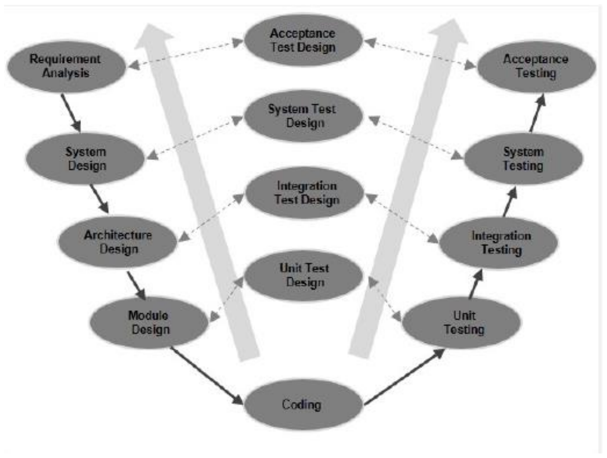
Week1

- V-model for software development



- Traceability matrix is a document that links each artifact of development phase to those of other phases.
- Validation is the process of checking if the software meets its requirements.
- Verification is 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.
- Levels of testing maturity

  Level 0: Testing = Debugging.
  Level 1: Show correctness.
  Level 2: Show that software doesn't work.
  Level 3: Reduce the risk of using the software.
  Level 4: Mental discipline that helps to develop higher quality software.

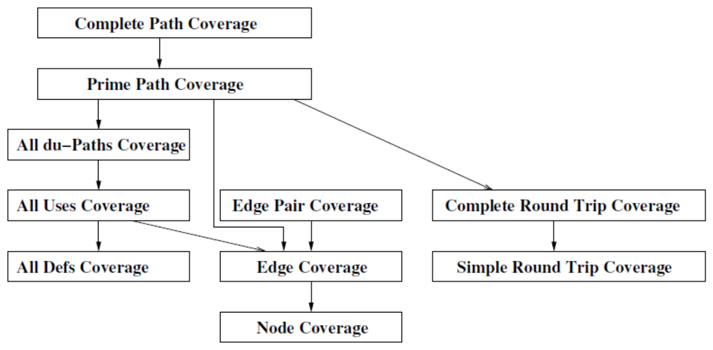
Week2

- For both directed and undirected graphs, adjacency list representation requires $\theta(|V|+|E|)$ memory.
- For both directed and undirected graphs, adjacency matrix representation requires $\theta(|V|^2)$ memory.
- Total running time of BFS and DFS is $O(|V|+|E|)$.
- In order to look for reachable nodes in a graph, use BFS or DFS.
- BFS identifies the shortest path in a graph.
- Forward edges (connects edge to its descendant) and backward edges (connects edge to its ancestor) in a graph can be identified using DFS. All other edges in a graph are called cross-edges.
- All nodes in a strongly connected component can be reached from each other, either through a forward or a backward edge.
- For Node coverage, TR contains every node in the graph.
- For Edge coverage, TR contains each reachable path of length up to 1 (this includes a node also)
- For Edge pair coverage, TR contains each reachable path of length up to 2.
- Covering all paths may not be feasible, if the graph contains loops.

- A path from one node to another is simple, if no node appears more than once.
- Prime path is a simple path that doesn't appear as a proper sub-path of any other simple path. In other words, prime path is a maximal simple path.
- Prime-path coverage subsumes node and edge coverage, but not edge-pair coverage.
- In some graphs, structural coverage criteria have infeasible TRs, when side trips are not allowed.

Week3

- A valid test path must start with the initial node and end with the terminal node.
- du-pairs may have multiple *defs* and *uses* in the path. Ensure that a path is traceable in the graph from the *def* to *use*.
- du-paths can trace any *simple path* (no node repeats), as far as it's *def-clear*. du-paths can have multiple *uses* in the path.
- There are three common data flow criteria:
  - All defs coverage: Each def reaches at least one use.
  - All uses coverage: Each def reaches all possible uses.
  - All du-paths coverage: Each def reaches all possible uses through all possible du-paths.
- Subsumption of various coverage cases.



Week4

- Types of interfaces are procedure call, shared memory and message-passing interfaces.
- Client-server and web-based systems are examples of message-passing interfaces.
- Scaffolding (Stub/driver) required to perform integration testing on components that're not ready yet.
- Top-down approach to software development required stubs and bottom-up approach requires drivers to perform integration testing.
- Call graphs are used as graph models in integration testing, where nodes are modules/stubs/drivers and edges are call interfaces.
- Call site refers to the statement where the actual call to the module appears in code.
- Different kinds of couplings in interfaces are parameter, shared data, external device and message passing.
- *Last def* refers to the node(s) that define a variable and has a def-clear path through the call site to the use in the callee module.

