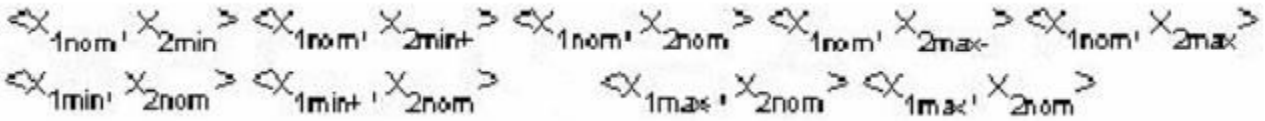
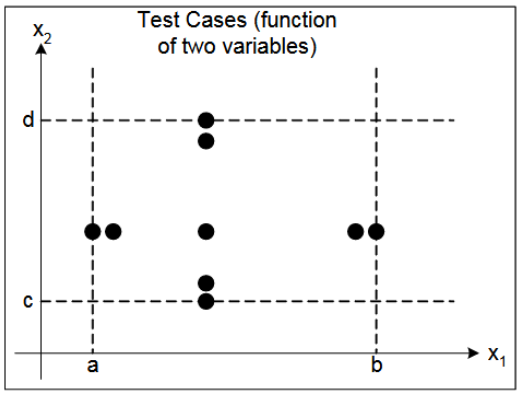
Answer Key – IAT2 – Software Testing(13MCA444)

1a. **BVA test case for two variables functions**

In the general application of Boundary Value Analysis can be done in a uniform manner. The basic form of implementation is to maintain all but one of the variables at their nominal (normal or average) values and allowing the remaining variable to take on its extreme values. The values used to test the extremities are:

•Min ---------- - Minimal
•Min+ ---------- - Just above Minimal
•Nom ---------- - Average
•Max- ---------- - Just below Maximum
•Max ---------- - Maximum





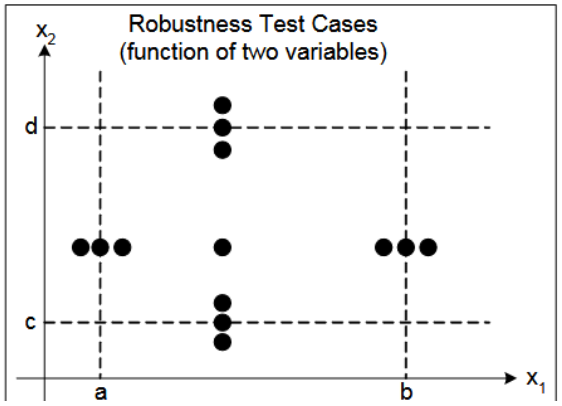
Limitations of BVA

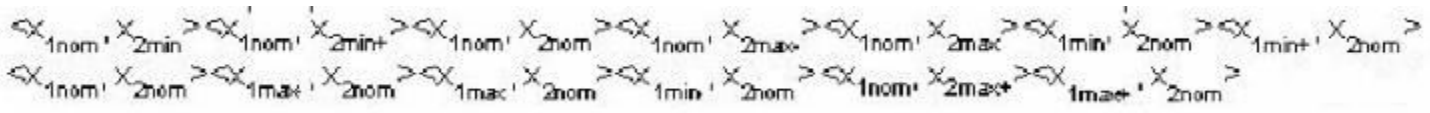
Boundary Value Analysis works well when the Program Under Test (PUT) is a "function of several independent variables that represent bounded physical quantities" [1]. When these conditions are met BVA works well but when they are not we can find deficiencies in the results. For example the NextDate problem, where Boundary Value Analysis would place an even testing regime equally over the range, tester's intuition and common sense shows that we require more emphasis towards the end of February or on leap years.
The reason for this poor performance is that BVA cannot compensate or take into consideration the nature of a function or the dependencies between its variables. This lack of intuition or understanding for the variable nature means that BVA can be seen as quite rudimentary.

