Rishabh Indoria
21F3001823

Software Testing

September 29, 2024

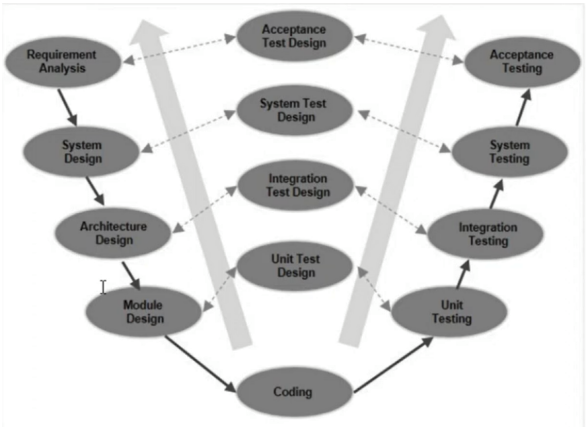
1 Introduction

1.1 Motivation

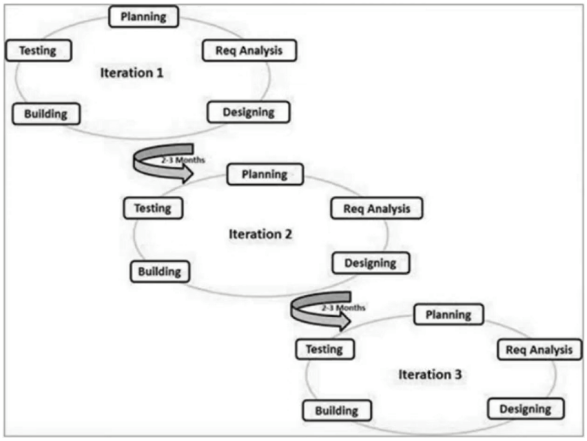
- **Introduction to Software Testing** by Paul Ammann and Jeff Offutt, **The Art of Software Testing** by Glenford J. Myers, **Software Testing: A Craftsman's Approach** by Paul C. Jorgensen, **Agile Testing: A practical guide for Testers and Agile Teams** by Lisa Crispin and Janet Gregory.
- Software is ubiquitous; Such software should be of very high quality, offer good performance in terms of response time, performance and also have no errors.
- It is no longer feasible to shut down a malfunctioning system in order to restore safety.
- Errors in software can cost lives, huge financial losses, or simply a lot of irritation.
- Testing is the **predominantly used** technique to find and eliminate errors in software.

1.2 Software Development Life Cycle

- **SDLC**: term used by the software industry to define a process for designing, developing, testing and maintaining a high quality software product.
- The goal is to use SDLC defined processes to develop a high quality software product that meets customer demands.
- **Planning**: Includes clearly identifying customer and/or market needs, pursuing a feasibility study and arriving at an initial set of requirements.
- **Requirements definition**: Includes documenting detailed requirements of various kinds: System-level, functional, software, hardware, quality requirements etc. They get approved by appropriate stakeholders.
- **Requirements analysis**: Includes checking and analyzing requirements to ensure that they are consistent, complete and match the feasibility study and market needs.
- **Design**: Identifies all the modules of the software product, details out the internals of each module, the implementation details and a skeleton of the testing details.
- **Architecture**: Defines the modules, their connections and other dependencies, the hardware, database and its access etc.
- **Development**: The design documents, especially that of low-level design, is used to implement the product. There are usually coding guidelines to be followed by the developers. Extensive unit testing and debugging are also done, usually by the developers. Tracking is done by project management team.
- **Testing**: Involves testing only where the product is thoroughly tested, defects are reported, fixed and re-tested, until all the functional and quality requirements are met.
- **Maintenance**: Done post deployment of product. Add new features as desired by the customer/market. Fix errors, if any, in the software product. Test cases from earlier phases are re-used here, based on need.
- **V-model**: It is a model that focuses on verification and validation. Follows the traditional SDLC life-cycle: Requirements, Design, Implementation, Testing, Maintenance.
- **Agile model**: Agile methodologies are adaptive and focus on fast delivery of features of a software product. All the SDLC steps are repeated in incremental iterations to deliver a set of features. Extensive customer interactions, quick delivery and rapid response to change in requirements.
- **Other Activities**: Project management, includes team management. Project documentation(Traceability matrix is a document that links each artifacts of development phase to those of other phases). Quality Inspection.



