UNIT –I
INTRODUCTION

(1) Purpose of Testing:

(i) What we do:

- Testing consumes at least half of the labor expended to produce a working program.
- Few programmers like testing and even fewer like test design—especially if test design and testing take longer than program design and coding.
- This attitude is understandable.
- Software is ephemeral: you can't point to something physical.
- I think, deep down, most of us don't believe in software—at least not the way we believe in hardware.
- If software is insubstantial, then how much more insubstantial does software testing seem? There isn't even some debugged code to point to when we're through with test design.
- The effort put into testing seems wasted if the tests don't reveal bugs.
- There's another, deeper, problem with testing that's related to the reason we do it (MILL78B, MYER79). It's done to catch bugs.
- There's a myth that if we were really good at programming, there would be no bugs to catch. If only we could really concentrate, if everyone used structured programming, top-down design, decision tables, if programs were written in SQUISH, if we had the right silver bullets, then there would be no bugs.
- So goes the myth. There are bugs, the myth says, because we are bad at what we do; and if we are bad at it, we should feel guilty about it.
- Therefore, testing and test design amount to an admission of failure, which instills a goodly dose of guilt. And the tedium of testing is just punishment for our errors.
- Punishment for what? For being human? Guilt for what? For not achieving inhuman perfection? For not distinguishing between what another programmer thinks and what he says? For not being telepathic? For not solving human communication problems that have been kicked around by philosophers and theologians for 40 centuries?
- The statistics show that programming, done well, will still have one to three bugs per hundred statements (AKIY71, ALBE76, BOEH75B, ENDR75, RADA81, SHOO75, THAY76, WEIS85B).*
- Certainly, if you have a 10% error rate, then you either need more programming education or you deserve reprimand *and* guilt.**
- There are some persons who claim that they can write bug-free programs. There's a saying among sailors on the Chesapeake Bay, whose sandy, shifting bottom outdates charts before they're printed, "If you haven't run aground on the Chesapeake, you haven't sailed the Chesapeake much."
- So it is with programming and bugs: I have them, you have them, we all have them—and the point is to do what we can to prevent them and to discover them as early as possible, but not to feel guilty about them.
- Programmers! Cast out your guilt! Spend half your time in joyous testing and debugging! Thrill to the excitement of the chase! Stalk bugs with care, methodology, and reason. Build traps for them.
- Be more artful than those devious bugs and taste the joys of guiltless programming! Testers! Break that software (as you must) and drive it to the ultimate—but don't enjoy the programmer's pain.

www.Jntufastupdates.com

(ii) Productivity and quality in Software:

- Consider the manufacture of a mass-produced widget. Whatever the design cost, it is a small part of the total cost when amortized over a large production run.
- Once in production, every manufacturing stage is subjected to quality control and testing from component source inspection to final testing before shipping.
- If flaws are discovered at any stage, the widget or part of it will either be discarded or cycled back for rework and correction.
- The assembly line's productivity is measured by the sum of the costs of the materials, the rework, and the discarded components, and the cost of quality assurance and testing.
- There is a trade-off between quality-assurance costs and manufacturing costs. If insufficient effort is spent in quality assurance, the reject rate will be high and so will the net cost.
- Conversely, if inspection is so good that all faults are caught as they occur, inspection costs will dominate, and again net cost will suffer.
- The manufacturing process designers attempt to establish a level of testing and quality assurance that minimizes net cost for a given quality objective.
- Testing and quality-assurance costs for manufactured items can be as low as 2% in consumer products or as high as 80% in products such as spaceships, nuclear reactors, and aircraft, where failures threaten life.
- The relation between productivity and quality for software is very different from that for manufactured goods.
- The "manufacturing" cost of a software copy is trivial: the cost of the tape or disc and a few minutes of computer time.
- Furthermore, software "manufacturing" quality assurance is automated through the use of check sums and other error-detecting methods.
- Software costs are dominated by development.
- Software maintenance is unlike hardware maintenance. It is not really "maintenance" but an extended development in which enhancements are designed and installed and deficiencies corrected.
- The biggest part of software cost is the cost of bugs: the cost of detecting them, the cost of correcting them, the cost of designing tests that discover them, and the cost of running those tests.
- The main difference then between widget productivity and software productivity is that for hardware quality is only one of several productivity determinants, whereas for software, quality and productivity are almost indistinguishable.

(iii) Goals for testing:

- Testing and test design, as parts of quality assurance, should also focus on bug prevention. To the extent that testing and test design do not prevent bugs, they should be able to discover symptoms caused by bugs.
- Finally, tests should provide clear diagnoses so that bugs can be easily corrected. Bug prevention is testing's first goal.
- A prevented bug is better than a detected and corrected bug because if the bug is prevented, there's no code to correct.
- Moreover, no retesting is needed to confirm that the correction was valid, no one is embarrassed, no memory is consumed, and prevented bugs can't wreck a schedule.
- More than the act of testing, the act of *designing* tests is one of the best bug preventers known.
- The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded—indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding, and the rest.

- For this reason, Dave Gelperin and Bill Hetzel (GELP87) advocate "Test, then code." The ideal test activity would be so successful at bug prevention that actual testing would be unnecessary because all bugs would have been found and fixed during test design.*
- Unfortunately, we can't achieve this ideal. Despite our effort, there will be bugs because we are human.
- To the extent that testing fails to reach its primary goal, *bug prevention*, it must reach its secondary goal, *bug discovery*. Bugs are not always obvious.
- A bug is manifested in deviations from expected behavior. A test design must document expectations, the test procedure in detail, and the results of the actual test—all of which are subject to error.
- But knowing that a program is incorrect does not imply knowing the bug. Different bugs can have the same manifestations, and one bug can have many symptoms.
- The symptoms and the causes can be disentangled only by using many small detailed tests.

(iv) Phases in a Tester's Mental Life:

(a) Why Testing:

- What's the purpose of testing? There's an attitudinal progression characterized by the following five phases:
  PHASE 0—There's no difference between testing and debugging. Other than in support of debugging, testing has no purpose.
  PHASE 1—The purpose of testing is to show that the software works.
  PHASE 2—The purpose of testing is to show that the software doesn't work.
  PHASE 3—The purpose of testing is not to prove anything, but to reduce the perceived risk of not working to an acceptable value.
  PHASE 4—Testing is not an act. It is a mental discipline that results in low-risk software without much testing effort.

(b) Phase 0 Thinking:

- I called the inability to distinguish between testing and debugging "phase 0" because it denies that testing matters, which is why I denied it the grace of a number. See Section 2.1 in this chapter for the difference between testing and debugging. If phase 0 thinking dominates an organization, then there can be no effective testing, no quality assurance, and no quality. Phase 0 thinking was the norm in the early days of software development and dominated the scene until the early 1970s, when testing emerged as a discipline.
- Phase 0 thinking was appropriate to an environment characterized by expensive and scarce computing resources, low-cost (relative to hardware) software, lone programmers, small projects, and throwaway software. Today, this kind of thinking is the greatest cultural barrier to good testing and quality software. But phase 0 thinking is a problem for testers and developers today because many software managers learned and practiced programming when this mode was the norm—and it's hard to change how you think.

(c) Phase 1 Thinking-The Software Works

- Phase I thinking represented progress because it recognized the distinction between testing and debugging. This thinking dominated the leading edge of testing until the late 1970s when its fallacy was discovered. This recognition is attributed to Myers (MYER79) who observed that it is self-corrupting. It only takes one failed test to show that software doesn't work, but even an infinite number of tests won't prove that it does. The objective of phase 1 thinking is unachievable. The process is corrupted because the probability of showing that the software works *decreases* as testing increases; that is, the more you test, the likelier you are to find a bug. Therefore, if your objective is to demonstrate a high probability of working, that objective is best achieved by not testing at all! Although this conclusion may seem silly to the

www.Jntufastupdates.com

conscious, rational mind, it is the kind of syllogism that our unconscious mind loves to implement.

(d) Phase 2 Thinking-The Software Doesn't Work:

- When, as testers, we shift our goal to phase 2 thinking we are no longer working in cahoots with the designers, but against them. The difference between phase 1 and 2 thinking is illustrated by analogy to the difference between bookkeepers and auditors. The bookkeeper's goal is to show that the books balance, but the auditor's goal is to show that despite the appearance of balance, the bookkeeper has embezzled. Phase 2 thinking leads to strong, revealing tests.
- While one failed test satisfies the phase 2 goal, phase 2 thinking also has limits. The test reveals a bug, the programmer corrects it, the test designer designs and executes another test intended to demonstrate another bug. Phase 2 thinking leads to a never-ending sequence of ever more diabolical tests. Taken to extremes, it too never ends, and the result is reliable software that never gets shipped. The trouble with phase 2 thinking is that we don't know when to stop.

(e) Phase 3 Thinking-Test for Risk Reduction:

- Phase 3 thinking is nothing more than accepting the principles of statistical quality control. I say "accepting" rather than "implementing" because it's not obvious how statistical quality control should be applied to software. To the extent that testing catches bugs and to the extent that those bugs are fixed, testing does improve the product. If a test is passed, then the product's quality does not change, but our perception of that quality does. Testing, pass or fail, reduces our perception of risk about a software product. The more we test, the more we test with harsh tests, the more confidence we have in the product. We'll risk release when that confidence is high enough.*

(f) Phase 4 Thinking-A State of Mind:

- The phase 4 thinker's knowledge of what testing can and can't do, combined with knowing what makes software testable, results in software that doesn't need much testing to achieve the lower-phase goals. Testability is the goal for two reasons. The first and obvious reason is that we want to reduce the labor of testing. The second and more important reason is that testable code has fewer bugs than code that's hard to test. The impact on productivity of these two factors working together is multiplicative. What makes code testable? One of the main reasons to learn test techniques is to answer that question.

(g) Cumulative Goals:

- The above goals are cumulative. Debugging depends on testing as a tool for probing hypothesized causes of symptoms. There are many ways to break software that have nothing to do with the software's functional requirements: phase 2 tests alone might never show that the software does what it's supposed to do. It's impractical to break software until the easy demonstrations of workability are behind you. Use of statistical methods as a guide to test design, as a means to achieve good testing at acceptable risks, is a way of fine-tuning the process. It should be applied only to large, robust products with few bugs. Finally, a state of mind isn't enough: even the most testable software must be debugged, must work, and must be hard to break.

(v) Test Design:

- Although programmers, testers, and programming managers know that code must be designed and tested, many appear to be unaware that tests themselves must be designed and tested—designed by a process no less rigorous and no less controlled than that used for code.

www.Jntufastupdates.com

- Too often, test cases are attempted without prior analysis of the program's requirements or structure. Such test design, if you can call it that, is just a haphazard series of ad-lib cases that are not documented either before or after the tests are executed.
- Because they were not formally designed, they cannot be precisely repeated, and no one is sure whether there was a bug or not. After the bug has been ostensibly corrected, no one is sure that the retest was identical to the test that found the bug.
- Ad-lib tests are useful during debugging, where their primary purpose is to help locate the bug, but adlib tests done in support of debugging, no matter how exhausting, are not substitutes for *designed* tests.
- The test-design phase of programming should be explicitly identified. Instead of "design, code, desk check, test, and debug," the programming process should be described as: "design, test design, code, test code, program inspection, test inspection, test debugging, test execution, program debugging, testing."
- Giving test design an explicit place in the scheme of things provides more visibility to that amorphous half of the labor that often goes under the name "test and debug." It makes it less likely that test design will be given short shrift when the budget's small and the schedule's tight and there's a vague hope that maybe this time, just this once, the system will come together without bugs.

(vi) Testing Isn't Everything:

- This is a book on testing techniques, which are only *part* of our weaponry against bugs. Research and practice (BASI87, FAGA76, MYER78, WEIN65, WHIT87) show that other approaches to the creation of good software are possible and essential. Testing, I believe, is still our most potent weapon, but there's evidence (FAGA76) that other methods may be as effective: but you can't implement inspections, say, *instead* of testing because testing and inspections catch or prevent different kinds of bugs. Today, if we want to prevent all the bugs that we can and catch those that we don't prevent, we must review, inspect, read, do walkthroughs, *and then test*. We don't know today the mix of approaches to use under what circumstances. Experience shows that the "best mix" *very much* depends on things such as development environment, application, size of project, language, history, and culture. The other major methods in decreasing order of effectiveness are as follows:
- **Inspection Methods**—In this category I include walkthroughs, desk checking, formal inspections (FAGA76), and code reading. These methods appear to be as effective as testing, but the bugs caught do not completely overlap.
- **Design Style**—By this term I mean the stylistic criteria used by programmers to define what they mean by a "good" program. Sticking to outmoded style, such as "tight" code or "optimizing" for performance destroys quality. Conversely, adopting stylistic objectives such as testability, openness, and clarity can do much to prevent bugs.
- **Static Analysis Methods**—These methods include anything that can be done by formal analysis of source code during or in conjunction with compilation. Syntax checking in early compilers was rudimentary and was part of the programmer's "testing," Compilers have taken that job over (thank the Lord). **Strong typing** and **type checking** eliminate an entire category of bugs. There's a lot more that can be done to detect errors by static analysis. It's an area of intensive research and development. For example, much of **data-flow anomaly** detection (see Chapters 5 and 8), which today is part of testing, will eventually be incorporated into the compiler's static analysis.
- **Languages**—The source language can help reduce certain kinds of bugs. Languages continue to evolve, and preventing bugs is a driving force in that evolution. Curiously, though, programmers find new kinds of bugs in new languages, so the bug rate seems to be independent of the language used.

- **Design Methodologies and Development Environment**—The design methodology (that is, the development process used and the environment in which that methodology is embedded), can prevent many kinds of bugs. For example, configuration control and automatic distribution of change information prevents bugs which result from a programmer's unawareness that there were changes.

(vii) The Pesticide Paradox and the Complexity Barrier:

- You're a poor farmer growing cotton in Alabama and the boll weevils are destroying your crop. You mortgage the farm to buy DDT, which you spray on your field, killing 98% of the pest, saving the crop. The next year, you spray the DDT early in the season, but the boll weevils still eat your crop because the 2% you didn't kill last year were resistant to DDT. You now have to mortgage the farm to buy DDT *and* Malathion; then next year's boll weevils will resist both pesticides and you'll have to mortgage the farm yet again. That's the pesticide paradox* for boll weevils and also for software testing.
- *First Law: The Pesticide Paradox*—Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual. That's not too bad, you say, because at least the software gets better and better. Not quite!
- *Second Law: The Complexity Barrier*—Software complexity (and therefore that of bugs) grows to the limits of our ability to manage that complexity.
- By eliminating the (previous) easy bugs you allowed another escalation of features and complexity, but this time you have subtler bugs to face, just to retain the reliability you had before. Society seems to be unwilling to limit complexity because we all want that extra bell, whistle, and feature interaction. Thus, our users always push us to the complexity barrier and how close we can approach that barrier is largely determined by the strength of the techniques we can wield against ever more complex and subtle bugs.

(2) Some Dichotomies:

(i) Testing Versus Debugging:

- Testing and debugging are often lumped under the same heading, and it's no wonder that their roles are often confused: for some, the two words are synonymous; for others, the phrase "test and debug" is treated as a single word. The **purpose of testing** is to show that a program has bugs. **The purpose of debugging** is find the error or misconception that led to the program's failure and to design and implement the program changes that correct the error. Debugging usually follows testing, but they differ as to goals, methods, and most important, psychology:

  1. Testing starts with known conditions, uses predefined procedures, and has predictable outcomes; only whether or not the program passes the test is unpredictable. Debugging starts from possibly unknown initial conditions, and the end cannot be predicted, except statistically.

  2. Testing can and should be planned, designed, and scheduled. The procedures for, and duration of, debugging cannot be so constrained.

  3. Testing is a demonstration of error or apparent correctness. Debugging is a deductive process.

  4. Testing proves a programmer's failure. Debugging is the programmer's vindication.

  5. Testing, as executed, should strive to be predictable, dull, constrained, rigid, and inhuman. Debugging demands intuitive leaps, conjectures, experimentation, and freedom.

  6. Much of testing can be done without design knowledge. Debugging is impossible without detailed design knowledge.

  7. Testing can often be done by an outsider. Debugging must be done by an insider.

www.Jntufastupdates.com

8. Although there is a robust theory of testing that establishes theoretical limits to what testing can and can't do, debugging has only recently been attacked by theorists—and so far there are only rudimentary results.

9. Much of test execution and design can be automated. Automated debugging is still a dream.

(ii) Function Versus Structure:

- Tests can be designed from a functional or a structural point of view. In **functional testing** the program or system is treated as a black box. It is subjected to inputs, and its outputs are verified for conformance to specified behavior. The software's user should be concerned only with functionality and features, and the program's implementation details should not matter. Functional testing takes the user's point of view.
- **Structural testing** does look at the implementation details. Such things as programming style, control method, source language, database design, and coding details dominate structural testing; but the boundary between function and structure is fuzzy. Good systems are built in layers—from the outside to the inside. The user sees only the outermost layer, the layer of pure function. Each layer inward is less related to the system's functions and more constrained by its structure: so what is structure to one layer is function to the next. For example, the user of an online system doesn't know that the system has a memory-allocation routine. For the user, such things are structural details. The memory-management routine's designer works from a specification for that routine. The specification is a definition of "function" at that layer. The memory-management routine uses a link-block subroutine. The memory-management routine's designer writes a "functional" specification for a link-block subroutine, thereby defining a further layer of structural detail and function. At deeper levels, the programmer views the operating system as a structural detail, but the operating system's designer treats the computer's hardware logic as the structural detail.
- Most of this book is devoted to models of programs and the tests that can be designed by using those models. A given model, and the associated tests may be first introduced in a structural context but later used again in a functional context, or vice versa. The initial choice of how to present a model was based on the context that seemed most natural for that model and in which it was likeliest that the model would be used for test design. Just as you can't clearly distinguish function from structure, you can't fix the utility of a model to structural tests or functional tests. If it helps you design effective tests, then use the model in whatever context it seems to work.
- There's no controversy between the use of structural versus functional tests: both are useful, both have limitations, both target different kinds of bugs. Functional tests can, in principle, detect all bugs but would take infinite time to do so. Structural tests are inherently finite but cannot detect all errors, even if completely executed. The art of testing, in part, is in how you choose between structural and functional tests.

(iii) The Designer Versus the Tester:

- If testing were wholly based on functional specifications and independent of implementation details, then the designer and the tester could be completely separated. Conversely, to design a test plan based only on a system's structural details would require the software designer's knowledge, and hence only she could design the tests. The more you know about the design, the likelier you are to eliminate useless tests, which, despite functional differences, are actually handled by the same routines over the same paths; but the more you know about the design, the likelier you are to have the same misconceptions as the designer. Ignorance of structure is the independent tester's best friend and worst enemy. The naive tester has no preconceptions about what is or is not possible and will, therefore, design tests that the program's designer would never think of—and many tests that never should be

www.Jntufastupdates.com

thought of. Knowledge, which is the designer's strength, brings efficiency to testing but also blindness to missing functions and strange cases. Tests designed and executed by the software's designers are by nature biased toward structural considerations and therefore suffer the limitations of structural testing. Tests designed and executed by an independent tester are bias-free and can't be finished. Part of the artistry of testing is to balance knowledge and its biases against ignorance and its inefficiencies.

- In this book I discuss the "tester," "test-team member," or "test designer" in contrast to the "programmer" and "program designer," as if they were distinct persons. As one goes from **unit testing** to **unit integration**, to **component testing** and integration, to **system testing**, and finally to formal **system feature testing**, it is increasingly more effective if the "tester" and "programmer" are different persons. The techniques presented in this book can be used for all testing—from unit to system. When the technique is used in system testing, the designer and tester are probably different persons; but when the technique is used in unit testing, the tester and programmer merge into one person, who sometimes acts as a programmer and sometimes as a tester.
- You must be a constructive schizophrenic. Be clear about the difference between your role as a programmer and as a tester. The tester in you must be suspicious, uncompromising, hostile, and compulsively obsessed with destroying, utterly destroying, the programmer's software. The tester in you is your Mister Hyde—your Incredible Hulk. He must exercise what Gruenberger calls "low cunning." (HETZ73) The programmer in you is trying to do a job in the simplest and cleanest way possible, on time, and within budget. Sometimes you achieve this by having great insights into the programming problem that reduce complexity and labor and are almost correct. And with that tester/Hulk lurking in the background of your mind, it pays to have a healthy paranoia about bugs. Remember, then, that when I refer to the "test designer" and "programmer" as separate persons, the extent to which they are separated depends on the testing level and the context in which the technique is applied. This saves me the effort of writing about the same technique twice and you the tedium of reading it twice.