1b. **Robustness Testing**

Robustness testing can be seen as and extension of Boundary Value Analysis. The idea behind Robustness testing is to test for clean and dirty test cases. By clean I mean input variables that lie in the legitimate input range. By dirty I mean using input variables that fall just outside this input domain. In addition to the aforementioned 5 testing values (min, min+, nom, max-, max) we use two more values for each variable (min-, max+), which are designed to fall just outside of the input range. If we adapt our function f to apply to Robustness testing we find the following equation: f= 6n+ 1
Equate this solution by the same reasoning that lead to the standard BVA equation. Each variable now has to assume 6 different values each whilst the other values are assuming their nominal value (hence the 6n), and there is again one instance whereby all variables assume their nominal value (hence the addition of the constant 1). Robustness testing ensures a sway in interest, where the previous interest lied in the input to the program, the main focus of attention associated with Robustness testing comes in the expected outputs when and input variab le has exceeded the given input domain. For example the NextDate problem when we an entry like the 31st

June we would expect an error message to the effect of "that date does not exist; please try again". Robustness testing has the desirable property that it forces attention on exception handling. Although Robustness testing can be somewhat awkward in strongly typed languages it can show up altercations. In Pascal if a value is defined to reside in a certain range then and values that falls outside that range result in the run time errors that would terminate any normal execution. For this reason exception handling mandates Robustness testing.





2a. **Equivalence Class Test**

EC Testing is when you have a number of test items (e.g. values) that you want to test but because of cost (time/money) you do not have time to test them all. Therefore you group the test item into class where all items in each class are suppose to behave exactly the same. The theory is that you only need to test one of each item to make sure the system works.

Example 1

Children under 2 ride the buss for free. Young people pay $10, Adults $15 and Senior Citizen pay $5.

Classes:

Price:0 -> Age:0-1

Price:10 -> Age:2-14

Price:15 -> Age:15-64

Price:5 -> Age:65-infinity

Example 2 (more than one parameter)

Cellphones K80, J64 and J54 run Java 5. K90 and J99 run Java 6. But there are two possible browsers FireFox and Opera, J models run FF and K models run O.

Classes:

Browser:FF, Java:5 -> Phones:J64,J54

Browser:FF, Java:6 -> Phones:J99

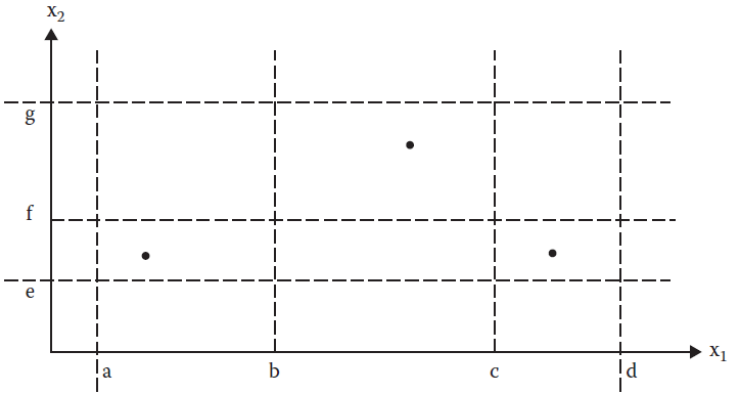
Browser:O, Java:5 -> Phones:K80

Browser:O, Java:6 -> Phones:K90

Weak Normal ECT

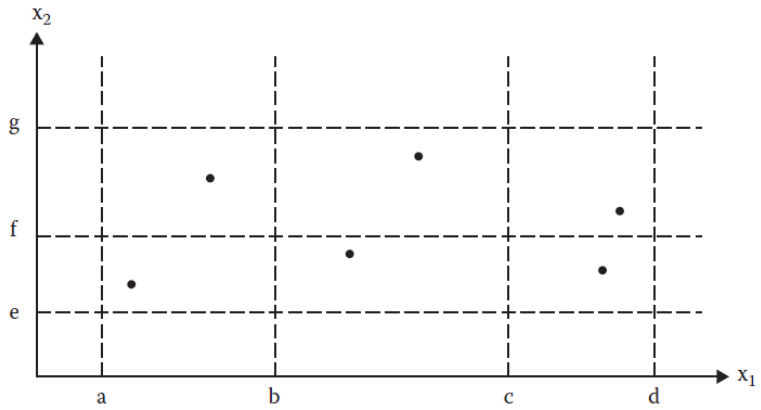
With the notation as given previously, weak normal equivalence class testing is accomplished by using one variable from each equivalence class (interval) in a test case. (Note the effect of the single fault assumption.) For the running example, we would end up with the three weak equivalence class tests. These three test cases use one value from each equivalence class. The test case in the lower left rectangle corresponds to a value of x1 in the class [a, b), and to a value of x2 in the class [e, f). The test case in the upper center rectangle corresponds to a value of x1 in the class [b, c) and to a value of x2 in the class [f, g]. The third test case could be in either

