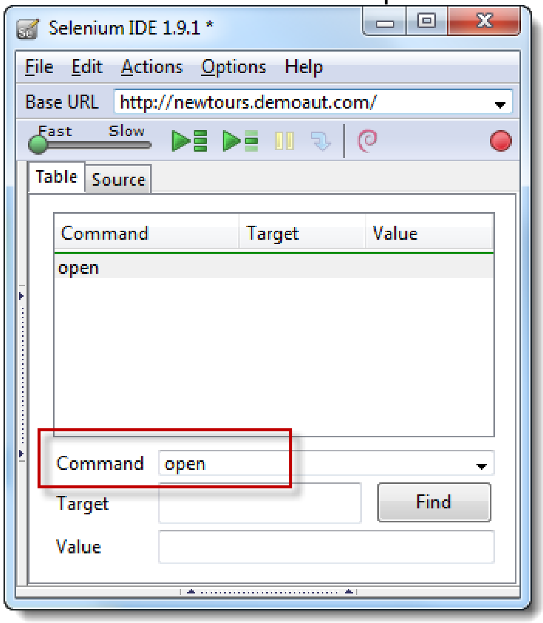


Step 2:

- Click on the topmost blank line in the Editor.

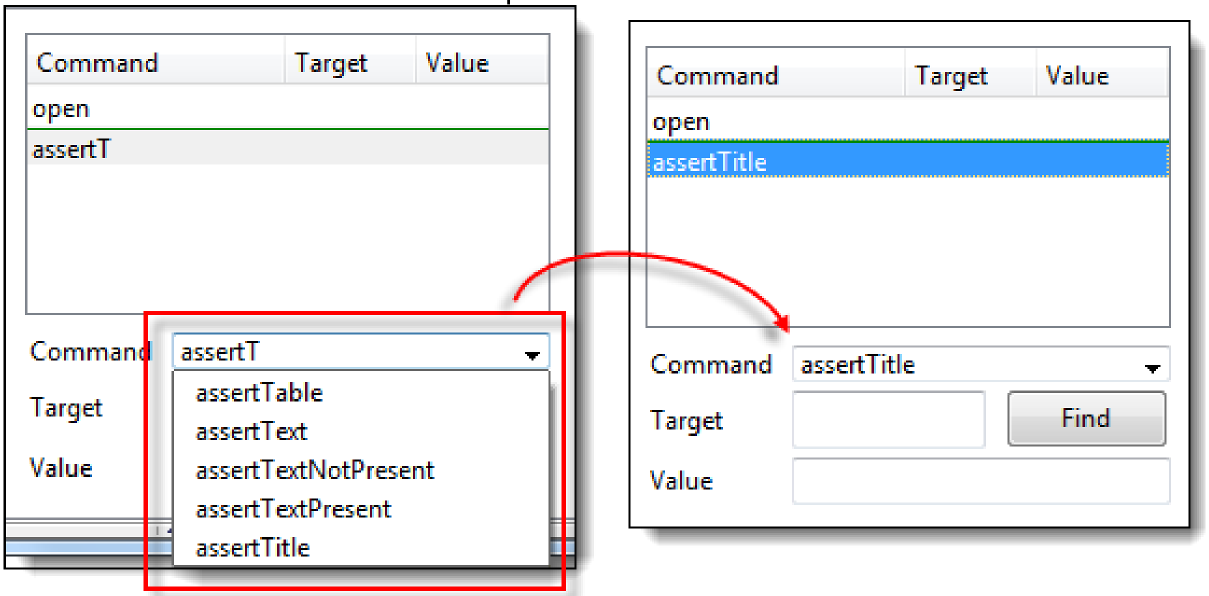


- Type “open” in the Command text box and press Enter.



