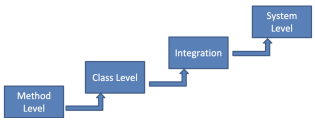
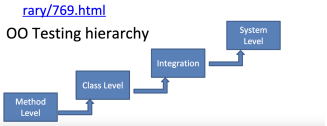
Testing Object Oriented Software

- OO Testing hierarchy

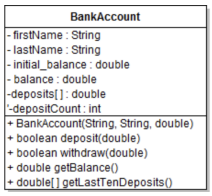


Testing Object Oriented Software

- Various UML artifacts
  - Use cases, Sequence diagrams, State Charts, Class Responsibility Collaborator (CRC) diagrams
    - https://www.ibm.com/developerworks/university/
  - http://www.ibm.com/developerworks/rational/library/769.html
- OO Testing hierarchy



OO Testing Example – Bank Account Example Revisit



Integration Testing Strategies

Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.

Options:

- **the "big bang" approach**
- **an incremental construction strategy**

Incremental Integration

- The program is constructed and tested in small increments, where errors are easier to isolate and correct.

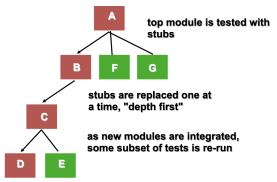
Three main approaches:

- Top-down
- Bottom-up
- Sandwich

Top Down Integration

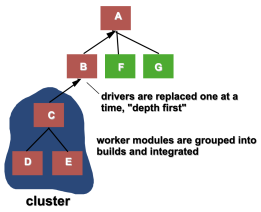
- Top Down Integration is an **incremental approach** to construction of the software architecture.
- Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program.)
- Modules subordinate to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

Top Down Integration

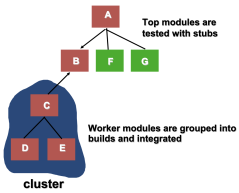


Depth-First Integration – A, B, C, D, E, F, G
Breadth First Integration – A, B, F, G, C, D, E

Bottom-Up Integration



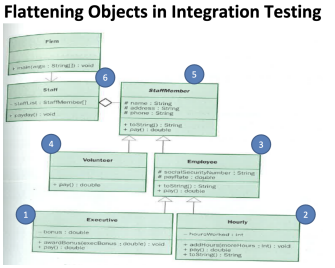
Sandwich Testing



OO Integration Testing

- If the class has to be very large, usually a method as a unit can be a viable choice. In that case, two levels of integration testing is required:
  - Integrate methods into a class
  - Integrate classes into other classes
- On the other hand, if more common, class as a unit is adopted Integrate classes into other classes is required
  - Once the unit testing is complete (a) if flatten classes are used, original class hierarchy must be restored.
  - Test methods (if any added) must be removed.

Flattening Objects in Integration Testing



Testing Firm Implementation

- One possible approach in testing this application is to test each class starting from the bottom most classes : Hourly and Executive. Then integrate other objects in bottom up manner
- Object Flattening can be used here. (See Executive.java example)

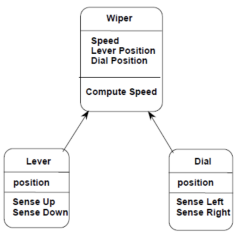
Saturn Windshield Wiper Controller

The windshield wiper on the Saturn automobiles is controlled by a lever with a dial. The four positions, OFF, INT (for intermittent), LOW, and HIGH, and the dial has three position numbered simply 1, 2, and 3. The dial positions indicate three intermittent speeds, and t position is relevant only when the lever is at the INT position. The decision table below s windshield wiper speeds (in wipes per minute) for the lever and dial positions.

Lever	OFF	INT	INT	INT	LOW	HIGH
Dial	n/a	1	2	3	n/a	n/a
Wiper	0	4	6	12	30	60

Input Event	Description	Output Event	Description
ie1	lever from OFF to INT	oe1	0 w.p.m.
ie2	lever from INT to LOW	oe2	4 w.p.m.
ie3	lever from LOW to HIGH	oe3	6 w.p.m.
ie4	lever from HIGH to LOW	oe4	12 w.p.m.
ie5	lever from LOW to INT	oe5	30 w.p.m.
ie6	lever from INT to OFF	oe6	60 w.p.m.
ie7	dial from 1 to 2		
ie8	dial from 2 to 3		
ie9	dial from 3 to 2		
ie10	dial from 2 to 1		

Windshield Wiper Objects

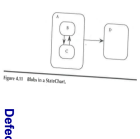


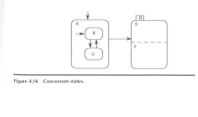
Windshield Wiper : Normal Usage (Use case)

UC 1	Normal Usage	
Description	The windshield wiper is in the OFF position, and the Dial is at the 1 position; the user moves the lever to INT, and then moves the dial first to 2 and then to 3; the user then moves the lever to LOW; the user moves the lever to INT, and then to OFF	
Preconditions	The windshield wiper is in the OFF position, and the Dial is at the 1 position; wiper speed is 0	
Event Sequence	User Action	System Response
1	Move lever to INT	Wiper speed is 6
2	Move dial to 2	Wiper speed is 6
3	Move dial to 3	Wiper speed is 12
4	Move lever to LOW	Wiper speed is 20
5	Move lever to INT	Wiper speed is 12
6	Move lever to OFF	Wiper speed is 0

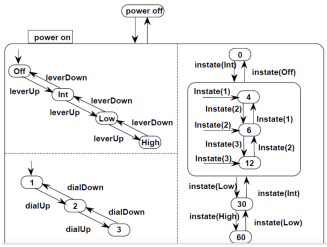
StateChart Based Testing

- StateChart is a techniques that provides visual representation of states of objects, transitions, and change of states through transitions.
- StateCharts can be used where the current action of an objects depends on previous actions.
- State chart is a collection of "blobs" that represent state and arrows that represent transitions.
- A blob can contain another blob
- Also StateCharts can be used to represent concurrent states of more than one object

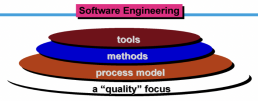




Windshield Wiper StateChart



What is Software Engineering?



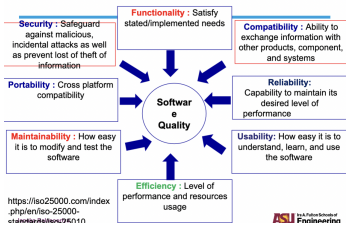
Quality: Conformance to explicitly stated **functional and performance requirements**, explicitly documented development standards, and **implicit characteristics that are expected** of all professionally developed software.

Process Model: Defines a framework that uses software engineering methods to build quality software.

Methods: technical details about how to's for building software.

Tools: Provides automated or semi-automated support for the method/ process

ISO - 25010

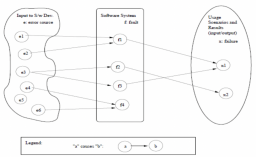


https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

Defect (or Bug) is a problem with the software, either with its external behavior or with its external characteristics.

Correctness, Defects (Bugs) and Quality

Good Quality = > Less Defects



Error, Fault, and Failure

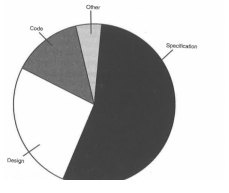
Error : A human action that produces an incorrect result

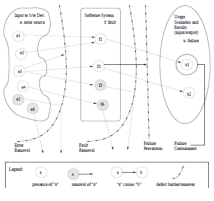
Fault: An incorrect step, process, or data definition in a program/ design/other software engineering artifact.

Failure: The inability of a system or a component to perform its required functions within specified performance requirements

Defect (or Bug): is a problem with the software, either with its external behavior or with its external characteristics. **Defect is an instance of a failure.**

Bugs are coming from ….





Three generic categories

Defect Prevention : Avoid error sources

Defect Reduction: Software Inspection, Testing Risk management

Defect Containment: Software fault tolerance …e

Software QA as dealing with defects

Defect prevention through error blocking

- Mitigating human errors by education and training (Product and domain specific knowledge, software development knowledge, knowledge about development methodology, technology and tools, development process knowledge)
- Better manage software process
- Using appropriate tools
- Using better software design/architecture

Defect reduction through fault detection and removal

- Inspection (code/design/requirement/test plan/test cases … etc)
- Testing : Failure observation and fault removal

Defect containment through failure prevention and containment

- Software fault tolerance (Executing the recovery block repeatedly, Duplication/parallel redundancy)

Software life cycle for a particular project depends on the nature of the project deliverables, time lines, resources available, completeness of requirements …etc

Major categories of software life cycle models are **Waterfall, Incremental, Iterative, Agile**

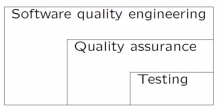
Aligning the SQA techniques with the software life cycle model used to develop the software is imperative in ensuring high quality software product.

Investigating software quality assurance activities in the waterfall model allows us to get a better perspective

Quality Assurance Activities in the Waterfall Model

- defect prevention in early phases
  - Error sources are associated with early phases
  - Inspections and reviews can be used to detect and remove error sources
- focused defect removal in testing phase
- defect containment in late phases
- phase transitions:
  - Inspection, review…etc except from testing to release.
  - From testing to release its acceptance testing

- Software Testing is the most important part of QA and the most commonly performed QA activity

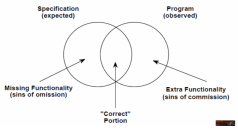


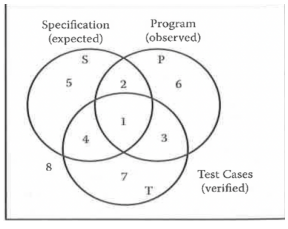
Software Quality vs. Testing [1]

What do we test?

