Completeness = all of the blocks in a characteristic consider all possible cases.
Disjointness = all of the blocks in a characteristic do not overlap.

Ex. Characteristic: Location of element in list
Block 1: element is first entry in list Block 2: element is last entry in list Block 3: element is in some position other than first or last
This example fails the completeness property because none of the blocks consider the case when the element is not in the list (e.g. [1,1,2] and target element 3).
This example fails the disjointness property because the element could be at multiple locations in the list (e.g. [1,1,2] and target element 1).

BCC **(base choice coverage)** = there must be at least one base block for each test case.

Ex. Characteristic 1: number of elements in lst
Block 1: $size = 1$ Block 2: $size > 1$ (base block)
Characteristic 2: the sign of sum
Block 1: $sum < 0$ Block 2: $sum = 0$ Block 3: $sum > 0$ (base block)
tr1 (base test): ($lst = [0,1]$, $sum = 1$). tr2: ($lst = [0,1]$, $sum = -1$). tr3: ($lst = [0,1]$, $sum = 0$). tr4: ($lst = [39]$, $sum = 14$).

PC **(predicate coverage)** = set of tests that evaluates $p = true$ and $p = false$.
CC **(clause coverage)** = set of tests that evaluates $a = true$ and $a = false$, $b = true$ and $b = false$, $c = true$ and $c = false$, etc.
$GACC$ **(general active clause coverage)** = set of pairs where the major clause predicate is true and the major clause varies.
$CACC$ **(correlated active clause coverage)** = same as $GACC$, except that it requires the predicate to vary.
$RACC$ **(restricted active clause coverage)** = same as $CACC$, except that it requires that the minor clauses to be the same.
$GICC$ **(general inactive clause coverage)** = set of pairs where the major clause predicate is false and the major clause varies, and cases when the predicate is true and false must be considered.
$RICC$ **(restricted inactive clause coverage)** = same as $GICC$, except that it requires that the minor clauses to be the same.

Ex. $p = (a \,\&\&\, b \,\&\&\, c) \,||\, (!a \,\&\&\, !b \,\&\&\, !c)$

Row#	a	b	c	P	P_a	P_b	P_c
1	T	T	T	T	T	T	T
2	T	T					T
3	T		T			T	
4	T				T		
5		T	T		T		
6		T				T	
7			T				T
8				T	T	T	T

$GACC$

Major Clause	Set of possible tests
a	(1,5), (1,8), (4,5), (4,8)
b	(1,3), (1,8), (6,3), (6,8)
c	(1,2), (1,8), (7,2), (7,8)

$CACC$

Major Clause	Set of possible tests
a	(1,5), (4,8)
b	(1,3), (6,8)
c	(1,2), (7,8)

$RACC$

Major Clause	Set of possible tests
a	(1,5), (4,8)
b	(1,3), (6,8)
c	(1,2), (7,8)

$GICC$

Major Clause	Set of possible tests	
a	$P =$ T: No feasible pairs	$P =$ F: (2,6), (2,7), (3,6), (3,7)
b	$P =$ T: No feasible pairs	$P =$ F: (2,4), (2,7), (5,4), (5,7)
c	$P =$ T: No feasible pairs	$P =$ F: (3,4), (3,6), (5,4), (5,6)

$RICC$

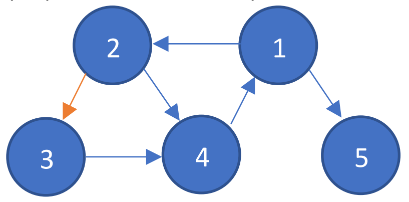
Major Clause	Set of possible tests	
a	$P =$ T: No feasible pairs	$P =$ F: (2,6), (3,7)
b	$P =$ T: No feasible pairs	$P =$ F: (2,4), (5,7)
c	$P =$ T: No feasible pairs	$P =$ F: (3,4), (5,6)

du**-pairs** = all the pairs that go from def to use for each variable. du**-paths** = all the paths that go from def to use for each variable.
TSC **(terminal symbol coverage)** = all the terminal symbols are the test requirements.
PDC **(production coverage)** = all the production rules are the test requirements.
DC **(derivation coverage)** = all the strings that can be produced by the grammar are the test requirements.

Ex. $A ::= B \mid B\ A$ $B ::= Init\ Dispose \mid Init\ C\ Dispose$ $C ::= PushS \mid PushS\ C \mid PushS\ D \mid PushS\ C\ D$ $D ::= Top \mid PopS \mid Top\ PopS$
TSC: tr: $\{Init, Dispose, PushS, PopS, Top\}$. $A \to B, Init\ C\ Dispose, Init\ PushS\ D\ Dispose, Init\ PushS\ Top\ PopS\ Dispose$.
PDC: tr: $\{A \to B, A \to B\ A, B \to Init\ Dispose, B \to Init\ C\ Dispose, C \to PushS, C \to PushS\ C, C \to PushS\ D, C \to PushS\ C\ D\}$.
DC: There are ∞ strings that are needed to satisfy derivation coverage.