- *First use* refers to the node(s) that uses the variable and has a def-clear and use-clear path to it.
- There are three common data flow criteria:
  - All coupling-defs coverage: Each last def reaches at least one first use.
  - All coupling-uses coverage: Each last def reaches all possible first uses.
  - All coupling-du-paths coverage: Each last def reaches all possible first uses through all possible du-paths.
- Sequencing constraints are rules that impose constraints on the order in which methods may be called. Thus in a queue, *enqueue* needs to be called before *dequeue*. Also, there must be at least as many calls to *enqueue* as there are calls to *dequeue*.
- Finite state machine (FSM) is a graph that describes how software variables are modified during execution. In FSM, nodes represent states and edges represent transitions (with/out guards or actions)
- CFG or Call graphs are significantly different from FSM.
- A queue can be represented using an FSM with 4 states. (null, null) transitions to (obj, null) on *enqueue*, which transitions to (obj, obj) on *enqueue*, which transitions to (null, obj) on *dequeue*. Further, (obj, null) transitions to (null, null) on *dequeue*. All other transitions are invalid.
- Cyclomatic complexity of CFG is M = E - N + 2P, where P = #connected components.
- Strongly connected components in CFG are obtained by connecting the terminal node(s) back to the initial node. Typically, it has a value equal to 1, thus in such cases, M = E – N + 2.
- # linearly independent paths (basis paths) = cyclomatic complexity.
- Basis path testing subsumes branch coverage. Complete path coverage subsumes basis path testing.
- Node coverage = Statement coverage; Edge coverage = Branch coverage; Prime path coverage = Loop coverage
- DD paths include terminal nodes, nodes with one incoming/outgoing edge, nodes with 2 or more incoming/outgoing edges (conditions), maximal chain of nodes each with one incoming/outgoing edge.

Week5

- Atomic proposition is a term that is either true or false, but not both.
- Propositions can be combined with the following logical connectives, to form formulas – and, or, negation, implies, equivalent
- Equivalent representations of logical connectives.
  - And: $\alpha \wedge \beta$: $\neg(\neg\alpha \vee \neg\beta)$
  - Implies: $\alpha \supset \beta$: $\neg\alpha \vee \beta$
  - Iff: $\alpha \equiv \beta$: $(\alpha \supset \beta) \wedge (\beta \supset \alpha)$.
- Truth table of *implies*.

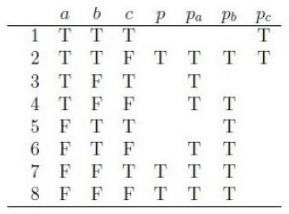
p	q	p ⊃ q
T	T	T
T	F	F
F	T	T
F	F	T

  NOTE: When *p* is False, ***p implies q*** is regarded True, irrespective of the value of *q*.
- A formula α is satisfiable, if there exists *any* valuation v such that v(α) = True.

- A formula α is valid or is called tautology, if for *every* valuation v, v(α) = True.
- For an atomic proposition **p**, the formula **p V not p** is considered valid, and the formula **p ^ not p** is not satisfiable (contradiction).
- Formula **p** is valid if **not p** is not satisfiable**.**
- Satisfiability problem for propositional logic is NP-complete. At best, the algorithm takes indefinite amount of time to solve.
- A clause is a predicate that doesn't contain any logical operator.
- In predicate coverage, there are two TRs – one when predicate evaluates to True, and another when predicate evaluates to False.
- For a set of predicates associated with branches, predicate coverage is the same as edge coverage.
- Predicate coverage doesn't subsume clause coverage. Likewise, clause coverage doesn't subsume predicate coverage.
- Combinatorial coverage contains 2^n TRs where each of the n clauses can be suitably combined to evaluate to truth/false values. This may an overkill many a time, and hence ACC.
- Major clause c_i **determines** predicate *p*, if the minor clauses take values such that changing the value of c_i will alter the value of *p*. TR has two requirements such that *p* evaluates to True and False.
- To determine the conditions under which the major clause determine the predicate, follow these steps:
    - Evaluate the predicate with major clause replaced as True ($p_{a=true}$)
    - Evaluate the predicate with major clause replaced as False($p_{a=false}$)
    - Find the XOR of $p_{a=true}$ and $p_{a=false}$
    - For example,
        - Consider $p = a \vee b$.
        - Then,

$$\begin{aligned} p_a &= p_{a=true} \oplus p_{a=false} && (1)\\ &= (true \vee b) \oplus (false \vee b) && (2)\\ &= true \oplus b && (3)\\ &= \neg b && (4) \end{aligned}$$

        - For major clause a to determine p, the only minor clause b must be false.
        - Symmetrically, $p_b = \neg a$.
- If p_a evaluates to true, then ICC criteria is infeasible. Similarly, if p_a evaluates to false, then ACC criteria is infeasible. In either case, the clause a is redundant in the predicate.
- For a predicate with n clauses, n + 1 distinct test requirements suffice to achieve ACC.
- MCDC, which is part of the conventional coverage, is same as ACC.
- GACC has two TRs, when c_i evaluates to True and False like in ACC and influences the predicate, but has no restrictions on minor clauses – could be same or different.
- CACC has two TRs, when c_i evaluates to True and False like in ACC and influences the predicate, but with different minor clauses.
- RACC has two TRs, when c_i evaluates to True and False like in ACC and influences the predicate, but with same minor clauses.
- For a predicate with the below truth table,



	a	b	c	p	p_a	p_b	p_c
1	T	T	T				T
2	T	T	F	T	T	T	T
3	T	F	T		T		
4	T	F	F		T	T	
5	F	T	T			T	
6	F	T	F		T	T	
7	F	F	T	T	T	T	
8	F	F	F	T	T	T	

To find the CACC pairs for clause a, follow these steps.

- List down all rows that has p_a = T
- Group the ones with a = T together, and a = F together.
- Each row from first group can be paired with each row in the second group, as far as the predicate p had different values.