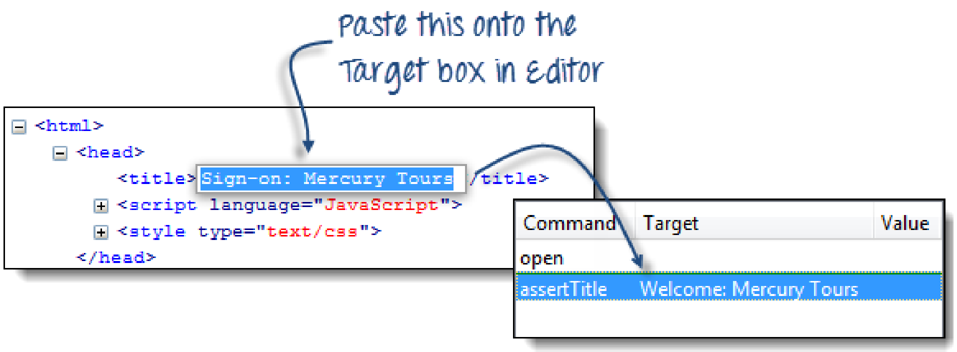
Step 3:

- Navigate Firefox to our base URL and activate Firebug
- In the Selenium IDE Editor pane, select the second line (the line below the "open" command) and create the second command by typing "assertTitle" on the Command box.
- Feel free to use the autocomplete feature.



Step 4:

- In Firebug, expand the <head> tag to display the <title> tag.
- Click on the value of the <title> tag (which is "Welcome: Mercury Tours") and paste it onto the Target field in the Editor.

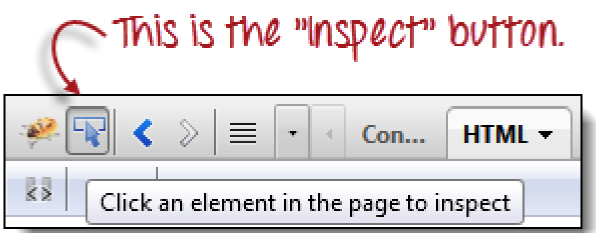


```
<html>
  <head>
    <title>Sign-on: Mercury Tours</title>
    <script language="JavaScript">
    <style type="text/css">
  </head>
```

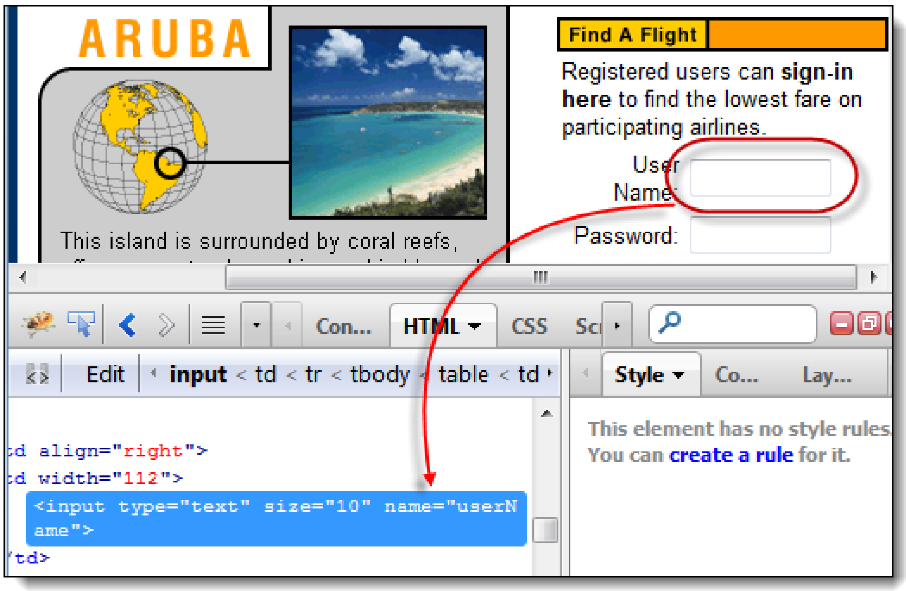
Command	Target	Value
open		
assertTitle	Welcome: Mercury Tours	

Step 5:

- To create the third command, click on the third blank line in the Editor and key-in "type" on the Command text box.
- In Firebug, click on the "Inspect" button.