(iv) Modularity Versus Efficiency:

- Both tests and systems can be modular. A **module** is a discrete, well-defined, small component of a system. The smaller the component, the easier it is to understand; but every component has interfaces with other components, and *all* interfaces are sources of confusion. The smaller the component, the likelier are interface bugs. Large components reduce external interfaces but have complicated internal logic that may be difficult or impossible to understand. Part of the artistry of software design is setting component size and boundaries at points that balance internal complexity against interface complexity to achieve an overall complexity minimization.
- Testing can and should likewise be organized into modular components. Small, independent test cases have the virtue of easy repeatability. If an error is found by testing, only the small test, not a large component that consists of a sequence of hundreds of interdependent tests, need be rerun to confirm that a test design bug has been fixed. Similarly, if the test has a bug, only that test need be changed and not a whole test plan. But microscopic test cases require individual setups and each such setup (e.g., data, inputs) can have bugs. As with system design, artistry comes into test design in setting the scope of each test and groups of tests so that test design, test debugging, and test execution labor are minimized without compromising effectiveness.

(v) Small Versus Large:

- I often write small analytical programs of a few hundred lines that, once used, are discarded. Do I use formal test techniques, quality assurance, and all the rest I so passionately advocate? Of course not, and I'm not a hypocrite. I do what everyone does in similar

circumstances: I design, I code, I test a few cases, debug, redesign, recode, and so on, much as I did 30 years ago. I can get away with such (slovenly) practices because I'm programming for a very small, intelligent, forgiving, user population—me. It's the ultimate of small programs and it is most efficiently done by intuitive means and complete lack of formality.

- Let's up the scale to a larger package. I'm still the only programmer and user, but now, the package has thirty components averaging 750 statements each, developed over a period of 5 years. Now I *must* create and maintain a data dictionary and do thorough unit testing. But I'll take my own word for it and not bother to retain all those test cases or to exercise formal configuration control.
- You can extrapolate from there or draw on your experiences. **Programming in the large** (DERE76) means constructing programs that consist of many components written by many different persons. **Programming in the small** is what we do for ourselves in the privacy of our own offices or as homework exercises in an undergraduate programming course. Size brings with it nonlinear scale effects, which are imperfectly understood today. Qualitative changes occur with size and so must testing methods and quality criteria. A primary example is the notion of **coverage**—a measure of test completeness. Without worrying about exactly what these terms mean, 100% coverage is essential for unit testing, but we back off this requirement as we deal with ever larger software aggregates, accept 75%-85% for most systems, and possibly as low as 50% for huge systems of 10 million lines of code or so.

(vi) The Builder Versus the Buyer:

- Most software is written and used by the same organization. Unfortunately, this situation is dishonest because it clouds accountability. Many organizations today recognize the virtue of independent software development and operation because it leads to better software, better security, and better testing. Independent software development does not mean that all software should be bought from software houses or consultants but that the software developing entity and the entity that pays for the software be separated enough to make accountability clear. I've heard of cases where the software development group and the operational group within the same company negotiate and sign formal contracts with one another—with lawyers present. If there is no separation between builder and buyer, there can be no accountability. If there is no accountability, the motivation for software quality disappears and with it any serious attempt to do proper testing.
- Just as programmers and testers can merge and become one, so can builder and buyer. There are several other persons in the software development cast of characters who, like the above, can also be separated or merged:

  1. The **builder**, who designs for and is accountable to
  2. The **buyer**, who pays for the system in the hope of profits from providing services to
  3. The **user**, the ultimate beneficiary or victim of the system. The user's interests are guarded by
  4. The **tester**, who is dedicated to the builder's destruction and
  5. The **operator**, who has to live with the builder's mistakes, the buyer's murky specifications, the tester's oversights, and the user's complaints.

(3) A Model For Testing:

(i) The Project:

- Testing is applied to anything from subroutines to systems that consist of millions of statements. The archetypical system is one that allows the exploration of all aspects of testing without the complications that have nothing to do with testing but affect any very large project.

- It's medium-scale programming. Testing the interfaces between different parts of your own mind is very different from testing the interface between you and other programmers separated from you by geography, language, time, and disposition.
- Testing a one-shot routine that will be run only a few times is very different from testing one that must run for decades and may be modified by some unknown future programmer.
- Although all the problems of the solitary routine occur for the routine that is embedded in a system, the converse is not true: many kinds of bugs just can't exist in solitary routines.
- There is an implied context for the test methods discussed in this book—a real-world context characterized by the following model project:
- **Application**—The specifics of the application are unimportant. It is a real-time system that must provide timely responses to user requests for services. It is an online system connected to remote terminals.
- **Staff**—The programming staff consists of twenty to thirty programmers—big enough to warrant formality, but not too big to manage—big enough to use specialists for some parts of the system's design.
- **Schedule**—The project will take 24 months from the start of design to formal acceptance by the customer. Acceptance will be followed by a 6-month cutover period. Computer resources for development and testing will be almost adequate.
- **Specification**—The specification is good. It is functionally detailed without constraining the design, but there are undocumented "understandings" concerning the requirements.
- **Acceptance Test**—The system will be accepted only after a formal acceptance test. The application is not new, so part of the formal test already exists. At first the customer will intend to design the acceptance test, but later it will become the software design team's responsibility.
- **Personnel**—The staff is professional and experienced in programming and in the application. Half the staff has programmed that computer before and most know the source language. One-third, mostly junior programmers, have no experience with the application. The typical programmer has been employed by the programming department for 3 years. The climate is open and frank. Management's attitude is positive and knowledgeable about the realities of such projects.
- **Standards**—Programming and test standards exist and are usually followed. They understand the role of interfaces and the need for interface standards. Documentation is good. There is an internal, semiformal, quality-assurance function. The database is centrally developed and administered.
- **Objectives**—The system is the first of many similar systems that will be implemented in the future. No two will be identical, but they will have 75% of the code in common. Once installed, the system is expected to operate profitably for more than 10 years.
- **Source**—One-third of the code is new, one-third extracted from a previous, reliable, but poorly documented system, and one-third is being rehosted (from another language, computer, operating system—take your pick).
- **History**—One programmer will quit before his components are tested. Another programmer will be fired before testing begins: excellent work, but poorly documented. One component will have to be redone after unit testing: a superb piece of work that defies integration. The customer will insist on five big changes and twenty small ones. There will be at least one nasty problem that nobody—not the customer, not the programmer, not the managers, nor the hardware vendor—suspected. A facility and/or hardware delivery problem will delay testing for several weeks and force second- and third-shift work. Several important milestones will slip but the delivery date will be met.

- Our model project is a typical well-run, successful project with a share of glory and catastrophe—neither a utopian project nor a slice of hell.

(ii)Overview:

- The process starts with a program embedded in an environment, such as a computer, an operating system, or a calling program. We understand human nature and its suceptibility to error. This understanding leads us to create three models: a model of the environment, a model of the program, and a model of the expected bugs. From these models we create a set of tests, which are then executed. The result of each test is either expected or unexpected. If unexpected, it may lead us to revise the test, our model or concept of how the program behaves, our concept of what bugs are possible, or the program itself. Only rarely would we attempt to modify the environment.

(iii)The Environment:

- A **program's environment** is the hardware and software required to make it run. For online systems the environment may include communications lines, other systems, terminals, and operators. The environment also includes all programs that interact with—and are used to create—the program under test, such as operating system, loader, linkage editor, compiler, utility routines.
- Programmers should learn early in their careers that it's not smart to blame the environment (that is, hardware and firmware) for bugs. Hardware bugs are rare. So are bugs in manufacturer-supplied software. This isn't because logic designers and operating system programmers are better than application programmers, but because such hardware and software is stable, tends to be in operation for a long time, and most bugs will have been found and fixed by the time programmers use that hardware or software.* Because hardware and firmware are stable, we don't have to consider all of the environment's complexity. Instead, we work with a simplification of it, in which only the features most important to the program at hand are considered. Our model of the environment includes our *beliefs* regarding such things as the workings of the computer's instruction set, operating system macros and commands, and what a higher-order language statement will do. If testing reveals an unexpected result, we may have to change our beliefs (our model of the environment) to find out what went wrong. But sometimes the environment could be wrong: the bug could be in the hardware or firmware after all.

(iv)The Program:

- Most programs are too complicated to understand in detail. We must simplify our concept of the program in order to test it. So although a real program is exercised on the test bed, in our brains we deal with a simplified version of it—a version in which most details are ignored. If the program calls a subroutine, we tend not to think about the subroutine's details unless its operation is suspect. Similarly, we may ignore processing details to focus on the program's control structure or ignore control structure to focus on processing. As with the environment, if the simple model of the program does not explain the unexpected behavior, we may have to modify that model to include more facts and details. And if that fails, we may have to modify the program.

(v)Bugs:

- Bugs are more insidious than ever we expect them to be. Yet it is convenient to categorize them: initialization, call sequence, wrong variable, and so on. Our notion of what is or isn't a bug varies. A bad specification may lead us to mistake good behavior for bugs, and vice versa. An unexpected test result may lead us to change our notion of what a bug is—that is to say, our model of bugs.
- While we're on the subject of bugs, I'd like to dispel some optimistic notions that many programmers and testers have about bugs. Most programmers and testers have beliefs

www.Jntufastupdates.com

about bugs that express a naivete that ranks with belief in the tooth fairy. If you hold any of the following beliefs, then disabuse yourself of them because as long as you believe in such things you will be unable to test effectively and unable to justify the dirty tests most programs need.

- **Benign Bug Hypothesis**—The belief that bugs are nice, tame, and logical. Only weak bugs have a logic to them and are amenable to exposure by strictly logical means. Subtle bugs have no definable pattern—they are wild cards.
- **Bug Locality Hypothesis**—The belief that a bug discovered within a component affects only that component's behavior; that because of structure, language syntax, and data organization, the symptoms of a bug are localized to the component's designed domain. Only weak bugs are so localized. Subtle bugs have consequences that are arbitrarily far removed from the cause in time and/or space from the component in which they exist.
- **Control Bug Dominance**—The belief that errors in the control structure of programs dominate the bugs. While many easy bugs, especially in components, can be traced to **control-flow** errors, **data-flow** and data-structure errors are as common. Subtle bugs that violate data-structure boundaries and data/code separation can't be found by looking only at control structures.
- **Code/Data Separation**—The belief, especially in HOL programming, that bugs respect the separation of code and data.* Furthermore, in real systems the distinction between code and data can be hard to make, and it is exactly that blurred distinction that permit such bugs to exist.
- **Lingua Salvator Est**—The hopeful belief that language syntax and semantics (e.g., structured coding, strong typing, complexity hiding) eliminates most bugs. True, good language features do help prevent the simpler component bugs but there's no statistical evidence to support the notion that such features help with subtle bugs in big systems.
- **Corrections Abide**—The mistaken belief that a corrected bug remains corrected. Here's a generic counterexample. A bug is believed to have symptoms caused by the interaction of components A and B but the real problem is a bug in C, which left a residue in a data structure used by both A and B. The bug is "corrected" by changing A and B. Later, C is modified or removed and the symptoms of A and B recur. Subtle bugs are like that.
- **Silver Bullets**—The mistaken belief that X (language, design method, representation, environment—name your own) grants immunity from bugs. Easy-to-moderate bugs may be reduced, but remember the pesticide paradox.
- **Sadism Suffices**—The common belief, especially by independent testers, that a sadistic streak, low cunning, and intuition are sufficient to extirpate most bugs. You only catch easy bugs that way. Tough bugs need methodology and techniques, so read on.
- **Angelic Testers**—The ludicrous belief that testers are better at test design than programmers are at code design.*

(vi)Tests:

- Tests are formal procedures. Inputs must be prepared, outcomes predicted, tests documented, commands executed, and results observed; all these steps are subject to error. There is nothing magical about testing and test design that immunizes testers against bugs. An unexpected test result is as often cause by a test bug as it is by a real bug.* Bugs can creep into the documentation, the inputs, and the commands and becloud our observation of results. An unexpected test result, therefore, may lead us to revise the tests. Because the tests are themselves in an environment, we also have a mental model of the tests, and instead of revising the tests, we may have to revise that mental model.

(vii)Testing and Levels:

www.Jntufastupdates.com

- We do three distinct kinds of testing on a typical software system: **unit/ component testing, integration testing**, and **system testing**. The objectives of each class is different and therefore, we can expect the mix of test methods used to differ. They are:
- **Unit, Unit Testing**—A **unit** is the smallest testable piece of software, by which I mean that it can be compiled or assembled, linked, loaded, and put under the control of a **test harness** or **driver**. A **unit** is usually the work of one programmer and it consists of several hundred or fewer, lines of source code. **Unit testing** is the testing we do to show that the unit does not satisfy its functional specification and/or that its implemented structure does not match the intended design structure. When our tests reveal such faults, we say that there is a **unit bug**.
- **Component, Component Testing**—A **component** is an **integrated aggregate** of one or more units. A unit is a component, a component with subroutines it calls is a component, etc. By this (recursive) definition, a component can be anything from a unit to an entire system. **Component testing** is the testing we do to show that the component does not satisfy its functional specification and/or that its implemented structure does not match the intended design structure.
- When our tests reveal such problems, we say that there is a **component bug**. **Integration, Integration Testing**—**Integration** is a *process* by which components are aggregated to create larger components. **Integration testing** is testing done to show that even though the components were individually satisfactory, as demonstrated by successful passage of component tests, the combination of components are incorrect or inconsistent. For example, components A and B have both passed their component tests.
- Integration testing is aimed as showing inconsistencies between A and B. Examples of such inconsistencies are improper call or return sequences, inconsistent data validation criteria, and inconsistent handling of data objects. Integration testing should not be confused with testing integrated objects, which is just higher level component testing. Integration testing is specifically aimed at exposing the problems that arise from the combination of components. The sequence, then, consists of component testing for components A and B, integration testing for the combination of A and B, and finally, component testing for the "new" component (A,B).*
- **System, System Testing**—A **system** is a big component. **System testing** is aimed at revealing bugs that cannot be attributed to components as such, to the inconsistencies between components, or to the planned interactions of components and other objects. System testing concerns issues and behaviors that can only be exposed by testing the entire integrated system or a major part of it. System testing includes testing for performance, security, accountability, configuration sensitivity, start-up, and recovery.
- This book concerns component testing, but the techniques discussed here also apply to integration and system testing. There aren't any special integration and system testing techniques but the mix of effective techniques changes as our concern shifts from components to integration, to system. How and where integration and system testing will be covered is discussed in the preface to this book. You'll find comments on techniques concerning their relative effectiveness as applied to component, integration, and system testing throughout the book. Such comments are intended to guide your selection of a mix of techniques that best matches your testing concerns, be it component, integration, or system, or some mixture of the three.

(viii)The Role of Models:

- Testing is a process in which we create mental models of the environment, the program, human nature, and the tests themselves. Each model is used either until we accept the behavior as correct or until the model is no longer sufficient for the purpose. Unexpected test results always force a revision of some mental model, and in turn may lead to a revision of

whatever is being modeled. The revised model may be more detailed, which is to say more complicated, or more abstract, which is to say simpler. The art of testing consists of creating, selecting, exploring, and revising models. Our ability to go through this process depends on the number of different models we have at hand and their ability to express a program's behavior.

(4) The Consequences of Bugs:

(i) The Importance of Bugs:

- The importance of a bug depends on frequency, correction cost, installation cost, and consequences.
- **Frequency**—How often does that kind of bug occur? See Table 2.1 on page 57 for bug frequency statistics. Pay more attention to the more frequent bug types.
- **Correction Cost**—What does it cost to correct the bug after it's been found? That cost is the sum of two factors: (1) the cost of discovery and (2) the cost of correction. These costs go up dramatically the later in the development cycle the bug is discovered. Correction cost also depends on system size. The larger the system the more it costs to correct the same bug.
- **Installation Cost**—Installation cost depends on the number of installations: small for a single-user program, but how about a PC operating system bug? Installation cost can dominate all other costs—fixing one simple bug and distributing the fix could exceed the entire system's development cost.
- **Consequences**—What are the consequences of the bug? You might measure this by the mean size of the awards made by juries to the victims of your bug.
- A reasonable metric for bug importance is:
  importance($) = frequency*(correction_cost + installation_cost + consequential_cost)
- Frequency tends not to depend on application or environment, but correction, installation, and consequential costs do. As designers, testers, and QA workers, you must be interested in bug importance, not raw frequency. Therefore you must create your own importance model. This chapter will help you do that.

(ii) How Bugs Affect Us-Consequences:

- Bug consequences range from mild to catastrophic. Consequences should be measured in human rather than machine terms because it is ultimately for humans that we write programs. If you answer the question, "What are the consequences of this bug?" in machine terms by saying, for example, "Bit so-and-so will be set instead of reset," you're avoiding responsibility for the bug. Although it may be difficult to do in the scope of a subroutine, programmers should try to measure the consequences of their bugs in human terms. Here are some consequences on a scale of one to ten:
- **1.** *Mild*—The symptoms of the bug offend us aesthetically; a misspelled output or a misaligned printout.
- **2.** *Moderate*—Outputs are misleading or redundant. The bug impacts the system's performance.
- **3.** *Annoying*—The system's behavior, because of the bug, is dehumanizing. Names are truncated or arbitrarily modified. Bills for $0.00 are sent. Operators must use unnatural command sequences and must trick the system into a proper response for unusual bug-related cases.
- **4.** *Disturbing*—It refuses to handle legitimate transactions. The automatic teller machine won't give you money. My credit card is declared invalid.
- **5.** *Serious*—It loses track of transactions: not just the transaction itself (your paycheck), but the fact that the transaction occurred. Accountability is lost.
- **6.** *Very Serious*—Instead of losing your paycheck, the system credits it to another account or converts deposits into withdrawals. The bug causes the system to do the wrong transaction.

- **7.** *Extreme*—The problems aren't limited to a few users or to a few transaction types. They are frequent and arbitrary instead of sporadic or for unusual cases.
- **8.** *Intolerable*—Long-term, unrecoverable corruption of the data base occurs and the corruption is not easily discovered. Serious consideration is given to shutting the system down.
- **9.** *Catastrophic*—The decision to shut down is taken out of our hands because the system fails.
- **10.** *Infectious*—What can be worse than a failed system? One that corrupts other systems even though it does not fail in itself; that erodes the social or physical environment; that melts nuclear reactors or starts wars; whose influence, because of malfunction, is far greater than expected; a system that kills.
- Any of these consequences could follow from that wrong bit. Programming is a serious business, and testing is more serious still. It pays to have nightmares about undiscovered bugs once in a while (SHED80). When was the last time one of your bugs violated someone's human rights?

(iii) Flexible Severity Rather Than Absolutes:

- Many programmers, testers, and quality assurance workers have an absolutist attitude toward bugs. "Everybody *knows* that a program must be *perfect* if it's to work: if there's a bug, it *must* be fixed." That's untrue, of course, even though the myth continues to be foisted onto an unwary public. Ask the person in the street and chances are that they'll parrot that myth of ours. That's trouble for us because we can't do it now and never could. It's *our* myth because we, the computer types, created it and continue to perpetuate it. Software never was perfect and won't get perfect. But is that a license to create garbage? The missing ingredient is our reluctance to quantify quality. If instead of saying that software has either 0 quality (there is at least one bug) or 100% (perfect quality and no bugs), we recognize that quality can be measured on some scale, say from 0 to 10. Quality can be measured as a combination of factors, of which the number of bugs and their severity is only one component. The details of how this is done is the subject of another book; but it's enough to say that many organizations have designed and use satisfactory, quantitative, quality metrics. Because bugs and their symptoms play a significant role in such metrics, as testing progresses you can see the quality rise from next to zero to some value at which it is deemed safe to ship the product.
- Examining these metrics closer, we see that how the parts are weighted depends on environment, application, culture, and many other factors.
- Let's look at a few of these:
- **Correction Cost**—The cost of correcting a bug has almost nothing to do with symptom severity. Catastrophic, life-threatening bugs could be trivial to fix, whereas minor annoyances could require major rewrites to correct.
- **Context and Application Dependency**—The severity of a bug, for the same bug with the same symptoms, depends on context. For example, a roundoff error in an orbit calculation doesn't mean much in a spaceship video game but it matters to real astronauts.
- **Creating Culture Dependency**—What's important depends on the creators of the software and their cultural aspirations. Test tool vendors are more sensitive about bugs in their products than, say, games software vendors.
- **User Culture Dependency**—What's important depends on the user culture. An R&D shop might accept a bug for which there's a workaround; a banker would go to jail for that same bug; and naive users of PC software go crazy over bugs that pros ignore.
- **The Software Development Phase**—Severity depends on development phase. Any bug gets more severe as it gets closer to field use and more severe the longer it's been around—

more severe because of the dramatic rise in correction cost with time. Also, what's a trivial or subtle bug to the designer means little to the maintenance programmer for whom all bugs are equally mysterious.