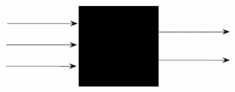
- In broader sense, we test if the software **does the right thing** and the software **does the things right.**
  - **Does the right thing :** based on the specification of the software
  - **Does the thing right:** perform specific tasks correctly and satisfactory

SRS document gives the specification (expected functionality) of the software
Actual program has the functionality implemented



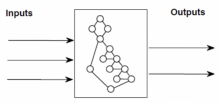


Functional Testing (Black Box)



Function is understood only in terms of it's inputs and outputs, with no knowledge of its implementation.

Structural Testing (White Box)



Function is understood only in terms of its implementation.

Functional vs. Structural Testing

- They are two fundamentally different approaches in designing test cases.
- **Functional testing uses the specification** while **structural approach uses the program** in designing test cases
- So, both approaches are needed in determining if we are building the right product and building product right

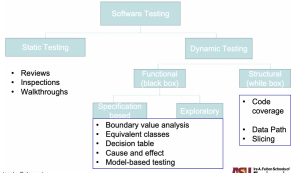
What is exploratory testing?

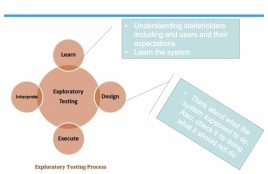
Is a blackbox testing approach where, a tester first study the software system to understand its purpose (intended usage, users, execution environments, social impact …etc) and test functionalities of the software by grouping them based in importance, similarity …etc.

Exploratory Testing Steps

- ❑ Identify the purpose of the product.
- ❑ Identify functions.
- ❑ Identify areas of potential instability.
- ❑ Test each function and record problems.
- ❑ Design and record a consistency verification test.

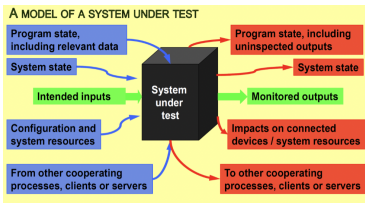
Various Approaches to Software Testing





Definition	Notes
Primary Function Any function or important that, in the estimation of a normal user, its inoperability or impairment would render the product unfit for its purpose.	A function is primary if you can associate it with the purpose of the product *even if* it is essential to that purpose. Primary functions define the product. For example, the function of editing text to a document in Microsoft Word is certainly so important that the product would be useless without it. Groups of functions, taken together, may constitute a primary function, too. For example, while perhaps no single function on the drawing toolbar of Word would be considered primary, the entire toolbar might be primary. If so, then most of the functions on that toolbar should be operable in order for the product to pass Certification.
Contributing Function Any function that contributes to the utility of the product, but is not a primary function.	Even though contributing functions are not primary, their inoperability *may* be grounds for refusing to grant Certification. For example, users may be technically able to do useful things with a product, even if it has an "Undo" function that never works, but most users will find that intolerable. Such a failure would violate fundamental expectations about how Windows products should work.

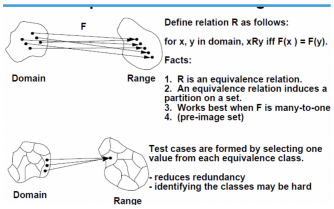
Equivalent Partition



Writing Test Cases : Anatomy of a Typical Test Case

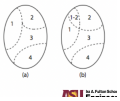
IEEE Standard 610 (1990) defines *test case* as follows:
"(1) A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.
"(2) (IEEE Std 829-1983) Documentation specifying inputs, predicted results, and a set of execution conditions for a test item."

Test case ID
Purpose
Preconditions
Inputs
Expected outputs
Post conditions
Execution History
Date Result Version Run By



- They all test the same thing so if one catches a defect, others probably will
- Thus, test case generation using **equivalence class** allows a tester to subdivide the input domain into a relatively small number of sub-domains, say N>1, as shown
- In strict mathematical terms, the sub-domains by **definition are disjoint.** The four subsets shown in (a) constitute a partition of the input domain while the subsets in (b) are not. Each subset is known as an **equivalence class.**

Finding equivalent classes
- Consider both valid and invalid inputs
- Look for equivalent output events
- Look for equivalent operating environments
- Organize them in a table

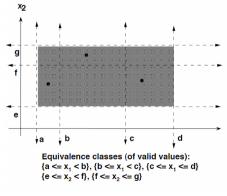


Classification of Equivalent Partitioning

Forms of Equivalence Class Testing (based on the textbook classification)
- **Weak Normal:** classes of valid values of inputs
- **Strong Normal: Valid inputs** one from each class
- **Weak Robust:** classes of valid and invalid values of inputs (single fault assumption)
- **Strong Robust:** (multiple fault assumption) one from each class in Cartesian Product

Let's compare these for a function of two variables, F(x1, x2)
- Extension to problems with 3 or more variables is not that difficult.

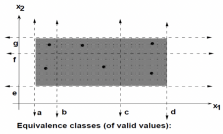
Weak Normal Equivalence Class Test Cases



Equivalence classes (of valid values):
{a <= x_1 < b}, {b <= x_1 < c}, {c <= x_1 <= d}
{e <= x_2 < f}, {f <= x_2 <= g}

- **Identify equivalence classes of valid values.**
- **Test cases have all valid values.**
- **OK for regression testing.**

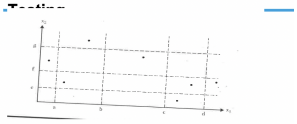
Strong Normal Equivalence Class Test Cases



Equivalence classes (of valid values):
{a <= x_1 < b}, {b <= x_1 < c}, {c <= x_1 <= d}
{e <= x_2 < f}, {f <= x_2 <= g}

Identify equivalence classes of valid values.
- **Test cases from Cartesian Product of valid values.**
- **Detects due to interactions with valid values of any number of variables.**
- **OK for regression testing, better for progression testing.**

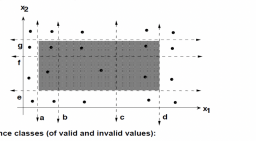
Weak Robust Equivalent Class



Equivalence classes (of valid and invalid values):
{a <= x_1 < b}, {b <= x_1 < c}, {c <= x_1 <= d}, {e <= x_2 < f}, {f <= x_2 <= g}
Invalid classes: {x_1 < a}, {x_1 > d}, {x_2 < e}, {x_2 > g}

Identify equivalence classes of valid values.
- **Test cases have all valid values except one invalid value.**
- **OK for regression testing.**

Strong Robust Equivalence Class Test Cases



Equivalence classes (of valid and invalid values):
{a <= x_1 < b}, {b <= x_1 < c}, {c <= x_1 <= d}, {e <= x_2 < f}, {f <= x_2 <= g}
Invalid classes: {x_1 < a}, {x_1 > d}, {x_2 < e}, {x_2 > g}

Identify equivalence classes of valid and invalid values.
- **Test cases from Cartesian Product of all classes.**

Triangle Problem Equivalent Partitioning : Approach 2 ... cont.

Again, we can try to generate EPs based on output. In addition, we can try to classify them based on weak and strong using the classification stated in the book.

Four possible outcomes

Equilateral
E1 = {<a, b, c> : the triangle with sides a, b, and c where a=b=c}

Isosceles
E2 = {<a, b, c> : the triangle with sides a, b, and c where a=b, a=c | b=c, b=a | a=c, a>b}

Scalene
E3 = {<a, b, c> : the triangle with sides a, b, and c where a<b<c }

Not A Triangle
E4 = {<a, b, c> : the triangle with sides a, b, and c where a + b <= c | a + c <= b | b + c <= a }

Weak Normal:

Test Case	a	b	c	Expected Output
WN1	5	5	5	Equilateral
WN2	2	2	3	Isosceles
WN3	3	4	5	Scalene
WN4	4	1	2	Not a Triangle

Weak Robust: considering invalid values for a, b, and c we get following **additional** test cases.

Test Case	a	b	c	Expected Output
WR1	-1	5	5	Value of a is not in the range of permitted values
WR2	5	-1	5	Value of b is not in the range of permitted values
WR3	5	5	-1	Value of c is not in the range of permitted values
WR4	201	5	5	Value of a is not in the range of permitted values
WR5	5	201	5	Value of b is not in the range of permitted values
WR6	5	5	201	Value of c is not in the range of permitted values

Strong Robust : In addition to weak robust test cases, we can consider multiple fault assumption for invalid inputs for a, b, and c

Test Case	a	b	c	Expected Output
SR1	-1	5	5	Value of a is not in the range of permitted values
SR2	5	-1	5	Value of b is not in the range of permitted values
SR3	5	5	-1	Value of c is not in the range of permitted values
SR4	-1	-1	5	Values of a, b are not in the range of permitted values
SR5	5	-1	-1	Values of b, c are not in the range of permitted values
SR6	-1	5	-1	Values of a, c are not in the range of permitted values
SR7	-1	-1	-1	Values of a, b, c are not in the range of permitted values

Strong Normal :
Since a, b, c do not have valid subintervals, Strong normal test cases are same as weak normal

Strong Robust : In addition to weak robust test cases, we can consider multiple fault assumption for invalid inputs for a, b, and c

Equivalent Class Testing Does Not Cover....

- Many faults can happen around upper and lower limits of variable data values. For example, loop conditions (< instead of ≤), counters
- Equivalent partitioning does not specifically address them.
- So, to uncover faults around boundaries of equivalent classes => Boundary Values Analysis

What is Boundary Value Analysis?