This is the "Inspect" button.
Con...
HTML
Click an element in the page to inspect

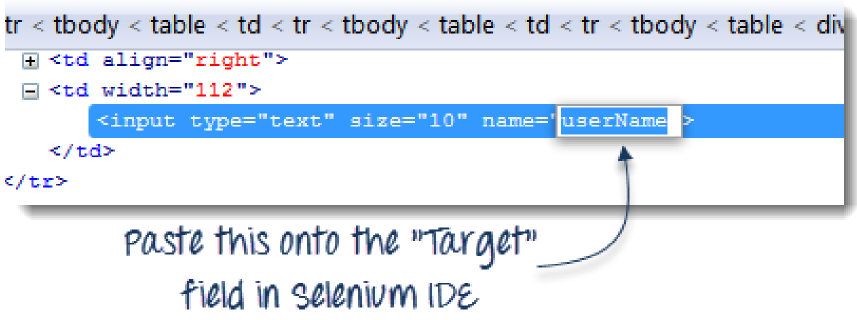
- Click on the User Name text box. Notice that Firebug automatically shows you the HTML code for that element.


ARUBA
This island is surrounded by coral reefs,
Find A Flight
Registered users can sign-in here to find the lowest fare on participating airlines.
User Name:
Password:
Con... HTML CSS Sc
Edit input < td < tr < tbody < table < td
Style Co... Lay...
This element has no style rules. You can create a rule for it.

```
d align="right">
d width="112">
  <input type="text" size="10" name="userN
  ame">
/td>
```

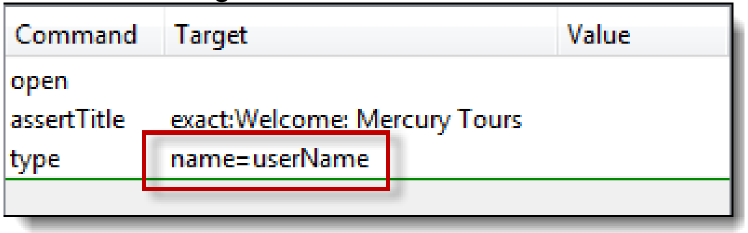
Step 6:

- Notice that the User Name text box does not have an ID, but it has a NAME attribute. We shall, therefore, use its NAME as the locator. Copy the NAME value and paste it onto the Target field in Selenium IDE.


tr < tbody < table < td < tr < tbody < table < td < tr < tbody < table < div
Paste this onto the "Target" field in Selenium IDE

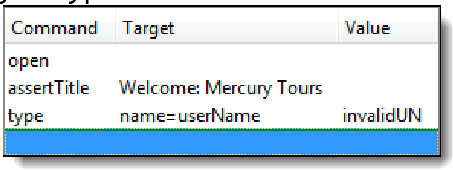
```
<td align="right">
<td width="112">
    <input type="text" size="10" name="userName">
  </td>
</tr>
```

- Still in the Target text box, prefix "userName" with "name=", indicating that Selenium IDE should target an element whose NAME attribute is "userName."



Command	Target	Value
open		
assertTitle	exact:Welcome: Mercury Tours	
type	name=userName	

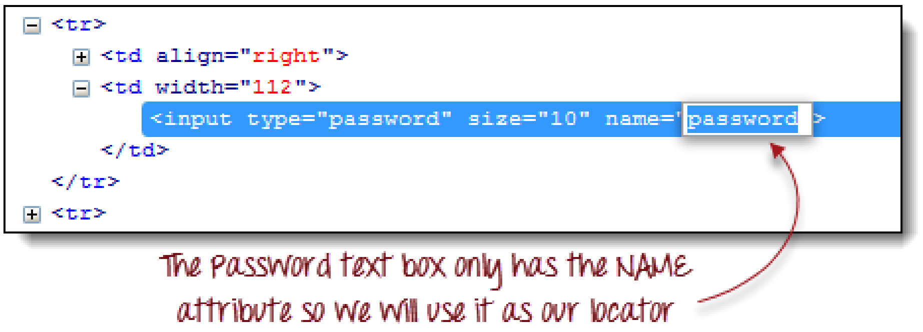
- Type "invalidUN" in the Value text box of Selenium IDE. Your test script should now look like the image below. We are done with the third command. Note: Instead of invalidUN , you may enter any other text string. But Selenium IDE is case sensitive and you type values/attributes exactly like in application.