rectangle on the right side of the valid values. We identified these in a systematic way, thus the apparent pattern. In fact, we will always have the same number of weak equivalence class test cases as classes in the partition with the largest number of subsets. What can we learn from a weak normal equivalence class test case that fails, that is, one for which the expected and actual outputs are inconsistent? There could be a problem with x1, or a problem with x2, or maybe an interaction between the two. This ambiguity is the reason for the "weak" designation. If the expectation of failure is low, as it is for regression testing, this can be an acceptable choice. When more fault isolation is required, the stronger forms, discussed next, are indicated.



Strong Normal ECT

Strong equivalence class testing is based on the multiple fault assumption, so we need test cases from each element of the Cartesian product of the equivalence classes. Notice the similarity between the pattern of these test cases and the construction of a truth table in propositional logic. The Cartesian product guarantees that we have a notion of "completeness" in two senses: we cover all the equivalence classes, and we have one of each possible combination of inputs. As we shall see from our continuing examples, the key to "good" equivalence class testing is the selection of the equivalence relation. Watch for the notion of inputs being "treated the same." Most of the time, equivalence class testing defines classes of the input domain. There is no reason why we could not define equivalence relations on the output range of the program function being tested; in fact, this is the simplest approach for the triangle problem.



2b. **DD-Path**

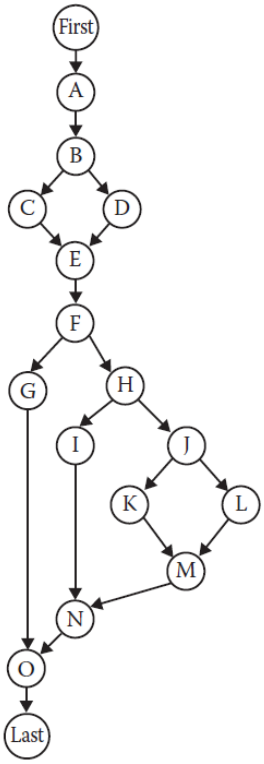
A *DD-path* is a sequence of nodes in a program graph such that
Case 1: It consists of a single node with indeg = 0.
Case 2: It consists of a single node with outdeg = 0.
Case 3: It consists of a single node with indeg ≥ 2 or outdeg ≥ 2.
Case 4: It consists of a single node with indeg = 1 and outdeg = 1.
Case 5: It is a maximal chain of length ≥ 1.
Cases 1 and 2 establish the unique source and sink nodes of the program graph of a structured program as initial and final DD-paths. Case 3 deals with complex nodes; it assures that no node is contained in more than one DD-path. Case 4 is needed for "short branches"; it also preserves the one-fragment, one DD-path principle. Case 5 is

the "normal case," in which a DD-path is a single entry, single-exit sequence of nodes (a chain). The "maximal" part of the case 5 definition is used to determine the final node of a normal (nontrivial) chain.

DD-Path for Triangle Problem

```
1  Program triangle2
2  Dim a,b,c As Integer
3  Dim IsATrinagle As Boolean
4  Output("Enter 3 integers which are sides of a triangle")
5  Input(a,b,c)
6  Output("Side A is", a)
7  Output("Side B is", b)
8  Output("Side C is", c)
9  If (a < b + c) AND (b < a + c) AND (c < a + b)
10    Then IsATriangle = True
11    Else IsATriangle = False
12 EndIf
13 If IsATriangle
14    Then  If (a = b) AND (b = c)
15             Then Output ("Equilateral")
16             Else  If (a≠b) AND (a≠c) AND (b≠c)
17                      Then Output ("Scalene")
18                      Else Output ("Isosceles")
19                EndIf
20             EndIf
21    Else  Output("Nota a Triangle")
22 EndIf
23 End triangle2
```

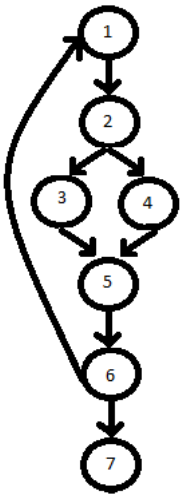
Nodes	DD-Path	Case of definition
4	First	1
5-8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3
21	G	4
22	O	3
23	Last	2