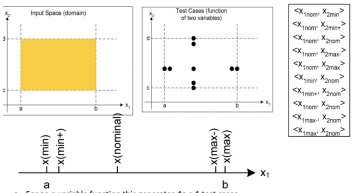
test cases for a variable x, where a <= x <= b

Method
- Hold all variables at their nominal value.
- Let one variable assume its boundary values.
- Repeat this for each variable.
- This will (hopefully) reveal all faults that can be attributed to a single variable

Boundary value analysis works well when the program to be tested is a function of several *independent* variables that represent bounded *physical* quantities

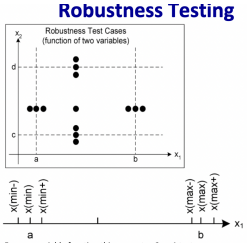
Boundary Value Analysis : Generating Test Cases

Let's consider a function of two variables, F(x1, x2)



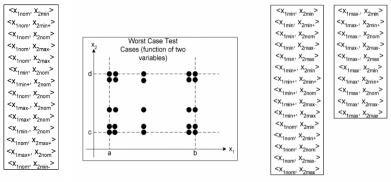
- For an n variable function this generates 4n + 1 test cases

Robustness Testing



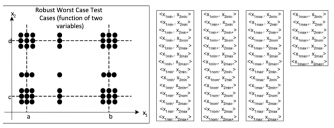
For an n variable function this generates 6n + 1 test cases

Worst Case Testing



For an n variable function this generates 5^n test cases
More than one variable has a boundary condition tested

Robust Worst Case Testing



- For an n variable function this generates 7 ^ n test cases

Determining Boundaries for Various Data Types : Guide Lines

Integer: One plus the max value and one less than the min values

Real: Depends on the decimal point values.

Character: Consider the ASCII sequence

Strings : consider the length or other character combination constrains. Sometimes, you may not be able to come up with one!

Boundary Value Based Test Cases for the Triangle Problem

Boundary conditions:
a={1 , 2, 100, 199, 200}
b={1 , 2, 100, 199, 200}
b={1 , 2, 100, 199, 200}

Table 3.3 Normal Boundary Value Test Cases

Case	a	b	c	Expected Output
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a Triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	100	100	Equilateral
9	100	199	100	Isosceles
10	100	200	100	Not a Triangle
11	1	100	100	Isosceles
12	2	100	100	Isosceles
13	100	100	100	Equilateral
14	199	100	100	Isosceles
15	200	100	100	Not a Triangle

Limitations of Boundary Value Analysis and Equivalent Partitioning

- Both equivalent partition and boundary value analysis test case generation focus on the input domain, output domain, runtime environment ...etc while designing test cases.
- Its quite likely that most of the logic paths of a complex program may not be executed and meaningful user interactions can easily be overlooked in this
- Sometimes it is more effective if test cases are generated by **combining inputs and outputs in more meaningful way** in the context of the application being tested.
- This technique is called the decision tables

Decision Table

- Originally known as Cause and Effect Graphing
  - Done with a graphical technique that expressed AND-OR-NOT logic.
  - Causes and Effects were graphed like circuit components
  - Inputs to a circuit "caused" outputs (effects)
- Equivalent to forming a decision table in which:
  - inputs are conditions
  - outputs are actions
- Test every (possible) rule in the decision table.
- Recommended for logically complex situations.

Decision Table Terminology

	Stub	Entry					
Conditions	c1	True		False	True		False
	c2	True	False		True	False	
	c3	T	F	--	T	F	--
Actions	a1	X	X		X		
	a2	X				X	
	a3		X		X	X	
	a4			X			X

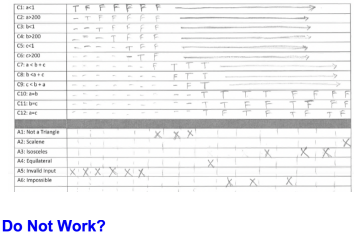
Decision Table – Triangle Problem

c1: a<b+c?	F	T	T	T	T	T	T	T	T	T
c2: b<a+c?	—	F	T	T	T	T	T	T	T	T
c3: c<a+b?	—	—	F	T	T	T	T	T	T	T
c4: a = b?	—	—	—	T	T	T	T	F	F	F
c5: a = c?	—	—	—	T	T	F	F	T	T	F
c6: b = c?	—	—	—	T	F	T	F	T	F	F
a1: Not a triangle	x	x	x							
a2: Scalene										x
a3: Isosceles								x	x	x
a4: Equilateral					x					
a5: Impossible						x	x		x	

Corresponding Test Cases

Case ID	a	b	c	Expected Output
DT1	4	1	2	Not a Triangle
DT2	1	4	2	Not a Triangle
DT3	1	2	4	Not a Triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

Triangle Problem – Decision Table (Length Restriction)



Decision Table for isLeap

Conditions	r1	r2	r3	r4	r5	r6	r7	r8
C1. year is a multiple of 4	T	T	T	T	F	F	F	F
C2. year is a century year	T	T	F	F	T	T	F	F
C3. year is a multiple of 400	T	F	T	F	T	F	T	F
Actions								
(logically impossible)			X		X	X	X	
A1. year is a common year		X						x
A2. year is a leap year	X			X				
test case: year =	2000	1900		2004				2007

Decision Table-Based Test Cases for isLeap

Test Case	Decision Table Rule	Year	Expected Output
1	1	2000	True
2	2	1900	False
3	3	?	Logically (numerically) impossible
4	4	2004	True
5	5	?	Logically (numerically) impossible
6	6	?	Logically (numerically) impossible
7	7	?	Logically (numerically) impossible
8	8	2007	False

Last Lecture: Decision Table

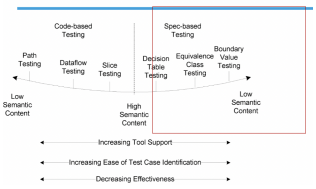
- Originally known as Cause and Effect Graphing
  - Done with a graphical technique that expressed AND-OR-NOT logic.
  - Causes and Effects were graphed like circuit components
  - Inputs to a circuit "caused" outputs (effects)
- Equivalent to forming a decision table in which:
  - inputs are conditions
  - outputs are actions
- Test every (possible) rule in the decision table.
- Recommended for logically complex situations.

When Decision Tables Do Not Work?

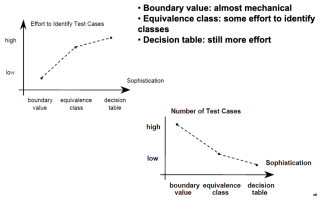
Well When...
- **The order of inputs is important.**
- **Time duration of input events is important.**
- **Times at which inputs occur is important.**

Special Considerations are needed When...
- **Conditions are mutually exclusive.**
- **There are many conditions. (n boolean conditions determine a table with 2^n rules!)**

The Test Method Pendulum



Test case development effort



- **Boundary value: almost mechanical**
- **Equivalence class: some effort to identify classes**
- **Decision table: still more effort**

Test Case Effectiveness

True trade-off between development effort and number of test cases.
Vulnerabilities
- Boundary value testing has gaps and redundancies, and many test cases.
- Equivalence class testing eliminates the gaps and redundancies, but cannot deal with dependencies among variables.
- Decision table testing extends equivalence class testing by dealing with dependencies, and supports algebraic reduction of test cases.

So, we need to apply these techniques in mix in most of the situation. See **Triangle problem solution.**

Triangle Problem Analysis

Triangle Problem
Decision Table Based
Test Cases : 14
Logic path coverage : 28/28

Equivalent Partitions
Weak Robust + Strong Normal
Test cases = 10
Logic Path Coverage = 24/28

Weak Robust + Strong Normal + BV
Test cases = 23
Logic Path Coverage = 28/28

Conclusion
More dependencies among inputs (Triangle Inequality in addition to valid and invalid). DT is good a covering such dependencies

Note: These numbers are from the code coverage of triangle Junit test cases

Limitations of BVA, EP, and DT

Program Behaviors :
- In general, software can be classified into two possible groups: **stateless and state-oriented.**
- Actions of state-oriented systems depend on the previous inputs/outputs of the system.
- So, in a situation where the system is state-oriented, we need to have a way to model the behavior of the system.
- Several behavior modeling approaches are being used in software testing : FSM (Finite State Model), Petri-Net, and StateCharts are more common

Limitations of BVA, EP, and DT

Common System Behavior Models

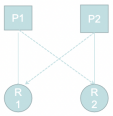
Behavior	Decision Table	Finite State Machine (FSM)	Petri-Net
Sequence	No	Yes	Yes
Selection	Yes	Yes	Yes
Repetition	No	Yes	Yes
Multiple-causes of output	Yes	Yes	Yes
Mutual Exclusion	No	No	Yes
Concurrency	No	No	Yes
Deadlock	No	No	Yes

Concurrency, Mutual Exclusion, and Deadlocks

Mutual Exclusion

- file sharing by reading and writing processes
- OS interrupt handling

if lock is not set
then set the lock
  enter critical section ...
  leave critical section
unset the lock



Finite State Model

Finite State Machines (FSM)

- A *Finite State Machine (FSM)* is a directed graph in which nodes are states and edges are transi
- States can be used to represent
  - Conditions
  - Stages of processing
- Transitions are caused by
  - Events
  - Data
  - Passage of time
- A well-formed FSM has one initial state and one final state
- Paths from the initial to the final state are test cases

FSM Notation

- **States are circles**
  - **s1, s2, s3, s4**
- **Events (e1, e2, e3)**
- **Actions (a1, a2, a3)**
- **Transitions are arrows**
- **Transition notation: the transition from s1 to s2 is caused by event e1, and generates action a1.**
- **e1 is a context-sensitive input event**

Deterministic and Non-Deterministic Nature of FSM

Two Definitions of "Deterministic"

- A program (or model) is *deterministic* if:
  - 1. Given the same input, it always produces the same output
  - OR
  - 2. Given the same input, it always produces a predictable output.

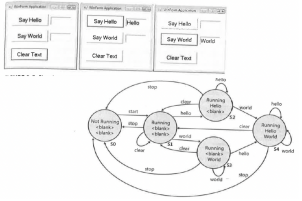
- A non-deterministic model cannot be tested using this technique.
- Why?