Ans: (2,6), (3,7), (3,8), (4,7), (4,8)

To find RACC pairs for clause a, follow these steps.

- List down all rows that has p_a = T
- Group the ones with a = T together, and a = F together.
- Each row from first group can be paired with each row in the second group, as far as the predicate p had different values, and b and c (minor clauses) remain the same in both rows.

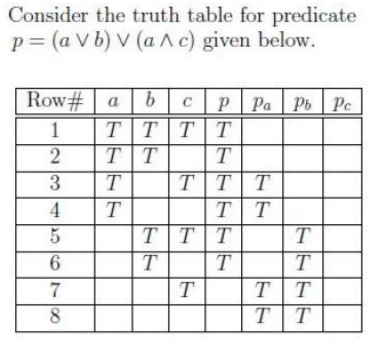
Ans: (2,6), (3,7), (4,8)

To find RICC pairs for clause a, follow these steps.

- List down all rows that has p_a = F
- Group the ones with a = T together, and a = F together.
- Each row from first group can be paired with each row in the second group, as far as the predicate p had same values, and b and c (minor clauses) remain the same in both rows. Typically, RICC have infeasible tests.

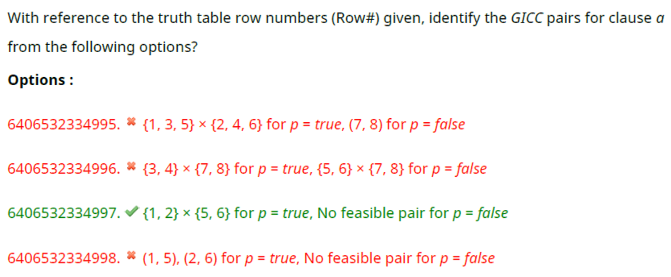
Ans: Nor feasible for p = T, as well as (1,5) for p = F

NOTE In certain cases, the options might use set cross-product representation instead of listing pairs. Consider this example from Sep '23 end-term.



Consider the truth table for predicate $p = (a \vee b) \vee (a \wedge c)$ given below.

Row#	a	b	c	p	p_a	p_b	p_c
1	T	T	T	T			
2	T	T		T			
3	T		T	T	T		
4	T			T	T		
5		T	T	T		T	
6		T		T		T	
7			T		T	T	
8					T	T	


With reference to the truth table row numbers (Row#) given, identify the *GICC* pairs for clause *a* from the following options?

Options :

6406532334995. ✖ {1, 3, 5} × {2, 4, 6} for *p = true*, (7, 8) for *p = false*

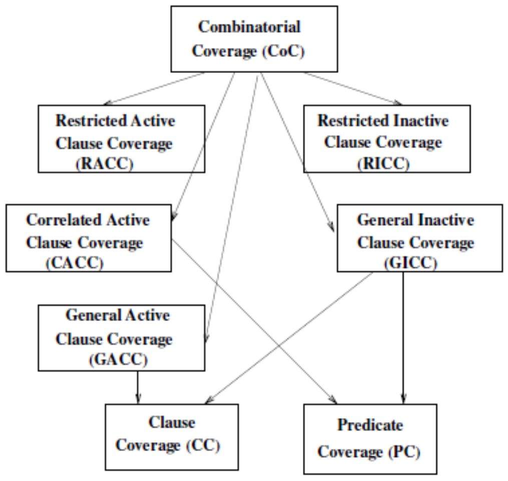
6406532334996. ✖ {3, 4} × {7, 8} for *p = true*, {5, 6} × {7, 8} for *p = false*

6406532334997. ✔ {1, 2} × {5, 6} for *p = true*, No feasible pair for *p = false*

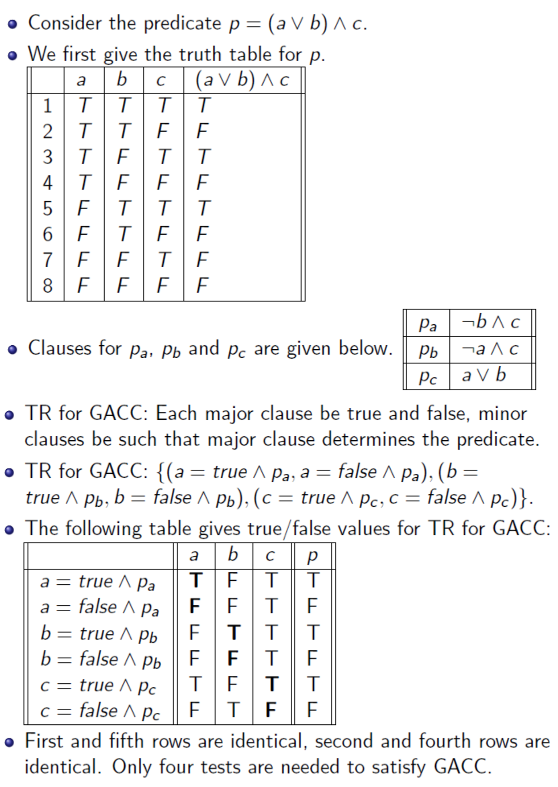
6406532334998. ✖ (1, 5), (2, 6) for *p = true*, No feasible pair for *p = false*

In this case, (1,5), (2,6) are right pairs of tests for p = true. But option #4 is not the correct option, because in addition to the above pairs, (1,6) and (2,5) are also right pairs of tests. Hence, the cross-product of sets {1,2} x {5,6} is the correct representation. So, option #3 is the right answer for this question.