3a.

```
1: IF A = 100
2: THEN IF B > C
3: THEN A = B
4: ELSE A= C
5: ENDIF
6: ENDIF
7: Print A
```

Cyclomatic complexity=3



3b. **Metric Based Testing**

Fundamental limitations of specification-based testing is that it is impossible to know either the extent of redundancy or the possibility of gaps corresponding to the way a set of functional test cases exercises a program. Test coverage metrics are a device to measure the extent to which a set of test cases covers (or exercises) a program.

Program Graph–Based Coverage Metrics: Given a set of test cases for a program, they constitute *node coverage* if, when executed on the program, every node in the program graph is traversed. Denote this level of coverage as *G*node, where the *G* stands for program graph. Since nodes correspond to statement fragments, this guarantees that every statement fragment is executed by some test case. If we are careful about defining statement fragment nodes, this also guarantees that statement fragments that are outcomes of a decision-making statement are executed.

E.F. Miller's Coverage Metrics: Having an organized view of the extent to which a program is tested makes it possible to sensibly manage the testing process. Most quality organizations now expect the *C*1 metric (DD-path coverage) as the minimum acceptable level of test coverage. These coverage metrics form a lattice in which some are equivalent and some are implied by others. The importance of the lattice is that there are always fault

types that can be revealed at one level and can escape detection by inferior levels of testing. Miller (1991) observes that when DD-path coverage is attained by a set of test cases, roughly 85% of all faults are revealed. The test coverage metrics tell us what to test but not how to test it. In this section, we take a closer look at techniques that exercise source code. We must keep an important distinction in mind: Miller's test coverage metrics are based on program graphs in which nodes are full statements, whereas our formulation allows statement fragments (which can be entire statements) to be nodes.

Miller's Test Coverage Metrics

Metric	*Description of Coverage*
C_0	Every statement
C_1	Every DD-path
C_{1p}	Every predicate to each outcome
C_2	C_1 coverage + loop coverage
C_d	C_1 coverage + every dependent pair of DD-paths
C_{MCC}	Multiple condition coverage
C_{ik}	Every program path that contains up to k repetitions of a loop (usually $k = 2$)
C_{stat}	"Statistically significant" fraction of paths
C_∞	All possible execution paths