Test Coverage Based on Finite State Machines

Develop a set of test cases, such that:

- Random Walk
- Weighted Traversal
- Shortest Path
- Every state is traversed
- Every transition is traversed

Program to Finite State Model



Test Cases for the HelloWorld Example

Test step	Action	Text box 1	Text box 2
0	Start Application	Empty	Empty
1	Say World	Empty	World
2	Clear	Empty	Empty
3	Say Hello	Hello	Empty
4	Say Hello	Hello	Empty
5	Clear	Empty	Empty
6	Say World	Empty	World
7	Say Hello	Hello	World
8	Say World	Hello	World
9	Clear	Empty	Empty
10	Stop Application	Empty	Empty

Reporting Bugs – Anatomy of a Bug Report

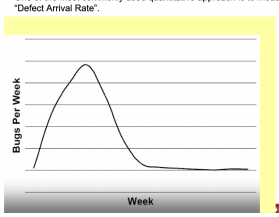
- Reporting bugs is an important process. Bug handing involves many and credible bug reporting is imperative.
- Your bug report should have necessary information to understand and fix the bug.
- Typically a bug report consists of
  - a) Title
  - b) Description
  - c) Status
  - d) Version Number
  - e) Feature Area
  - f) Reproduction steps
  - g) Assignment
  - h) Severity
  - i) Customer Impact
  - j) Environment
  - k) Resolution

Bug Workflow

1. Report the bug
2. Confirm the bug - Bug Verification
3. Programmer evaluates the bug and fixes it, or not to fix it due to various reasons such as bug is not reproducible, not a credible bug…etc
   - The process of deciding which bugs to fix and which to leave in the product is called **bug triage**.
4. Programmer decides not to fix it and sends it to the project manager
5. If testers and programmers disagree whether a bug should be fixed: Send it to a project team (representatives of the key stakeholder groups) reviews bugs. Such a team is called a **Triage team**
6. If the bug is fixed, the programmers mark the bug fixed in the bug tracking system

How Much Testing Should Be Performed?

- Various parameters are used to determine if the product is ready to be delivered. One of the most commonly used quantitative approach is to measure the "Defect Arrival Rate".



The Weibull Curve

http://testing education.org/BBST

What We Have Discussed So Far?

- All the testing techniques that we have discussed so far are based on either inputs, outputs, or the behavior of the software.

- It is quite likely, many defects are hidden in the software after applying some of above testing techniques as needed.
- Experimental results show that code based (structural testing) can achieve the practical testable limit .

What is Code Based Testing ?

- Simply put, code based testing involves developing test cases and testing the software system based on the code.
- Thus, the tester has access to the code and knowledgeable about the programming language used to develop the software.
- Code based testing focus on covering the code in various angles. We call is test coverage.

Code Based Testing Strategies:

a) Statement coverage
b) Basic path testing (Control flow coverage)
c) Decision-to-Decision testing
d) Dataflow coverage
e) Condition testing
f) Code based risk assessments (complexity of the code, object oriented matrices …etc)

```
int X=0, Y=10;
if(X>0)
X = X + Y;
else
X = X – Y;
cout<<X;
```

Program as a Graph

- Given a program written in an imperative programming language, its *program graph* is a directed graph in which…

nodes are either entire statements or fragments of a statement, and edges represent flow of control (there is an edge from node i to node j iff the statement (fragment) corresponding to node j can be executed immediately after the statement or statement fragment corresponding to node i)

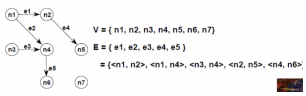
Directed Graphs

9. A directed graph (or digraph) D = (V, E) consists of:

a finite set V = { n(1), n(2), … n(k) } of nodes (or vertices), and

a set E = { <n(i), n(j)>, where i ≠ j and 1 <= i, j <= k} of edges.

Notice that the (undirected) edges of an ordinary graph become ordered pairs of nodes. We say that a directed edge goes from its starting node to its finish node.



V = { n1, n2, n3, n4, n5, n6, n7}

E = { e1, e2, e3, e4, e5 }

= {<n1, n2>, <n1, n4>, <n3, n4>, <n2, n5>, <n4, n6>

Indegrees and Outdegrees

10. The indegree of a node in a directed graph is the number of distinct edges that have the node as an endpoint. We write indeg(n)

11. The outdegree of a node in a directed graph is the number of distinct edges that have the node as an start point. We write outdeg(n)

indeg(n1) = 0 outdeg(n1) = 2
indeg(n2) = 1 outdeg(n2) = 1
indeg(n3) = 0 outdeg(n3) = 1
indeg(n4) = 2 outdeg(n4) = 1
indeg(n5) = 1 outdeg(n5) = 0
indeg(n6) = 1 outdeg(n6) = 0
indeg(n7) = 0 outdeg(n7) = 0

Types of Nodes

12. A node with indegree = 0 is a source node.
13. A node with outdegree = 0 is a sink node.
14. A node with indegree ≠ 0 and outdegree ≠ 0 is a transfer node. (AKA an interior node)

n1, n3 and n7 are source nodes,

n5, n6 and n7 are sink nodes,

n2 and n4 are transfer nodes.

Adjacency Matrix of a Digraph

15. The adjacency matrix of a directed graph D = (V, E) with m nodes is an m x m matrix A = (a(i, j)) , where a(i, j) is a 1 if and only if there is an edge from node i and node j, otherwise the element is 0.

The adjacency matrix of a directed graph is not necessarily symmetric . A row sum is the outdegree of the node; a column sum is the indegree of a node.

	n1	n2	n3	n4	n5	n6	n7
n1	0	1	0	1	0	0	0
n2	0	0	0	0	1	0	0
n3	0	0	0	1	0	0	0
n4	0	0	0	0	0	1	0
n5	0	0	0	0	0	0	0
n6	0	0	0	0	0	0	0
n7	0	0	0	0	0	0	0

Reachability Matrix

18. The reachability matrix of a directed graph D = (V, E) with m nodes is an m x m matrix R = (r(i, j)) , where r(i, j) is a 1 if and only if there is a true path from node i and node j, otherwise the element is 0.

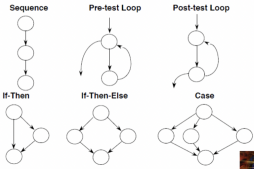
	n1	n2	n3	n4	n5	n6	n7
n1	0	1	0	1	1	1	0
n2	0	0	0	0	1	0	0
n3	0	0	0	1	0	1	0
n4	0	0	0	0	0	1	0
n5	0	0	0	0	0	0	0
n6	0	0	0	0	0	0	0
n7	0	0	0	0	0	0	0

Cyclomatic Number of a Graph

Here, V(G) is 3

Cyclomatic Number V(G) of a graph is equal to number of simple decision nodes + 1

Program Graphs of Structured Programming Constructs



- Code based testing focus more on the statements, logic paths, and data in a program under test and test cases are directly based on testing the correctness of them.

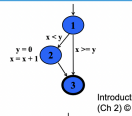
Code Based Testing Strategies:

a) Basic path testing (Control flow coverage)
b) Statement coverage
c) Decision-to-Decision testing
d) Dataflow coverage
e) Condition testing
f) Code based risk assessments (complexity of the code, object oriented matrices …etc)

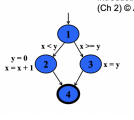
- Most of these testing techniques see the program as a graph.

Control Flow Graph Example : The if, and if-else Statements

```
if (x < y)
{
  y = 0;
  x = x + 1;
}
```



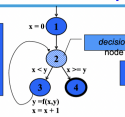

```
if (x < y)
{
  y = 0;
  x = x + 1;
}
else
{
  x = y;
}
```



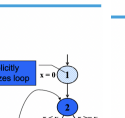
Introduction to Software Testing (Ch 2) © Ammann & Offutt

Control Flow Graph Example : while and for Loops

```
x = 0;
while (x < y)
{
  y = f (x, y);
  x = x + 1;
}
```




```
for (x = 0; x < y; x++)
{
  y = f (x, y);
}
```

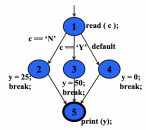


Introduction to Software Testing (Ch 2) © Ammann & Offutt

Janaka Balasooriya

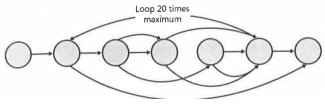
Control Flow Graph Example : The case (switch) Structure

```
read ( c) ;
switch ( c )
{
  case 'N':
    y = 25;
    break;
  case 'Y':
    y = 50;
    break;
  default:
    y = 0;
    break;
}
print (y);
```

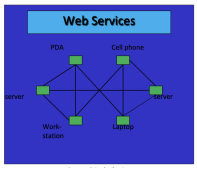


Introduction to Software Testing (Ch 2) © Ammann & Offutt

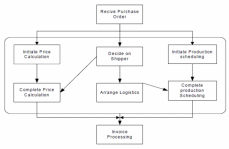
How many Independent Paths Here?



Web Service Architecture

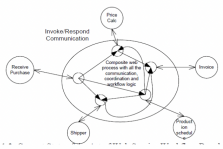


Web Service Application Example



Purchase Order Workflow

Web Service Application Example



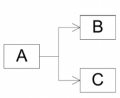
Service Order Workflow Implementation

Main differences Between Component Based and SOA applications

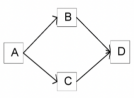
- Component based systems are usually tightly-coupled.
- Web service applications are loosely coupled and the implementation/design is not available most of the time.
- Web services can be dynamically bound at runtime

Different Set of Control Logic – Workflow control flow patterns

http://www.workflowpatterns.com/patterns/control/



AND-Split



Synchronous merge

Levels of Testing in SOA Applications

- Test web services – at web service level
  - Availability
  - Response time
  - Correctness of output
  - Other quality factors …..

Web Service Composition Level Testing

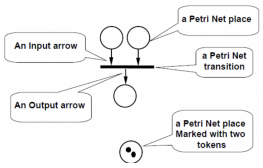
- Testing web service composition logic
- How to model the behavior of a composite web service?