Command	Target	Value
open		
assertTitle	Welcome: Mercury Tours	
type	name=userName	invalidUN

Step 7:

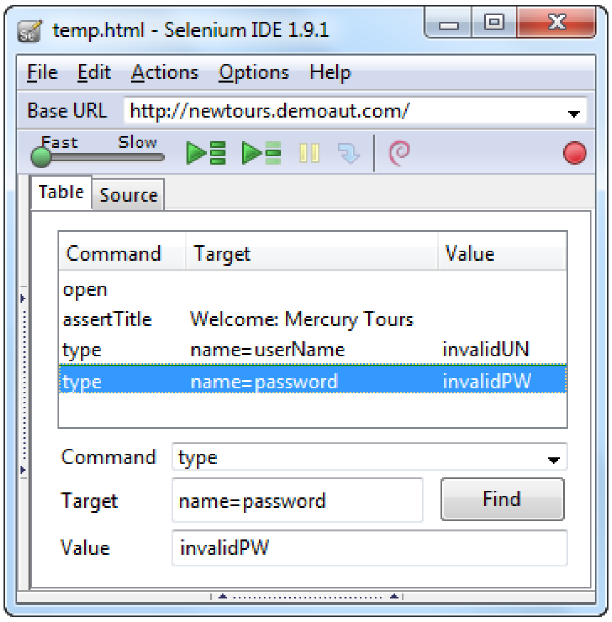
- To create the fourth command, key-in “type” on the Command text box.
- Again, use Firebug’s “Inspect” button to get the locator for the “Password” text box.



```
<tr>
    <td align="right">
    <td width="112">
        <input type="password" size="10" name="password">
    </td>
</tr>
<tr>
```

The password text box only has the NAME attribute so we will use it as our locator

- Paste the NAME attribute (“password”) onto the Target field and prefix it with “name=”
- Type “invalidPW” in the Value field in Selenium IDE. Your test script should now look like the image below.



temp.html - Selenium IDE 1.9.1
File Edit Actions Options Help
Fast Slow
Table Source
Find

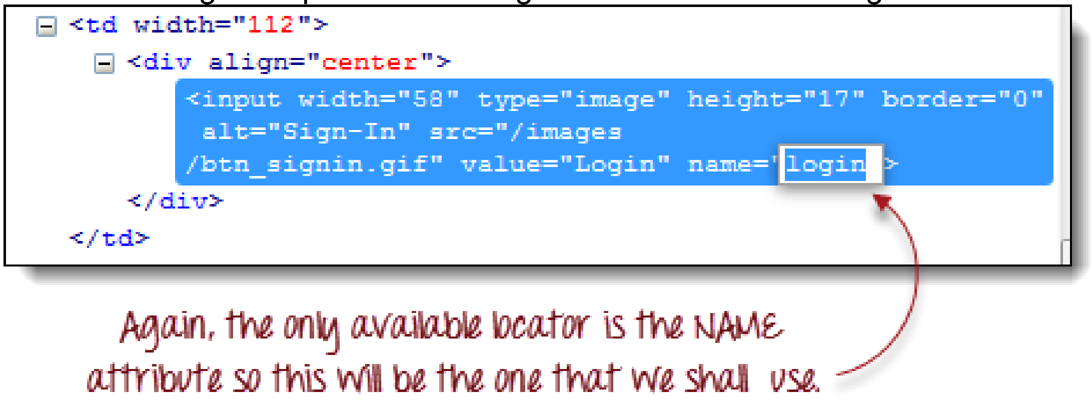
Base URL http://newtours.demoaut.com/

Command	Target	Value
open		
assertTitle	Welcome: Mercury Tours	
type	name=userName	invalidUN
type	name=password	invalidPW