(a) V-Model



(b) Agile Model

Figure 1: Model Visualization

1.3 Testing Terminologies

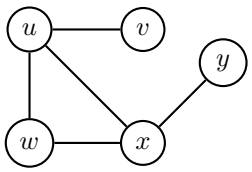
- **Validation**: The process of evaluating software at the end of software development to ensure compliance with intended usage. i.e., checking if the software meets its requirements.
- **Verification**: The process of determining whether the products of a given phase of the software development process fulfill the requirements established at the start of that phase.
- **Fault**: A static defect in the software. It could be a missing function or a wrong function in code.
- **Failure**: An external, incorrect behavior with respect to the requirements or other description of the expected behavior. A failure is a manifestation of a fault when software is executed.
- **Error**: An incorrect internal state that is the manifestation of some fault.
- **Test case**: A test case typically involves inputs to the software and expected outputs. A failed test case indicates an error. A test case also contains other parameters like test case ID, traceability details etc.
- **Unit Testing**: Done by developer during coding.
- **Integration Testing**: Various components are put together and tested. Components could be only software or software and hardware components.
- **System Testing**: Done with full system implementation and the platform on which the system will be running.
- **Acceptance Testing**: Done by end customer to ensure that the delivered products meet the committed requirements.
- **Beta Testing**: Done in a (so-called) beta version of the software by end users, after release.
- **Functional Testing**: Done to ensure that the software meets its specified functionality.
- **Stress Testing**: Done to evaluate how the system behaves under peak/unfavorable conditions.
- **Performance Testing**: Done to ensure the speed and response time of the system.
- **Usability Testing**: Done to evaluate the user interface, aesthetics.
- **Regression Testing**: Done after modifying/upgrading a component, to ensure that the modification is working correctly, and other components are not damaged by the modification.
- **Black-Box Testing**: A method of testing that examines the functionalities of a software/system without looking into its internal design or code.
- **White-Box Testing**: A method of testing that test the internal structure of the design or code of a software.
- **Test Design**: Most critical job in testing. Need to design effective test cases. Apart from specifying the inputs, this involves defining the expected outputs too. Typically, cannot be automated.

- **Test Automation**: Involves converting the test cases into executable scripts. Need to specify how to reach deep parts of the code using just inputs, Observability and Controllability.
- **Test Execution**: Involves running the test on the software and recording the results. Can be fully automated.
- **Test Evaluation**: Involves evaluating the results of testing, reporting identified errors. A difficult problem is to isolate faults, especially in large software and during integration testing.
- **Testing goals**: Organizations tend to work with one or more of the following levels
  Level 0: There is no difference between testing and debugging.
  Level 1: The purpose of testing is to show correctness.
  Level 2: The purpose of testing is to show that software doesn't work.
  Level 3: The purpose of testing is not to prove anything specific, but to reduce the risk of using the software.
  Level 4: Testing is a mental discipline that helps all IT professionals develop higher quality software.
- **Controllability**: Controllability is about how easy it is to provide inputs to the software module under test, in terms of reaching the module and running the test cases on the module under test.
- **Observability**: Observability is about how easy it is to observe the software module under test and check if the module behaves as expected.

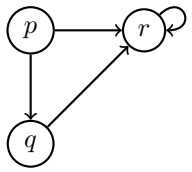
2 Graphs

2.1 Basics

- A graph is a tuple $G = (V, E)$ where V is a set of **nodes/vertices**, and $E \subseteq (V \times V)$ is a set of **edges**.
- Graphs can be **directed** or **undirected**.



(a) A simple undirected graph



(b) A directed graph

- Graphs can finite or infinite.
- The **degree** of a vertex is the number of edges that are connected to it. Edges connected to a vertex are said to be **incident** on the vertex.
- There are designated special vertices like **initial** and **final** vertices. These vertices indicate beginning and end of a property that the graph is modeling.
- Typically, there is only one initial vertex, but there could be several final vertices.