Behavior	Decision Table	Finite State Machine (FSM)	Petri-Net
Sequence	No	Yes	Yes
Selection	Yes	Yes	Yes
Repetition	No	Yes	Yes
Multiple-causes of output	Yes	Yes	Yes
Mutual Exclusion	No	No	Yes
Concurrency	No	No	Yes
Deadlock	No	No	Yes

Petri-Net model for SOC Applications

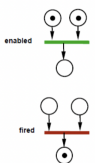
- **Subject of Carl Adam Petri's dissertation in 1963.**
- **A *Petri Net* is a bipartite directed graph, consisting of two sets of nodes (places, and transitions) and edges showing places that are either inputs to, or outputs of transitions.**
- **Formally, a *Petri Net* is a four-tuple (P, T, In, Out) where**
  - **P is a non-empty set of places**
  - **T is a non-empty set of transitions**
  - **In is a mapping from P to T, i.e., a subset of P x T.**
  - **Out is a mapping from T to P, i.e., a subset of T x P.**

Petri Net Drawings



Transition Firing

1. **A transition is *enabled* if each of its input places is marked.**
2. **When a *transition fires*, a token is removed from each of its input places, and a token is put in each of its output places.**
3. **Analyses:**
   - **conflict**
   - **liveness**
   - **deadlock**
   - **token conservation**

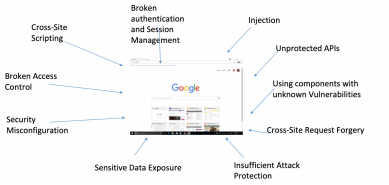


Usage Suggestions

1. Use transitions to represent actions.
2. Use places to represent any of the following:
   - data
   - conditions (pre- and post-)
   - states
   - messages
   - events
3. Use the Input relationship to represent prerequisites and inputs.
4. Use the Output relationship to represent consequences and outputs.
5. Use markings to represent "states" of a net, memory, counters.
6. Tasks can be considered to be individual transitions or subnets.

Security Vulnerabilities of Web Applications

According to Open Web Application Security Project, 10 diffident vulnerabilities are identified. When security testing is carried out, these categories can be considered as classes of inputs for security tests, similar to equivalent partitions.



Injection

In general, injection is the term use to describe the scenario where

a) user input for the web application is altered

b) give an input that can harm the web application/access resources intentionally or data in the web application without proper authentication.

c) Sql injection is one of the common issues related web applications security. It is an application security weakness that allows attackers to control the web application's database such as unauthorized login, delete/alter tables ...etc. This happens due to the nature of how we write SQL queries and how databases process SQL queries

SQL Injection Examples:

1) Returning more data than expected:

```
SELECT accountNumber, balance FROM accounts WHERE account_owner_id = 984
```

If we change the query so that

```
SELECT accountNumber, balance FROM accounts WHERE account_owner_id = 0 OR 1=1
```

2) Multi-Statement (S1; S2)

```
SELECT * FROM `table`; DROP TABLE `table`;
```

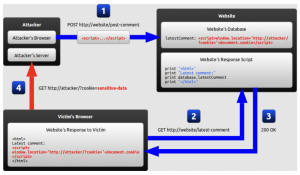
Preventing SQL Injection

Using prepared statements in the server-side code:

```
String accountBalanceQuery =
  "SELECT accountNumber, balance FROM accounts WHERE account_owner_id = ?";

try {
   PreparedStatement statement = connection.prepareStatement(accountBalanceQuery);
   statement.setInt(1, request.getParameter("user_id"));
   ResultSet rs = statement.executeQuery();
   while (rs.next()) {
      page.addTableRow(rs.getInt("accountNumber"), rs.getFloat("balance"));
   }
} catch (SQLException e) { ... }
```

XSS : Cross-Site Scripting

Cross-site scripting (XSS) is a code injection attack that allows an attacker to execute malicious script in another user's browser. It essentially, helps the attacker to gain access to a website through another user.

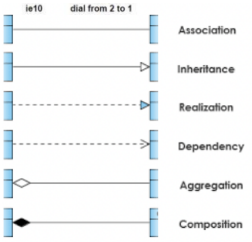


The attacker uses one of the website' forms to insert a malicious string into the website's database.

The victim requests a page from the website.

The website includes the malicious string from the database in the response and sends it to the victim.

The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server.



UML Support for Integration Testing:

- In OO integration testing, sequence diagrams and collaboration diagrams are the basic for integration testing

Quick Review Sequence Diagram:

- Sequence diagrams describes the **dynamic behavior of objects**
- It describes **interactions between objects by depicting the time ordering**. In general, each use case has one of more sequence diagrams.

The windshield wiper on the Saturn automobiles is controlled by a lever with a dial. The lever has four positions, OFF, INT (for intermittent), LOW, and HIGH, and the dial has three positions, numbered simply 1, 2, and 3. The dial positions indicate three intermittent speeds, and the dial position is relevant only when the lever is at the INT position. The decision table below shows the windshield wiper speeds (in wipes per minute) for the lever and dial positions.

Junit 4.x for testing programs

Import the JUnit 4 classes you need

```
import org.junit.*;
import static org.junit.Assert.*;
```

Declare your (conventional) Java class

```
public class MyProgramTest {  }
```

Declare any variables you are going to use, e.g., an instance of the class being tested

```
MyProgram program;
int [ ] array;
int solution;
```

Junit 4.x for testing programs

If needed, define *one* method to be executed *just once,* when the class is first loaded. For instance, when we need to connecting to a database

```
@BeforeClass
public static void setUpClass() throws Exception {
    // one-time initialization code
}
```

If needed, define *one* method to be executed *just once,* to do cleanup after all the tests have been completed

```
@AfterClass
public static void tearDownClass() throws Exception {
    // one-time cleanup code
}
```

Junit 4.x for testing programs

If needed, define *one or more* methods to be executed before each test, e.g., typically for initializing values

```
@Before
public void setUp() {
    program = new MyProgram();
    array = new int[] { 1, 2, 3, 4, 5 };
}
```

If needed, define *one or more* methods to be executed after each test, e.g., typically for releasing resources (files, etc.)

```
@After
public void tearDown() {
}
```

- Software quality and quality assurance activities
- So far, we focused on dynamic testing where code / software is available and test cases are designed and executed against the code.

- Relative strengths and weaknesses of software testing approaches.
- Now its time for us to think about few other aspects of software quality assurance.

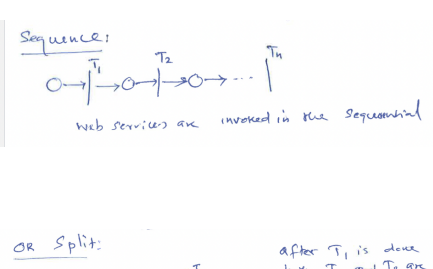
```
<suite name="Test-method Suite" parallel="methods" thread-count="2">
    <test name="Test-method test" group-by-instances="true">
        <classes>
            <class name="ngTest" />
        </classes>
    </test>
</suite>
```

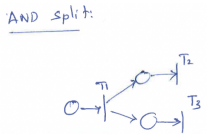
```
import static org.junit.Assert.*;
import org.junit.Test;
import static org.hamcrest.CoreMatchers.*;
import static org.hamcrest.MatcherAssert.assertThat;

public class datapathTesting {

	@Test
	public void path1() {
		String data ="";
		int[] testData = {};
		Ismod10.IsValidMod10Number(data);
		try{
		assertEquals(0, Ismod10.getTotalSum());
		}catch(AssertionError  e)
		{
			System.out.println(e.toString());
		}
		try{
		// report only the first mismatch
		assertArrayEquals(testData, Ismod10.getNumArray());
		}catch(AssertionError  e)
		{
			System.out.println(e.toString());
		}
	}
```

```
import static org.testng.AssertJUnit.assertEquals;
import org.testng.annotations.Test;

public class ngTest {
	@Test
	public void T1() {
		assertEquals(1, cse360demo.f1(1, 1));
		long id = Thread.currentThread().getId();
		System.out.println("T1 Thread id is: " + id);
	}

	@Test
	public void T3() {
		assertEquals(1, cse360demo.f1(1, 2));
		long id = Thread.currentThread().getId();
		System.out.println("T3 Thread id is: " + id);
	}

	@Test
	public void T2() {
		assertEquals(1, cse360demo.f1(2, 1));
		long id = Thread.currentThread().getId();
		System.out.println("T2 Thread id is: " + id);
	}
}
```

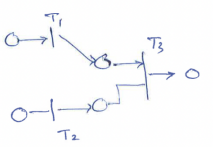
```
import static org.junit.jupiter.api.Assertions.*;
import java.util.stream.Stream;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
import org.junit.jupiter.params.provider.MethodSource;
import org.junit.jupiter.params.provider.ValueSource;

class loanTestParameter {

	@Test
	void oddEvenTest1() {
		assertEquals(true, loanCalc.oddEven(10));
	}

	@ParameterizedTest
	@ValueSource(ints = { 10, 7, 16 })
	void oddEvenTest2(int nums) {
	    assertTrue(loanCalc.oddEven(nums));
	}

	private static Object[] inputs()
	{
		return new Object[][] {
			{2000.50,600,7},
			{30000,650,5},
			{90000,800,5}

		};

	}
```

```
@ParameterizedTest
@MethodSource("inputs")
void loanTest1(double n1, int n2, int n3) {
    assertEquals(n3, loanCalc.calculateRate(n1, n2));
}


@ParameterizedTest
@CsvSource({
                "2000.50,600,7",
                "30000,650,5",
                "90000,800,5"
        }
                )
void loanTest2(double n1, int n2, int n3) {
    assertEquals(n3, loanCalc.calculateRate(n1, n2));
}
```

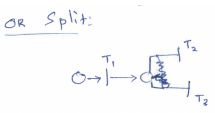




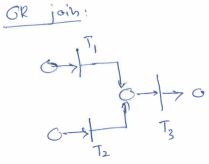
when T_1 is done both T_2 and T_3 will be executed in parallel (splitting the control to both web services at the same)



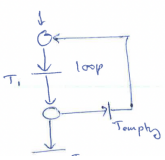
T_3 will be executed once T_1 and T_2 (both) are done.