- ACC subsumption.


Combinatorial Coverage (CoC)
Restricted Active Clause Coverage (RACC)
Restricted Inactive Clause Coverage (RICC)
Correlated Active Clause Coverage (CACC)
General Inactive Clause Coverage (GICC)
General Active Clause Coverage (GACC)
Clause Coverage (CC)
Predicate Coverage (PC)

- If the minor clauses are such that, the major clause does NOT determine the predicate, it's called ICC. Similar to ACC, GICC has no restrictions on the choice of minor clauses, but the minor clauses are required to be same in RICC. CICC doesn't make sense, since major clause anyway does NOT determine the predicate.
- RACC is often not feasible in many practical situations. In such cases, if CACC is feasible, consider CACC.
- When a predicate only has one clause, CoC, ACC, ICC, and CC all collapse to predicate coverage (PC).
- https://discourse.onlinedegree.iitm.ac.in/t/live-session-6-20th-july-t2-2023/95935/3?u=anand
- Here is an example case that finds the GACC for a predicate.



- Consider the predicate $p = (a \vee b) \wedge c$.
- We first give the truth table for p.

	a	b	c	$(a \vee b) \wedge c$
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

- Clauses for p_a, p_b and p_c are given below.

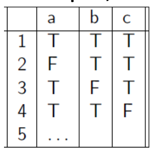
p_a	$\neg b \wedge c$
p_b	$\neg a \wedge c$
p_c	$a \vee b$

- TR for GACC: Each major clause be true and false, minor clauses be such that major clause determines the predicate.
- TR for GACC: $\{(a = true \wedge p_a, a = false \wedge p_a), (b = true \wedge p_b, b = false \wedge p_b), (c = true \wedge p_c, c = false \wedge p_c)\}$.
- The following table gives true/false values for TR for GACC:

	a	b	c	p
$a = true \wedge p_a$	**T**	F	T	T
$a = false \wedge p_a$	**F**	F	T	F
$b = true \wedge p_b$	F	**T**	T	T
$b = false \wedge p_b$	F	**F**	T	F
$c = true \wedge p_c$	T	F	**T**	T
$c = false \wedge p_c$	F	T	**F**	F

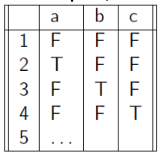
- First and fifth rows are identical, second and fourth rows are identical. Only four tests are needed to satisfy GACC.

- A predicate is in Conjunctive Normal Form (CNF) if it consists of clauses or disjuncts connected by the conjunction (and) operator. In this case, ACC TR is all True, and then a diagonal of False. For example, for the predicate (a ^ b ^ c), ACC TR is given by



	a	b	c
1	T	T	T
2	F	T	T
3	T	F	T
4	T	T	F
5	. . .		

- A predicate is in Disjunctive Normal Form (DNF) if it consists of clauses or conjuncts connected by the disjunction (or) operator. In this case, ACC TR is all False, and then a diagonal of True. For example, for the predicate (a V b V c), ACC TR is given by



	a	b	c
1	F	F	F
2	T	F	F
3	F	T	F
4	F	F	T
5	. . .		

Week7

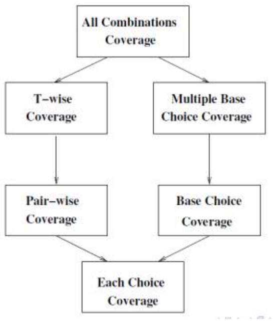
- Program proving involves formal methods to prove programs correct.
- 3 techniques in formal methods: Model checking, Theorem proving, Program analysis
- Symbolic execution uses symbolic values instead of concrete data values as input.
- 3-part series on symbolic execution (https://youtu.be/wOO5jpoFIss)
- Symbolic state σ maps variables to symbolic expressions.
- Symbolic path constraint, PC, is a quantifier-free, first-order formula over symbolic expressions.
- Here are the steps for symbolic execution.

    - At the beginning of symbolic execution, σ is initialized to an empty map and PC is initialized to True.
    - At every assignment, v = e, σ is updated by mapping v to σ(e) and PC is updated to ^ *σ(e)*. A new path constraint is created with PC' = ^ *not σ(e).*
    - The *then* and *else* branches result in two different execution paths.
    - If any of PC or PC' is not satisfiable, symbolic execution terminates along the corresponding path.
    - At the end of symbolic execution along an execution path of the program, PC (or PC') is solved using a constraint solver to generate concrete input values.
- In general, symbolic execution of code containing loops or recursion may result in an infinite number of paths if the termination condition for the loop or recursion is symbolic.
- Disadvantages of symbolic execution.
    - If the PC generated during a symbolic execution contains formulas that cannot be (efficiently) solved by a constraint solver, symbolic execution cannot generate test inputs.
    - Many real-life programs have huge number of program paths, resulting in several path constraints.
    - When parts of the programs rely on libraries that don't have available code, PC cannot be solved as well.
- Concolic testing starts by assigning random concrete values to the variables. It maintains a concrete state and a symbolic state at every step of execution.