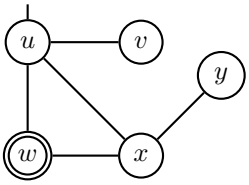
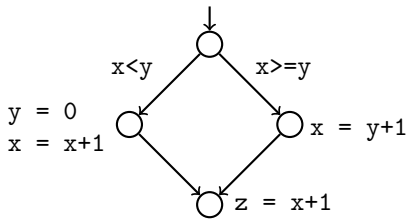


Figure 3: Graph with initial and final vertices

- Most of these graphs will have **labels** associated with vertices and edges. Labels or annotations could be details about the artifact that the graphs are modelling. Tests are intended to cover the graph in some way.

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

(a) A sample code



(b) A Control Flow Graph

Figure 4: Example of a CFG

- A **path** is a sequence of vertices $v_1, v_2, ..., v_n$ such that $(v_i, v_{i+1}) \in E$.
- **Length** of a path is the number of edges that occur in it. A single vertex path has length 0.
- **Sub-path** of a path is a sub-sequence of vertices that occur in the path.
- A vertex v is **reachable** from some other vertex if there is a path connecting them.
- An edge $e = (u, v)$ is **reachable** if there is a path that goes to vertex u and then goes to vertex v.
- A **test path** is a path that starts in an initial vertex and ends in a final vertex. These represent execution of test cases.
- Some test paths can be executed by many test cases: **Feasible paths**.
- Some test paths cannot be executed by any test case: **Infeasible paths**.
- A test path p **visits** a vertex v if v occurs in path p. A test path p **visits** an edge e if e occurs in the path p.
- A test path p **tours** a path q if q is a sub-path of p.
- When a test case t executes a path, we call it the **test path** executed by t, denoted by $path(t)$.
- The set of test paths executed by a set of test cases T is denoted by $path(T)$.
- **Test requirement** describes properties of test paths.
- **Test Criterion** are rules that define test requirements.
- **Satisfaction**: Given a set TR of test requirements for a criterion C, a set of tests T satisfies C on a graph iff for every test requirement in $t \in TR$, there is a test path in $path(T)$ that meets the test requirement t.
- **Structural Coverage Criteria**: Defined on a graph just in terms of vertices and edges.
- **Data Flow Coverage Criteria**: Requires a graph to be annotated with references to variables and defines criteria requirements based on the annotations.

2.2 Elementary Graph Algorithms

- Two standard ways of representing graphs: **adjacency matrix** or **adjacency lists**.
- Adjacency list representation provides a compact way to represent **sparse** graphs, i.e., graphs for which $|E|$ is much less than $|V|^2$. For each $u \in V$, $Adj[u]$ contains all vertices v such that $(u, v) \in E$, i.e., it contains all edges incident with u. Represented in $\Theta(|V| + |E|)$ memory.
- Adjacency matrix representation provides a compact way to represent **dense** graphs, i.e., graphs for which $|E|$ is close to $|V|^2$. This is a $|V| \times |V|$ matrix, where $a_{ij} = 1$ if there is an edge going from i to j.
- **Breadth First Search**: Computes the "distance" from s to each reachable vertex.

Algorithm 1 Breadth First Search

```
    BFS(G)
 1: for each vertex u ∈ G, V − {s} do
 2:     u.color = WHITE, u.d = ∞, u.π = NIL
 3: end for
 4: s.color = BLUE, s.d = 0, s.π = NIL
 5: Q = φ
 6: ENQUEUE(Q, s)
 7: while Q ≠ φ do
 8:     u = DEQUEUE(Q)
 9:     for each v ∈ G.Adj[u] do
10:         if v.color == WHITE then
11:             v.color = BLUE
12:             v.d = u.d + 1
13:             v.π = u
14:             ENQUEUE(Q, v)
15:         end if
16:     end for
17:     u.color = BLACK
18: end while
```

- **Depth First Search**

Algorithm 2 Depth First Search

```
    DFS(G)
 1: for each vertex u ∈ G.V do
 2:     u.color = WHITE
 3:     u.π = NIL
 4: end for
 5: time = 0
 6: for each vertex u ∈ G.V do
 7:     if u.color == WHITE then
 8:         DFS-VISIT(G, u)
 9:     end if
10: end for

    DFS-VISIT(G, u)
11: time = time + 1
12: u.d = time
13: u.color = GRAY
14: for each v ∈ G.Adj[u] do
15:     if v.color == WHITE then
16:         v.π = u
17:         DFS-VISIT(G, v)
18:     end if
19: end for
20: u.color = BLACK
21: time = time + 1
22: u.f = time
```

2.3 Structural Graph Coverage

- **Node Coverage** requires that the test cases visit each node in the graph once. Test set T satisfies node coverage on graph G iff for every syntactically reachable node $n \in G$, there is some path p in $path(T)$ such that p visits n.
- **Edge Coverage**: TR contains each reachable path of length up to 1, inclusive, in G. Edge coverage is slightly stronger than node coverage. Allowing length up to 1 allows edge coverage to subsume node coverage.
- **Edge-Pair Coverage**: TR contains each reachable path of length up to 2, inclusive in G. Paths of length up to 2 correspond to pairs of edges.