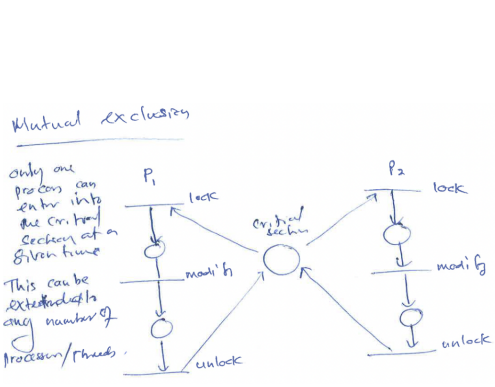
after T_1 is done both T_2 and T_3 are enabled, but only one of them can fire.

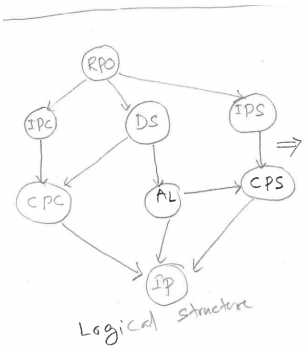


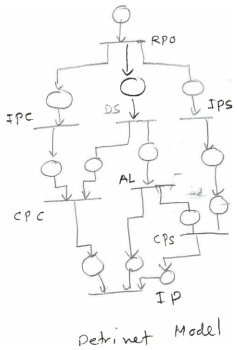
T_3 will be enabled with either T_1 or T_2 firing. So, T_3 doesn't need to wait for both.

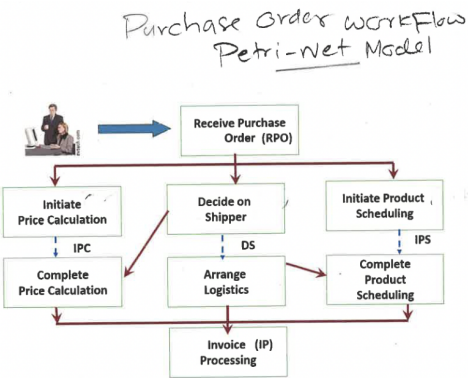


when T_1 is done it will be enabled again if Temp0 fires.

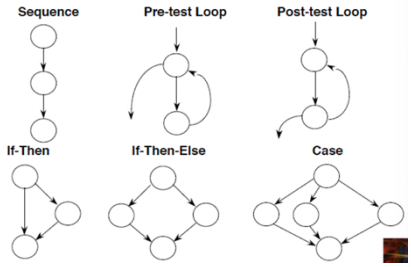




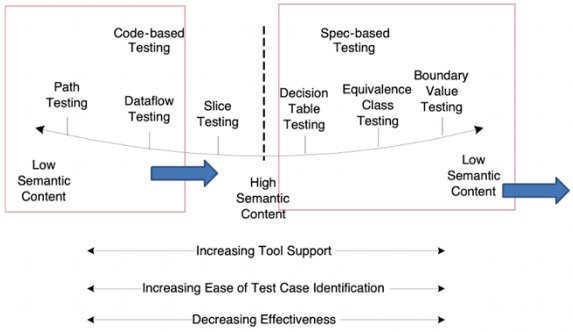




- **Code Based Testing Strategies:** Statement coverage, Basic path testing (Control flow coverage), Decision-to-Decision testing, Dataflow coverage, Condition testing, Code based risk assessments (complexity of the code, object-oriented matrices ...etc.)
- **Program as a graph:** Source node has no arrows entering it; sink node has no arrows leaving it; All others are transfer nodes
- **Cyclomatic Number V(G):** Make sure graph has 1 source and 1 sink, then use Edges – Nodes + 2
- **Graphs of structured programming constructs:**

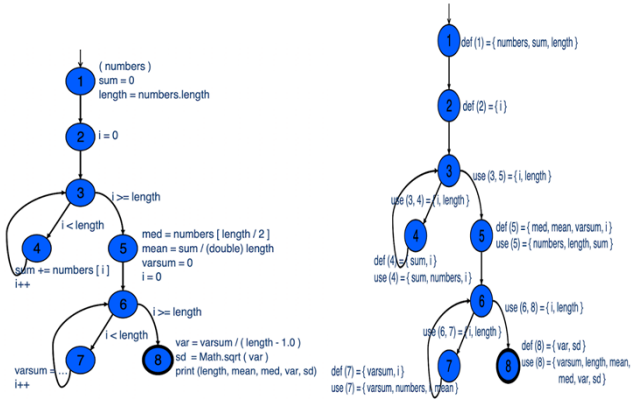


- **Test Method Pendulum:**



- **D(definition)-U(use) Path Testing:** Definition = change of data stored in var; Use = usage of data stored in var
- **D-U Example:**

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0.0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2 ];
    mean = sum / (double) length;