Command: type
Target: name=password
Value: invalidPW

Step 8:

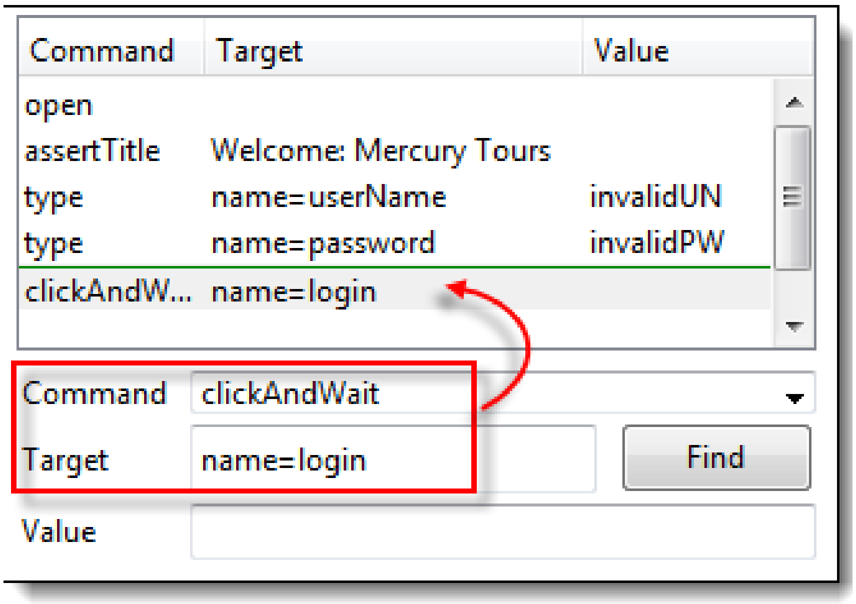
- For the fifth command, type “clickAndWait” on the Command text box in Selenium IDE.
- Use Firebug’s “Inspect” button to get the locator for the “Sign In” button.



```
<td width="112">
    <div align="center">
        <input width="58" type="image" height="17" border="0"
         alt="Sign-In" src="/images
        /btn_signin.gif" value="Login" name="login">
    </div>
</td>
```

Again, the only available locator is the NAME attribute so this will be the one that we shall use.

- Paste the value of the NAME attribute ("login") onto the Target text box and prefix it with "name=".
- Your test script should now look like the image below.



Command	Target	Value
open		
assertTitle	Welcome: Mercury Tours	
type	name=userName	invalidUN
type	name=password	invalidPW
clickAndW...	name=login	

Command: clickAndWait
Target: name=login
Value:

Step 9:

- Save the test case

Practical No 11

Title: Test Automation using Selenium Webdriver

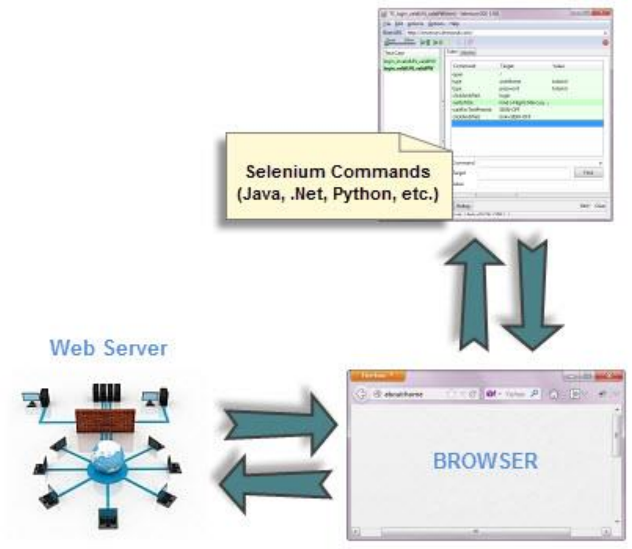
Problem Statement:
Automate web test using Selenium Webdriver.