Week8

- Characteristics (inputs) that define partitions must ensure that the partition satisfies two properties.
    - Completeness: The partitions must cover the entire domain
    - Disjoint: The partitions must not overlap.
- ISP can lead to valid or invalid tests, typically when outside the partition.
- Input domain can be modelled either using Interface, or Functionality.
- Typical sources for identifying functionality based characteristics are:
    - Pre-post conditions
    - Implicit-explicit relationships.
    - Missing functionality.
- If we have three partitions as [A,B], [1,2,3] and [x,y], then All Combinations coverage (ACoC) will have twelve tests, given by $\Pi_{i=1}^{n} B_i.$
- If we have three partitions as [A,B], [1,2,3] and [x,y], then Each Choice coverage (ECC) will have 3 tests, given by $Max_{i=1}^{n} B_i$
- If we have three partitions as [A,B], [1,2,3] and [x,y], then Pair-wise coverage (PWC) will have 8 tests, given by $(Max_{i=1}^{n} B_i)^2$
- T-wise coverage will have $(Max_{i=1}^{n} B_i)^t$ tests. If T = #partitions, then T-wise coverage and ACoC are same.
- Base-choice coverage considers an important choice from each characteristic. Tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic. Number of such tests is given by $1 + \Sigma_{i=1}^{n}(B_i - 1)$
  https://discourse.onlinedegree.iitm.ac.in/t/previous-paper-question/124027/4

- Multiple Base-choice coverage considers multiple important choices from each characteristic. Tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic. Number of such tests is given by $M + \Sigma_{i=1}^{n}(M * (B_i - m_i))$
- Subsumption of ISP tests.



Week9

- Syntax can be used to generate artifacts that are valid and those that are invalid.
- Levels of syntax in programming languages – words (uses regex), phrases (uses BNF), context.
- Words can be represented using the regex $r ::= \emptyset \mid a \mid r + r \mid r \cdot r \mid r^*$, where $a \in A$.
- Semantics of the language of words are defined as

$$\begin{array}{lcl} L(\emptyset) & = & \{\} \\ L(a) & = & \{a\} \\ L(r + r') & = & L(r) \cup L(r') \\ L(r \cdot r') & = & L(r) \cdot L(r') \\ L(r^*) & = & L(r)^*. \end{array}$$

- Production rule for context-free grammar.

$$\begin{array}{lcl} S & \rightarrow & aX \\ X & \rightarrow & aX \\ X & \rightarrow & bX \\ X & \rightarrow & b \end{array}$$

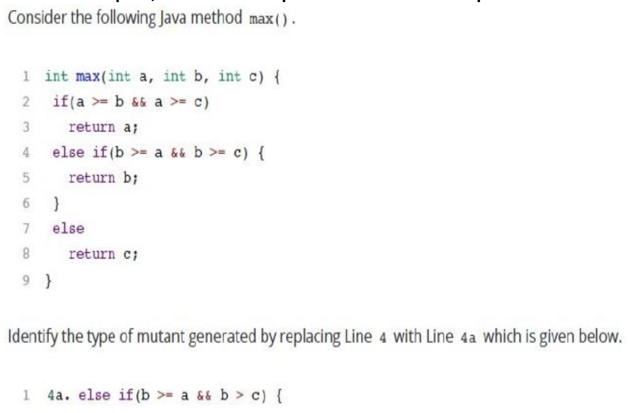
  This set of production rules can generate **ab, *abb***, ***aab***, ***aabb etc.***
  NOTE: Production rules always starts with the symbol S.
- Context-free grammar (CFG) is mathematically represented as
  G = (N, A, S, P)
  where
  - N is a finite set of non-terminal symbols.
  - A is a finite set of terminal symbols.
  - S is the start non-terminal symbol
  - P is a finite subset of N x (N U A)*, called the set of productions or rules.

NOTE: The word *terminal* symbol denotes a symbol that cannot be further sub-divided, or is atomic.

- Program to which mutation is applied is called ground-string. It can be mutated by applying a mutation operator.
- When the ground-string is mutated, the resultant is called a *mutant*. When a test runs on the mutant, if it generates an output that is different from that generated when the test runs on the ground-string, the test is said to kill the mutant. The objective of mutation testing is to kill the mutants by adding more tests into the test suite. Note that a single test can kill multiple mutants.

- The amount of coverage is usually written as the % mutants killed by the test suite and is called mutation score. Note that a higher mutation score doesn't mean more effective testing. It only means more mutants were killed. Effectiveness depends on the value-add of the mutants themselves.
- As mentioned before, goal of mutation testing is to kill (fail) the mutant. If this can't be achieved with existing tests, you'll add a new test that can pass the original program and kill (fail) the mutant. Typically, the older test could be removed in favor of the new test, but might be retained, depending on whether it's made completely redundant.
- Variants of mutants.
    - A mutant that can't be compiled is called a stillborn mutant.
    - A mutant that can be killed by almost any test is called a trivial mutant.
    - A mutant that is functionally equivalent to the ground-string is called an equivalent mutant.
    - A mutant that can be killed by a test case is called as dead mutant.

    For example, here is a question from Sep '22 term.