    varsum = 0.o;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:              " + length);
    System.out.println ("mean:                " + mean);
    System.out.println ("median:              " + med);
    System.out.println ("variance:            " + var);
    System.out.println ("standard deviation: " + sd);
}
```



Node	Def	Use
1	{ numbers, sum, length }	{ numbers }
2	{ i }	
3		{i, length}
4	{ sum, i }	{ numbers, i, sum }
5	{ med, mean, varsum, i }	{ numbers, length, sum }
6		
7	{ varsum, i }	{ varsum, numbers, i, mean }
8	{ var, sd }	{ varsum, length, var, mean, med, var, sd }

variable	DU Pairs
numbers	(1, 4) (1, 5) (1, 7)
length	(1, 5) (1, 8) (1, (3,4)) (1, (3,5)) (1, (6,7)) (1, (6,8))
med	(5, 8)
var	(8, 8)
sd	(8, 8)
mean	(5, 7) (5, 8)
sum	(1, 4) (1, 5) (4, 4) (4, 5)
varsum	(5, 7) (5, 8) (7, 7) (7, 8)
i	(2, 4) (2, (3,4)) (2, (3,5)) (2, 7) (2, (6,7)) (2, (6,8)) (4, 4) (4, (3,4)) (4, (3,5)) (4, 7) (4, (6,7)) (4, (6,8)) (5, 7) (5, (6,7)) (5, (6,8)) (7, 7) (7, (6,7)) (7, (6,8))

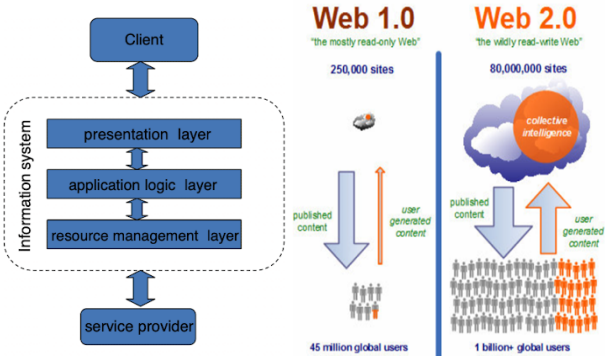
variable	DU Pairs	DU Paths
numbers	(1, 4)	[1, 2, 3, 4]
	(1, 5)	[1, 2, 3, 5]
	(1, 7)	[1, 2, 3, 5, 6, 7]
length	(1, 5)	[1, 2, 3, 5]
	(1, 8)	[1, 2, 3, 5, 6, 8]
	(1, (3,4))	[1, 2, 3, 4]
	(1, (3,5))	[1, 2, 3, 5]
	(1, (6,7))	[1, 2, 3, 5, 6, 7]
	(1, (6,8))	[1, 2, 3, 5, 6, 8]
med	(5, 8)	[5, 6, 8]
var	(8, 8)	No path needed
sd	(8, 8)	No path needed
sum	(1, 4)	[1, 2, 3, 4]
	(1, 5)	[1, 2, 3, 5]
	(4, 4)	[4, 3, 4]
	(4, 5)	[4, 3, 5]

variable	DU Pairs	DU Paths
mean	(5, 7)	[5, 6, 7]
	(5, 8)	[5, 6, 8]
varsum	(5, 7)	[5, 6, 7]
	(5, 8)	[5, 6, 8]
	(7, 7)	[7, 6, 7]
	(7, 8)	[7, 6, 8]
i	(2, 4)	[2, 3, 4]
	(2, (3,4))	[2, 3, 4]
	(2, (3,5))	[2, 3, 5]
	(4, 4)	[4, 3, 4]
	(4, (3,4))	[4, 3, 4]
	(4, (3,5))	[4, 3, 5]
	(5, 7)	[5, 6, 7]
	(5, (6,7))	[5, 6, 7]
	(5, (6,8))	[5, 6, 8]
	(7, 7)	[7, 6, 7]
	(7, (6,7))	[7, 6, 7]
	(7, (6,8))	[7, 6, 8]

- **Static Testing:** inspect and review various artifacts of software without executing code; NOT replacement for dynamic testing
- **Code Reviews:** Management= systematic eval of software acquisition, supply, dev, operation, or maintenance process to determine status of plans/schedules, confirm requirements and system allocation, or eval effectiveness of management approaches used. Technical= Eval product to determine suitability for intended use. ID discrepancies from specs and standards. Inspection= Formal eval where software requirements, design, or code are examined in detail by person(s) other than the author, also code/docs given to reviewers beforehand. Walkthrough= formal process where author formally presents requirements, design, or code to small group of reviewers who listen and ask questions. Audit= Independent examination of software product, process, or set of processes to assess compliance with specs, standards, contracts, etc.
- **Informal (Technical) Reviews:** include simple desk check of work with colleague, casual meeting (2+ ppl) for reviewing product, or review-oriented aspects of pair programming (encourages continuous review as product is created, benefit is immediate discovery of errors and better product quality)
- **Formal Technical Reviews (FTR):** A class of reviews that includes walkthroughs and inspections. Objectives: uncover errors in function, logic, or implementation for any representation of the software, verify that software meets its requirements, ensure that the software has been represented according to predefined standards, achieve software that is developed in uniform manner, make projects more manageable
- **Review Meeting:** 3-5ppl typically, prep shouldn't be 2+hrs, meeting duration < 2hr, focus on a work product(portion of requirements model, detailed component design, source code for component). **Ppl involved**: Producer= individual who developed product, informs project leader that work product is complete and that review is required. Review Leader= evals product for readiness, generates copies of

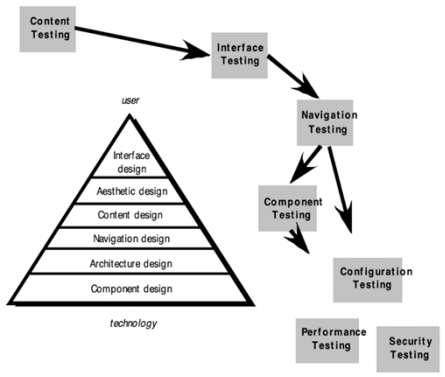
product materials and distributes to reviewers for meeting prep. Reviewer(s)= should spend 1-2hr reviewing product, make notes, and become familiar with the work. Recorder= reviewer who records all important issues raised

- **Conducting Review:** Review the product, not the producer. Set an agenda and maintain it. Limit debate and rebuttal. Enunciate problem areas, but don't attempt to solve every problem noted. Take written notes. Limit the number of participants and insist upon advance preparation. Develop a checklist for each product that is likely to be reviewed. Allocate resources and schedule time for FTRs. Conduct meaningful training for all reviewers. Review your early reviews.
- **Code Writing Errors:** data declaration errors, computation errors, comparison errors, control flow errors
- **Code Design Issues:** duplicated code, long methods, large class, long parameter list, temporary field
- **Code Complexity Risk:** 1-10=simple module, little risk; 11-20=moderate risk; 21-50=complex module, high risk; 51+=almost untestable, very high risk
- **Fundamental OO Concepts:**
  - Abstraction(data, interface, control): information hiding, inheritance and polymorphism
  - Modularity(compartmentalization of data and function): functional independence
  - Architecture(overall structure of software): patterns
- **Why information hiding?:** reduced likelihood of "side effects", limits global impact of local design decisions, emphasizes communication through controlled interfaces, discourages use of global data, leads to encapsulation(good), results in higher quality software
- **Cohesion:** the degree to which a module performs one and only one function (high = good)
- **Coupling:** the degree to which a module is "connected" to other modules in the system (low = good)
- **Types of Coupling:** Content= one module directly connected to the inner working of another module. Common= two modules share a global data item. Control= data in one module is used to determine the order of execution of another module. Stamp= one module passes non-global data structures/objects to another module. Data= one module passes elementary data types as parameters to the other.
- **Coupling Spectrum:** (least) data, stamp, control, common, content (most)

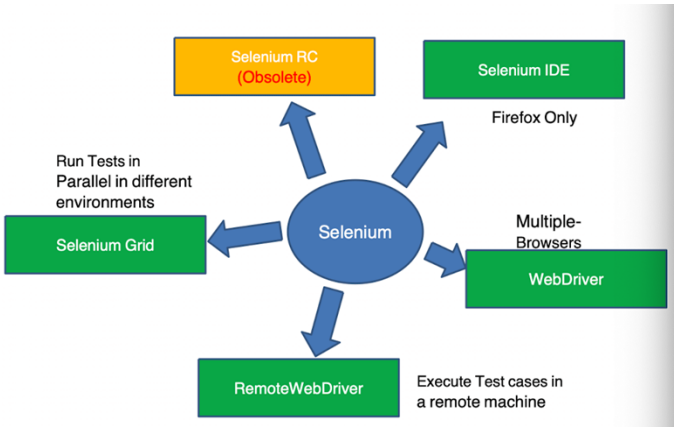


- **Web App Categories:** Client server, SOA, Cloud

- **Testing Web Apps: Bigger picture**

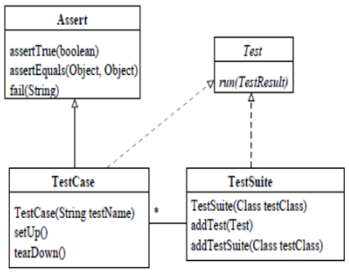


- **Gray-Box Testing:** Most appropriate technique for web apps. Incorporates elements of both black and white box. It considers testing outcome on end user, system specific technical knowledge, operating env, app design in context of interoperability of system components
- **Selenium:** Toolkit that can automate web app testing, can be run on multiple browsers and several languages, has assertion statements.

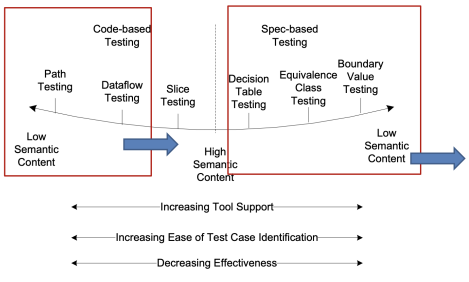


- **Verification/Assertions:** Selenese(Lang selenium uses) allows multiple ways of checking for UI elements. These are used to check if an element/text is present somewhere on the page, specific text is at a specific location, etc. NOT THE SAME, if assertion fails, script will be aborted but if verification fails script will continue.
- **Limitations of Selenium:** no programming logic(loops, conditionals), can only execute scripts in selenese, difficult to use for checking complex test cases involving dynamic contents
- **Junit annotations to provide resource initialization and reclamation methods:** @Before, @BeforeClass, @After, @AfterClass. A variety of assert methods make it easy to check results of tests. Integrated with popular IDEs

The `junit.framework` package contains a simple class hierarchy for developing unit tests:



The Test Method Pendulum



IEEE standard for software review identify four different types of reviews

a) Management reviews
b) Technical reviews
c) Inspections
d) Walk-throughs
e) Audits

What do we review? : Requirements, Specification, Design, Code, User guides, Test cases, …etc

Data declaration errors
- Are the variables assigned the correct length, type, storage class?
  - E.g. should a variable be declared a string instead of an array of characters?
- If a variable is initialized at its declaration, is it properly initialized and consistent with its type?

Computation errors
- Do any calculations that use variables have different data types?
  E.g., add a floating-point number to an integer
- Do any calculations that use variables have the same data type but are different size?
  E.g., add a long integer to a short integer
- Is overflow or underflow in the middle of a numeric calculation possible?

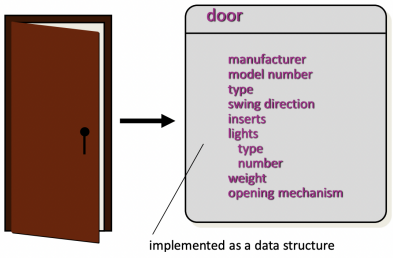
Comparison errors
- Are the comparisons correct?
  E.g., < instead of <=
- Properly comparing decimal values?
  Use EPSILON

Control flow errors
- Do the loops terminate? If not, is that by design?
- Does every `switch` statement have a `default` clause/break after cases?

- **Duplicated Code**
  - **bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!**
- **Long Method**
  - **long methods are more difficult to understand**
- **Large Class**
  - **classes try to do too much, which reduces cohesion (will talk about cohesion and coupling next time)**
- **Long Parameter List**
  - **hard to understand, can become inconsistent**
- **Temporary Field**
  - **An attribute of an object is only set in certain circumstances; but an object should need all of its attributes**

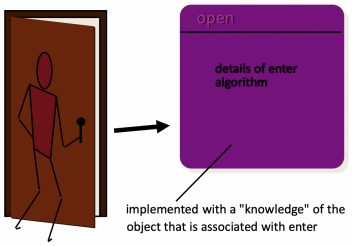
1. abstraction—data, interface, control
   - Information hiding—controlled interfaces
   - Inheritance and polymorphism
2. modularity—compartmentalization of data and function
   - Functional independence—single-minded function and low coupling
3. architecture—the overall structure of the software
   - patterns—"conveys the essence" of a proven design solution

Data Abstraction



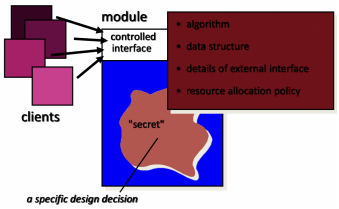
implemented as a data structure

Procedural Abstraction



implemented with a "knowledge" of the object that is associated with enter

Information Hiding



a specific design decision

OO Application Example : Inheritance and Polymorphism

- The term *polymorphism* literally means "having many forms"
- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time
- The method called through a polymorphic reference can change from one invocation to the next
- All object references in Java are potentially polymorphic

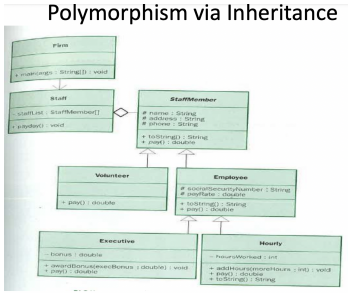
Polymorphism

- Suppose we create the following reference variable:

```
Vehicle car;
```

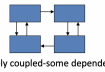
- This reference can point to a `Vehicle` object, or to any object of any compatible type such as **Car**
- This compatibility can be established using inheritance or using interfaces
- Careful use of polymorphic references can lead to elegant, robust software designs

Polymorphism via Inheritance



Coupling: Degree of dependence among components (objects)

No dependencies



Loosely coupled-some dependencies

highly coupled-many dependencies

High coupling makes modifying parts of the system difficult, e.g., modifying a component affects all the components to which the component is connected

rence:

Cohesion

The degree to which all elements of a component are directed towards a single task and all elements directed towards that task are contained in a single component.