NC **(node coverage)** = all the reachable nodes are the test requirements.
EC **(edge coverage)** = all the reachable paths of length up to 1, inclusive.
EPC **(edge-pair coverage)** = all the reachable paths of length up to 2, inclusive.
PPC **(prime path coverage)** = all the simple paths that do not appear as a proper subpath of any other simple paths are the test requirements

Ex. $N = \{1, 2, 3, 4, 5\}$ $N_0 = \{1\}$ $N_f = \{5\}$ $E = \{(1,2), (1,5), (2,3), (2,4), (3,4), (4,1)\}$
NC: tr: $\{1,2,3,4,5\}$ EC: tr: $\{(1,2), (1,5), (2,3), (2,4), (3,4), (4,1)\}$
EPC: tr: $\{(1,2,3), (1,2,4), (1,5), (2,3,4), (2,4,1), (3,4,1), (4,1,2), (4,1,5)\}$
Simple Paths: $\{(1,2), (1,5), (2,3), (2,4), (3,4), (4,1), (1,2,3), (1,2,4), (2,3,4), (2,4,1), (3,4,1),$
$(4,1,2), (4,1,5), (1,2,3,4), (1,2,4,1), (2,3,4,1), (2,4,1,2), (2,4,1,5),$
$(3,4,1,2), (3,4,1,5), (4,1,2,3), (4,1,2,4), (1,2,3,4,1), (2,3,4,1,2),$
$(2,3,4,1,5), (3,4,1,2,3), (4,1,2,3,4)\}$
Prime Paths: $\{(2,3,4,1,2), (1,2,3,4,1), (2,3,4,1,5), (4,1,2,3,4), (3,4,1,2,3), (2,4,1,2), (1,2,4,1),$
$(2,4,1,5), (4,1,2,4)\}$