(iv) The Nightmare List and When to Stop Testing:

- In George Orwell's novel, *1984*, there's a torture chamber called "room 101"—a room that contains your own special nightmare. For me, sailing through 4-foot waves, the boat heeled over, is exhilarating; for my seasick passengers, that's room 101. For me, rounding Cape Horn in winter, with 20-foot waves in a gale is a room 101 but I've heard round-the-world sailboat racers call such conditions "bracing."
- The point about bugs is that you or your organization must define your own nightmares. I can't tell you what they are, and therefore I can't ascribe a severity to bugs. Which is why I treat all bugs as equally as I can in this book. And when I slip and express a value judgment about bugs, recognize it for what it is because I can't completely rid myself of my own values.
- How should you go about quantifying the nightmare? Here's a workable procedure:
- **1.** List your worst software nightmares. State them in terms of the symptoms they produce and how your user will react to those symptoms. For end users and the population at large, the categories of Section 2.2 above are a starting point. For programmers the nightmare may be closer to home, such as: "I might get a bad personal performance rating."
- **2.** Convert the consequences of each nightmare into a cost. Usually, this is a labor cost for correcting the nightmare, but if your scope extends to the public, it could be the cost of lawsuits, lost business, or nuclear reactor meltdowns.
- **3.** Order the list from the costliest to the cheapest and then discard the low-concern nightmares with which you can live.
- **4.** Based on your experience, measured data (the best source to use), intuition, and published statistics postulate the kinds of bugs that are likely to create the symptoms expressed by each nightmare. Don't go too deep because most bugs are easy. This is a bug design process. If you can "design" the bug by a one-character or one statement change, then it's a good target. If it takes hours of sneaky thinking to characterize the bug, then either it's an unlikely bug or you're worried about a saboteur in your organization, which could be appropriate in some cases. Most bugs are simple goofs once you find and understand them.
- **5.** For each nightmare, then, you've developed a list of possible causative bugs. Order that list by decreasing probability. Judge the probability based on your own bug statistics, intuition, experience, etc. The same bug type will appear in different nightmares. The importance of a bug type is calculated by multiplying the expected cost of the nightmare by the probability of the bug and summing across all nightmares:
- **6.** Rank the bug types in order of decreasing importance to you.
- **7.** Design tests (based on your knowledge of test techniques) and design your quality assurance inspection process by using the methods that are most effective against the most important bugs.
- **8.** If a test is passed, then some nightmares or parts of them go away. If a test is failed, then a nightmare is possible, but upon correcting the bug, it too goes away. Testing, then, gives you information you can use to revise your estimated nightmare probabilities. As you test, revise the probabilities and reorder the nightmare list. Taking whatever information you get from testing and working it back through the exercise leads you to revise your subsequent test strategy, either on this project if it's big enough or long enough, or on subsequent projects.
- **9.** Stop testing when the probability of all nightmares has been shown to be inconsequential as a result of hard evidence produced by testing.

www.Jntufastupdates.com

- The above prescription can be implemented as a formal part of the software development process, or it can be adopted as a guideline or philosophical point of view. The idea is not that you implement elaborate metrics (unless that's appropriate) but that you recognize the importance of the feedback that testing provides to the testing process itself and, more important, to the kinds of tests you will design.
- The mature tester's problem has never been how to design tests. If you understand testing techniques, you will know how to design several different infinities of justifiable tests. The tester's central problem is how to best cull a reasonable, finite, number of tests from that multifold infinity—a test suite that, as experience and logic leads us to predict, will have a high probability of putting the nightmares to rest—that is to say, an effective, revealing, set of tests. Look at the pesticide paradox again and observe the following consequence:
- Corollary to the First Law—Test suites wear out.
- Yesterday's elegant, revealing, effective, test suite will wear out because programmers and designers, given feedback on their bugs, do modify their programming habits and style in an attempt to reduce the incidence of bugs they know about. Furthermore, the better the feedback, the better the QA, the more responsive the programmers are, the faster those suites wear out. Yes, the software is getting better, but that only allows you to approach closer to, or to leap over, the previous complexity barrier. True, bug statistics tell you nothing about the coming release, only the bugs of the previous release—but that's better than basing your test technique strategy on general industry statistics or on myths. If you don't gather bug statistics, organized into some rational taxonomy, you don't know how effective your testing has been, and worse, you don't know how worn out your test suite is. The consequences of that ignorance is a brutal shock.
- How many horror stories do you want to hear about the sophisticated outfit that tested long, hard, and diligently—sent release 3.4 to the field, confident that it was the best tested product they had ever shipped—only to have it bomb more miserably than any prior release?

(5) A Taxonomy For Bugs:

(i) General:

- There is no universally correct way to categorize bugs. This taxonomy is not rigid. Bugs are difficult to categorize. A given bug can be put into one or another category depending on its history and the programmer's state of mind. For example, a one-character error in a source statement changes the statement, but unfortunately it passes syntax checking. As a result, data are corrupted in an area far removed from the actual bug. That in turn leads to an improperly executed function. Is this a typewriting error, a coding error, a data error, or a functional error? If the bug is in our own program, we're tempted to blame it on typewriting;** if in another programmer's code, on carelessness. And if our job is to critique the system, we might say that the fault is an inadequate internal data-validation mechanism. A detailed taxonomy is presented in the appendix.
- The major categories are: requirements, features and functionality, structure, data, implementation and coding, integration, system and software architecture, and testing. A first breakdown is provided in Table 2. 1, whereas in the appendix the breakdown is as fine as makes sense. Bug taxonomy, as testing, is potentially infinite. More important than adopting the "right" taxonomy is that you adopt *some* taxonomy and that you use it as a statistical framework on which to base your testing strategy. Because there's so much effort required to develop a taxonomy, don't redo my work—you're invited to adopt the taxonomy of the appendix (or any part thereof) and are hereby authorized to copy it (with appropriate attribution) without guilt or fear of being sued by me for plagiarism. If my taxonomy doesn't turn you on, adopt the IEEE taxonomy (IEEE87B).

www.Jntufastupdates.com

(ii) Requirements, Features, and Functionality Bugs:

(a) Requirements and Specifications:

- Requirements and the specifications developed from them can be incomplete, ambiguous, or self-contradictory. They can be misunderstood or impossible to understand. The specification may assume, but not mention, other specifications and prerequisites that are known to the specifier but not to the designer. And specifications that don't have these flaws may change while the design is in progress. Features are modified, added, and deleted. The designer has to hit a moving target and occasionally misses.
- Requirements, especially as expressed in a specification (or often, as *not* expressed because there is no specification) are a major source of expensive bugs. The range is from a few percent to more than 50%, depending on application and environment. What hurts most about these bugs is that they're the earliest to invade the system and the last to leave. It's not unusual for a faulty requirement to get through all development testing, beta testing, and initial field use, only to be caught after hundreds of sites have been installed.

(b) Feature Bugs:

- Specification problems usually create corresponding feature problems. A feature can be wrong, missing, or superfluous. A missing feature or case is the easiest to detect and correct. A wrong feature could have deep design implications. Extra features were once considered desirable. We now recognize that "free" features are rarely free. Any increase in generality that does not contribute to reliability, modularity, maintainability, and robustness should be suspected. Gratuitous enhancements can, if they increase complexity, accumulate into a fertile compost heap that breeds future bugs, and they can create holes that can be converted into security breaches. Conversely, one cannot rigidly forbid additional features that might be a consequence of good design. Removing the features might complicate the software, consume more resources, and foster more bugs.

(c) Feature Interaction:

- Providing clear, correct, implementable, and testable feature specifications is not enough. Features usually come in groups of related features. The features of each group and the interaction of features within each group are usually well tested. The problem is unpredictable interactions between feature groups or even between individual features. For example, your telephone is provided with call holding and call forwarding. Call holding allows you to put a new incoming call on hold while you continue talking to the first caller. Call forwarding allows you to redirect incoming calls to some other telephone number. Here are some simple feature interaction questions: How about holding a third call when there is already a call on hold? Forwarding forwarded calls (i.e., the number forwarded to is also forwarding calls)? Forwarding calls in a loop? Holding while forwarding is active? Initiating forwarding when there is a call on hold? Holding for forwarded calls when the telephone forwarded to does (doesn't) have forwarding? . . . If you think these variations are brain twisters, how about feature interactions for your income tax return, say between federal, state, and local tax laws? Every application has its peculiar set of features and a much bigger set of unspecified feature interaction potentials and therefore feature interaction bugs. We have very little statistics on these bugs, but the trend seems to be that as the earlier, simpler, bugs are removed, feature interaction bugs emerge as a major category. Other than deliberately preventing some interactions and testing the important combinations, we have no magic remedy for these problems.

(d) Specification and Feature Bug Remedies:

- Most feature bugs are rooted in human-to-human communication problems. One solution is the use of high-level, formal specification languages or systems (BELF76, BERZ85,

www.Jntufastupdates.com

DAVI88A, DAV18813, FISC79, HAYE85, PROG88, SOFT88, YEHR80). Such languages and systems provide short-term support but, in the long run, do not solve the problem.

- **Short-Term Support**—Specification languages (we'll call them all "languages" hereafter, even though some may be interactive dialogue systems) facilitate formalization of requirements and (partial)* inconsistency and ambiguity analysis. With formal specifications, partially to fully automatic test case generation is possible. Generally, users and developers of such products have found them to be cost-effective.
- **Long-Term Support**—Assume that we have a great specification language and that it can be used to create unambiguous, complete specifications with unambiguous, complete tests and consistent test criteria. A specification written in that language could theoretically be compiled into object code (ignoring efficiency and practicality issues). But this is just programming in HOL squared. The specification problem has been shifted to a higher level but not eliminated. Theoretical considerations aside, given a system which can generate functional tests from specifications, the likeliest impact is a further complexity escalation facilitated by the reduction of another class of bugs (the complexity barrier law).
- The long-term impact of formal specification languages and systems will probably be that they will influence the design of ordinary programming languages so that more of *current* specification can be formalized. This approach will reduce, but not eliminate, specification bugs. The pesticide paradox will work again to eliminate the kinds of specification bugs we now have (simple ambiguities and contradictions), leaving us a residue of tougher specification bugs that will need an even higher order specification system to expose.

(e) Testing Techniques:

- Most **functional test techniques**—that is, those techniques which are based on a behavioral description of software, such as **transaction flow testing** (Chapter 4), **syntax testing** (Chapter 9), **domain testing** (Chapter 6), **logic testing** (Chapter 10), and **state testing** (Chapter 11) are useful in testing functional bugs. They are also useful in testing for requirements and specification bugs to the extent that the requirements can be expressed in terms of the model on which the technique is based.

(iii) Structural Bugs:

(a) Control and Sequence Bugs:

- Control and sequence bugs include paths left out, unreachable code, improper nesting of loops, loop-back or loop-termination criteria incorrect, missing process steps, duplicated processing, unnecessary processing, rampaging GOTO's, ill-conceived switches, **spaghetti code**, and worst of all, **pachinko code**.
- Although much of testing and software design literature focuses on control flow bugs, they are not as common in new software as the literature might lead one to believe. One reason for the popularity of control-flow problems in the literature is that this area is amenable to theoretical treatment. Fortunately, most control-flow bugs (in new code) are easily tested and caught in unit testing.
- Another source of confusion and therefore research concern is that novice programmers working on toy problems do tend to have more control-flow bugs than experienced programmers. A third reason for concern with control-flow problems is that dirty old code, especially assembly language and COBOL code, can be dominated by control-flow bugs. In fact, a good reason to rewrite an application from scratch is that the old control structure has become so complicated and so arbitrary after decades of rework that no one dare modify it further and, further, it defies testing.
- Control and sequence bugs at all levels are caught by testing, especially structural testing, more specifically, path testing (Chapter 3), combined with a bottom-line functional test based on a specification. These bugs are partially prevented by language choice (e.g., languages

www.Jntufastupdates.com

that restrict control-flow options) and style, and most important, lots of memory. Experience shows that many control-flow problems result directly from trying to "squeeze" 8 pounds of software into a 4-pound bag (i.e., 8K object into 4K). Squeezing for short execution time is as bad.

(b) Logic Bugs:

- Bugs in logic, especially those related to misunderstanding how case statements and logic operators behave singly and in combinations, include nonexistent cases, improper layout of cases, "impossible" cases that are not impossible, a "don't-care" case that matters, improper negation of a boolean expression (for example, using "greater than" as the negation of "less than"), improper simplification and combination of cases, overlap of exclusive cases, confusing "exclusive OR" with "inclusive OR."
- Another problematic area concerns misunderstanding the semantics of the order in which a boolean expression is evaluated for specific compilers, especially in the context of deeply nested IF-THEN-ELSE constructs. For example, the truth or falsity of a logical expression is determined after evaluating a few terms, so evaluation of further terms (usually) stops, but the programmer expects that further terms will be evaluated. In other words, although the boolean expression appears as a single statement, the programmer does not understand that its components will be evaluated sequentially. See index entries on **predicate coverage** for more information.
- If these bugs are part of logical (i.e., boolean) processing not related to control flow, then they are categorized as processing bugs. If they are part of a logical expression (i.e., **control-flow predicate**) which is used to direct the control flow, then they are categorized as control-flow bugs.
- Logic bugs are not really different in kind from arithmetic bugs. They are likelier than arithmetic bugs because programmers, like most people, have less formal training in logic at an early age than they do in arithmetic. The best defense against this kind of bug is a systematic analysis of cases. Logic-based testing (Chapter 10) is helpful.

(c) Processing Bugs:

- Processing bugs include arithmetic bugs, algebraic, mathematical function evaluation, algorithm selection, and general processing. Many problems in this area are related to incorrect conversion from one data representation to another. This is especially true in assembly language programming. Other problems include ignoring overflow, ignoring the difference between positive and negative zero, improper use of greater-than, greater-than-or-equal, less-than, less-than-or-equal, assumption of equality to zero in floating point, and improper comparison between different formats as in ASCII to binary or integer to floating point.
- Although these bugs are frequent (12%), they tend to be caught in good unit testing and also tend to have localized effects. Selection of covering test cases, especially domain-testing methods (Chapter 6) are the testing remedies for this kind of bug.

(d) Initialization Bugs:

- Initialization bugs are common, and experienced programmers and testers know they must look for them. Both improper and superfluous initialization occur. The latter tends to be less harmful but can affect performance. Typical bugs are as follows: forgetting to initialize working space, registers, or data areas before first use or assuming that they are initialized elsewhere; a bug in the first value of a loop-control parameter; accepting an initial value without a validation check; and initializing to the wrong format, data representation, or type.
- The remedies here are in the kinds of tools the programmer has. The source language also helps. Explicit declaration of all variables, as in Pascal, helps to reduce some initialization problems. Preprocessors, either built into the language or run separately, can detect some,

but not all, initialization problems. The test methods of Chapter 5 are helpful for test design and for debugging initialization problems.

(e) Data Flow Bugs and Anomalies:

- Most initialization bugs are a special case of data-flow anomalies. A **data-flow anomaly** occurs when there is a path along which we expect to do something unreasonable with data, such as using an uninitialized variable, attempting to use a variable before it exists, modifying data and then not storing or using the result, or initializing twice without an intermediate use. Although part of data-flow anomaly detection can be done by the compiler based on information known at compile time, much can be detected only by execution and therefore is a subject for testing. It is generally recognized today that data-flow anomalies are as important as control-flow anomalies. The methods of Chapters 5 and 12 will help you design tests aimed at data-flow problems.

(iv) Data Bugs:

(a) General:

- Data bugs include all bugs that arise from the specification of data objects, their formats, the number of such objects, and their initial values. Data bugs are at least as common as bugs in code, but they are often treated as if they did not exist at all. Underestimating the frequency of data bugs is caused by poor bug accounting. In some projects, bugs in data declarations are just not counted, and for that matter, data declaration statements are not counted as part of the code. The separation of code and data is, of course, artificial because their roles can be interchanged at will. At the extreme, one can write a twenty-instruction program that can simulate any computer (a Turing machine) and have all "programs" recorded as data and manipulated as such. Furthermore, this can be done in any language on any computer—but who would want to?
- Software is evolving toward programs in which more and more of the control and processing functions are stored in tables. I call this the third law:
- *Third Law*—Code migrates to data.
- Because of this law there is an increasing awareness that bugs in code are only half the battle and that data problems should be given equal attention. The bug statistics of Table 2.1 support this concept; that is, structural bugs and data bugs each have frequencies of about 25%. If you examine a piece of contemporary source code, you may find that half of the statements are data declarations. Although these statements do not result in executable code, because they are specified by humans, they are as subject to error as operative statements. If a program is designed under the assumption that a certain data object will be set to zero and it isn't, the operative statements of the program are not at fault. Even so, there is still an initialization bug, which, because it is in a data statement, could be harder to find than if it had been a bug in executable code.
- This increase in the proportion of the source statements devoted to data definition is a direct consequence of two factors: (1) the dramatic reduction in the cost of main memory and disc storage, and (2) the high cost of creating and testing software. Generalized software controlled by tables is not efficient. Computer costs, especially memory costs, have decreased to the point where the inefficiencies of generalized table-driven code are not usually significant. The increasing cost of software as a percentage of system cost has shifted the emphasis in the software industry away from single-purpose, unique software to an increased reliance on prepackaged, generalized programs. This trend is evident in the computer manufacturers' software, in the existence of a healthy proprietary software industry, and in the emergence of languages and programming environments that support code reusability (e.g., object-oriented languages). Generalized packages must satisfy a wide range of options, host configurations, operating systems, and computers. The designer of a

www.Jntufastupdates.com

generalized package achieves generality, in part, by making many things parametric, such as array sizes, memory partition, and file structure. It is not unusual for a big application package to have several hundred parameters. Setting the parameter values particularizes the program to the specific installation. The parameters are interrelated, and errors in those relations can cause illogical conditions and, therefore, bugs.

- Another source of database complexity increase is the use of control tables in lieu of code. The simplest example is the use of tables that turn processing options on and off. A more complicated form of control table is used when a system must execute a set of closely related processes that have the same control structure but are different in details. An early example is found in telephony, where the details of controlling a telephone call are table-driven. A generalized call-control processor handles calls from and to different kinds of lines. The system is loaded with a set of tables that corresponds to the protocols required for that telephone exchange. Another example is the use of generalized device-control software which is particularized by data stored in device tables. The operating system can be used with new, undefined devices, if those devices' parameters can fit into a set of very broad values. The culmination of this trend is the use of complete, internal, transaction-control languages designed for the application. Instead of being coded as computer instructions or language statements, the steps required to process a transaction are stored as a sequence of constants in a transaction-processing table. The state of the transaction, that is, the current processing step, is stored in a transaction-control block. The generalized transaction-control processor uses the combination of transaction state and the control tables to direct the transaction to the next step. The transaction-control table is actually a program which is processed interpretively by the transaction-control processor. That program may contain the equivalent of addressing, conditional branch instructions, looping statements, case statements, and so on. In other words, a **hidden programming language** has been created. It is an effective design technique because it enables fixed software to handle many different transaction types, individually and simultaneously. Furthermore, modifying the control tables to install new transaction types is usually easier than making the same modifications in code.
- In summary, current programming trends are leading to the increasing use of undeclared, internal, specialized programming languages. These are languages—make no mistake about that—even if they are simple compared to normal programming languages; but the syntax of these languages is rarely debugged. There's no compiler for them and therefore no source syntax checking. The programs in these languages are inserted as octal or hexadecimal codes—as if we were programming back in the early days of UNIVAC-I. Large, low-cost memory will continue to strengthen this trend and, consequently, there will be an increased incidence of code masquerading as data. Bugs in this kind of hidden code are at least as difficult to find as bugs in normal code. The first step in the avoidance of data bugs—whether the data are used as pure data, as parameters, or as hidden code—is the realization that *all* source statements, including data declarations, must be counted, and that all source statements, whether or not they result in object code, are bug-prone.
- The categories used for data bugs are different from those used for code bugs. Each way of looking at data provides a different perspective. These categories for data bugs overlap and are no stricter than the categories used for bugs in code.

(b) Dynamic Versus Static:

- Dynamic data are transitory. Whatever their purpose, they have a relatively short lifetime, typically the processing time of one transaction. A storage object may be used to hold dynamic data of different types, with different formats, attributes, and residues. Failure to initialize a shared object properly can lead to data-dependent bugs caused by residues from a previous use of that object by another transaction. Note that the culprit transaction is long

www.Jntufastupdates.com

gone when the bug's symptoms are discovered. Because the effect of corruption of dynamic data can be arbitrarily far removed from the cause, such bugs are among the most difficult to catch. The design remedy is complete documentation of all shared-memory structures, defensive code that does thorough data-validation checks, and centralized-resource managers.