Theory:
Selenium Web Driver:
WebDriver enables you to **use a programming language** in creating your test scripts(not possible in Selenium IDE).

- You can now use **conditional operations** like if-then-else or switch-case
- You can also perform **looping** like do-while.
- Following programming languages are supported by WebDriver
    - Java
    - .Net
    - PHP
    - Python
    - Perl
    - Ruby

- **You do not have to know all of them. You just need to be knowledgeable in one.**
- Before advent of WebDriver in 2006, there was another, **automation tool called Selenium Remote Control.** Both WebDriver and Selenium RC have following features:
- They both allow you to **use a programming language** in designing your test scripts.
- They both allow you to **run your tests against different browsers.**

- **WebDriver's architecture is simple.**
- It controls the browser from the OS level
- All you need are your programming language's IDE (which contains your Selenium commands) and a browser.



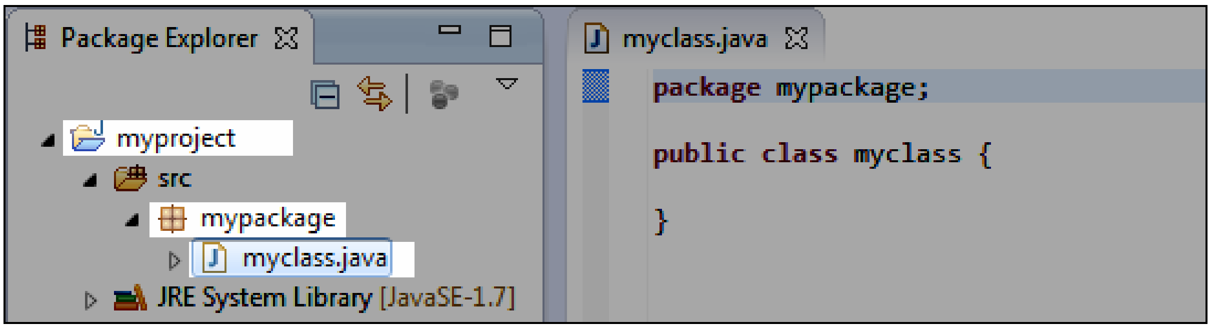
- **WebDriver can support the headless HtmlUnit browser**.
- HtmlUnit is termed as “headless” because it is an invisible browser – it is GUI-less.
- It is a very fast browser because no time is spent in waiting for page elements to load. This accelerates your test execution cycles.
- Since it is invisible to the user, it can only be controlled through automated means.
- **Selenium RC cannot support the headless HtmlUnit browser.** It needs a real, visible browser to operate on.

Limitations of Webdriver:

- WebDriver Cannot Readily Support New Browsers
- WebDriver operates on the OS level. Also remember that different browsers communicate with the OS in different ways. If a new browser comes out, it may have a different process of communicating with the OS as compared to other browsers.
- However, it is up to the WebDriver’s team of developers to decide if they should support the new browser or not.

Installing Selenium WebDriver:

- Step 1 – Install Java on your computer
- Step 2 – Install Eclipse IDE
- Step 3 – Download the Selenium Java Client Driver
- Step 4 – Configure Eclipse IDE with WebDriver
- Launch the “eclipse.exe” file inside the “eclipse” folder that we extracted in step 2. If you followed step 2 correctly, the executable should be located on C:\eclipse\eclipse.exe.
- When asked to select for a workspace, just accept the default location.
- Create a new project through File > New > Java Project. Name the project as “myproject”.
- Right-click on the newly created project and select New > Package, and name that package as “mypackage”.
- Create a new Java class under *mypackage* by right-clicking on it and then selecting New > Class, and then name it as “myclass”.