- **Complete path coverage**: TR contains all paths in G. Unfortunately, this can be an infeasible test requirement, due to loops.
- **Specified path coverage**: TR contains a set S of paths, where S is specified by the user/tester.
- A path from n_i to n_j is **simple** if no node appears more than once, except possible the first and last node.
- A **prime path** is a simple path that does not appear as a proper sub-path of any other simple path.
- **Prime path coverage**: TR contains each prime path in G. Ensures that loops are skipped as well as executed. It subsumes node and edge coverage.
- **Tour with side trips**: A test path p tours a sub-path q with side trips iff every edge q is also in p in the same order. the tour can include a side trip, as long as it comes back to the same node.
- **Tours with detours**: A test path p tours a sub-path q with detours iff every node in q is also in p in the same order. The tour can include a detour from node n as long as it comes back to the prime path at a successor of n.
- **Best Effort Touring**: Satisfy as many test requirements as possible without sidetrips. Allow sidetrips to try to satisfy remaining test requirements.
- **Round trip path**: A prime path that starts and ends at the same node.
- **Simple round trip coverage**: TR contains at least one round trip path for each reachable node in G that begins and ends in a round trip path.
- **Complete round trip coverage**: TR contains all round trip paths for each reachable node in G.

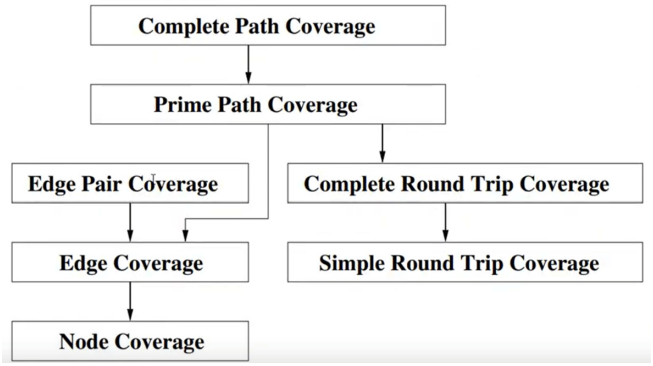


Figure 5: Structure Coverage Criteria Subsumption

2.4 Algorithms: Structural Graph Coverage Criteria

- There are two entities related to coverage criteria: Test requirement, and Test case as a test path, if the test requirement is feasible.
- Test requirements for node and edge coverage are already given as a part of the graph modeling the software artifact. Test paths to achieve node and edge coverage can be obtained by simple modifications to BFS.
- TR for edge-pair coverage is all paths of length two in a given graph. Need to include paths that involve self-loops too.

Algorithm 3 Simple Edge-Pair Algorithm

```
1: for each node u in the graph do
2:     for each node v in Adj[u] do
3:         for each node w in Adj[v] do
4:             Output path u − −v − −w
5:         end for
6:     end for
7: end for
```

- Prime Path Algorithm

Algorithm 4 Computing prime paths

```
1: Loops = [ ]
2: Terminate = [ ]
3: Q = φ
4: for each v ∈ G do
5:     ENQUEUE(Q, [v])
6: end for
7: while Q ≠ φ do
8:     path = DEQUEUE(Q)
9:     for each v ∈ G.Adj[path[−1]] do
10:        if v is a final vertice then
11:            Terminate = Terminate ++ (path ++ v)
12:        else if v in path and v == path[0] then
13:            Loops = Loops ++ (path ++ v)
14:        else if v in path then
15:            continue
16:        else
17:            ENQUEUE(Q, path ++ v)
18:        end if
19:    end for
20: end while
21: return Loops, Terminate
```