- The basic problem is leftover garbage in a shared resource. This can be handled in one of three ways: (1) cleanup after use by the user, (2) common cleanup by the resource manager, and (3) no cleanup. The latter is the method usually used. Therefore, resource users must program under the assumption that the resource manager gives them garbage-filled resources. Common cleanup is used in very secure systems where subsequent users of a resource must never be able to read data left by a previous user in another security or privacy category.
- Static data are fixed in form and content. Whatever their purpose, they appear in the source code or data base, directly or indirectly, as, for example, a number, a string of characters, or a bit pattern. Static data need not be explicit in the source code. Some languages provide **compile-time processing**, which is especially useful in general-purpose routines that are particularized by interrelated parameters. Compile-time processing is an effective measure against parameter-value conflicts. Instead of relying on the programmer to calculate the correct values of interrelated parameters, a program executed at compile time (or assembly time) calculates the parameters' values. If compile-time processing is not a language feature, then a specialized preprocessor can be built that will check the parameter values and calculate those values that are derived from others. As an example, a large commercial telecommunications system has several hundred parameters that dictate the number of lines, the layout of all storage media, the hardware configuration, the characteristics of the lines, the allowable user options for those lines, and so on. These are processed by a site-adapter program that not only sets the parameter values but builds data declarations, sizes arrays, creates constants, and inserts processing routines from a library. A bug in the site adapter, or in the data given to the site adapter, can result in bugs in the static data used by the object programs for that site.
- Another example is the postprocessor used to install many personal computer software packages. Here the configuration peculiarities are handled by generalized table-driven software, which is particularized at run (actually, installation) time.
- Any preprocessing (or postprocessing) code, any code executed at compile or assembly time or before, at load time, at installation time, or some other time can lead to faulty static data and therefore bugs—even though such code (and the execution thereof) does not represent object code at run time. We tend to take compilers, assemblers, utilities, loaders, and configurators for granted and do not suspect them to be bug sources. This is not a bad assumption for standard utilities or translators. But if a highly parameterized system uses site-adapter software or preprocessors or compile-time/assembly-time processing, and if such processors and code are developed concurrently with the working software of the application—watch out!
- Software used to produce object code is suspect until validated. All new software must be rigorously tested even if it isn't part of the application's mainstream. Static data can be just as wrong as any other kind and can have just as many bugs. Do not treat a routine that creates static data as "simple" because it "just stuffs a bunch of numbers into a table." Subject such code to the same testing rigor that you apply to running code.*
- The design remedy for the preprocessing situation is in the source language. If the language permits compile-time processing that can be used to particularize parameter values and data structures, and if the syntax of the compile-time statements is identical to the syntax of the

www.Jntufastupdates.com

rest of the language, then the code will be subjected to the same validation and syntax checking as ordinary code. Such language facilities eliminate the need for most specialized preprocessors, table generators, and site adapters. For postprocessors, there is no magic, other than to recognize that users judge developers by the entire picture, installation software included.

(c) Information, Parameter, and Control:

- Static or dynamic data can serve in one of three roles, or in a combination of roles: as a parameter, for control, or for information. What constitutes control or information is a matter of perspective and can shift from one processing level to another. A scheduler receives a request to start a process. To the scheduler the identity of the process is information to be processed, but at another level it is control. My name is used to generate a hash code that will be used to access a disc record. My name is information, but to the disc hardware its translation into an address is control (e.g., move to track so-and-so).
- Information is usually dynamic and tends to be local to a single transaction or task. As such, errors in information (when data are treated as information, that is) may not be serious bugs. The bug, if any, is in the lack of protective data-validation code or in the failure to protect the routine's logic from out-of-range data or data in the wrong format. The only way we can be sure that there is data-validation code in a routine is to put it there. Assuming that the other routine will validate data invites latent bugs and maintenance problems. The program evolves and changes, and it is forgotten that the modified routine did the data validation for several other routines. *If* a routine is vulnerable to bad data, the only sane thing to do is to block such data within the routine; but it's even better to redesign it so that it is no longer vulnerable.
- Inadequate data validation often leads to finger pointing. The calling routine's author is blamed, the called routine's author blames back, they both blame the operators. This scenario leads to a lot of ego confrontation and guilt. "If only the other programmers did their job correctly," you say, "we wouldn't need all this redundant data validation and defensive code. I have to put in this extra junk because I'm surrounded by slobs!" This attitude is understandable, but not productive. Furthermore, if you really feel that way, you're likely to feel guilty about it. Don't blame your fellow programmer and don't feel guilt. Nature has conspired against us but given us a scapegoat. One of the unfortunate side effects of large-scale integrated circuitry stems from the use of microscopic logic elements that work at very low energy levels. Modern circuitry is vulnerable to electronic noise, electromagnetic radiation, cosmic rays, neutron hits, stray alpha particles, and other noxious disturbances. No kidding—alpha-particle hits that can change the value of a bit are a serious problem, and the semiconductor manufacturers are spending a lot of money and effort to reduce the random modification of data by alpha particles. Therefore, even if your fellow programmers did thorough, correct data validation, dynamic data, static data, parameters, and code can be corrupted. Program without rancor and guilt! Put in the data-validation checks and blame the necessity on sun spots and alpha particles!*

(d) Contents, Structure, and Attributes:

- Data specifications consist of three parts:
- **Contents**—The actual bit pattern, character string, or number put into a data structure. Content is a pure bit pattern and has no meaning unless it is interpreted by a hardware or software processor. All data bugs result in the corruption or misinterpretation of content.
- **Structure**—The size and shape and numbers that describe the data object, that is, the memory locations used to store the content (e.g., 16 characters aligned on a word boundary, 122 blocks of 83 characters each, bits 4 through 14 of word 17). Structures can have substructures and can be arranged into superstructures. A hunk of memory may have

www.Jntufastupdates.com

several different structures defined over it—e.g., a two-dimensional array treated elsewhere as *N* one-dimensional arrays.

- **Attributes**—The specification of meaning, that is, the semantics associated with the contents of a data object (e.g., an integer, an alphanumeric string, a subroutine).
- The severity and subtlety of bugs increases as we go from content to attributes because things get less formal in that direction. Content has been dealt with earlier in this section. Structural bugs can take the form of declaration bugs, but these are not the worst kind of structural bugs. A serious potential for bugs occurs when data are used with different structures. Here is a piece of clever design. The programmer has subdivided the problem into eight cases and uses a 3-bit field to designate the case. Another programmer has four different cases to consider and uses a 2-bit field for the purpose. A third programmer is interested in the combination of the other two sets of cases and treats the whole as a 5-bit field that leads to thirty-two combined cases. We cannot judge, out of context, whether this is a good design or an abomination, but we can note that there is a different structure in the minds of the three programmers and therefore a potential for bugs. The practice of interpreting a given memory location under several different structures is not intrinsically bad. Often, the only alternative would be increased memory and many more data transfers.
- Attributes of data are the meanings we associate with data. Although some bugs are related to misinterpretation of integers for floating point and other basic representation problems, the more subtle attribute-related bugs are embedded in the application. Consider a 16-bit field. It could represent, among other things, a number, a loop-iteration count, a control code, a pointer, or a link field. Each interpretation is a different attribute. There is no way for the computer to know that it is proper or improper to add a control code to a link field to yield a loop count. We have used the same data with different meanings. In modern parlance, we have changed the **data type**. It is generally incorrect to logically or arithmetically combine objects whose types are different. Conversely, it is almost impossible to create an efficient system without doing so. Shifts in interpretation usually occur at interfaces, especially the human interface that is behind every software interface. See GANN76 for a summary of **type bugs**.
- The preventive measures for data-type bugs are in the source language, documentation, and coding style. Explicit documentation of the contents, structure, and attributes of all data objects is essential. The database documentation should be centralized. All alternate interpretation of a given data object should be listed along with the identity of all routines that have access to that object. A proper **data dictionary** (which is what the database documentation is called) can be as large as the narrative description of the code. The data dictionary and the database it represents must also be designed. This design is done by a high-level design process, which is as important as the design of the software architecture. My point of view here is dogmatic. Routines should not be administratively treated as if they have their “own” data declarations.* All data structures should be globally defined and centrally administered. Exceptions, such as a private work area, should be individually justified. Such private data structures must never be used by any other routine but the structure must still be documented in the data dictionary.
- It’s impossible to properly test software of any size (say 10,000+ statements) without central database management and a configuration-controlled data dictionary. I was once faced with such a herculean challenge. My first step was to try to create the missing data dictionary preparatory to any attempt to define tests. The act of dragging the murky bottoms of a hundred minds for hidden data declarations and semiprivate space in an attempt to create a data dictionary revealed so many data bugs that it was obvious that the system would defy

www.Jntufastupdates.com

integration. I never did get to design tests for that project—it collapsed; and a new design was started surreptitiously from scratch.

- The second remedy is in the source language. **Strongly typed languages** prevent the inadvertent mixed manipulation of data that are declared as different types. A conversion in usage from pointer type to counter type, say, requires an explicit statement that will do the conversion. Such statements may or may not result in object code. Conversion from floating point to integer, would, of course, require object code, but conversion from pointer to counter might not. Strong typing forces the explicit declaration of attributes and provides compiler facilities to check for mixed-type operations. The ability of the user to specify types, as in Pascal, is mandatory. These data-typing facilities force the specification of data attributes into the source code, which makes them more amenable to automatic verification by the compiler and to test design than when the attributes are described in a separate data dictionary. In assembly language programming, or in source languages that do not have user-defined types, the remedy is the use of **field-access macros**. No programmer is allowed to directly access a field in the database. Access can be obtained only through the use of a field-access macro. The macro code does all the extraction, stripping, justification, and type conversion necessary. If the database structure has to be changed, the affected field-access macros are changed, but the source code that uses the macros does not (usually) have to be changed. The attributes of the data are documented with the field-access macro documentation. Another advantage of this approach is that the data dictionary can be automatically produced from the specifications of the field-access macro library.

(v) Coding Bugs:

- Coding errors of all kinds can create any of the other kinds of bugs. Syntax errors are generally not important in the scheme of things if the source language translator has adequate syntax checking. Failure to catch a syntax error is a bug in the translator. A good translator will also catch undeclared data, undeclared routines, dangling code, and many initialization problems. Any programming error caught by the translator (assembler, compiler, or interpreter) does not substantially affect test design and execution because testing cannot start until such errors are corrected. Whether it takes a programmer one, ten, or a hundred passes before a routine can be tested should concern software management (because it is a programming productivity issue) but not test design (which is a quality-assurance issue). But if a program has many source-syntax errors, we should expect many logic and coding bugs—because a slob is a slob is a slob.
- Given good source-syntax checking, the most common pure coding errors are typographical, followed by errors caused by not understanding the operation of an instruction or statement or the by-products of an instruction or statement. Coding bugs are the wild cards of programming. Unlike logic or process bugs, which have their own perverse rationality, wild cards are arbitrary.
- The most common kind of coding bug, and often considered the least harmful, are documentation bugs (i.e., erroneous comments). Although many documentation bugs are simple spelling errors or the result of poor writing, many are actual errors—that is, misleading or erroneous comments. We can no longer afford to discount such bugs because their consequences are as great as “true” coding errors. Today, programming labor is dominated by maintenance. This will increase as software becomes even longer-lived. Documentation bugs lead to incorrect maintenance actions and therefore cause the insertion of other bugs. Testing techniques have nothing to offer for these bugs. The solution lies in inspections, QA, automated data dictionaries, and specification systems.

www.Jntufastupdates.com

(vi) Interface, Integration, and System Bugs:

(a) External Interfaces:

- The external interfaces are the means used to communicate with the world. These include devices, actuators, sensors, input terminals, printers, and communication lines. Often there is a person on the other side of the interface. That person may be ingenious or ingenuous, but is frequently malevolent. The primary design criterion for an interface with the outside world should be **robustness**. All external interfaces, human or machine, employ a protocol. Protocols are complicated and hard to understand. The protocol itself may be wrong, especially if it's new, or it may be incorrectly implemented. Other external interface bugs include: invalid timing or sequence assumptions related to external signals; misunderstanding external input and output formats; and insufficient tolerance to bad input data. The test design methods of Chapters 6, 9, and 11 are suited to testing external interfaces.

(b) Internal Interfaces:

- Internal interfaces are in principle not different from external interfaces, but there are differences in practice because the internal environment is more controlled. The external environment is fixed and the system must adapt to it but the internal environment, which consists of interfaces with other components, can be negotiated. Internal interfaces have the same problems external interfaces have, as well as a few more that are more closely related to implementation details: protocol-design bugs, input and output format bugs, inadequate protection against corrupted data, wrong subroutine call sequence, call-parameter bugs, misunderstood entry or exit parameter values.
- To the extent that internal interfaces, protocols, and formats are formalized, the test methods of Chapters 6, 9, and 11 will be helpful. The real remedy is in the design and in standards. Internal interfaces should be standardized and not just allowed to grow. They should be formal, and there should be as few as possible. There's a trade-off between the number of different internal interfaces and the complexity of the interfaces. One universal interface would have so many parameters that it would be inefficient and subject to abuse, misuse, and misunderstanding. Unique interfaces for every pair of communicating routines would be efficient, but N programmers could lead to N^2 interfaces, most of which wouldn't be documented and all of which would have to be tested (but wouldn't be). The main objective of integration testing is to test all internal interfaces (BEIZ84).

(c) Hardware Architecture:

- It's easy to forget that hardware exists. You can have a programming career and never see a mainframe or minicomputer. When you are working through successive layers of application executive, operating system, compiler, and other intervening software, it's understandable that the hardware architecture appears abstract and remote. It is neither practical nor economical for every programmer in a large project to know all aspects of the hardware architecture. Software bugs related to hardware architecture originate mostly from misunderstanding how the hardware works. Here are examples: paging mechanism ignored or misunderstood, address-generation error, I/O-device operation or instruction error, I/O-device address error, misunderstood device-status code, improper hardware simultaneity assumption, hardware race condition ignored, data format wrong for device, wrong format expected, device protocol error, device instruction-sequence limitation ignored, expecting the device to respond too quickly, waiting too long for a response, ignoring channel throughput limits, assuming that the device is initialized, assuming that the device is not initialized, incorrect interrupt handling, ignoring hardware fault or error conditions, ignoring operator malice.

www.Jntufastupdates.com

- The remedy for hardware architecture and interface problems is two-fold: (1) good programming and testing and (2) centralization of hardware interface software in programs written by hardware interface specialists. Hardware interface testing is complicated by the fact that modern hardware has very few buttons, switches, and lights. Old computers had lots of them, and you could abuse those buttons and switches to create wonderful anomalous interface conditions that could not be simulated any other way. Today's highly integrated black boxes rarely have such controls and, consequently, considerable ingenuity may be needed to simulate and test hardware interface status conditions. Modern hardware is better and cheaper without the buttons and lights, but also harder to test. This paradox can be resolved by hardware that has special test modes and test instructions that do what the buttons and switches used to do. The hardware manufacturers, as a group, have yet to provide adequate features of this kind. Often the only alternative is to use an elaborate hardware simulator instead of the real hardware. Then you're faced with the problem of distinguishing between real bugs and hardware simulator implementation bugs.

(d) Operating System:

- Program bugs related to the operating system are a combination of hardware architecture and interface bugs, mostly caused by a misunderstanding of what it is the operating system does. And, of course, the operating system could have bugs of its own. Operating systems can lull the programmer into believing that all hardware interface issues are handled by it. Furthermore, as the operating system matures, bugs in it are found and corrected, but some of these corrections may leave quirks. Sometimes the bug is not fixed at all, but a notice of the problem is buried somewhere in the documentation—if only you knew where to look for it.
- The remedy for operating system interface bugs is the same as for hardware bugs: use operating system interface specialists, and use explicit interface modules or macros for all operating system calls. This approach may not eliminate the bugs, but at least it will localize them and make testing easier.

(e) Software Architecture:

- Software architecture bugs are often the kind that are called "interactive." Routines can pass unit and integration testing without revealing such bugs. Many of them depend on load, and their symptoms emerge only when the system is stressed. They tend to be the most difficult kind of bug to find and exhume. Here is a sample of the causes of such bugs: assumption that there will be no interrupts, failure to block or unblock interrupts, assumption that code is reentrant or not reentrant, bypassing data interlocks, failure to close or open an interlock, assumption that a called routine is resident or not resident, assumption that a calling program is resident or not resident, assumption that registers or memory were initialized or not initialized, assumption that register or memory location content did not change, local setting of global parameters, and global setting of local parameters.
- The first line of defense against these bugs is the design. The first bastion of that defense is that there *be* a design for the software architecture. Not designing a software architecture is an unfortunate but common disease. The most elegant test techniques will be helpless in a complicated system whose architecture "just growed" without plan or structure. All test techniques are applicable to the discovery of software architecture bugs, but experience has shown that careful integration of modules and subjecting the final system to a brutal stress test are especially effective (BEIZ84).*

(f) Control and Sequence Bugs:

- System-level control and sequence bugs include: ignored timing; assuming that events occur in a specified sequence; starting a process before its prerequisites are met (e.g., working on data before all the data have arrived from disc); waiting for an impossible combination of

www.Jntufastupdates.com

prerequisites; not recognizing when prerequisites have been met; specifying wrong priority, program state, or processing level; missing, wrong, redundant, or superfluous process steps.

- The remedy for these bugs is in the design. Highly structured sequence control is helpful. Specialized, internal, sequence-control mechanisms, such as an internal job control language, are useful. Sequence steps and prerequisites stored in tables and processed interpretively by a sequence-control processor or dispatcher make process sequences easier to test and to modify if bugs are discovered. **Path testing** as applied to **transaction flowgraphs**, as discussed in Chapter 4, is especially effective at detecting system-level control and sequence bugs.

(g) Resource Management Problems:

- Memory is subdivided into dynamically allocated resources such as buffer blocks, queue blocks, task control blocks, and overlay buffers. Similarly, external mass storage units such as discs, are subdivided into memory-resource pools. Here are some resource usage and management bugs: required resource not obtained (rare); wrong resource used (common, if there are several resources with the same structure or different kinds of resources in the same pool); resource already in use; race condition in getting a resource; resource not returned to the right pool; fractionated resources not properly recombined (some resource managers take big resources and subdivide them into smaller resources, and Humpty Dumpty isn't always put together again); failure to return a resource (common); **resource deadlock** (a type A resource is needed to get a type B, a type B is needed to get a type C, and a type C is needed to get a type A); resource use forbidden to the caller; used resource not returned; resource linked to the wrong kind of queue; forgetting to return a resource.
- A design remedy that prevents bugs is always preferable to a test method that discovers them. The design remedy in resource management is to keep the resource structure simple: the fewest different kinds of resources, the fewest pools, and no private resource management.
- Complicated resource structures are often designed in a misguided attempt to save memory and not because they're essential. The software has to handle, say, large-, small-, and medium-length transactions, and it is reasoned that memory will be saved if three different-sized resources are implemented. This reasoning is often faulty because:
- **1.** Memory is cheap and getting cheaper.
- **2.** Complicated resource structures and multiple pools need management software; that software needs memory, and the increase in program space could be bigger than the expected data space saved.
- **3.** The complicated scheme takes additional processing time, and therefore all resources are held in use a little longer. The size of the pools will have to be increased to compensate for this additional holding time.
- **4.** The basis for sizing the resource is often wrong. A typical choice is to make the buffer block's length equal to the length required by an average transaction—usually a poor choice. A correct analysis (see BEIZ78, pp. 301-302) shows that the optimum resource size is usually proportional to the square root of the transaction's length. However, square-root laws are relatively insensitive to parameter changes and consequently the waste of using many short blocks for long transactions or large blocks to store short transactions isn't as bad as naive intuition suggests.
- The second design remedy is to centralize the management of all pools, either through centralized resource managers, common resource-management subroutines, resource-management macros, or a combination of these.
- I mentioned resource loss three times—it was not a writing bug. Resource loss is the most frequent resource-related bug. Common sense tells you why programmers lose resources.

www.Jntufastupdates.com

You need the resource to process—so it's unlikely that you'll forget to get it; but when the job is done, the successful conclusion of the task will not be affected if the resource is not returned. A good routine attempts to get resources as soon as possible at a common point and also attempts to return them at a common point; but strange paths may require more resources, and you could forget that you're using several resource units instead of one. Furthermore, an exception-condition handler that responds to system-threatening illogical conditions may bypass the normal exit and jump directly to an executive level—and there goes the resource. The design remedies are to centralize resource fetch-and-return within each routine and to provide macros that return all resources rather than just one. Resource-loss problems are exhumed by path testing (Chapter 3), by transaction-flow testing (Chapter 4), data-flow testing (Chapter 5), and by stress testing (BEIZ84).

(h) Integration Bugs:

- **Integration bugs** are bugs having to do with the integration of, and with the interfaces between, presumably working and tested components. Most of these bugs result from inconsistencies or incompatibilities between components. All methods used to transfer data directly or indirectly between components and all methods by which components share data can host integration bugs and are therefore proper targets for integration testing. The communication methods include data structures, call sequences, registers, semaphores, communication links, protocols, and so on. Integration strategies and special testing considerations are discussed in more detail in BEIZ84. While integration bugs do not constitute a big bug category (9%) they are an expensive category because they are usually caught late in the game and because they force changes in several components and/or data structures, often during the height of system debugging. Test methods aimed at interfaces, especially domain testing (Chapter 6), syntax testing (Chapter 9), and data-flow testing when applied across components (Chapter 5), are effective contributors to the discovery and elimination of integration bugs.

(i) System Bugs:

- **System bugs** is a catch-all phrase covering all kinds of bugs that cannot be ascribed to components or to their simple interactions, but result from the totality of interactions between many components such as programs, data, hardware, and the operating system. System testing as a discipline is discussed in BEIZ84. The only test technique that applies obviously and directly to system testing is transaction-flow testing (Chapter 4); but the reader should keep in mind two important facts: (1) all test techniques can be useful at all levels, from unit to system, and (2) there can be no meaningful system testing until there has been thorough component and integration testing. System bugs are infrequent (1.7%) but very important (expensive) because they are often found only after the system has been fielded and because the fix is rarely simple.