Consider the following Java method `max()`.

```
1  int max(int a, int b, int c) {
2    if(a >= b && a >= c)
3      return a;
4    else if(b >= a && b >= c) {
5      return b;
6    }
7    else
8      return c;
9  }
```

Identify the type of mutant generated by replacing Line `4` with Line `4a` which is given below.

```
4a. else if(b >= a && b > c) {
```

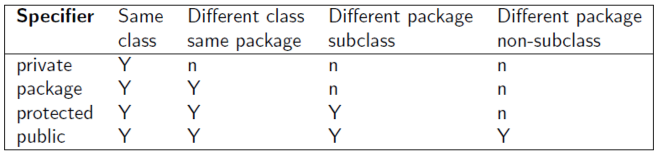
    This is an equivalent mutant, since if b equals c, it doesn't matter if line#5 runs or line #8 runs.

- Mutant is not always *required* to generate an output that's different from the ground-string. Instead, it could result in a state change. In this case, we say that *reachability* and *infection* conditions were satisfied, but not *propagation*.
- When propagation is satisfied with a mutant, it's said to be strongly killed. When the mutation coverage focuses on strongly killing mutants, it's called Strong Mutation Coverage (SMC).
- When propagation is NOT satisfied with a mutant, it's said to be weakly killed. When the mutation coverage focuses on weakly killing mutants, it's called Weak Mutation Coverage (WMC).
- There are many possible mutation operators that can be used in a programming language. Which operators can be used to generate mutants on the ground-string depends on the specific context and the requirements. Also, typically, only one operator is applied at a time to generate mutants.
- Some examples of mutation operators are:
    - Absolute value insertion (Abs, negAbs, failOnZero)
    - Arithmetic Operator Replacement
    - Relational Operator Replacement
    - Conditional Operator Replacement
    - Shift Operator Replacement
    - Logical Operator Replacement
    - Assignment Operator Replacement

- Unary Operator Replacement
    - Unary Operator Deletion/Insertion
    - Scalar variable replacement
    - Bomb Statement Replacement
- Mutation testing subsumes node and edge coverage, all-defs coverage.
- Mutation testing subsumes predicate, clause and general active clause coverage.
- Mutation testing does not subsume combinatorial coverage and CACC, RACC.
- Interface (integration) mutation:
    - Parameter Variable Replacement (PVR)
    - Unary Operator Insertion (UOI)
    - Parameter Exchange (PEX)
    - Method Call Deletion (MCD)
    - Return Expression Modification (REM)

Week10

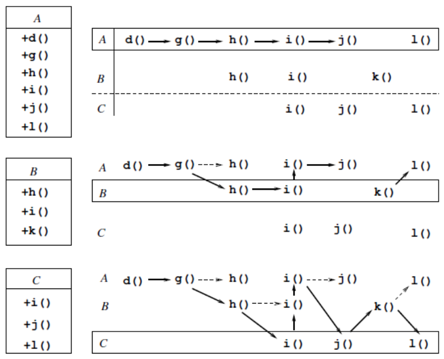
- Access specifier rules in Java



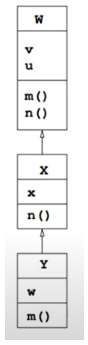
Specifier	Same class	Different class same package	Different package subclass	Different package non-subclass
private	Y	n	n	n
package	Y	Y	n	n
protected	Y	Y	Y	n
public	Y	Y	Y	Y

- Sub-classes can explicitly use parent's variables and methods (including constructors) using the keyword *super* (super.methodname();)
- When the parent constructor is used in the constructor of the sub-class, the former must be the first line inside the latter. However, this rule doesn't apply for using parent methods (or variables) elsewhere in the program. It can appear anywhere.
- Following are some of the mutation operators employed in OO programs.
    - Access Modifier Change (AMC)
    - Hiding Variable Deletion (HVD)
    - Hiding Variable Insertion (HVI)
    - Overriding Method Deletion (OMD)
    - Overriding Method Moving (OMM)
    - Overridden Method Rename (OMR)
    - Super Keyword Deletion (SKD)
    - Parent Constructor Deletion (PCD)
    - Actual Type Change (ATC)
    - Declared/Parameter Type Change (DTC/PTC)
    - Reference Type Change (RTC)
    - Overloading Method Change (OMC)
    - Overloading Method Deletion (OMD)
    - Argument Order Change (AOC)
    - Argument Number Change (ANC)
    - this Keyword Deletion (TKD)
    - Static Modifier Change (SMC)
    - Variable Initialization Deletion (VID)
    - Default Constructor Deletion (DCD)
- Polymorphic call set represents a set of methods from parent and child classes with the same name, that can be called polymorphically.

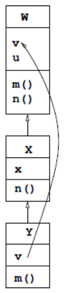
- OO Classes are tested at any of the following 4 levels:
    - Intra-method (unit) testing: Individual methods are tested separately.
    - Inter-method testing: Multiple methods are tested in concert.
    - Intra-class testing: Sequence calls to methods within the same class.
    - Inter-class testing: Integration of two or more classes.
- Yo-yo graph of a class hierarchy A, B, C where B and C inherits A and overrides certain methods of A.