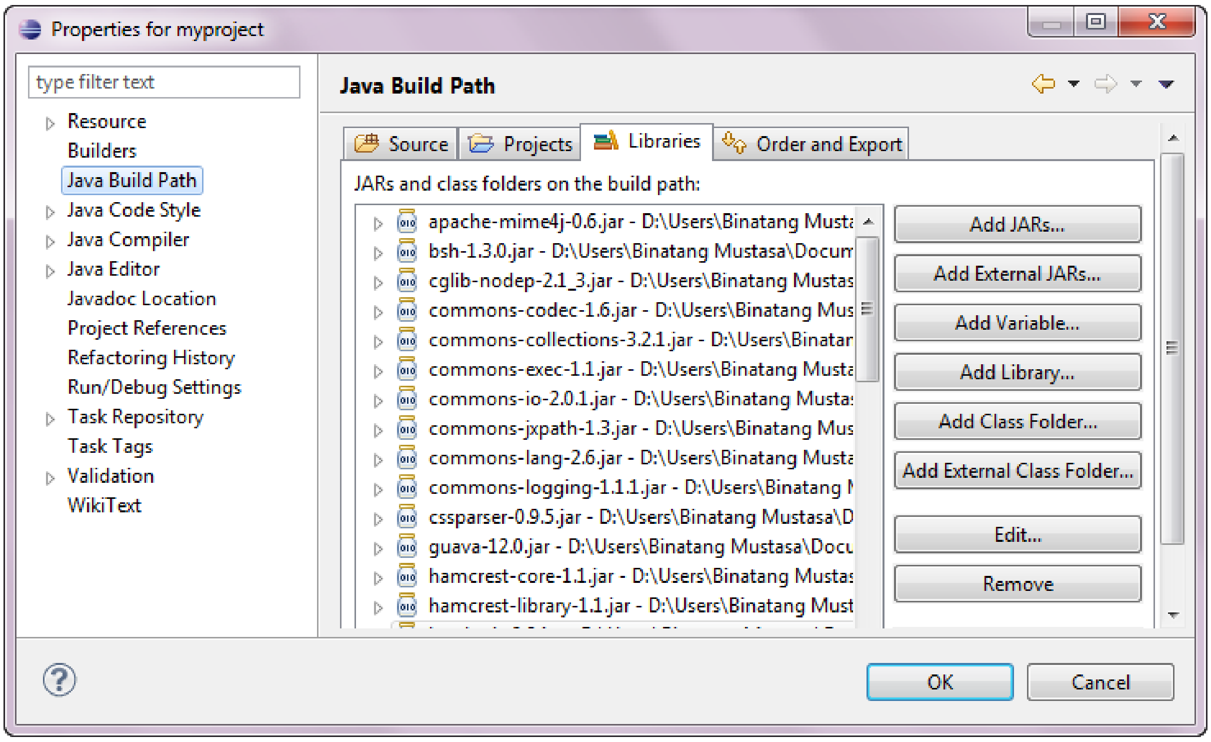
```
package mypackage;

public class myclass {

}
```

- Right-click on *myproject* and select Properties.
- On the Properties dialog, click on “Java Build Path”.
- Click on the Libraries tab, and then click “Add External JARs..”
- Navigate to C:\selenium-2.25.0\ (or any other location where you saved the extracted contents of “selenium-2.25.0.zip” in step 3).
- Add all the JAR files inside and outside the “libs” folder.

- Finally, click OK and we are done importing Selenium libraries into our project.



JARs and class folders on the build path:

```
apache-mime4j-0.6.jar - D:\Users\Binatang Must
bsh-1.3.0.jar - D:\Users\Binatang Mustasa\Docum
cglib-nodep-2.1_3.jar - D:\Users\Binatang Mustas
commons-codec-1.6.jar - D:\Users\Binatang Mus
commons-collections-3.2.1.jar - D:\Users\Binatar
commons-exec-1.1.jar - D:\Users\Binatang Must
commons-io-2.0.1.jar - D:\Users\Binatang Musta
commons-jxpath-1.3.jar - D:\Users\Binatang Mus
commons-lang-2.6.jar - D:\Users\Binatang Must
commons-logging-1.1.1.jar - D:\Users\Binatang M
cssparser-0.9.5.jar - D:\Users\Binatang Mustasa\D
guava-12.0.jar - D:\Users\Binatang Mustasa\Docu
hamcrest-core-1.1.jar - D:\Users\Binatang Mustas
hamcrest-library-1.1.jar - D:\Users\Binatang Must
```