(vii) Test and Test Design Bugs:

(a) Testing:

- Testers have no immunity to bugs (see the footnote on page 20). Tests, especially system tests, require complicated scenarios and databases. They require code or the equivalent to execute, and consequently they can have bugs. The virtue of independent functional testing is that it provides an unbiased point of view; but that lack of bias is an opportunity for different, and possibly incorrect, interpretations of the specification. Although test bugs are not software bugs, it's hard to tell them apart, and much labor can be spent making the distinction. Also, consider the maintenance programmer—does it matter whether she's worked 3 days to chase and fix a real bug or wasted 3 days chasing a chimerical bug that was really a faulty test specification?

www.Jntufastupdates.com

(b) Test Criteria:

- The specification is correct, it is correctly interpreted and implemented, and a seemingly proper test has been designed; but the criterion by which the software's behavior is judged is incorrect or impossible. How would you, for example, "prove that the entire system is free of bugs?" If a criterion is quantitative, such as a throughput or processing delay, the act of measuring the performance can perturb the performance measured. The more complicated the criteria, the likelier they are to have bugs.

(c) Remedies:

- The remedies for test bugs are: test debugging, test quality assurance, test execution automation, and test design automation.
- **Test Debugging**—The first remedy for test bugs is testing and debugging the tests. The differences between test debugging and program debugging are not fundamental. Test debugging is usually easier because tests, when properly designed, are simpler than programs and do not have to make concessions to efficiency. Also, tests tend to have a localized impact relative to other tests, and therefore the complicated interactions that usually plague software designers are less frequent. We have no magic prescriptions for test debugging—no more than we have for software debugging.
- **Test Quality Assurance**—Programmers have the right to ask how quality in independent testing and test design is monitored. Should we implement test testers and test—tester tests? This sequence does not converge. Methods for test quality assurance are discussed in *Software System Testing and Quality Assurance* (BEIZ84).
- **Test Execution Automation**—The history of software bug removal and prevention is indistinguishable from the history of programming automation aids. Assemblers, loaders, compilers, and the like were all developed to reduce the incidence of programmer and/or operator errors. Test execution bugs are virtually eliminated by various test execution automation tools, many of which are discussed throughout this book. The point is that "manual testing" is self-contradictory. If you want to get rid of test execution bugs, get rid of manual execution.
- **Test Design Automation**—Just as much of software development has been automated (what is a compiler, after all?) much test design can be and has been automated. For a given productivity rate, automation reduces bug count—be it for software or be it for tests.

(viii) Testing and Design Style:

- This is a book on test design, yet this chapter has said a lot about programming style and design. You might wonder why the productivity of one programming group is as much as 10 times higher than that of another group working on the same application, the same computer, in the same language, and under similar constraints. It should be obvious—bad designs lead to bugs, and bad designs are difficult to test; therefore, the bugs remain. Good designs inhibit bugs before they occur and are easy to test. The two factors are multiplicative, which explains the large productivity differences. The best test techniques are useless when applied to abominable code: it is sometimes easier to redesign a bad routine than to attempt to create tests for it. The labor required to produce new code plus the test design and execution labor for the new code can be much less than the labor required to design thorough tests for an undisciplined, unstructured monstrosity. Good testing works best on good code and good designs. And no test technique can ever convert garbage into gold.

www.Jntufastupdates.com

FLOW GRAPHS AND PATH TESTING

(1) Basics concepts of path testing:

(i) Motivation and Assumptions:

(a) Path testing

- A sequence of statements which starts at an entry and ends at an exit by passing all the existing junctions, decisions etc is known as path.
- Path testing is a process which involves all the available paths in a program from an entry to an exit in such a way that the entire path is thoroughly tested.
- If the set of paths is properly chosen, then we have achieved some measure of test thoroughness.

(b) Motivation

- Path–testing techniques are the oldest of all structural test techniques.
- Path–testing techniques were also the first techniques to come under theoretical scrutiny.
- There is considerable evidence that path testing was independently discovered and used many times in many different places.
- Path testing is most applicable to new software for unit testing. It is a structural technique. It requires complete knowledge of the program's structure (i.e., source code).
- It is most often used by programmers to unit–test their own code.

(c) The Bug Assumption:

- The bug assumption for the path–testing strategies is that something has gone wrong with the software that makes it take a different path than intended.
- As an example, "GOTO X" where "GOTO Y" had been intended. As another example, "IF A is *true* THEN DO X ELSE DO Y", instead of "IF A is *false* THEN . . ."
- We also assume, in path testing, that specifications are correct and achievable, that there are no processing bugs other than those that affect the control flow, and that data are properly defined and accessed.

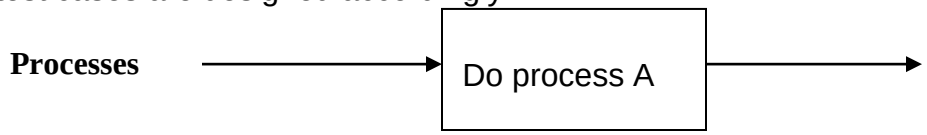
(ii) Control Flowgraphs:

(a) About control flowgraphs:

- The control flowgraph is a graphical representation of a program's control structure.
- A control flowgraph is a form of a flowchart which does not deal with the internal structure of the process rather it shows the data flow and the control flow between the processes.
- It uses the elements process blocks, decisions and junctions.

(i) Process Block

- A process block* is a sequence of program statements uninterrupted by either decisions or junctions.
- Formally, it is a sequence of statements such that if any one statement of the block is executed, then all statements are executed.
- Here once a process block is initiated, every statement within it will be executed.
- Every process has an entry and an exit and consists of a single or series of statements.
- Control flow graph are not concerned with the details of operations in a process block so, the test cases are designed accordingly.

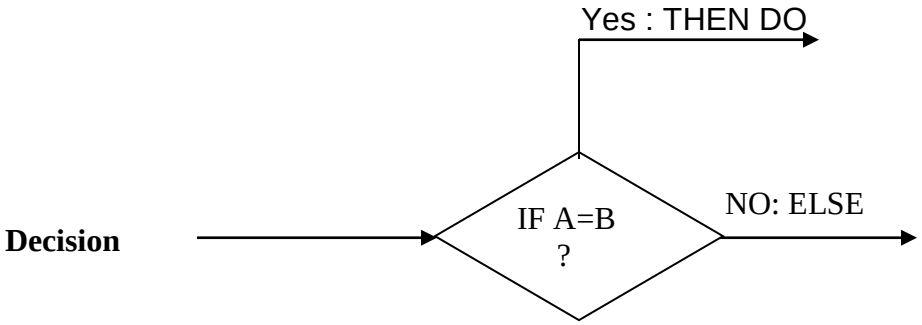


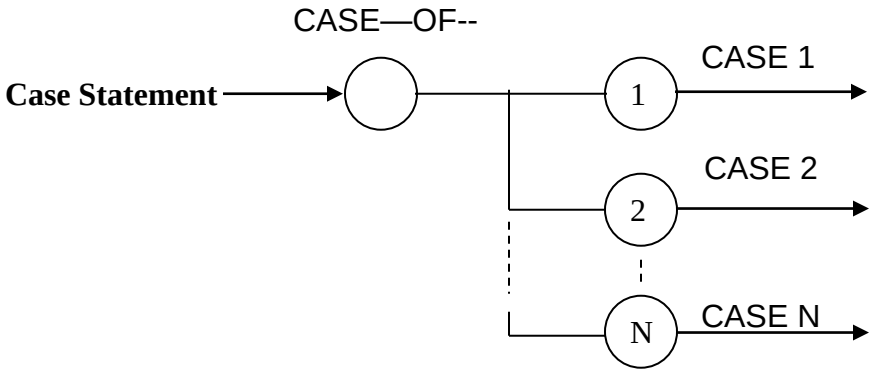
(ii) Decisions and Case Statements:

- A decision is a program point at which the control flow can split.

www.Jntufastupdates.com

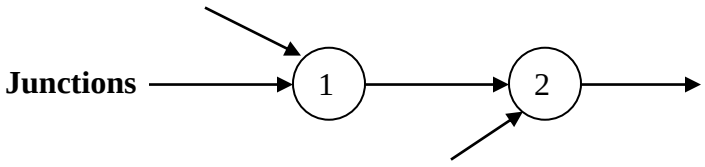
- Machine language conditional branch and conditional skip instructions are examples of decisions.
- The FORTRAN IF and the Pascal IF–THEN–ELSE constructs are decisions, although they also contain processing components.
- While most decisions are two–way or binary**,** some (such as the FORTRAN IF) are three–way branches in control flow.
- The design of test cases is generally easier with two–way branches than with three–way branches, and there are also more powerful test–design tools that can be used.
- Any decision can split the control flow into different way branches.
- This multi way branches can be termed as case statements.
- The designing of test cases for decision and case statements are same.





(iii) Junctions:

- A junction is a point in the program where the control flow can merge.
- That is all the control flows can merge at a point in a program which is known as junction.
- In other words a node with more than one input line is known as junction.
- Examples of junctions are: the target of a jump or skip instruction in assembly language, a label that is the target of a GOTO, the END–IF and CONTINUE statements in FORTRAN, and the Pascal statement labels, END and UNTIL.



www.Jntufastupdates.com

Control flowgraph advantages:

- ❖ Control flowgraph eliminates the occurrence of some problems which results from expanding the visual complexities.
- ❖ Control flowgraph treats all the steps inside a process as a single process entity and shows only data and control flow to and from that entity there by reducing the complexity of structure.
- ❖ Control flowgraphs can be referred to as a modern approach for representation of flows.
- ❖ Control flowgraph gives the precise and clear view of the program structure, the directions of data flow etc.

Control flowgraph disadvantages:

- ❖ Control flowgraph plays an important role in representing the program control structure, but are sparsely available due to the scarcity of control flowgraph generators.
- ❖ The information needed to produce a control flowgraph is not provided by most of the compilers.
- ❖ Although the control flowgraphs are informative, but causes inconveniency while working.
- ❖ Control flowgraph structure is similar to many programming structures and is very difficult to differentiate..

(b) Control Flowgraphs Versus Flowcharts

- ➢ Flowchart is a graph which represents the control structure of the program, as well as the internal structure of each and every process or process block.
- ➢ Control flowgraph is also a graph which represents the control structure of a program, but it excludes the detailed structure of process blocks.
- ➢ All the steps inside a process are shown using flowchart in addition to the control flows, but control flowgraph considers all the steps as a single process entity and shows only the control flows to and from that process entity.
- ➢ Flowchart shows the internal flows of each process so, it is difficult to identify the actual control flows between different processes.
- ➢ Whereas control flowgraphs shows the control and data flow only between processes, there by complexity is reduced.
- ➢ Flowcharts had lost its importance because of the detailed information, it provides which is not in use for process design.
- ➢ We can also use flowchart for representing the control and data flows in a traditional way and control flowgraphs as the modern approach for representation of flows.
- ➢ Flowcharts can easily be drawn manually using available flowchart generators whereas control flowgraph can be drawn difficult.
- ➢ In control flowgraphs, we don't show the details of what is in a process block; indeed, the entire block, no matter how many statements in it, is shown as a single process.
- ➢ In flowcharts, conversely, every part of the process block is drawn: if a process block consists of 100 steps, the flowchart may have 100 boxes.
- ➢ Flowchart has a box to represent each and every process step which is not the case with control flowgraph, only the outline of process block is shown in control flowgraph.

(c) Notational Evolution

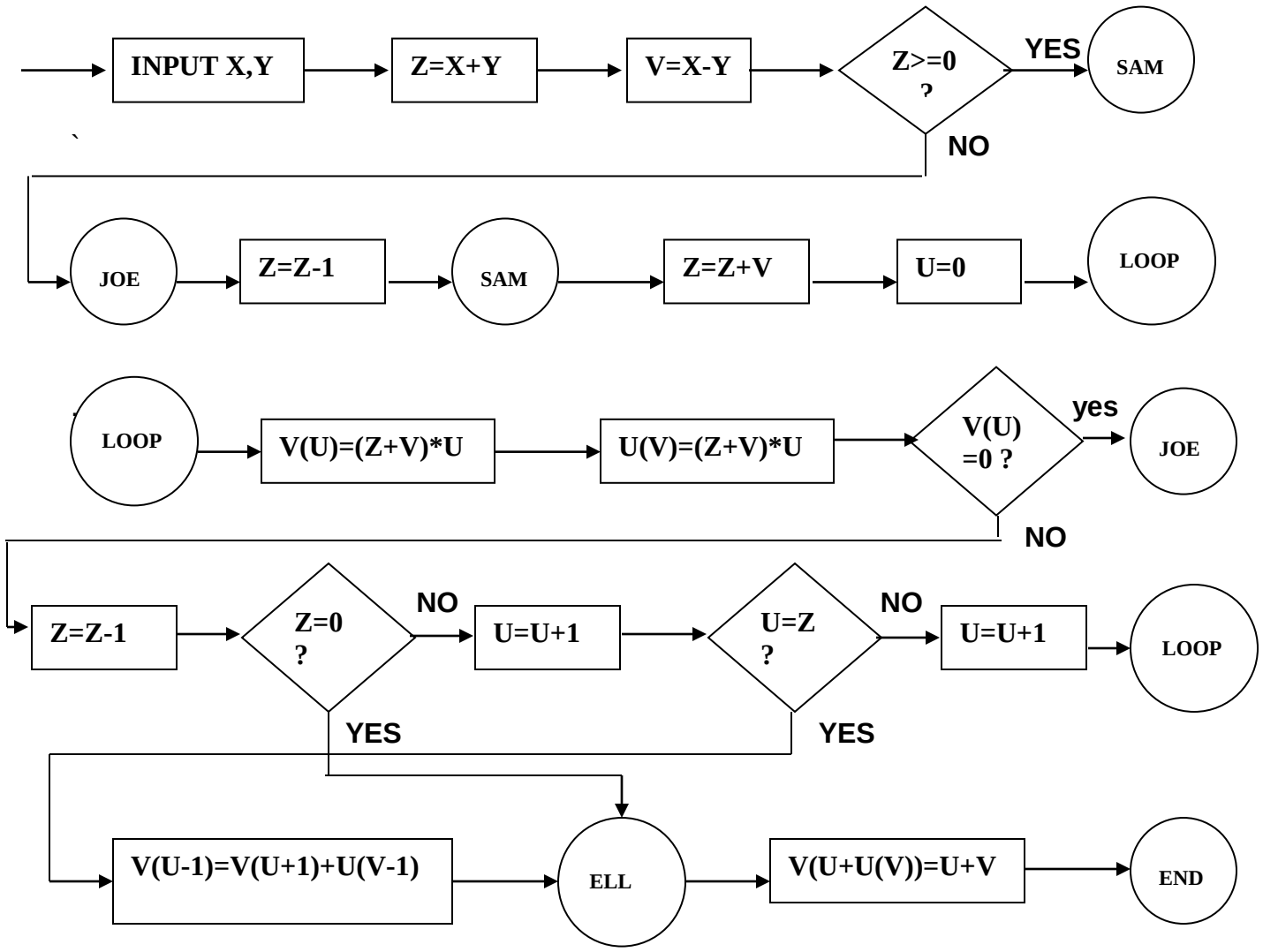
- ➢ The control flowgraph is a simplified representation of the program's structure.
- ➢ To understand its creation and use, we'll go through an example, written in a FORTRAN–like **program design language** (PDL).
- ➢ The code is given below.

www.Jntufastupdates.com

CODE* (PDL)

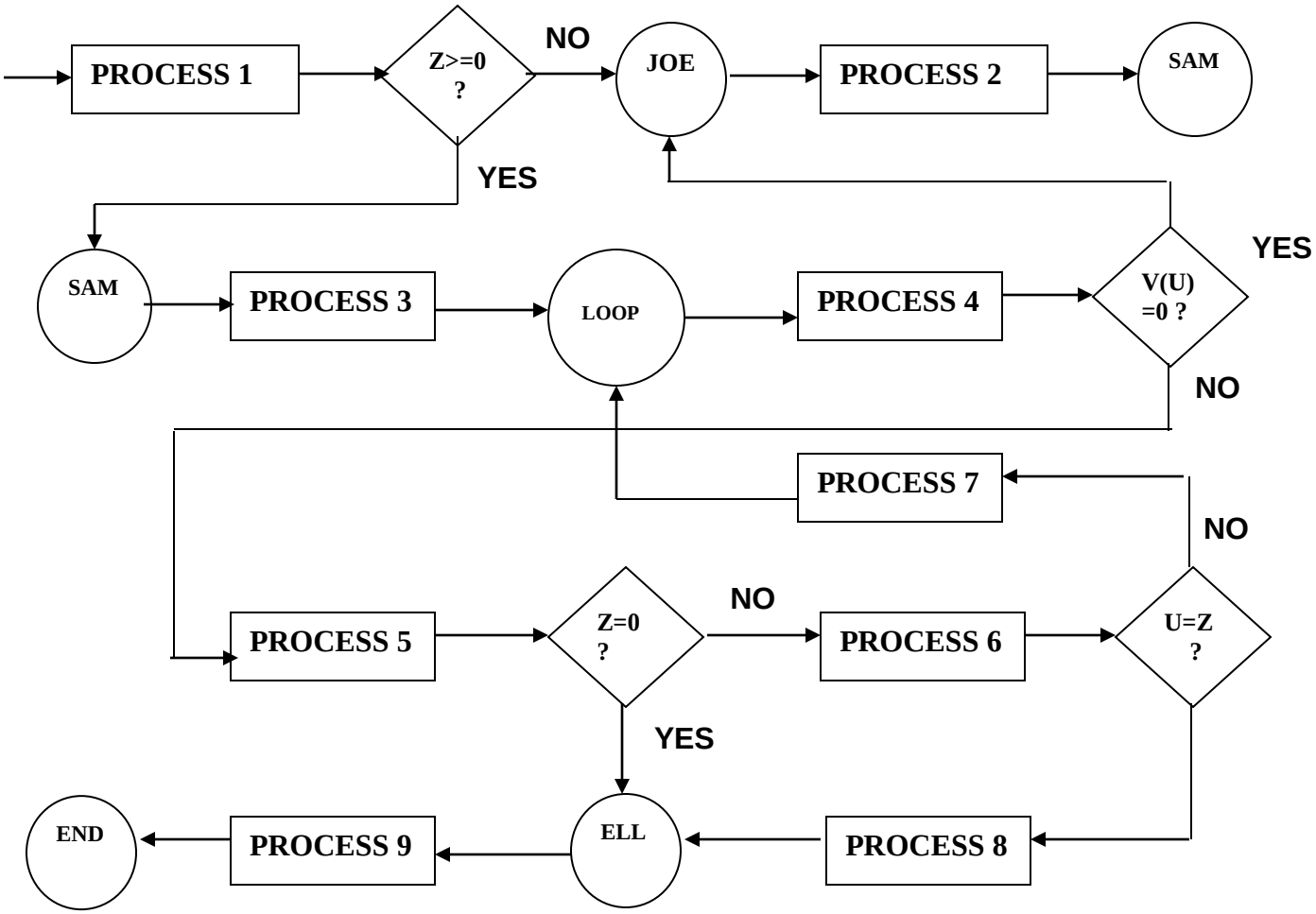
```
INPUT X,Y
Z:=X+Y
V:=X-Y
IF Z>=0 GOTO SAM
JOE:Z:=Z-1
SAM:Z:=Z+V
FOR U=0 TO Z
V(U),U(V):=(Z+V)*V
IF V(U)=0 GOTO JOE
Z:=Z-1
IF Z=0 GOTO ELL
U:=U+1
NEXT U
V(U+1)+U(V-1)
ELL:V(U+U(V)):=U+V
END
```

- One-to-one Flowchart for the above code is given by



www.Jntufastupdates.com

- Control flowgraph for the above example is given by



(d) Flowgraph–Program Correspondence

- A flowgraph is a pictorial representation of a program and not the program itself.
- We can't always associate the parts of a program in a unique way with flowgraph parts because many program structures, such as IF–THEN–ELSE constructs, consist of a combination of decisions, junctions, and processes.
- Furthermore, the translation from a flowgraph element to a statement and vice versa is not always unique.
- A FORTRAN DO has three parts: a decision, an end–point junction, and a process that iterates the DO variable.
- The FORTRAN IF–THEN–ELSE has a decision, a junction, and three processes (including the processing associated with the decision itself).
- Therefore, neither of these statements can be translated into a single flowgraph element.
- Some computers have looping, iterating, and EXECUTE instructions or other instruction options and modes that prevent the direct correspondence between instructions and flowgraph elements.
- Such differences are so familiar to us that we often code without conscious awareness of their existence.
- It is, however, important that the distinction between a program and its flowgraph representation be kept in mind during test design.

www.Jntufastupdates.com

- An improper translation from flowgraph to code during coding can lead to bugs, and an improper translation (in either direction) during test design can lead to missing test cases and consequently, to undiscovered bugs.