Fault=a static defect in source code. **Error**=incorrect internal state, manifestation of some fault. **Failure**=external, incorrect behavior w/respect to requirements(propagation). **Error State**=first different state in execution compared to correct execution. **RIPR**=Reachability, Infection (execution leads to incorrect program state), Propagation (infected state propagates to final state), Revealability (tester observes the incorrect program state). **Black-box testing**=external descriptions of software ->Tests. **White-box Testing**=source code -> Tests. **Model-based testing**=derive tests from a model of software. **Human-based approach**=design test values on: Domain knowledge of program, human knowledge of testing, knowledge of user interface (No CS degree needed). **Criteria-based approach**=design test values to satisfy coverage criteria, require CS degree. **Test Criterion:** a collection of rules and a process that define TRs. **Test Requirements:** specific things that must be satisfied or covered during tests. **Model Driven Test Design**=software->model->TRs->refine requirements->input values->test cases->results>pass/fail. OR: TestDesign, Automation, Test Execution, Test Eval. **Test Automation**=use of software to control execution of tests, comparison of actual outcomes to predicted outputs, the setting up of test preconditions, and other test control and test reporting functions. -Benefits: reduce cost, human error, etc. **Testability**: how easy is it to establish test criteria and whether those criteria have been met. How easy to 1)provide test values (controllability) 2)observe the results (observability). **Test Case Contains:** test case values, expected results, prefix vals (put SW into state to receive test vals), postfix vals (vals needed to see output, exit values (to terminate program)). **Software testing should be embedded in system, otherwise may cost too much time/money if we only do after development. Agile methods:** recognizes that the assertions that "modeling and analysis can identify potential problems early in development" and "saving by cost-of-change curve justifies the cost of modeling and analysis over the life of the project" are invalid (because requirements not always complete/valid), i.e. requirements go out of date for a quickly. ->software starts small and evolves. ->don't derive system tests from requirements. **Correctness:** Traditionally= universal (define all behavior before tests): Agile Correctness= existential (define only some behavior, correct if follows that). **Test Harnesses:** run all automated tests efficiently and reports results to developer. **Continuous Integration**: rebuilds the system, and the reruns/verifies tests. **Test Doubles:** an object that can stand in for a real object in a test. **TDD**=1)write a failing test, write code to make it work, refactor. **Good Tests**=tests better for guiding work, a good test is atomic (small, focused), isolated (doesn't depend on other tests). Breadth First Implementation= Higher level functionality first by faking low level. **Coverage Criteria=** ***rule of collection of rules that impose TRs on a test set.*** describe a finite subset of test cases to execute, divide input space to maximize the #faults found per test case, provide useful rules for when to stop testing → help us decide when we have enough tests, when to test more, automate tests. Focus on structures (input spaces, graphs, logical expressions, grammars). **Test Requirement=** a specific element of a software artifact that a test case must cover. **Test Case=** a set of test inputs, execution conditions, expected results, etc. **Test Set=** a set of test cases. **Subsuming-> C1 subsumes C2 if every set of test cases that satisfied C1 and satisfies C2. why use criteria?** Fewer tests, more effective at finding faults, increase traceability, provides how many tests needed, supports automation, efficient. **Testing=** process of finding test input values to check against a software (choosing finite sets). **All possible values, can be infinite. Input Space Partitioning=** Pairwise disjoint (**no overlap**), together blocks cover domain (**complete**). At least one value used from each block. **Benefits of ISP:** Fewer tests, no implementation knowledge needed, equally applied at different levels of testing. Identify testable functs -> identify parameters, model input domain, apply test criterion to choose combination of blocks, derive test vals. **Characteristics=** define the structure of the input domain. **Interfaced-based=** develop characteristics from individual parameters, consider params separately (STRENGTH: easy to identify characteristics, easy to translate to executable test cases. WEAK: some info not be used, ignore parameter relationships). **Functionality-based=** developed characteristics from a behavior view, can use relationships among parameters, incorporate domain and semantic knowledge (Strong: incorporate semantic knowledge, WEAK: difficult to design reasonable characteristics, hard to generate tests). **Extra: typical to create multiple small IDMs, some may overlap. All Combined Coverage (ACoC): All combinations of blocks from all characteristic must be used in all tests [drop infeasible pairs]. Each Choice Coverage (EEC): One value from each block for each characteristic must be used in at least one test case [change value to find feasible combination if infeasible]. -Certain values are important. TO strengthen ECC, domain knowledge of the program must be incorporated (most important block for each partition. Base Choice Coverage (BCC): base choice block is chosen for each characteristics, a base test is formed by using the base choice for each characteristic, a base test is formed by using the base choice for each characteristic and holding all but one base choice constant. Base Test must be feasible. Up to implemented which base choice to use. [change a value to another non-base choice to find a feasible combination] Multiple Base Choice Coverage (MBCC): At least one, maybe more, base choice blocks chosen for each characteristic. Base tests are formed by using each base choice for each characteristic at least once. Subsequent tests are chosen by holding all but one base choice constant for each base test**. **Subsumption →ACoC -> MBCC -> BCC -> ECC. Constraints=infeasible tests. Two types, either a block can't be combined with another, or it must be combined with a specific other.