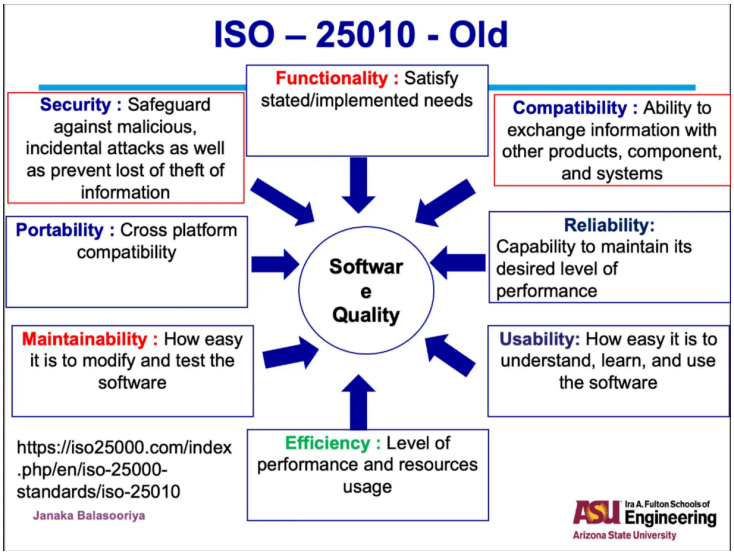
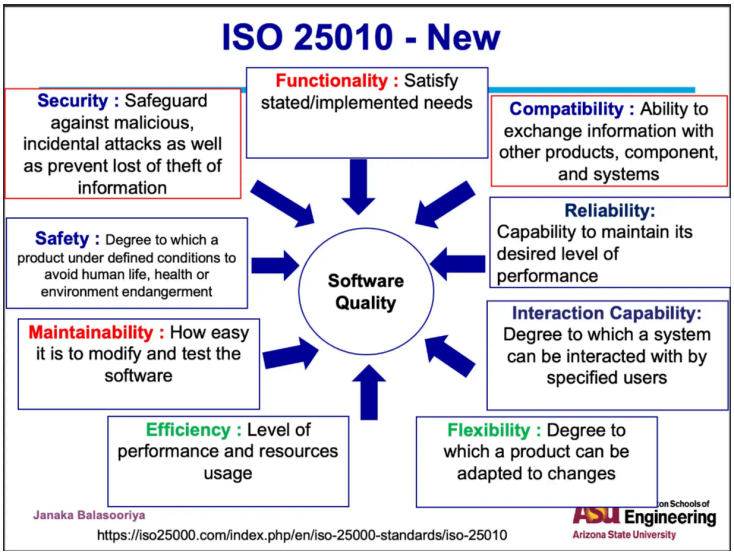




QA spans three complementary strategies: Defect prevention includes education/training, domain and process knowledge, appropriate tooling, and sound architecture/design practices to reduce error injection. Defect reduction includes inspections/reviews and testing to observe failures, diagnose faults, fix them, and re-verify corrections before release. Defect containment includes software fault tolerance (e.g., recovery blocks, redundancy) to limit failure impact when faults escape earlier stages.

Functional Testing (Blackbox testing): Specification-derived tests focusing on required behaviors. Requirements, SRS, and user-facing behaviors. Validating observable outputs for inputs and scenarios. Equivalence classes, boundary value analysis, decision tables, model-based tests. Catches specification conformance issues and user-visible failures. May miss internal faults if outputs look correct. Acceptance/system testing and user-oriented validation. Blackbox testing is synonymous with functional testing, relying on external specifications to select inputs and check outputs without internal knowledge. Code is not required.

Structural testing (Whitebox testing): Code/structure-derived tests focusing on internal elements and paths. Control flow, data flow, code structure, and coverage goals. Exercising statements, branches, paths, and data interactions. Code coverage, path testing, slicing, data-flow testing. Reveals untested code, dead code, and logic faults not visible externally. May miss missing or ambiguous requirements even with full coverage. Unit/integration testing and structural adequacy measurement. Whitebox testing is synonymous with structural testing, using internal knowledge to design tests that meet coverage criteria over code and data structures. Code is required.

Venn Diagram (SPT) Representative regions: $S \cap P \cap T$ (implemented as specified and tested), $P \cap T - S$ (extra functionality tested), $S \cap P - T$ (known missing tests), $S \cap T - P$ (tests for specified but missing features), $S - (P \cup T)$ and $P - (S \cup T)$ (likely unknown gaps), and $T - (S \cup P)$ (tested unspecified behaviors, often quality-related).