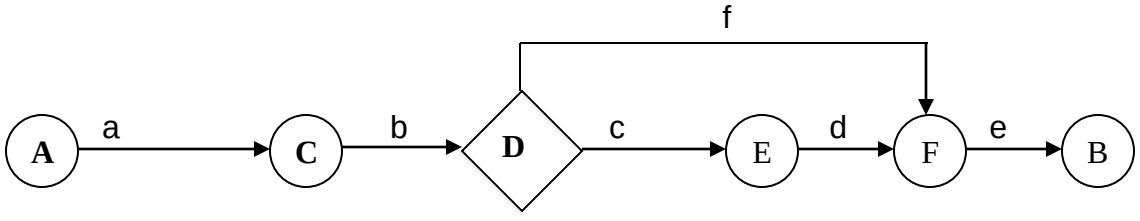
(e) Flowgraph and Flowchart Generation

- The control flowgraph is a simplified version of the earlier flowchart.
- Flowcharts can be (1) hand–drawn by the programmer, (2) automatically produced by a flowcharting program based on a mechanical analysis of the source code, or (3) semiautomatically produced by a flowcharting program based in part on structural analysis of the source code and in part on directions given by the programmer.
- The semiautomatic flowchart is most common with assembly language source code.
- A flowcharting package that provides controls over how statements are mapped into process boxes can be used to produce a flowchart that is reasonably close to the control flowgraph.
- You do this by starting process boxes just after any decision or GOTO target and ending them just before branches or GOTOs.

(iii) Path Testing:

(a) Paths, Nodes, and Links

- A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit.
- A path may go through several junctions, processes, or decisions, one or more times.
- Every path consists of a set of processes known as links.
- A direct connection between two nodes is also called a “process”.
- Links can be denoted by an arrow and can represented by the lower case letters.
- A path segment is a succession of consecutive links that belongs to some path.
- The length of a path is measured by the number of links in it and not by the number of instructions or statements executed along the path.
- An alternative way to measure the length of a path is by the number of nodes traversed.
- Nodes are mainly denoted by small circles. A node which has more than one input link is known as a junction, and a node which has more than one output link is referred to as a decision.
- Nodes can be labeled by an alphabets or numbers.
- If programs are assumed to have an entry and an exit node, then the number of links traversed is just one less than the number of nodes traversed.
- Because links are named by the pair of nodes they join, the name of a path is the name of the nodes along the path.



- There are two different paths from an entry (A) to an exit (B), they are ACDEFB and ACDFB respectively. In these two ACDFB is the shortest path between an entry and an exit.
- In all the nodes (A,B,C,D,E,F), D is the decision which has 2 output links, and F is a junction which has two input links.
- The a,b,c,d,e,f are all the available links.

www.Jntufastupdates.com

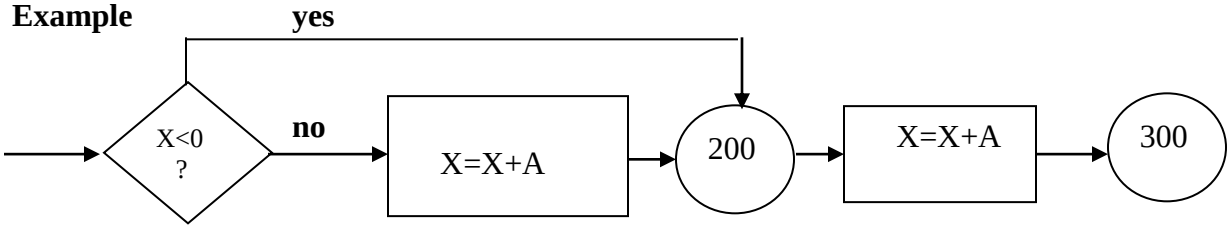
(b) Multi–Entry/Multi–Exit Routines

- Multi-entry means, multiple entry points and multi-exit refers to multiple exit points.
- Generally all routines and programs have a single entry and a single exit.
- There are certain situations in which it is appropriate to change the routine and choose an alternate way to normal control structure.
- There is no justifiable reason which forces you to change the routine.
- You may want to choose an alternate routine, when an illegitimate condition occur and will damage the system's data, if that path is continued further.
- The other reason might be the occurrence of several fluctuations during the processing of same path.
- Hence changing of route is advantageous in such situations by placing an entry point in a routine which sends the flow to appropriate location.
- If a routine can have several different kinds of outcomes, then an exit parameter should be used.
- As there is no direct connection between entry and exit so control flow is managed by reviewing the parameter values of entry and exit in both directions of the routine.
- The main drawback of multi-entry and multi-exit routines is that all the test cases are difficult to cover because the control flow between various processes can't be determined easily due to multiple entry and exit points.

(c) Fundamental Path Selection Criteria

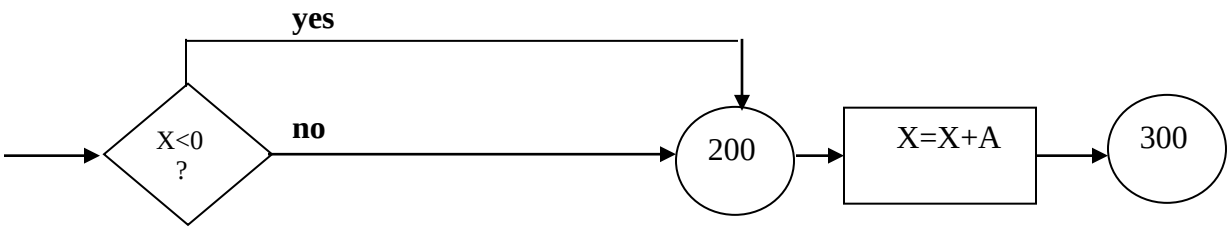
- There are many paths between the entry and exit of a typical routine.
- Path selection mainly deals with the selection of an optimal path between its entry and exit.
- If a routine contains decisions or loops inside it, then there will be more number of paths.
- For example every decision doubles the number of potential paths, and every loop multiplies the number of potential paths by the number of different iteration values possible for the loop.
- If a routine has one loop, each pass through that loop (once, twice, three times, and so on) constitutes a different path through the routine, even though the same code is traversed each time.
- A lavish test approach might consist of testing all paths, but that would not be a complete test, because a bug could create unwanted paths or make mandatory paths unexecutable.
- Complete testing involves
    1. Exercise every path from entry to exit.
    2. Exercise every statement or instruction at least once.
    3. Exercise every branch and case statement, in each direction, at least once.
- If prescription 1 is followed then prescriptions 2 and 3 are automatically followed, but prescription 1 is impractical for most routines.

Example



- For X is less than zero, the output is X+A while X is greater than or equal to zero the output is X+2A because decision doubles the number of paths.
- If we execute all the statements but not the branches in the above example we would get the bug.

www.Jntufastupdates.com



- For the above example if X is less than zero the output is correct, but for any positive value the output will be X=X+A which is wrong.
- A static analysis that is an analysis based on examining the source code or structure cannot determine whether a piece of code is or is not reachable.
- Only a dynamic analysis that is an analysis based on the code's behavior while running can determine whether code is reachable or not.

(d) Path–Testing Criteria

- There are three path testing criteria.
- The notation **P_1, P_2, …, P_∞** should alert you to the fact that there is an infinite number of such strategies, but even that's insufficient to exhaust testing.

(i) Path Testing (P_∞):

- Path testing deals with the execution of paths if we have tested all the available control flow paths we have achieved 100% path coverage which is mostly impossible.
- The word coverage refers to combinational value of 100% statement coverage and branch coverage.
- It is represented as (C_1 +C_2), where C_1 refers to statement coverage and C_2 refers to branch coverage.
- Hence this type of coverage is also referred as completed coverage.

(ii) Statement Testing (P_1):

- Statement testing deals with the execution of all the statements inside a program at least once.
- The process of performing possible tests in order to achieve statement testing is called 100% statement coverage.
- Statement coverage is also known as 100% node coverage.
- We denote this by C_1.

(iii) Branch Testing (P_2):

- Branch testing deals with the execution of all the branches at least once in the program.
- The process of performing possible tests in order to achieve branch testing is called 100% branch coverage.
- Branch coverage is also known as link coverage.
- We denote branch coverage by C_2.

(e) Common Sense and Strategies

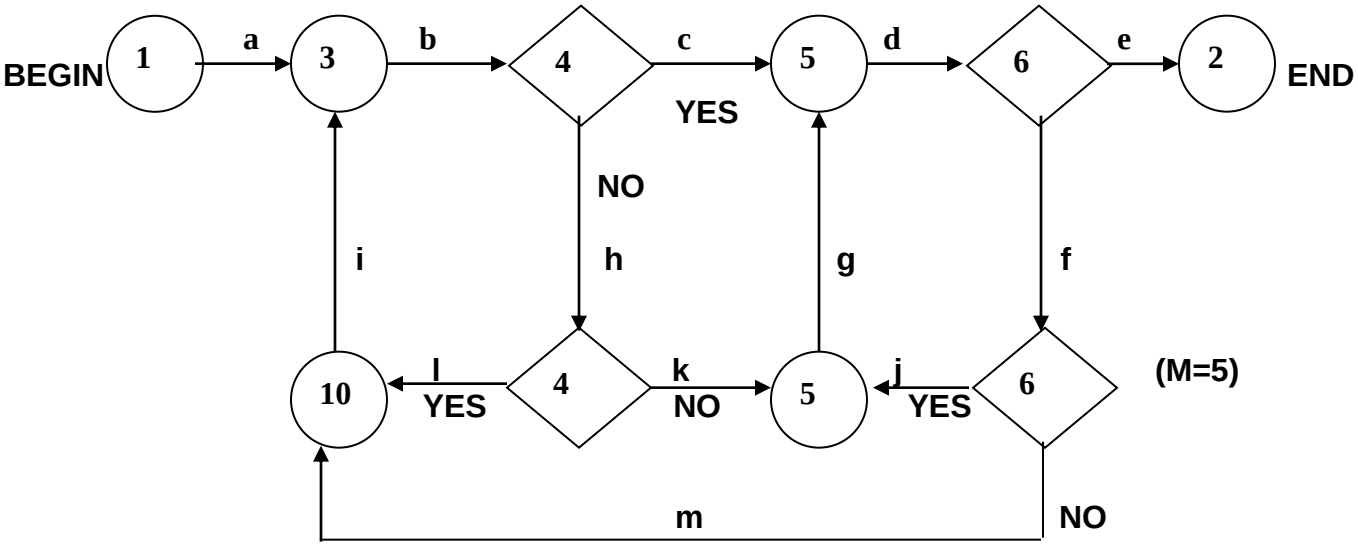
- Branch and statement coverage are accepted today as the minimum mandatory testing requirement.
- Statement coverage is established as a minimum testing requirement in the IEEE unit test standard.
- Statement and branch coverage have also been used for more than two decades as minimum mandatory unit test requirements for new code at IBM and other major computer and software companies.
- The justification for insisting on statement and branch coverage isn't based on theory but on common sense.

www.Jntufastupdates.com

- Also with our common sense, we can classify code with much probability of having bugs and code with less probability, separately.
- Keeping the code with lower probability of bugs untested may not be wrong because this code will probably have less or no bugs.
- The code with higher probability of bugs is tested thoroughly to remove all the bugs. Even if we are skipping some part of this code it will not create a big one because this portion is tested many times in the entire testing process.

(f) Which Paths

- We must pick enough paths to achieve C1 + C2.
- It's better to take many simple paths than a few complicated paths.
- An example of path selection is given below.



- As we trace the paths, create a table that shows the paths, the coverage status of each process, and each decision.
- Start at the beginning and take the most obvious path to the exit—it typically corresponds to the normal path.
- The most obvious path in above figure is (1,3,4,5,6,2), if we name it by nodes, or *abcde* if we name it by links.
- Then take the next most obvious path, *abhkgde.* All other paths in this example lead to loops.
- Take a simple loop first—building, if possible, on a previous path, such as *abhlibcde.*
- Then take another loop, *abcdfjgde.* And finally, *abcdfmibcde.*
- The above paths lead to the following table.

PATHS	DECISIONS				PROCESS-LINK
	4	6	7	9	a b c d e f g h i j k l m
abcde	YES	YES			
abhkgde	NO	YES		NO	
abhlibcde	NO,YES	YES		YES	
abcdfjgde	YES	NO,YES	YES		
abcdfmibcde	YES	NO,YES	NO		

www.Jntufastupdates.com

- After you have traced a covering path set on the master sheet and filled in the table for every path, check the following.
  1. Does every decision have a YES and a NO in its column? (C2)
  2. Has every case of all case statements been marked? (C2)
  3. Is every three–way branch (less, equal, greater) covered? (C2)
  4. Is every link (process) covered at least once? (C1)*
- Select successive paths as small variations of previous paths.
- Try to chance only one thing at a time that is only one decision's outcome if possible.
- It is better to have several paths, each differing by only one thing, than one path that covers more but along which several things change.
- The abcd segment in the above example is common to many paths

(g) Path selection rules:

(a) Selection of simple path:

- Select an entry/exit path which is simple and assign selected path with either nodes or links.

(b) Selection of additional paths:

- After selection of simple path, the next obvious path is selected.
- This method of selecting successive paths can be done by making small changes to the previous paths.
- Unlike long and complex paths, various small paths are selected which involves gradual variations.
- In path selection Select paths with no loops, Select shorter paths and Select simple and sensible paths.

(c) Selection of Non-functional Sensible paths:

- Select additional paths in such a way that coverage is achieved through the non-functional sensible paths.
- This type of selection should be preferred only if coverage is essential.

(d) Meet the user Requirements:

- All possible paths should be selected in order to meet the requirements of a user.
- This process is repeated until statement (C_1) and branch (C_2) coverages are achieved.
- During this process checking is carried out on each and every decision statement, branch covering, link covering etc.
- Statement coverage and branch coverage ($C_1 + C_2$) does not support loop-related bugs.

(iv) Loops:

(a) The Kinds of Loops

- There are three kinds of loops.
- They are nested, concatenated and horrible loops.

(i) Cases for a Single Loop:

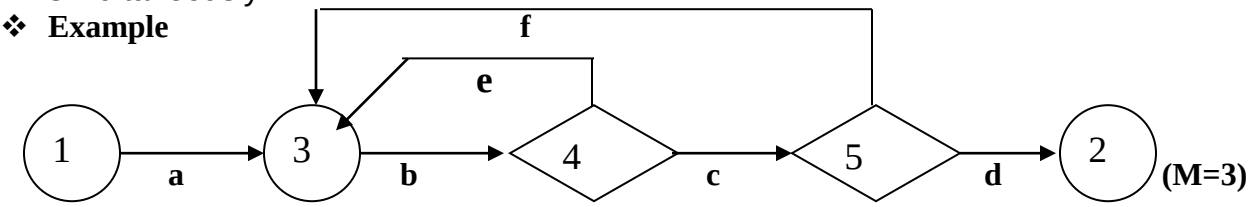
- A single loop can be covered with two cases: looping and not looping.
- The different cases for a single loop are
- Case 1—Single Loop, Zero Minimum, N Maximum, No Excluded Values.
- Case 2—Single Loop, Nonzero Minimum, No Excluded Values.
- Case 3—Single Loops with Excluded Values.

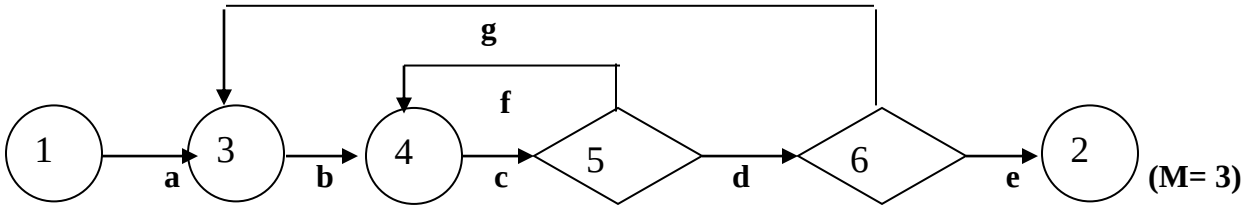
. **(ii) Nested Loops:**

- The nested loops are quite complicated i.e. a loop within another loop is known as nested loop.
- It is very expensive to test the path which contains nested loop because of its complexity.

www.Jntufastupdates.com

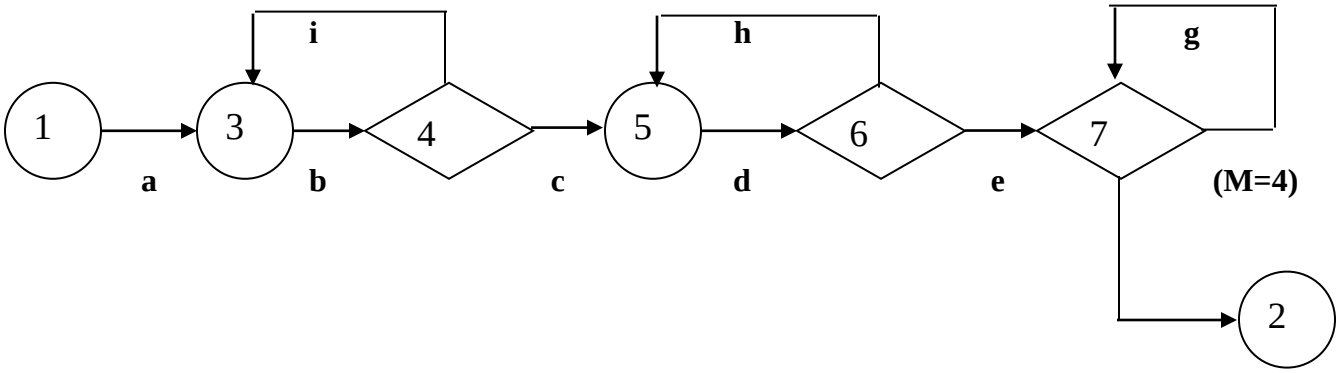
- If you had five tests for one loop, a pair of nested loops would require 25 tests, and three nested loops would require125.
- To overcome this complexity we have to follow some steps.
  1. Start at the innermost loop. Set all the outer loops to their minimum values.
  2. Test the minimum, minimum + 1, typical, maximum – 1, and maximum for the innermost loop, while holding the outer loops at their minimum–iteration–parameter values. Expand the tests as required for out–of–range and excluded values.
  3. If you've done the outermost loop, GOTO step 5, ELSE move out one loop and set it up as in step 2—with all other loops set to typical values.
  4. Continue outward in this manner until all loops have been covered.
  5. Do the five cases for all loops in the nest simultaneously.
- This procedure works out to twelve tests for a pair of nested loops, sixteen for three nested loops, and nineteen for four nested loops.
- Practicality may prevent testing in which all loops achieve their maximum values simultaneously**.**
- **Example**





(iii) Concatenated Loops:

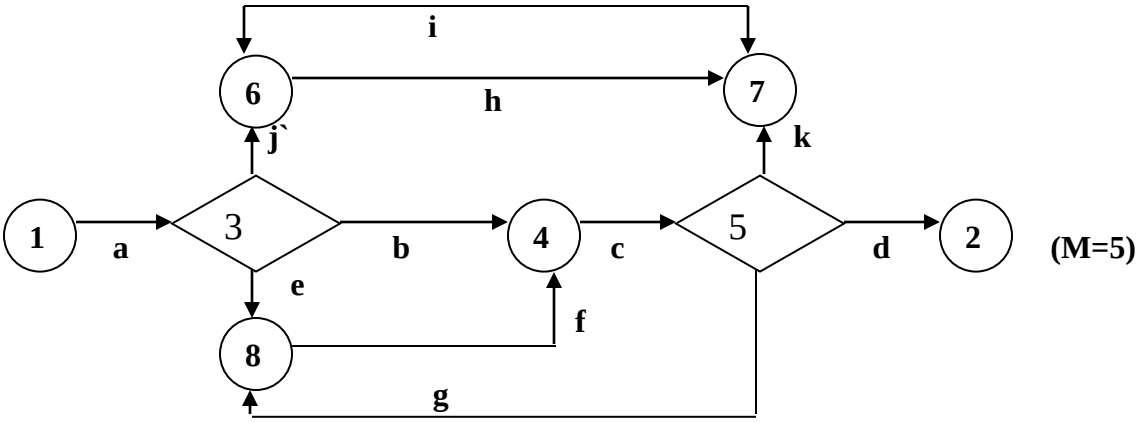
- Concatenated loops are the loops which reside one beside the other on the same path.
- In other words, when there exits two adjacent loops on the same path such that, an exit of one loop serves as an entry point for the other loop, then the loops are said to be concatenated.
- If the loops cannot be on the same path, then they are not concatenated and can be treated as individual loops.
- If one loop's repetition value depends on the repetition value of other loop and both lie on same path they can be termed as nested loops.



www.Jntufastupdates.com

(iv) Horrible Loops:

- If the loops cannot be on the same path, then they are not concatenated and can be treated as individual loops.
- Horrible loops are the complexed of all the three loops. This complex structure of horrible loops makes it very difficult to be tested.
- The design of test cases for horrible loops is indefinite and is too many to execute. Hence horrible loops must be avoided.