NOTE: The methods called are determined dynamically depending on the object that calls these methods. For example, if object of class B calls **d**, **d** calls **g**, which calls **h** defined in the class B (not class A). Continuing on the sequence of calls, **h** calls **i** defined in class B. However, **i** calls **j** defined in class A, since **j** is not defined in class B.
- Following are some more OO faults:
    - Inconsistent Type Use (ITU):
      A descendent class does not override any inherited method – hence no polymorphic behavior. Object is used as child, then as parent which can cause inconsistency in the state of the object and using it as a child again, could cause issues.
    - State Definition Anomaly (SDA):

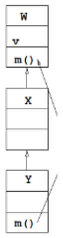


      Suppose v is defined by *W::m()* and used in *W::n()* as well as *X::n()*. Note that *Y::m()* doesn't define v. Now, if the method m() is called on an object of class Y, and calls the method n() in sequence from either of the parent classes, it'll give rise to a fault (since v doesn't exist)
    - State Definition Inconsistency Anomaly (SDA):



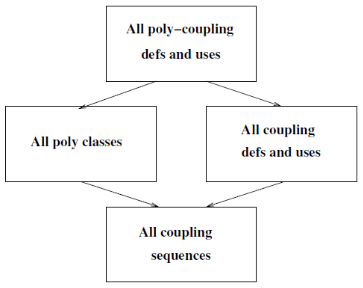
Suppose v is defined by *W::m()* and *Y::m()* used in *W::n()* as well as *X::n()*. Now, if the method m() is called on an object of class Y, and calls the method n() in sequence from either of the parent classes, it'll give rise to a fault (since Y::v gets defined, but W::v gets used)

- State Visibility Anomaly (SVA):



If W declares a private variable v, it won't be accessible from Y::m() because it's private.

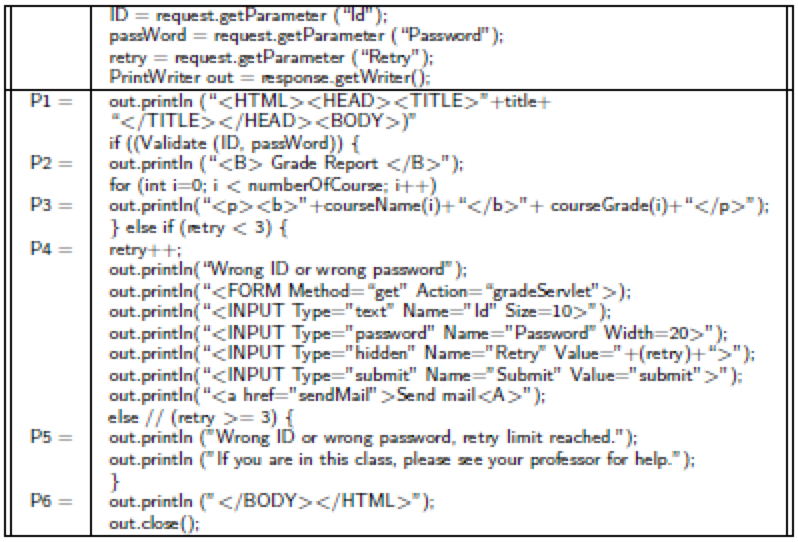
- Calling method calls *antecedent* method (that defines a variable), which in turn calls *consequent* method (that uses the variable).
- OO coupling coverage criteria subsumption



Week11

- Static Vs Dynamic websites. Dynamic websites need client-side and server-side testing.
- Communication between client and server happens through messages, and sometimes through shared memory (session objects).
- HTTP is a stateless protocol. State must be managed by using cookies, session objects etc.
- Web applications are inherently concurrent and distributed.
- Web apps deployment:
  - Bundled (pre-installed on computer)
  - Shrink-wrap (bought and installed by end-users)
  - Contract (Developer builds for the purchaser)
  - Embedded (Installed on hardware device)
  - Web (Cloud)

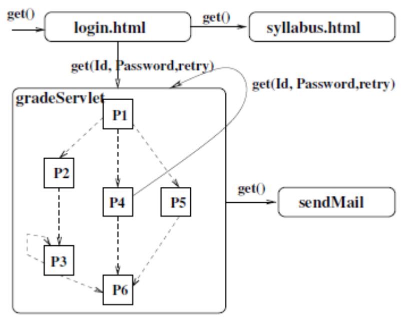
- In a 3-tier model of the web app, there are presentation, application and storage layers.
- In a 4-tier model, there are presentation, data content (computation and data access), data representation (in-memory data storage) and data storage (permanent data storage) layers.
- Static web page testing involves checking for dead links.
- Test techniques for client-side testing is agnostic of technologies used to develop client.
- In client-side testing, inputs can be one of user supplied, generated randomly, generated from user-session data, or bypass testing.
- In bypass testing, client-side tests are removed and sent directly to the server and tested using inputs that violate client-side rules. Following are types of bypass testing.
    - Value-level: uses input that uses invalid datatype, invalid length, corrupts data, causes security vulnerability
    - Parameter-level: uses input that causes mismatch in relationship between input parameters.
    - Control-flow level: forcefully alter the control-flow using browser navigation buttons
- User session data can be used to test client. Following are variations of it:
    - Directly reuse entire sessions
    - Replay a mixture of sessions. Use *1-i* requests from user session a, and use *i-n* requests from user session b.
    - Replay sessions with some targeted modifications. Same as above, but change the form data being sent to the server.
- Invalid input passed to the server might cause:
    - Valid server response (Invalid inputs adequately processed)
        - Generate an explicit message from the server.
        - Generate a generic error message.
        - Input might be ignored.
    - Faults and failures (Invalid inputs cause abnormal server error)
    - Exposure (Invalid inputs not recognized and users exposed abnormal behavior)
        - Corruption of data.
- For software in presentation layer, two graph models exist to test it.
    - Component Interaction Model (CIM)
        - Models individual components.
        - Combines atomic sections. Each atomic section might use external data called "content" variables. Atomic sections can be empty.
        - Intra-component
    - Application Transition Graph (ATG)
        - Each node is one CIM
        - Edges are transitions among CIMs
        - Inter-component.
- CIM can be represented using a language similar to reg-expressions. For example, for the *gradeServlet* code used by an academic institution is shown below.



```
       ID = request.getParameter ("Id");
       passWord = request.getParameter ("Password");
       retry = request.getParameter ("Retry");
       PrintWriter out = response.getWriter();
P1 =   out.println ("<HTML><HEAD><TITLE>"+title+
       "</TITLE></HEAD><BODY>)"
       if ((Validate (ID, passWord)) {
P2 =   out.println ("<B> Grade Report </B>");
       for (int i=0; i < numberOfCourse; i++)
P3 =   out.println("<p><b>"+courseName(i)+"</b>"+ courseGrade(i)+"</p>");
       } else if (retry < 3) {
P4 =   retry++;
       out.println("Wrong ID or wrong password");
       out.println("<FORM Method="get" Action="gradeServlet">);
       out.println("<INPUT Type="text" Name="Id" Size=10>");
       out.println("<INPUT Type="password" Name="Password" Width=20>");
       out.println("<INPUT Type="hidden" Name="Retry" Value="+(retry)+">");
       out.println("<INPUT Type="submit" Name="Submit" Value="submit">");
       out.println("<a href="sendMail">Send mail<A>");
       else // (retry >= 3) {
P5 =   out.println ("Wrong ID or wrong password, retry limit reached.");
       out.println ("If you are in this class, please see your professor for help.");
       }
P6 =   out.println ("</BODY></HTML>");
       out.close();
```