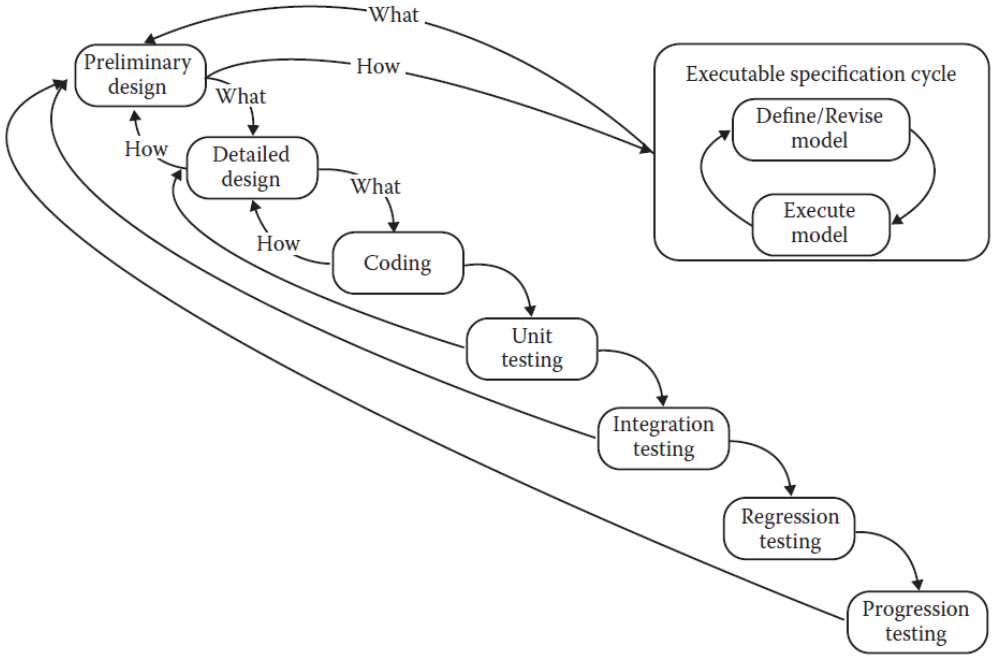
4a. **Waterfall Spin Off**

- Development in stages
  - Level use of staff across all types
  - Testing now entails both
    - Regression
    - Progression
- Main variations involve constructing a sequence of systems
  - Incremental
  - Evolutionary
  - Spiral
- Waterfall model is applied to each build
  - Smaller problem than original
  - System functionality does not change
- Incremental
  - Have high-level design at the beginning
  - Low-level design results in a series of builds
    - Incremental testing is useful
  - System testing is not affected
  - Level off staffing problems
- Evolutionary
  - First build is defined
  - Priorities and customer define next build
  - Difficult to have initial high-level design
    - Incremental testing is difficult
    - System testing is not affected
- Spiral
  - Combination of incremental and evolutionary
  - After each build assess benefits and risks
    - Use to decide go/no-go and direction

    - Difficult to have initial high-level design
        - Incremental testing is difficult
        - System testing is not affected
- Advantage of spiral models
    - Earlier synthesis and deliverables
    - More customer feedback
    - Risk/benefit analysis is rigorous

Specification Based Life Cycle Models

When systems are not fully understood (by either the customer or the developer), functional decomposition is perilous at best. Barry Boehm jokes when he describes the customer who says "I don't know what I want, but I'll recognize it when I see it." The rapid prototyping life cycle deals with this by providing the "look and feel" of a system. Thus, in a sense, customers can recognize what they "see." In turn, this drastically reduces the specification-to-customer feedback loop by producing very early synthesis. Rather than build a final system, a "quick and dirty" prototype is built and then used to elicit customer feedback. Depending on the feedback, more prototyping cycles may occur. Once the developer and the customer agree that a prototype represents the desired system, the developer goes ahead and builds to a correct specification. At this point, any of the waterfall spin-offs might also be used. The agile life cycles are the extreme of this pattern. Rapid prototyping has no new implications for integration testing; however, it has very interesting implications for system testing. Where are the requirements? Is the last prototype the specification? How are system test cases traced back to the prototype? One good answer to questions such as these is to use the prototyping cycles as information-gathering activities and then produce a requirements specification in a more traditional manner. Another possibility is to capture what the customer does with the prototypes, define these as scenarios that are important to the customer, and then use these as system test cases. These could be precursors to the user stories of the agile life cycles. The main contribution of rapid prototyping is that it brings the operational (or behavioral) viewpoint to the requirements specification phase. Usually, requirements specification techniques emphasize the structure of a system, not its behavior. This is unfortunate because most customers do not care about the structure, and they do care about the behavior. Executable specifications are an extension of the rapid prototyping concept. With this approach, the requirements are specified in an executable format (such as finite state machines, StateCharts, or Petri nets). The customer then executes the specification to observe the intended system behavior and provides feedback as in the rapid prototyping model. The executable models are, or can be, quite complex. This is an understatement for the full-blown version of StateCharts. Building an executable model requires expertise, and executing it requires an engine. Executable specification is best applied to event-driven systems, articularly when the events can arrive in different orders. David Harel, the creator of StateCharts, refers to such systems as "reactive" (Harel, 1988) because they react to external events. As with rapid prototyping, the purpose of an executable specification is to let the customer experience scenarios of intended behavior. Another similarity is that executable models might have to be revised on the basis of customer feedback. One side benefit is that a good engine for an executable model will support the capture of "interesting" system transactions, and it is often a nearly mechanical process to convert these into true system test cases. If this is done carefully, system testing can be traced directly back to the requirements. Once again, this life cycle has no implications for integration testing. One big difference is that the requirements specification document is explicit, as opposed to a prototype. More important, it is often a mechanical process to derive system test cases from an executable specification. Although more work is required to develop an executable specification, this is partially offset by the reduced effort to generate system test cases. Here is another important distinction: when system testing is based on an executable specification, we have an interesting form of structural testing at the system level. Finally, as we saw with rapid prototyping, the executable specification step can be combined with any of the iterative life cycle models.