(f) Loop–Testing Time

- Any kind of loop can lead to long testing time, especially if all the extreme value cases are to be attempted (MAX – 1, MAX, MAX + 1).
- This situation is obviously worse for nested and dependent concatenated loops.
- In the context of real testing, most tests take a fraction of a second to execute, and even deeply nested loops can be tested in seconds or minutes.
- The unreasonably long test execution times (i.e., hours or centuries) could indicate bugs in the software or the specification.
- Consider nested loops in which testing the combination of extreme values leads to long test times. You have several options:

  1. Show that the combined execution time results from an unreasonable or incorrect specification. Fix the specification.

  2. Prove that although the combined extreme cases are hypothetically possible, they are not possible in the real world. That is, the combined extreme cases cannot occur.

  3. Put in limits or checks that prevent the combined extreme cases. Then you have to test the software that implements such safety measures.

  4. Test with the extreme–value combinations, but use different numbers.

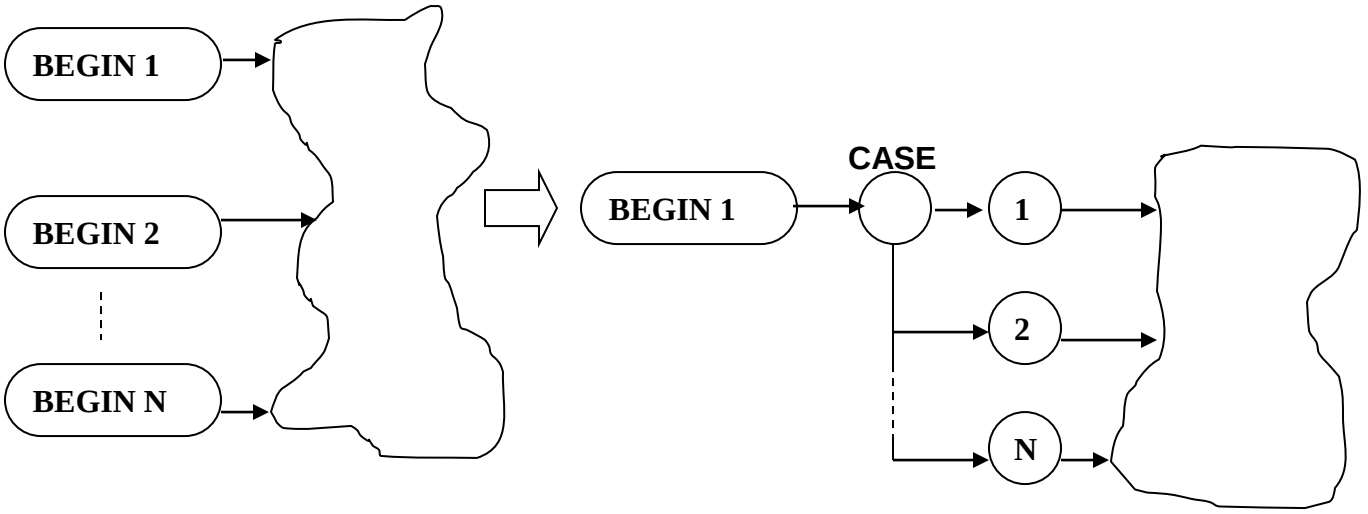
(v) More on Testing Multi–Entry/Multi–Exit Routines:

(a) A Weak Approach

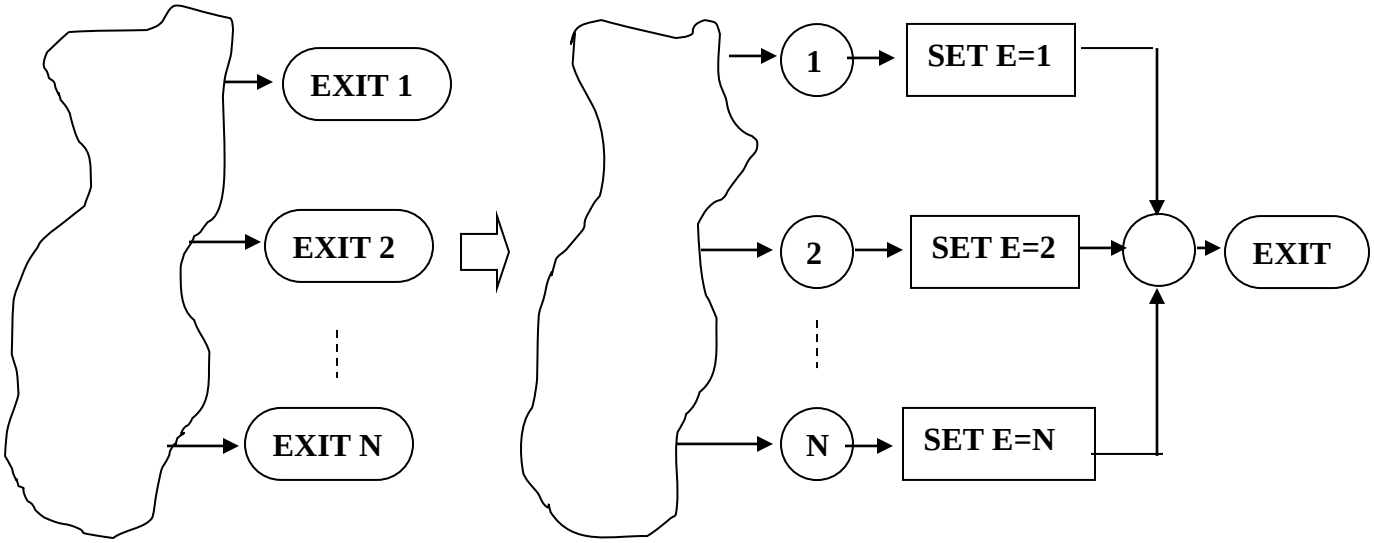
- To test the program with multi-entry and multi-exit routines are as follows.
- First, built the fictitious single entry routine and fictitious exit routine with fictitious case statements and processes respectively.
- Secondly concentrate on fictitious common junction. This fictitious code will help you to organize the test case design for multi-entry and multi-exit routines.
- This technique involves a lot of extra work because you must examine the cross-reference listings to find all references to the labels that correspond to the multiple entries.
- All the designers of routines should know how they want to exit, but it's difficult to control an entry that can be initiated by many other programmers.
- The Conversion of Multi-exit or Multi-entry routines is given by the following figures.

www.Jntufastupdates.com

(i) A Multi-entry routine is converted to an equivalent single-entry routine with an entry parameter and a controlling case statement.



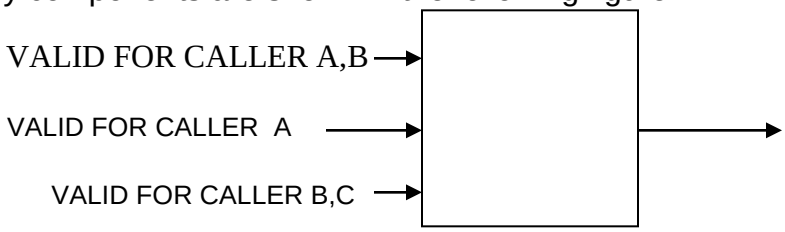
(ii) A Multi-exit routine is converted to an equivalent single-exit routine with an exit parameter.



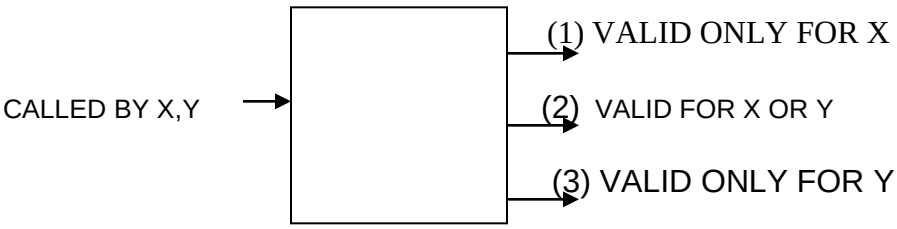
(b) The Integration Testing Issue

- Treating the multi–entry/multi–exit routine by using a fictional entry case statement and a fictional exit parameter is a weak approach because it does not solve the essential testing problem.
- The essential problem is an integration testing issue and has to do with paths within called components.
- For example we have a multi–entry routine with three entrances and three different callers. The first entrance is valid for callers A and B, the second is valid only for caller A, and the third is valid for callers B and C.
- Just testing the entrances doesn't do the job because in integration testing it's the interface, the validity of the call that must be established.
- In integration testing, we would have to do at least two tests for the A and B callers—one for each of their entrances. Note also that, in general, during unit testing we have no idea who the callers are to be.

- Multi-entry components are shown in the following figure.



- Multi-exit routine is shown in the following figure.



- The above multi-exit routine has three exits labeled 1, 2, and 3.
- It can be called by components X or Y. Exits 1 and 2 are valid for the X calls and 2 and 3 are valid for the Y calls.
- Component testing must not only confirm that exits 1 and 2 are taken for the X calls, but that there are no paths for the X calls that lead to exit 3—and similarly for exit 1 and the Y calls.
- But when we are doing unit tests, we do not know who will call this routine with what restrictions. As for the multi–entry routine, we must establish the validity of the exit for every caller.
- Note that we must confirm that not only does the caller take the expected exit, but also that there is no way for the caller to return via the wrong exit.
- When we combine the multi–entry routine with the multi–exit routine, we see that in integration testing we must examine every combination of entry and exit for every caller.
- Since we don't know, during unit design, which combinations will or will not be valid, unit testing must at least treat each such combination as if it were a separate routine.
- Thus, a routine with three entrances and four exits results in twelve routines' worth of unit testing.
- Integration testing is made more complicated in proportion to the number of exits, or fourfold.

(c) The Theory and Tools Issue

- A well-formed software is a software, which has single entry and single exit with a rigid structure.
- Software which does not have this property is called ill-formed.
- The other characteristic of well–formed software is to insist on strict structuring in addition to single–entry/single–exit.
- An assumption that multi-entry and multi-exit routines can't occur in testing theory has been followed.
- Such multi-entry and multi-exit routines come under ill formed routines.
- Before applying the theoretical rules, it is better to confirm whether the software is well-formed or ill-formed.
- Ill-formed (multi-entry and multi-exit) software does not have any structure so, testing of one component does not guarantee the test results for another.
- Even test generators may not be able to generate test cases for ill-formed software.

www.Jntufastupdates.com

(d) Strategy Summary

The proper way to test multi–entry or multi–exit routines is:

1. Get rid of them.
2. Completely control those you can't get rid of.
3. Supply the imaginary input case statements, and exit parameters to control flowgraph in order to design test cases for these routines.
4. Do stronger unit testing by treating each and every entry/exit combination considered as a completely different routine.
5. Multi-entry and multi-exit routines are assumed to be more unusual and dangerous so, integration testing is performed with more efforts and concentration.
6. Be sure you understand that test cases designed based on your assumption are suitable for multi-entry and multi-exit routines.

(vi) Effectiveness of Path Testing:

(a) Effectiveness and Limitations

- Unit testing is comparatively stronger than path testing which is stronger than statement and branch testing.
- Unit testing can catch up to 65% of bugs in overall structure, this implies that path testing captures approximately 35% of bugs in the overall structure as per statistical reports.
- Path testing is more effective for unstructured than for structured software.
- Apart from effectiveness, path testing also has certain limitations.

1. Planning to cover does not mean you will cover. Path testing may not cover if you have bugs.
2. Path testing has to be combined with other methods to improve the overall performance in terms of percentage.
3. Unit level path testing does not concentrate on integration issues which may result in interface errors.
4. Database and data–flow errors may not be caught.
5. Illegitimate functions or missed functions cannot be identified during path testing.
6. Not all initialization errors are caught by path testing.
7. Specification errors can't be caught.

(b) A Lot of Work?

- Path testing involves a lot of work that is.
  - Development of control flowgraph.
  - Choosing a route that can cover all the paths, decisions and junctions in a flowgraph.
  - Determining the input values which satisfies each path expression for selecting the respective paths.
  - Writing test cases for loops.
- The statistics indicate that you will spend half of your time testing and debugging—presumably that time includes the time required to design and document test cases.
- Furthermore, the act of careful, complete, systematic, test design will catch as many bugs as the act of testing.
- It is worth that, the test design process, at all levels, and is at least as effective at catching bugs as is running the test designed by that process.

(c)More on How to Do It

- To trace the path from your code, you need a marking pen, a copying machine and a source code list.
- At first you may want to create the control flowgraph and use that as a basis for test design, but as you gain experience with practice, you'll find that you can select the paths directly on the source code without bothering to draw the control flowgraph.

www.Jntufastupdates.com

- If you can path trace through code for debugging purposes then you can just as easily trace through code for test design purposes.
- And if you can't trace a path through code, are you a programmer then you do it with code almost the same way as you would with a pictorial control flowgraph.
- Choose your path and mark only the executed statements in case of "if-then-else statements".
- Also mark all the ongoing statements on a path with a marking pen by doing this you will accomplish C_1.
- Place or draw your marking on a master sheet with the marking pen (yellow).
- For achieving C_2 we need to identify and mark all the statements irrespective of its execution even for the if-then-else statements.

(vii) Variations:

- Branch and statement coverage as basic testing criteria are well established as effective, reasonable, and easy to implement.
- There are two main classes of variations:
  1. Strategies between P_2 and total path testing.
  2. Strategies weaker than P_1 or P_2.
- The stronger strategies typically require more complicated path selection criteria, most of which are impractical for human test design.
- Typically, the strategy has been embedded in a tool that either selects a covering set of paths based on the strategy or helps the programmer to do so.
- While research can show that strategy A is stronger than B in the sense that all tests generated by B are included in those generated by A, it is much more difficult to ascertain cost–effectiveness.
- For example, if strategy A takes 100 times as many cases to satisfy as B, the effectiveness of A would depend on the probability that there are bugs of the type caught by A and not by B.
- We have almost no such statistics and therefore we know very little about the pragmatic effectiveness of this class of variations.
- As an example of how we can build a family of path–testing strategies, consider a family in which we construct paths out of segments that traverse one, two, or three nodes or more.
- If we build all paths out single–node segments P_1 (hardly to be called a "path," then we have achieved C1. If we use two–node segments (e.g., links = P_2) to construct paths, we achieve C2.

(2) Predicates, Path Predicates, and Achievable Paths:

(i) General

- Selecting a path does not mean that it is achievable.
- If all decisions are based on variables whose values are independent of the processing and of one another, then all combinations of decision outcomes are possible (2^n outcomes for n binary decisions) and all paths are achievable: in general, this is not so.
- Every selected path leads to an associated boolean expression, called the path predicate expression, which characterizes the input values (if any) that will cause that path to be traversed.

(ii) Predicates

(a) Definition and Examples

- The direction taken at a decision depends on the value of decision variables.
- For binary decisions, decision processing ultimately results in the evaluation of a logical (i.e., boolean) function whose outcome is either TRUE or FALSE.

www.Jntufastupdates.com

- Although the function evaluated at the decision can be numeric or alphanumeric, when the decision is made it is based on a logical function's truth value.
- The logical function evaluated at a decision is called a predicate.
- That is Predicate is a function which is logically executed during the decision processing.
- The result of this function decides the direction of flow.

Example

- "A is greater than zero," "the fifth character has a numerical value of 31," "X is either negative or equal to 10," "$X + Y = 3Z^2 – 44$," "Flag 21 is set.".
- Every path corresponds to a succession of TRUE/FALSE values for the predicates traversed on that path.
- As an example:
  " 'X is greater than zero' is TRUE."
  AND
  " '$X + Y = 3Z^2 – 44$' is FALSE."
  AND
  " 'W is either negative or equal to 10' is TRUE."

- is a sequence of predicates whose truth values will cause the routine to take a specific path. A predicate associated with a path is called a path predicate**.**

(b) Multiway Branches

- The path taken through a multiway branch such as computed GOTO's (FORTRAN), case statements (Pascal), or jump tables (assembly language) cannot be directly expressed in TRUE/FALSE terms.
- Although it is possible to describe such alternatives by using multivalued logic, an easier expedient is to express multiway branches as an equivalent set of IF . . . THEN . . . ELSE statements.
- For example, a three–way case statement can be written as:
  IF case=1 DO A1 ELSE
  (IF case=2 DO A2 ELSE DO A3 ENDIF) ENDIF
- The translation is not unique because there are many ways to create a tree of IF . . . THEN . . . ELSE statements that simulates the multiway branch.
- We treat multiway branches this way as an analytical convenience in order to talk about testing.
- we don't replace multiway branches with nested IF's just to test them.

(c)Inputs

- In testing, the word input is not restricted to direct inputs, such as variables in a subroutine call, but includes all data objects referenced by the routine whose values are fixed prior to entering it.
- for example, inputs in a calling sequence, objects in a data structure, values left in a register.
- Although inputs may be numerical, set members, boolean, integers, strings, or virtually any combination of object types, we can talk about data as if they are numbers.

(iii) Predicate Expressions

(a) Predicate Interpretation

- Predicate interpretation refers to the process of expressing the predicate in terms of the given input vector by performing various symbolic replacement of operations.
- For example if X_1 and X_2 are inputs, the predicate might be "$X_1 + X_2 > 0$".
- Now let the value of X_2 be given using another predicate as X_2:=Y+5
- The substitution of X_2 value in the first predicate gives you another predicate which is X+Y+5 > 0. This process is known as predicate interpretation.

www.Jntufastupdates.com

- The path predicates are the specific form of the predicates of the decisions along the selected path after interpretation.

(b) Independence and Correlation of Variables and Predicates

- The path predicates take on truth values (TRUE/FALSE) based on the values of input variables, either directly (interpretation is not required) or indirectly (interpretation is required).
- If a variable's value does not change as a result of processing, that variable is independent of the processing.
- Conversely, if the variable's value can change as a result of the processing the *variable* is process dependent.
- Similarly, a *predicate* whose truth value can change as a result of the processing is said to be process dependent and one whose truth value does not change as a result of the processing is process independent.
- Process dependence of a predicate does not always follow from dependence of the input variables on which that predicate is based.
- For example, the input variables are X and Y and the predicate is "X + Y = 10".
- The processing increments X and decrements Y.
- Although the numerical values of X and Y are process dependent, the predicate "X + Y = 10" is process independent.
- Variables, whether process dependent or independent, may be correlated to one another.
- Two variables are correlated if every combination of their values cannot be independently specified.
- Variables whose values can be specified independently without restriction are uncorrelated.
- By analogy, a pair of predicates whose outcomes depend on one or more variables in common (whether or not those variables are correlated) are said to be correlated predicates.

(c) Path Predicate Expressions

- Path predicate expressions are the collection of expressions that must be fulfilled in order to achieve the desired path.
- This collection of expressions is satisfied based on input values provided.
- These input values must meet all the expressions. If all the expressions are met then the path is chosen else the path is rejected.
- This is shown by means of an example

  X_1 =18

  X_2 +5 X_3 +2>0

  X_4 – X_2 >=10 X_3

  Let the input values of X_2, X_3, X_4 be 2,1,12 respectively.

  Substituting the values in above predicates, we get

  X_1 =18

  X_2 +5 X_3 +2=2+5*1+2=9 >0

  X_4 – X_2 >=10 X_3 i.e. 12-2 >=10(1) i.e. 10 >=10
- All the conditions appear to be correct as per the values so this path can be chosen.

(iv) Predicate Coverage

(a) Compound Predicates

- Most programming languages permit compound predicates at decisions—that is, predicates of the form A .OR. B or A .AND. B. and more complicated boolean expressions.
- The branch taken at such decisions is determined by the truth value of the entire boolean expression.
- Simply the compound predicate is the combination of two predicates.

- Even if a given decision's predicate is not compound, it may become compound after interpretation because interpretation may require us to carry forward a compound term.

(b) Predicate Coverage

- Predicate coverage is the process of testing all the truth values related to a specific path in all the possible ways.
- If all the values are tested in all possible directions then we can say that 100% predicate coverage is achieved which needs lots of efforts.
- Predicate coverage is slightly comparable to path coverage and is much powerful than the branch coverage.
- If we are using a compound predicate then predicate coverage involves testing of both the predicates in any order.

(v) Testing Blindness

(a) The Problem

- Blindness is a situation which results in the correct path via wrong route unintentionally.
- Testing blindness is a pathological situation in which the desired path is achieved for the wrong reason.
- It can occur because of the interaction of two or more statements that makes the buggy predicate "work" despite its bug and because of an unfortunate selection of input values that does not reveal the situation.
- There are three kinds of predicate blindness: assignment blindness, equality blindness, and self–blindness

(b) Assignment Blindness

- Assignment blindness comes into consideration when both the predicates irrespective of their correctness are satisfied by a value assigned to the assignment statement.
- Assignment blindness may also lead to wrong path selection.

Correct	***Buggy* (Incorrect)**
X := 7	X := 7
.....	.....
IF Y > 0 THEN	IF X + Y > 0 THEN

- If the test case sets Y := 1 the desired path is taken in either case, but there is still a bug.
- Some other path that leads to the same predicate could have a different assignment value for X, so the wrong path would be taken because of the error in the predicate.

(c)Equality Blindness

- Equality blindness occurs when the path selected by a prior predicate results in a value that works both for the correct and buggy predicate.