```
class compXYX
{
  string x, y, z;
  int p, q;

  A(){  p = x.length();}

  B() { q = y.length() – z.length()}

}
```

Lack of Cohesion

Object Oriented Design Matrices

Chidamber and Kemerer (1998) proposed six desirable attributes known as C-K matrices of a good OO design.

– **Weighted methods per class (WMC) :**
Number of methods weighted by their procedural complexity (~complexity of class). Using unity weights gives simply number of methods in a class.
Difficult to set exact limits for metric, but WMC should be kept as low as possible

– **Depth of inheritance (DIT) :**
As DIT grows, it is likely that classes on lower level inherits lots of methods and overrides some. Thus predicting behavior for an object of a class becomes difficult. Max value recommended is 5

- **Number of children (NOC) :**

High value of NOC might indicate misuse of subclassing (=implementation inheritance instead of is-a relationship)
Class with very high NOC might be a candidate for refactoring to create more maintainable hierarchy. Max recommended is 10

- **Response for a class (RFC):**

Number of methods that can be invoked in response to a message
Measure of potential communication between classes
High RFC -> there could be a better class subdivision (e.g. merge classes) – More than 50 is not acceptable

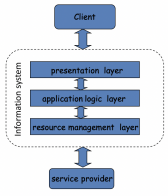
- **Coupling between object classes (CBO):**

coupling = class x is coupled to class y iff x uses y's methods or instance variables (includes inheritance related coupling)
High coupling between classes means modules depend on each other too much
High coupling makes maintenance more difficult since changes in a class might propagate to other parts of software

- **Lack of cohesion in methods (LCOM) :**

Cohesion measures "togetherness" of a class: high cohesion means good class subdivision (encapsulation)
LCOM counts the sets of methods that are not related through the sharing of some of the class's instance variables

Distributed Information Systems

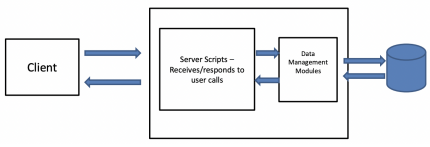


- Any information system has requesters and service providers.
- In general, information systems are designed around three layers: presentation layer, application layer, resource management layer.
- The way these layers are arranged between service provider and the service requester determine if it is 1-tier, 2-tier,3-tier or n-tier.

Web Application Categories

- Client server architecture
- SOA architecture
- Cloud computing

A Typical 3-tier Web Application



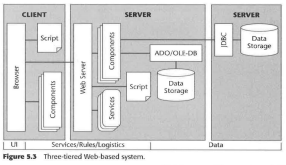


Figure 5.3 Three-tiered Web-based system.

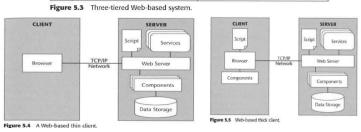
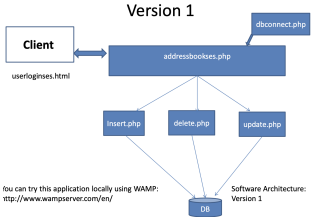


Figure 5.4 A Web-based thin client.

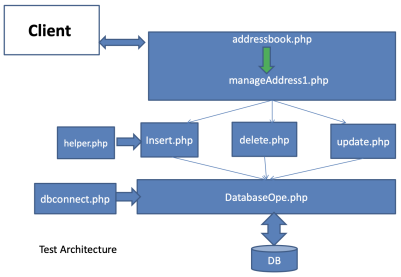
Issues in Testing a Typical Web Software

- Web applications are loosely coupled and components are distributed
- Data pertaining to one application is scattered among different devices
- Heterogeneity of devices, configurations and software systems
- Dynamic adaptation to local settings

Web Application Testing : Addressbook Version 1

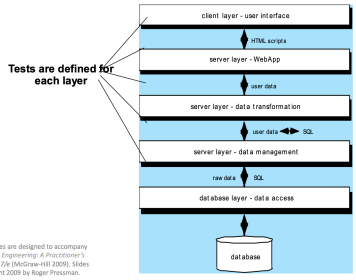


Addressbook Version 2 – Better Design

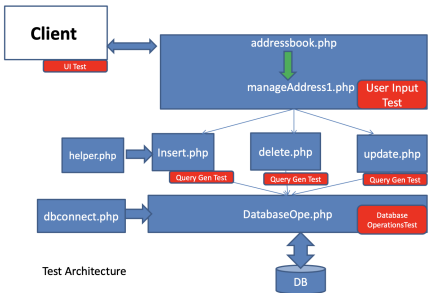


Test Architecture

Database Testing – A Generic View



Addressbook Test Architecture



Test Architecture

Tools in Testing Web Applications

- Some unit testing framework to test server side code (phpUnit, simpleTest …etc)
- Framework such as DBUnit to test database related functionalities.
- Tool such as Selenium to test user scenarios through UI.

What is Test Automation

- In general, test automation entails automating a manual process that uses a **formalized testing process**.
- Test Automation
  - Saves money and time
  - Increases the accuracy
  - Faster
  - Can do certain things that manual testing can't do
    - Simulating load balancing
    - Network traffic …etc

Some Commonly Used Test Automation Tools

- Quick Test Professional (QTP) – Java, Windows, and Web
- Watir – Ruby
- Selenium – Web Applications
- LoadRunner (HP) – Mobile and Many varieties of web applications
- Rational Performance Tester – Web and Server Applications

What is Selenium

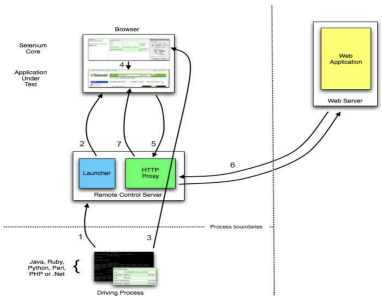
- Selenium is a toolkit that can automate web application testing
- The Selenium tests can be run on multiple browsers and supports several languages like Java, C#, PHP and Python.
- Many assertion statements provide an efficient way of comparing expected and actual results similar to Junit. Here assertion statements can be used to test web content.

Supporting Languages and Browsers

•Google Chrome
•Internet Explorer 6, 7, 8, 9, 10 - 32 and 64-bit where applicable
•Firefox: latest ESR, previous ESR, current release, one previous release
•Safari
•Opera
•HtmlUnit
•phantomjs
•Android (with Selendroid or appium)
•iOS (with ios-driver or appium)

C#,
Ruby
Python,
Java.

Selenium RC - Architecture



Adding Assertions to the Script

- Selenese allows multiple ways of checking for UI elements.
- Verifications and assertions are used to check if
  - an element is present somewhere on the page?
  - specific text is somewhere on the page?
  - specific text is at a specific location on the page?
- Verifications and assertions are not one and the same.
- If an assertion fails, the script will be aborted but if a verification fails the script will continue.

Verification Commands

verifyTitle

- This command is used to check if the page title is correct or not.
- To add this command , While recording the test steps right click any where on the page that you want verify. Once right clicked you can find an option "Show all commands". On Clicking it you will find an option "verifyTitle", select it

Verification Commands

verifyElementPresent

- This command is used to verify if a page element is present in the page or not.
- To add this command , While recording the test steps right click any element on the page that you want verify. Once right clicked you can find an option "Show all commands". On Clicking it you will find an option "verifyElementPresent", select it

Verification Commands

verifyValue

- This method is used to check if edit box has particular value or if the check box is on. Basically this method returns the value of present in the object.
- To add this command , While recording the test steps right click any element on the page that you want verify the value of. Once right clicked you can find an option "Show all commands". On Clicking it you will find an option "verifyValue", select it.

- Assertions are same as Verifications. The only difference is, if the assertions fail the script will abort. But the script will continue run in case a verification point fails.
- The steps for inserting the assertions is same as that of verification point.
- While recording Right Click → Show all commands → select an assertion.

- **assertTextPresent**
  This will assert if the text is present in the page.
- **assertText**
  This will assert if a particular element is having the particular text.
- **assertTitle**
  This will assert if the page is having a proper title.
- **assertValue**
  This will assert if a Text box or check box has a particular value
- **assertElementPresent**
  This will assert if a particular UI Element is present in the page.

Limitations of Selenium IDE

- Can run the test only on Some browsers (Firefox, chrome)
- No Programming login (like loops, conditional statements) can be applied
- Selenium IDE can execute scripts created in Selenese only.
- It is difficult to use Selenium IDE for checking complex test cases involving dynamic contents

If needed, define *one or more* methods to be executed before each test, e.g., typically for initializing values

```
@Before
public void setUp() {
   program = new MyProgram();
   array = new int[] { 1, 2, 3, 4, 5 };
}
```

If needed, define *one or more* methods to be executed after each test, e.g., typically for releasing resources (files, etc.)

```
@After
public void tearDown() {
}
```

```
import static org.testng.AssertJUnit.assertEquals;
import org.testng.annotations.Test;

public class ngTest {
    @Test
    public void T1() {
        assertEquals(1, cse360demo.f1(1, 1));
        long id = Thread.currentThread().getId();
        System.out.println("T1 Thread id is: " + id);
    }

    @Test
    public void T3() {
        assertEquals(1, cse360demo.f1(1, 2));
        long id = Thread.currentThread().getId();
        System.out.println("T3 Thread id is: " + id);
    }

    @Test
    public void T2() {
        assertEquals(1, cse360demo.f1(2, 1));
        long id = Thread.currentThread().getId();
        System.out.println("T2 Thread id is: " + id);
    }
}
```

```
<suite name="Test-method Suite" parallel="methods" thread-count="2">
  <test name="Test-method test" group-by-instances="true">
    <classes>
      <class name="ngTest"/>
    </classes>
  </test>
</suite>
```

If needed, define *one* method to be executed *just once,* when the class is first loaded. For instance, when we need to connecting to a database

```
@BeforeClass
public static void setUpClass() throws Exception {
    // one-time initialization code
}
```

If needed, define *one* method to be executed *just once,* to do cleanup after all the tests have been completed

```
@AfterClass
public static void tearDownClass() throws Exception {
    // one-time cleanup code
}
```