** Graph **Node=**statement, state, method, basic block **Edge=** Branch, Transition, Method Call**, Path =** a sequence of nodes **Length =** number of edges **Subpath,** subsequence of nodes in p, possibly p itself. **Cycle** = a path that begins and ends at the same node **Invalid Path =** path where the two nodes are not connected by an edge. **Syntactically Reachable:** depends if there is a path to a node or edge then it is syntactically reachable. **Semantically Reachable:** If *possible* to execute at least one path to a node or edge. **Test path:** Starts at initial Node and ends at final node. **Single Entry Single Exit graphs =**one initial and one final node. **Visit =** path visits node if node is in path, same with edge, **Touring** = a test path p tours subpath q if q is a subpath of p. **Path(t) =** Test path executed by test case t, **Path(T) =** Test paths executed by set of tests T. Minimal set of test paths is the smallest set of test paths to satisfy a requirement. **Deterministic software=** test always executes the same test path. **Non-deterministic software=** the same test can execute different test paths. **Test Requirements (TR) =** Describe properties of test paths. **Test criterion =** Rules that define test requirements. **Structural coverage criteria=** Define a graph just in terms of nodes and edges. **Data flow coverage criteria**= Requires a graph to be annotated with references to variables. **Simple Path:** Simple if no node appears more than once, except possibly the first and last. **Prime Path:** A simple path that is not a strict subset of another SP. **Structural Coverage Criteria Node Coverage:** TR contains each reachable node in G. **Edge Coverage:** TR contains each reachable path of length up to 1 inclusive, in G. **Edge-Pair Coverage:** TR contains each reachable path of length up to 2, inclusive, in G. **Complete path Coverage:** TR contains all paths in G. Specified Path Coverage: contains all paths in S, where S is defined. **Prime Path Coverage:** TR contains each prime path in G. **Tour w/sidetrips =** a test path p tours q with sidetrips iff every e in q is in p in the same order (allowing side trips weakens test criteria). **Tour w/ detours=** same as w/ side trips but with nodes, not edges. **Best effort touring=** satisfy as many test requirements as possible w/out sidetrips. **Round Trip Path:** prime path that starts and ends at the same node. **Simple RTP** = TR contains at least one round-trip path for each reachable node in G that begins and ends a round trip path. **Complete Round Trip Coverage:** TR contains all round trip paths for each reachable node in G. **Data Flow Coverage Criteria Definition:** a location where a value/stored in memory. **def(n) or def(e)=** set of variables that are defined by n or e. **Use:** a location where a variable is used. **use(n) or use(e):** the set of variables that are used by n or e. **DU-Pair =** pair of locations such that something is defined and then used. **Def-clear=** a path from I to J is def clear if there isn't a definition of v in the path. **Reach:** if there is a def-clear path from I to j with respect to v, then a def of v reaches j. **DU=path:** simple subpath that is def-clear with respect to v to a use of v. Du(I, j, v) is a set of du-paths from I to j . Du(I,v) is a set of du paths that start at I. **Du tour** is a tour that is def clear. **Criterion: use every def, get to every use, follow all du paths. All-defs Coverage=** For each set of du-paths S = du(n,v) TR contains at least one path in S (USE EVERY DEF). **All-uses Coverage=** for each set of du-paths to uses S = du(I,j,v) TR contains at least one path d in S (GET TO EVERY USE). **All-du-paths Coverage=** For each set S = du(I, j, v) TR contains every path d in S (FOLLOW ALL DU PATHS). **Control Flow Graph=** source code graph. **Decision Nodes →** represent choices in control flow**, Loops:** require dummy nodes. **Call Graph:** Nodes represent methods, edges are method calls, method coverage and call coverage are what they sound like. **Parameter Coupling:** defined by parameter passing from caller to callee**, Return Value Coupling:** defined by return value passing from callee to caller, **Shared data coupling:** defined by shared variables between caller and callee **external device coupling:** defined by shared use of a device by caller and callee. **Last def of x** set of locations that last define x in a unit (caller or callee), **First Use:** set of locations that first use x (**define coupling du-pairs). All-Coupling-Defs Coverage:** For each last-def of x, cover at least one first use. **All Coupling Uses Coverage:** for each all first uses. **All coupling DU paths coverage:** for each last-def of x, cover every first use. **Logic Coverage allows for reach but also ensures that states are infected. Predicate:** expression that evaluates to a Boolean value. **Clause:** predicate with no logical operators. **Predicate Coverage:** p evaluates to True and False. **Clause Coverage:** ensure each clause evaluates to true and false. **Combinatorial Coverage:** all possible combination of truth values (a lot of tests). **Active Clauses:** determining factor in the value of the predicate. **Active Clause Coverage=**major clause evaluates to true and false while minor held constant. **General Active Clause Coverage=** For each major clause, choose minor clauses so C determines predicate, major clause = T&F, minor clauses need not be same. **Correlated Active Clause Coverage=**same but predicate value evaluates to true and false and minor clauses do not need to be the same. **Restricted Active Clause Coverage=**same but minor clauses are the same and p evaluates to true and false. **ICC:** clause shluld be True with P=T, False with P=T, True with P=F, and False with P=F. **GICC:** find minor clauses such that major clause does not determine P. **RICC:** Minor clauses must be the same. **Grammer Coverage Criteria:**Terminal Symbol Coverage: all TSs reached**,** Production Coverage: Each production rule reached**,** Derivation Coverage: every possible derivation described, usually not possible. **Mutation Testing:** grammars describe both valid and invalid strings, mutants are invalid or valid changes made to valid strings (based on mutation operators). **Mutation Operators**: process of changing the software artifact based on defined rules. **Grammar:** rules are defined on syntactic description. **Ground String=**thing in grammar, **Mutation operator:** rule that specifies syntactic variation in a string, **Mutant:** result of one application of mutation op. **Killing mutant:** if mutant changes the output then the test kills it. **Mutant Score=** # killed mutants/total #mutants. **Mutant Coverage=** one requirement to kill each mutant. **Mutant Operator Coverage=** for each operator, TR contains exactly one requirement to create a mutant using that operator. **Mutant production Coverage=** for each mutation op, TR contains several requirements to create a mutant that contains everything that operator can make. **Syntax Dead Mutant:** a test case has killed it. **Uncompilable Mutant:** syntactically illegal. **Trivial Mutant:** almost every test kills it. **Equivalent Mutant:** no test can kill it. **Mutation Coverage:** Tr contains one requirement to kill M. **Strong Mutation Coverage=** one TR to strongly kill m. **Weak Mutation Coverage=** one TR to weakly kill m (no propagation).