Exploratory testing is a black box approach where testers learn the system, form goals, design on-the-fly experiments, and adapt based on findings while documenting insights. Heuristics such as SFDIPOT guide exploration across Structure, Function, Data, Interfaces, Platform, Operations, and Time to uncover risk areas efficiently. Function categorization, stakeholder understanding, and coverage of unspecified but expected behaviors complement scripted functional testing.

Equivalent classes: Equivalent Partitioning subdivides the input domain into disjoint subsets where any one representative test from a class is assumed to reveal the same fault as any other in that class, which keeps tests focused and non-redundant. Steps to identify EPs: Consider both valid and invalid inputs, look for equivalent output events, look for equivalent operating environments, and organize them in a table. **EP classification: Weak normal**: choose one representative from each valid class. **Strong normal**: cartesian product over valid classes across variables. **Weak robust**: introduce a single invalid value while others are valid. **Strong robust**: cartesian product including invalid classes, assuming multiple simultaneous faults. **Limitation**: EP does not specifically target off-by-one and boundary faults. EP alone does not model inter-variable dependencies well.

Boundary value analysis: BVA holds all but one variable at nominal values and drives the remaining variable through its boundary points to reveal edge-sensitive faults such as loop, comparison, and off-by-one errors. It works best when inputs are independent, bounded physical quantities, and is often paired with EP to close the most common gaps. Integers: test min, min+1, max−1, max, plus one outside each side. Reals and chars: adapt to representable granularity or ASCII ordering. Strings: length constraints and character-class constraints define edges. **Limitation**: BVA can produce gaps and redundancies if applied naively and doesn't capture logical dependencies across variables.

Decision table: DT formalizes logical combinations by mapping conditions to actions, ensuring each (feasible) rule is considered and enabling algebraic minimization of redundant cases. Originating from **cause–effect graphing**, DTs are recommended for logically complex situations with interacting conditions. **Limitation:** Ordering of inputs, durations, and timing of events are critical, or the number of conditions explodes, requiring careful reduction.

Model-based testing: Finite State Machines represent states as nodes and transitions as edges labeled by events/data/time, with paths from start to final state forming test cases. Determinism is required for this technique; otherwise, the same input may not produce a predictable output to assert against. **Limitation**: Requires determinism and modeling effort; pure FSMs don't cover concurrency without other models. **Use Cases**: Stateful workflows, UI flows, protocols, and device interactions.

Effective Combination: A practical "pendulum" strategy balances effort and effectiveness: start with EP to remove redundancy, add BVA to probe edges, and then DT to capture cross-variable dependencies.

Bug Management: A bug report must contain enough information for teams to understand, reproduce, diagnose, and fix the problem, emphasizing clarity and credibility in reporting. Core fields typically include title, description, status, version no., feature area, reproduction steps, assignment, severity, customer impact, environment, and 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. The 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.
7. The test group retests the fixed bugs and close or reopen.

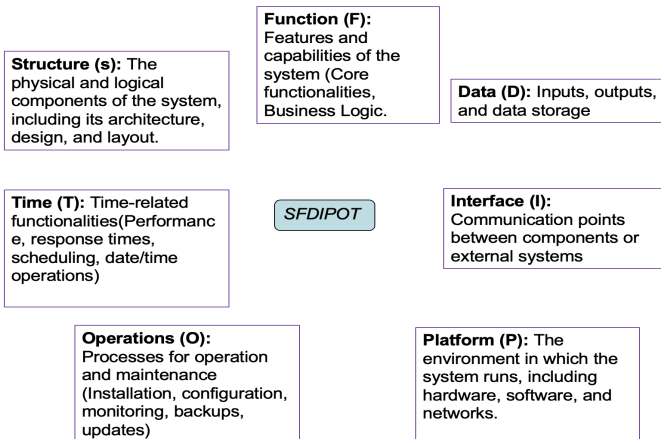
One of the most commonly used quantitative approaches is to measure the **"Defect Arrival Rate"**. The **Weibull curve** is used to model defect arrival behavior that typically rises and then falls over time; many practitioners expect bug-count curves to exhibit this general shape during testing.

Code-Based Testing: It 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: Basic path testing (Control flow coverage), Statement coverage, Decision-to-Decision testing, Dataflow coverage, Condition testing, Code based risk assessments (complexity of the code, object oriented matrices …etc).