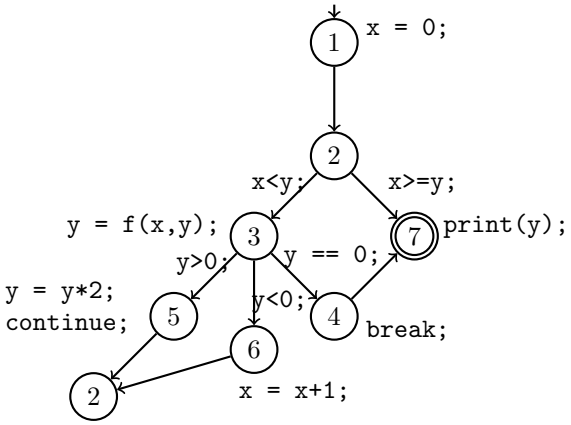
- To enumerate test paths for prime path coverage: Start with the longest prime paths and extend each of them to the initial and the final nodes in the graph.

2.5 Control Flow Graphs for Code

- Modelling control flow in code as graphs. Using structural coverage criteria to test control flow in code.
- Typically used to test a particular function or procedure or a method.
- A **Control Floe Graph** models all executions of a method by describing control structures.
- **Nodes**: Statements or sequences of statements (basic blocks).
- **Basic Block**: A sequence of statements such that if the first statement is executed, all statements will be (no branches).
- **Edges**: Transfer of control from one statement to the next.
- CFGs are often annotated with extra information to model data, this includes branch predicates, Definitions and/or uses.

```
x = 0;
while(x<y){
    y = f(x,y);
    if(y == 0){
        break;
    }else if(y<0){
        y = y*2;
        continue;
    }
    x = x+1;
}
print(y);
```

(a) While loop with break and continue



(b) A Control Flow Graph

Figure 6: Example of a CFG

2.6 Data Flow in Graphs

- Graph models of programs can be tested adequately by including values of variables(data values) as a part of the model.
- Data values are created at some point in the program and use later. They can be used several times.
- A **definition (def)** is a location where a value of a variable is stored into memory.
- A **use** is a location where a value of a variable is accessed.
- As a program executes, data values are carried from their defs to uses. We call these du-pairs or def-use pairs.
- A **du-pair** is a pair of location (l_i, l_j) such that a variable v is defined at l_i and used at l_j.
- Let V be the set of variables that are associated with the program artifact being modelled as a graph.
  The subset of V that each node n (edge e) defines is called $def(n)(def(e))$.
  The subset of V that each node n (edge e) uses is called $use(n)(use(e))$.
- A def of a variable may or may not reach a particular use.
- A path from l_i to l_j is **def-clear** with respect to variable v if v is not given another value on any of the nodes or edges in the path.
- If there is a def-clear path from l_i to l_j with respect to v, the def of v at l_i **reaches** the use at l_j.
- A **du-path** with respect to a variable v is a simple path that is def-clear from a def of v to a use of v.
- $du(n_i, n_j, v)$: The set of du-paths from n_i to n_j for variable v.
- $du(n_i, v)$: The set of du-paths that start at n_i for variable v.
- In testing literature, there are two notions of uses available.
  If v is used in a computational or output statement, the use is referred to as **computation use** (or **c-use**).
  If v is used in a conditional statement, its use is called as **predicate use** (or **p-use**).
- Data flow coverage criteria will be defined as sets of du-paths. Such du-paths will be grouped to define the data flow coverage criteria.
- The **def-path set** $du(n_i, v)$ is the set of du-paths with respect to variable v that start at node n_i.
- A **def-pair set**, $du(n_i, n_j, v)$ is the set of du-paths with respect to variable v that start at node n_i and end at node n_j.
- It can be clearly seen that $du(n_i, v) = \bigcup_{n_j} du(n_i, n_j, v)$.
- A test path p is said to **du tour** a sub-path d with respect to v if p tours d and the portion of p to which d corresponds is def-clear with respect to v.
- We can allow **def-clear side trips** with respect to v while touring a du-path, if needed.
- There are three common data flow criteria
  1. TR: Each def reaches at least one use.
  2. TR: Each def reaches all possible uses.
  3. TR: Each def reaches all possible uses through all possible du-paths.
- We assume every use is preceded by a def, every def reaches at least one use, and for every node with multiple out-going edges, at least one variable is used on each out edge, and the same variables are used on each out edge.
- **Subsumption**: ③→②→①
- Prime path coverage subsumes all-du-paths coverage