4b. **Pros and cons of traditional waterfall model**

Some of the major advantages of the Waterfall Model are as follows −

- Simple and easy to understand and use
- Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.
- Clearly defined stages.
- Well understood milestones.
- Easy to arrange tasks.
- Process and results are well documented.

The major disadvantages of the Waterfall Model are as follows −

- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing. So, risk and uncertainty is high with this process model.
- It is difficult to measure progress within stages.
- Cannot accommodate changing requirements.
- Adjusting scope during the life cycle can end a project.
- Integration is done as a "big-bang. at the very end, which doesn't allow identifying any technological or business bottleneck or challenges early.

5a. A **decision table** is an excellent tool to use in both testing and requirements management. Essentially it is a structured exercise to formulate requirements when dealing with complex business rules. **Decision tables** are used to model complicated logic.

Decision Table for the NextDate function

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*		
c1: Month in	M1	M1	M1	M1	M1	M2	M2	M2	M2	M2		
c2: Day in	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5		
c3: Year in	—	—	—	—	—	—	—	—	—	—		
Actions												
a1: Impossible					X							
a2: Increment day	X	X	X			X	X	X	X			
a3: Reset day				X						X		
a4: Increment month				X						X		
a5: Reset month												
a6: Increment year												
	11	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*
c1: Month in	M3	M3	M3	M3	M3	M4	M4	M4	M4	M4	M4	M4
c2: Day in	D1	D2	D3	D4	D5	D1	D2	D2	D3	D3	D4	D5
c3: Year in	—	—	—	—	—	—	Y1	Y2	Y1	Y2	—	—
Actions												
a1: Impossible										X	X	X
a2: Increment day	X	X	X	X		X	X					
a3: Reset day					X			X	X			
a4: Increment month								X	X			
a5: Reset month					X							
a6: Increment year					X							

5b. **Path testing** is an approach to testing where you ensure that every path through a program has been executed at least once. You normally use a dynamic analyzer tool or test coverage analyzer to check that all of the code in a program has been executed.

Path Testing is a structural testing method based on the source code or algorithm and NOT based on the specifications. It can be applied at different levels of granularity.

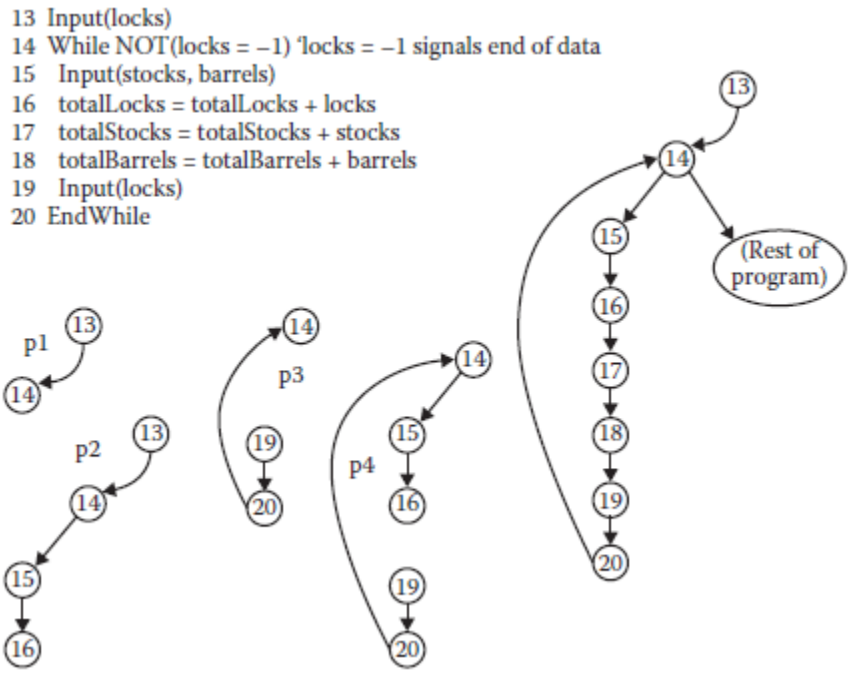
Path Testing Assumptions:

- The Specifications are Accurate
- The Data is defined and accessed properly
- There are no defects that exist in the system other than those that affect control flow

Path Testing Techniques:

- Control Flow Graph (CFG) - The Program is converted into Flow graphs by representing the code into nodes, regions and edges.
- Decision to Decision path (D-D) - The CFG can be broken into various Decision to Decision paths and then collapsed into individual nodes.
- Independent (basis) paths - Independent path is a path through a DD-path graph which cannot be reproduced from other paths by other methods.