Control-flow graph: A CFG models a program as a directed graph where nodes represent statements or statement fragments and edges represent possible transfers of control between them. **Cyclomatic Complexity:** Edges - Nodes + 2.

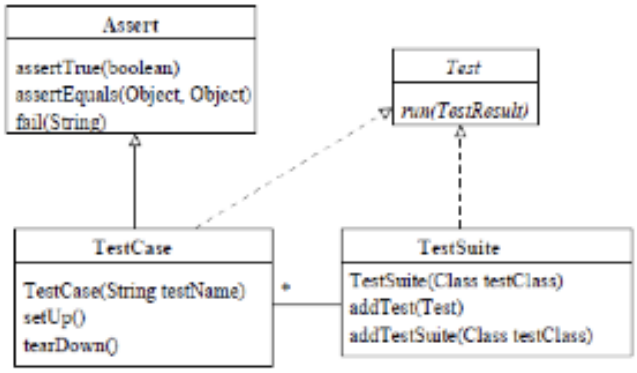
JUnit Testing: JUnit is a simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks written by Kent Beck and Erich Gamma. JUnit 4 introduces annotations for one-time and per-test setup/teardown, with @BeforeClass and @AfterClass executing once per class, and @Before and @After executing before and after each test, respectively. Tests are grouped into a conventional class with imports for org.junit and static imports for Assert methods, using assertEquals and related assertions to check expected outcomes for each path. Choose a basis set of independent paths equal to the cyclomatic number, then design one test input per path that forces execution along that path, using assertions tied to the path's observable outcome.

A framework is a semi-complete application. A framework provides a reusable, common structure to share among applications. Developers incorporate the framework into their own application and extend it to meet their specific needs. Frameworks differ from toolkits by providing a coherent structure, rather than a simple set of utility classes.



Characteristics of a Good Test Engineer: Analytical Problem Solving, Customer-focused Innovation, Technical excellence, Project/Time Management, Passion for quality, Strategic insight, Confidence, Impact and Influence, Cross-Boundary Collaboration, Interpersonal Awareness.

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



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;
```


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;
```


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() {
}
```

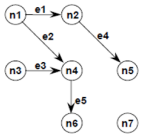
Directed Graphs

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>}

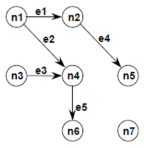
Indegrees and Outdegrees

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

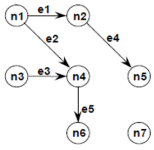
indeg(n7) = 0 outdeg(n7) = 0

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,

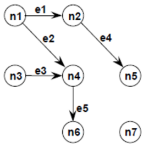
n5, n6 and n7 are sink nodes,

n2 and n4 are transfer nodes.

Adjacency Matrix of a Digraph

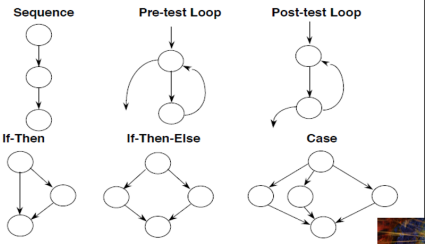
15. The adjacency matrix of a directed graph D = (V, E) with m nodes is an m x m matrix A = (a(i, j)) , where a(i, j) is a 1 if and only if there is an edge from node i and node j, otherwise the element is 0.

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.

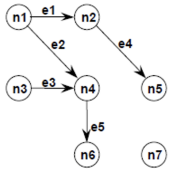


	n1	n2	n3	n4	n5	n6	n7
n1	0	1	0	1	0	0	0
n2	0	0	0	0	1	0	0
n3	0	0	0	1	0	0	0
n4	0	0	0	0	0	1	0
n5	0	0	0	0	0	0	0
n6	0	0	0	0	0	0	0
n7	0	0	0	0	0	0	0

Program Graphs of Structured Programming Constructs



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

Weak normal	Pick one representative from each valid class while keeping others nominal; cover month-type classes and leap vs non-leap at least once without all combinations.
Strong normal	Take the Cartesian product over valid classes to cover every valid class combination across variables (e.g., month type × leap/non-leap).
Weak robust	Include invalid classes but only one invalid input at a time; others remain valid (single-fault assumption).

Strong robust	Cartesian product over valid and invalid classes, allowing multiple invalids simultaneously (multiple-fault assumption).