Using the Java class “myclass” that we created in the previous tutorial, let us try to create a WebDriver script that would:

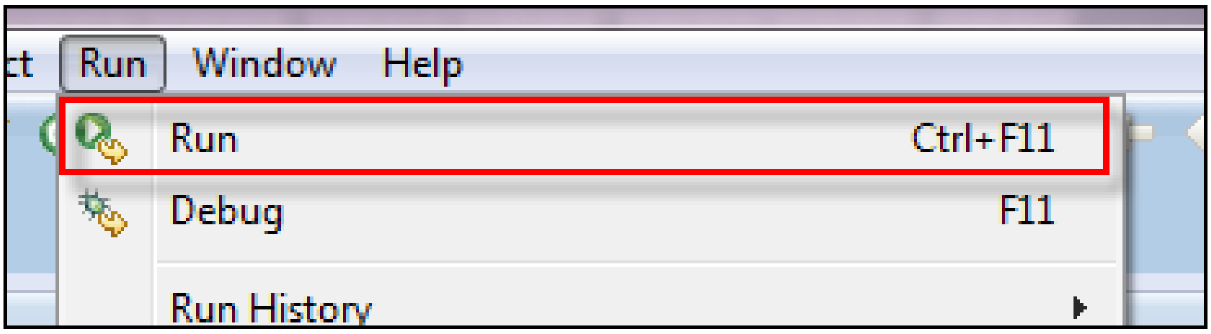
- fetch Mercury Tours’ homepage
- verify its title
- print out the result of the comparison
- close it before ending the entire program.

```
package mypackage;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
public class myclass {
    public static void main(String[] args) {
          // declaration and instantiation of objects/variables
          WebDriver driver = new FirefoxDriver();
          String baseUrl = "http://newtours.demoaut.com";
          String expectedTitle = "Welcome: Mercury Tours";
               String actualTitle = "";
               // launch Firefox and direct it to the Base URL
               driver.get(baseUrl);
               // get the actual value of the title
               actualTitle = driver.getTitle();
               /*
                * compare the actual title of the page witht the expected one and print
                * the result as "Passed" or "Failed"
               */
```

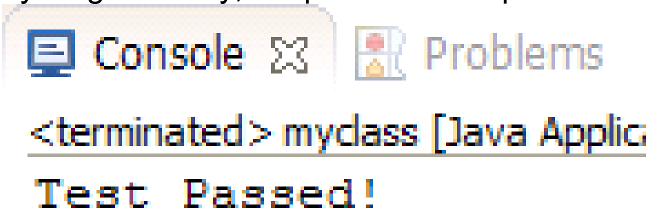
```
 if (actualTitle.contentEquals(expectedTitle)){
                       System.out.println("Test Passed!");
                } else {
                       System.out.println("Test Failed");
                }
                //close Firefox
                driver.close();
                // exit the program explicitly
                System.exit(0);
           }
      }
```

Running the Test:

- There are two ways to execute code in Eclipse IDE.
- On Eclipse's menu bar, click Run > Run.
- Press Ctrl+F11 to run the entire code.



- If you did everything correctly, Eclipse would output "Test Passed!"



```
<terminated> myclass [Java Applic
Test Passed!
```

These are basic examples. More on selenium need to be studied.

References:

1. IEEE829
2. "*Effective Methods for Software Testing*" 2nd Edition, W.E. Perry, John Wiley.
3. *"Foundations of Software Testing"*, Dorothy Graham, Erik van Veenendaal, Isabel Evans, Rex Black
4. http://www.guru99.com/selenium-tutorial.html