6a. A ***definition/use path*(DU-path)** with respect to a variable *v* (denoted du-path) is a path in PATHS(*P*) such that, for some $v \in V$, there are define and usage nodes DEF(*v*, *m*) and USE(*v*, *n*) such that *m* and *n* are the initial and final nodes of the path.



6b. **Slice-Based Testing Definitions**

- Given a program P, and a program graph G(P) in which statements and statement fragments are numbered, and a set V of variables in P, the ***slice on the variable set V at statement fragment n***, written S(V,n), is the set node numbers of all statement fragments in P prior to n that contribute to the values of variables in V at statement fragment n
- The idea of slices is to separate a program into components that have some useful meaning

- We will include CONST declarations in slices
- Five forms of usage nodes
  - P-use (used in a predicate (decision))
  - C-use (used in computation)
  - O-use (used for output, e.g. writeln())
  - L-use (used for location, e.g. pointers)
  - I-use (iteration, e.g. internal counters)
- Two forms of definition nodes
  - I-def (defined by input, e.g. readln())
  - A-def (defined by assignment)
- For now, we presume that the slice S(V,n) is a slice on one variable, that is, the set V consists of a single variable, v

- If statement fragment n (in S(V,n)) is a defining node for v, then n is included in the slice
- If statement fragment n (in S(V,n)) is a usage node for v, then n is not included in the slice
- P-uses and C-uses of other variables are included to the extent that their execution affects the value of the variable v
- O-use, L-use, and I-use nodes <u>are excluded</u> from slices
- Consider making slices compliable

Slice-Based Testing Examples

- Find the following program slices
- S(commission,48)
- S(commission,40)
- S(commission,39)
- S(commission,38)
- S(sales,35)
- S(num_locks,34)
- S(num_stocks,34)
- S(num_barrels,34)
- S(commission,48)
    - {1-5,8-11,13,14,19-30,36,47,48,53}
- S(commission,40), S(commission,39), S(commission,38)
    - {Ø}
- S(sales,35)
    - {Ø}
- S(num_locks,34)
    - {1,8,9,10,13,14,19, 22,23,24,26,29,30, 53}
- S(num_stocks,34)
    - {1,8,9,10,13,14,20, 22-25,27,29,30,53}
- S(num_barrels,34)
    - {1,8,9,10,13,14,21-25,28,29,30,53}

7a. **Unit Testing** - As the name suggests, this method tests at the object level. Individual software components are tested for any errors. Knowledge of the program is needed for this test and the test codes are created to check if the software behaves as it is intended to.

Integration Testing - Individual modules that are already subjected to unit testing are integrated with one another. Generally the two approaches are followed:

1) Top-Down
2) Bottom-Up

System Testing - Is carried out without any knowledge of the internal working of the system. The tester will try to use the system by just following requirements, by providing different inputs and testing the generated outputs. This test is also known as closed-box testing or black-box.

7b. **Top down Integration Testing:** In this approach testing is conducted from main module to sub module. if the sub module is not developed a temporary program called STUB is used for simulate the submodule.
Advantages:
- Advantageous if major flaws occur toward the top of the program.
- Once the I/O functions are added, representation of test cases is easier.
- Early skeletal Program allows demonstrations and boosts morale.
Disadvantages:
- Stub modules must be produced
- Stub Modules are often more complicated than they first appear to be.
- Before the I/O functions are added, representation of test cases in stubs can be difficult.

- Test conditions ma be impossible, or very difficult, to create.
- Observation of test output is more difficult.
- Allows one to think that design and testing can be overlapped.
- Induces one to defer completion of the testing of certain modules.

Bottom up Integration Testing: In this approach testing is conducted from sub module to main module, if the main module is not developed a temporary program called DRIVERS is used to simulate the main module.

Advantages:

- Advantageous if major flaws occur toward the bottom of the program.
- Test conditions are easier to create.
- Observation of test results is easier.

Disadvantages:

- Driver Modules must be produced.
- The program as an entity does not exist until the last module is added.