The code can be represented as P1.((P2.P3*)|P4|P5).P6.

- ATG for this use case can be represented as



Once the student logs in using *login.html*, he/she can choose to get the syllabus using *syllabus.html,* or display his/her grade using the CIM *gradeServlet* (discussed above)*,* followed by sending email to the student using *sendMail.* Note that if the *P4* fails to login, it'll redirect to login.html, as far as the number of trials is less than 3.

ATG has 5 types of transitions/edges.

- Simple link transition (*login.html* to *syllabus.html*)
- Form link transition (*login.html* to *gradeServlet)*
- Component Expression Transition (internal to *gradeServlet*)
- Operation Transition (Navigation buttons in browser)
- Redirect Transition (Server-side transition - invisible)

Week12

- Maintenance (new features, fixing bugs etc) necessitates regression testing.
- Regression testing detects whether new errors have been introduced into previous testing code.
- Test selection is very important is regression testing, and focus is to reduce the number of tests used during the process.
  - In case of spec changes identify tests that are made *obsolete*.
  - Requires domain knowledge.
  - Tests are selected based on the area of frequent defects.
  - Tests are selected to include the areas that has undergone most code changes.

    - Tests are selected based on the criticality of features.
    - Select all def-use pairs that have been deleted, added or modified in the newer version of the software.
    - Use explicitly stated safety conditions to determine the regression tests.
- Low quality increases cost.
- Types of software quality:
    - Functional quality: degree to which correct software was produced
    - Structural quality: degree to which it meets non-functional requirements, like robustness, security, usability, or maintainability.
- Software Quality Assurance is a guarantee on quality and involves process-focused action.
- Software Quality Control focuses on determining defects in software products, and involves product-focused action. Software Testing is part of Software Quality Control.
- Software metrics provide quantifiable measurements to various software development processes and resulting product.
    - Product metrics: Size (KLOC, binary size), Complexity, Coupling (interdependence between software modules), Cohesion (strength of relationship between methods and data in a class).
    - Process metrics: Focuses on processes used in software development, maintenance etc.
    - Project metrics: Team size, cost, schedule, productivity etc.
- Software quality metrics
    - Product: MTF, Defect density (DDE = #defects/KLOC), #customer-reported defects, CSAT score.
    - Process: Defect density, Defect arrival pattern (#defects per week/month), defect removal pattern, defect removal effectiveness (DRE = #defects removed/#defects introduced)
    - Maintenance: Fix backlog (BMI = #problems closed/#problems found), Fix response time.
- Non-functional testing is performed during system testing phase.
- Some types of non-functional tests:
    - Performance – Responsiveness, Stability. Load, Stress, Soak, Spike testing.
    - Interoperability –Compatibility (forward/backward) with other third-party products
    - Security – Confidentiality, Integrity, Availability, Authorization, Authentication
    - Reliability – Ability of system to keep operating over specified period of time. Example is MTF.
    - Scalability – Tests system limits.
    - Documentation – Read test, Hands-on test, Functional test
    - Regulatory – Safety critical systems
- Test Driven Programming (TDD) is part of Xtreme Programming (XP). It means Test first, code next per user story.
- After writing code for each story, they're put together (composed) to construct the final code the entire set of user stories. This could give rise to *code smell*. To get rid of the *code smell*, refactoring is performed.
- Advantages of TDD:
    - Fault isolation is easier.
    - Pair programming.
- Disadvantages of TDD:
    - Heavy dependence on test frameworks.
    - Without additional step of refactoring, *code smell* happens.