Correct	***Buggy***
IF Y = 2 THEN. . .	IF Y = 2 THEN. . .
.....	.....
IF X + Y > 3 THEN. . .	IF X > 1 THEN. . .

- The first predicate (IF Y = 2) forces the rest of the path, so that for any positive value of X, the path taken at the second predicate will be the same for the correct and buggy versions.

(d) Self–Blindness

- Self–blindness occurs when the buggy predicate is a multiple of the correct predicate and as a result is indistinguishable along that path.

www.Jntufastupdates.com

Correct	*Buggy*
X := A	X := A
.....	.....
IF X – 1 > 0 THEN...	IF X + A – 2 > 0 THEN

- The assignment (X := A) makes the predicates multiples of each other (for example, A – 1 > 0 and 2A – 2 > 0), so the direction taken is the same for the correct and buggy version.

(3) Path Sensitizing:

(i) Review :Achievable and Unachievable Paths.

- In order to accomplish test completeness (i.e. C_1 or C_2) for sufficient paths the procedure is as follows.
  1. Extract the programs control flowgraph and select a set of tentative covering paths.
  2. After path selection, determine the predicates for all paths that exist in the selected path set. This makes the basic nature of each predicate compound.
  3. In order to achieve a Boolean expression, the path is traced by multiplying the individual compound predicates. For instance, let the compound predicate be
     (A+BC)(D+E)(FGH)(IJ)(K)(L)
     where the terms in the parentheses are the compound predicates met at each decision along the path and each letter (A,B,…) stands for simple predicates.
  4. The Boolean expression is converted into SOP (Sum of Products) format by multiplying the terms in the given expression as follows
     ADFGHIJKL + AEFGHIJKL + BCDFGHIJKL + BCEFGHIJKL
- Path predicate expressions are the collection of expressions that must be fulfilled in order to achieve the desired path.
- If all the expressions are met then the path is achievable else the path is not achievable.
- The act of finding a set of solutions to the path predicate expression is called path sensitization.

(ii) Pragmatic Observations

- The purpose of the above discussion has been to explore the sensitization issues and to provide insight into tools that help us sensitize paths.
- If in practice you really had to do the above in the manner indicated then test design would be a difficult procedure suitable only to the mathematically inclined.
- It doesn't go that way in practice: it's much easier

(iii) Heuristic Procedures for Sensitizing Paths

- Heuristic procedures are the most optimistic ways for sensitizing paths.
- The first preference for selecting a path must be given to the paths which can be easily sensitized there by delaying the paths whose solution to the path predicate expression is difficult to obtain.
- This convention is followed just for the sake of coverage. Heuristic procedures for path sensitization involve discovery and problem solving using past experience and reasoning.

1. All the process dependent process independent and correlated input variables are first determined and classified accordingly. Show the type of relation that is (logical, arithmetic, functional) and dependency by means of equations for the correlated and dependent variables respectively.

2. After classifying the variables, determine and classify the predicates depending on the input variables into dependent, independent or correlated predicates and also show the type of relation that exists among them.

www.Jntufastupdates.com

3. Consider the uncorrelated and independent predicates for selection or path. During the selection, if you have found any dependent predicate, then there may be a classification error or there might be a bug or complete path coverage is not yet achieved.

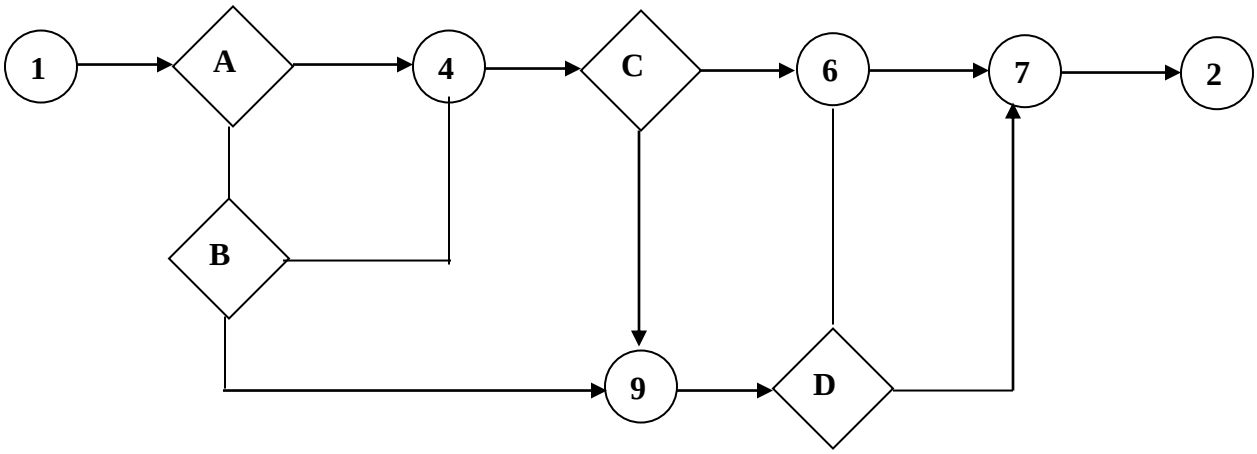
4. Now, consider the correlated and independent predicates if they are not covered then start considering the dependent and uncorrelated, predicates. If the complete coverage is not yet accomplished then move on to the last selection i.e. consider correlated, dependent variables.

5. Display all the input variables, its values, relationship among the variables, type of links for all independent, dependent and correlated variables respectively of every selected path.

6. Every path will produce some set of inequalities, which must be met in order to select that path.

(iv) Examples

(a) Simple, Independent, Uncorrelated Predicates



- Consider the independent, uncorrelated predicates.
- The uppercase letters in the decision boxes of the above figure represent the predicates.
- There are four decisions in this example and, consequently, four predicates.
- False predicates are denoted by a bar on the variable. True predicates are represented by the variables without any bar over them.
- From the above figure, we can retrieve the entire covering path and the predicate values which can be represented as follows.

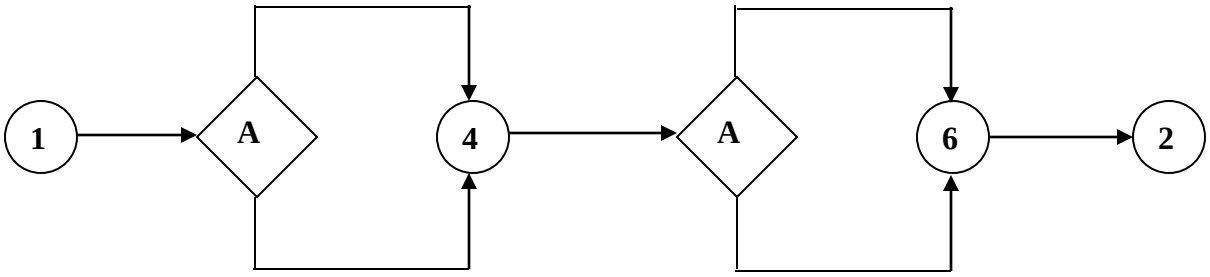
Path	Predicate values
abcdef	AC
aghcimkf	ABCD
aglmjef	ABD

- Using a few more but simpler paths with fewer changes to cover the same flowgraph is

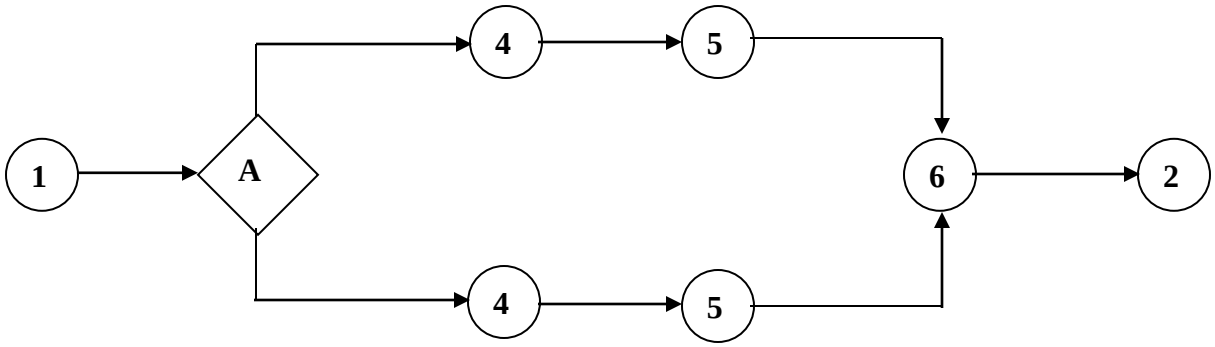
Path	Predicate values
abcdef	AC
abcimjef	ACD
abcimkf	ACD
aghcdef	ABC
aglmkf	ABD

www.Jntufastupdates.com

(b) Correlated, Independent Predicates



- The two decisions in the above figure are correlated because they used the identical predicate (A).
- If you picked paths *abdeg* and *acdfg,* which seem to provide coverage, you would find that neither of these paths is achievable.
- If the A branch (*c*) is taken at the first decision, then the A branch (*e*) must also be taken at the second decision.
- There are two decisions and therefore a potential for four paths, but only two of them, *abdfg* and *acdeg,* are achievable.



- The flowgraph can be replaced with the above figure, in which we have reproduced the common code, or alternatively, we can embed the common link *d* code into a subroutine.

(c) Dependent Predicates

- Finding sensitizing values for dependent predicates may force you to "play computer."
- Usually, and thankfully, most of the routine's processing does not affect the control flow and consequently can be ignored.
- Simulate the computer only to the extent necessary to force paths.
- Loops are the most common kind of dependent predicates; the number of times a typical routine will iterate in the loop is usually determinable in a straightforward manner from the input variables' values.
- Consequently it is usually easy to work backward to determine the input value that will force the loop a specified number of times

(d) The General Case

- There is no simple procedure for the general case. It is easy to state the steps involved but much harder to accomplish them.

1. Select cases to provide coverage on the basis of functionally sensible paths. If the routine is well structured, you should be able to force most of the paths without deep analysis. Intractable paths should be examined for potential bugs before investing time solving equations or whatever you might have to do to find path–forcing input values.

www.Jntufastupdates.com

2. Tackle the path with the fewest decisions first. Give preference to non looping paths over looping paths.

3. Start at the end of the path and not the beginning. Trace the path in reverse and list the predicates in the order in which they appear. The first predicate (the last on the path in the normal direction) imposes restrictions on subsequent predicates (previous when reckoned in the normal path direction). Determine the broadest possible range of values for the predicate that will satisfy the desired path direction.

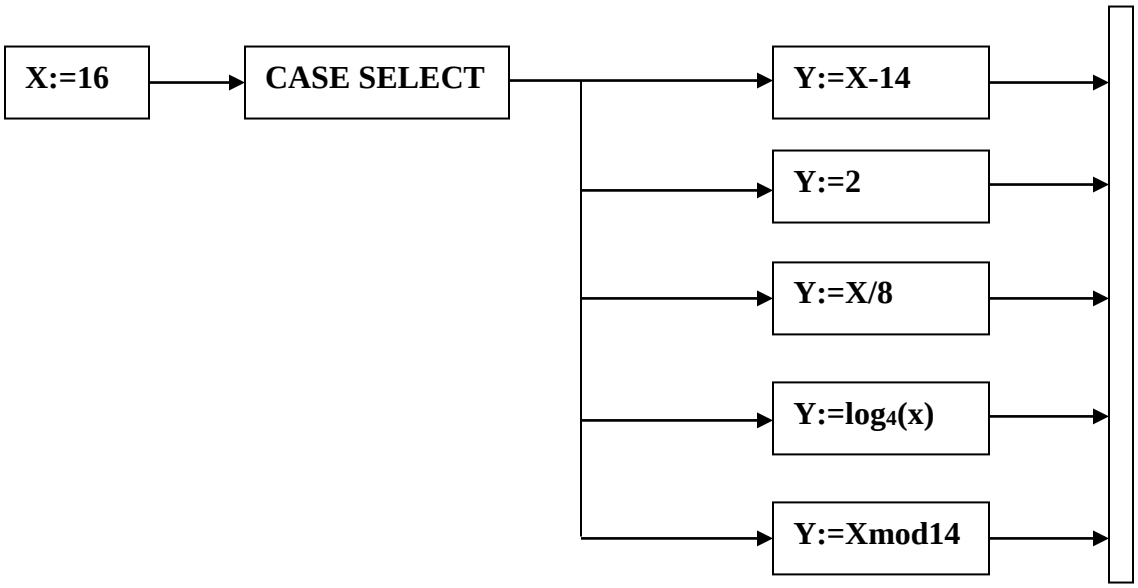
4. Continue working backward along the path to the next decision. The next decision may be restricted by the range of values you determined for the previous decision (in the backward direction). Pick a range of values for the affected variables as broad as possible for the desired direction and consistent with the set of values thus far determined.

5. Continue until you reach the entrance and therefore have established a set of input conditions for the entire path.

(4) Path Instrumentation:

(i) Coincidental Correctness:

- Coincidental Correctness is described as follows.



- Since the test outcome is considered as a part of design process, the test is made to run for comparing the actual outcome with the desired outcome.
- Even if the desired outcome is equal to the actual outcome, only some of the conditions are satisfied by the test which are not sufficient enough.
- This type of condition is named as coincidental correctness.
- Simply it can be defined as a condition in which we check whether the expected outcome of a test is generated truly.
- For instance, the coincidental correctness is represented as follows.
- Let us consider an input variable X with an initial value 16 (X=16) which produces a single outcome Y with a value 2 (Y=2) no matter which case we select.
- Therefore the tests chosen this way will not tell us whether we have achieved coverage.
- For example, the five cases could be totally jumbled and still the outcome would be the same.
- Path instrumentation is what we have to do to confirm that the outcome was achieved by the independent path.

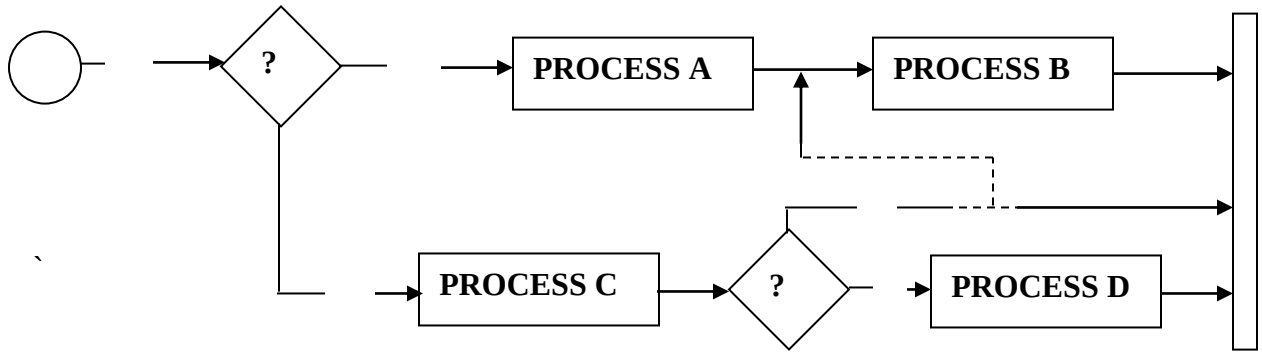
www.Jntufastupdates.com

(ii) Path Instrumentation.

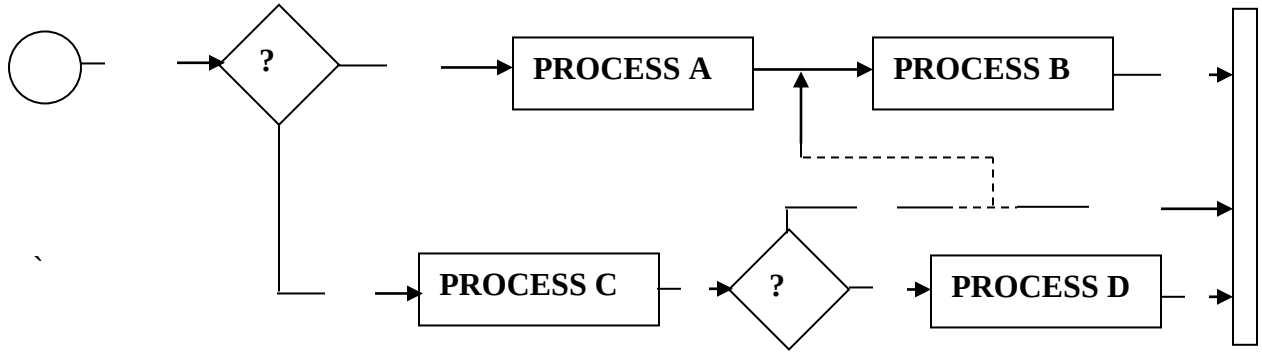
- Path instrumentation is a technique used for identifying whether the outcome of a test is achieved through the desired path or a wrong path.
- Path instrumentation technique is another form of interpretive trace program, which will run each and every statement sequentially there by storing all labels and values of the statements covered for.
- The trouble with traces is that they give us far more information than we need, which is of no use.
- To overcome this drawback many different instrumentation methods have evolved.

(iii) Link Markers

- A simple and effective form of instrumentation is called a traversal marker or link marker.
- Name every link by a lowercase letter. Whenever a link is passed, it's name is recorded in the marker.
- The concatenation of the names of all the links starting from an entry to an exit gives the path name.
- The single link marker may not serve the purpose, because there is every possibility of bug which may result in a new link in the middle of the link being traversed.



- We intended to traverse the ikm path, but because of a GOTO in the middle of the m link, we go to process B.
- If coincidental correctness is against us, the outcomes will be the same and we won't know about the bug.
- The solution is to implement two markers per link: one at the beginning of each link and one at the end.
- The two link markers now specify the path name and confirm both the beginning and end of the link.
- The double link markers are shown in the following figure.



www.Jntufastupdates.com

(iv) Link Counters

- Link counter is one of the instrumentation techniques which usually based on the concept of counters.
- This method provides comparatively less information than interpretive trace method.
- Link counter method of instrumentation follows same procedure as that of link marker but make use of counters instead of using labels for each link which has executed.
- Counters in this method goes on increasing with respect to each link traversed.
- Single counter may not serve the purpose so, we move little deeper and introduce a separate counter for every link.
- With this in practice, we can cross check the total link count against the expected path length.
- This format is not reliable because there is every possibility of having a bug, which may result in a new link in the middle of the link being traveled.
- The same problem that led us to double link markers also leads us to double link counters.

(iv) Other Instrumentation Methods.

- The methods you can use to instrument paths are limited only by your imagination. Here's a sample:

1. Mark each link by a unique prime number and multiply the link name into a central register. The path name is a unique number and you can recapture the links traversed by factoring.

2. Use a bit map with a single bit per link and set that bit when the link is traversed.

3. Use a hash coding scheme over the link names, or calculate an error–detecting code over the link names, such as a check sum.

4. Use your symbolic debugger or trace to give you a trace only of subroutine calls and return.

5. Set a variable value at the beginning of the link to a unique number for that link and use an assertion statement at the end of the link to confirm that you're still on it.

- Every instrumentation probe (marker, counter) you insert gives you more information, but with each probe the information is further removed from reality.

(vi) Implementation

- For unit testing, path instrumentation and verification can be provided by a comprehensive test tool that supports your source language.
- It is easiest to install probes when programming in languages that support conditional assembly or conditional compilation.
- The probes are written in the source code and tagged into categories. Both counters and traversal markers can be implemented, and one need not be parsimonious with the number and placement of probes because only those that are activated for that test will be compiled or assembled.
- For any test or small set of tests, only some of the probes will be active. Rarely would you compile with all probes activated and then only when all else failed.

(5) Implementation and Application of path testing:

- Path testing is a process which involves all the available paths in a program from an entry to an exit in such a way that the entire path is thoroughly tested.
- Path testing implementation and application can be categorized as follows.

(i) Integration, Coverage, and Paths in Called Components

- Path–testing methods are mainly used in unit testing, especially for new software.
- Classical unit testing mainly involves the use of stubs for replacement of all called components and corequisite components thereby testing the new component individually.

www.Jntufastupdates.com

- Path testing process which is carried out at this phase is to analyze the control flow errors rather than focusing on bugs in called or corequisite components.
- We then integrate the component with its called subroutines and corequisite components, one at a time, carefully probing the interface issues.
- Once the interfaces have been tested, we retest the integrated component, this time with the stubs replaced by the real subroutines and corequisite component.
- The component is now ready for the next level of integration. This bottom–up integration process continues until the entire system has been integrated.
- Coverage issue arises since, subroutines and corequisite components are considered to be a part of the component and hence, increasing the complexity as large code need to be processed which makes path sensitization much difficult.
- The main intention behind path testing is that, testing each level at any time increases the effectiveness of the test but the drawback associated with this approach is that it results in i.e. predicate coverage and blindness i.e. outcome of one level may not be compatible with the outcome of other consecutive levels.

(ii) New Code

- The new code (components) has to be given higher priority for testing than the old trusted components.
- Stubs are used where it is clear that the bug potential for the stub is significantly lower than that of the called component.
- That means that old, trusted components will not be replaced by stubs.
- Some consideration is given to paths within called components, but only to the extent that we have to do so to assure that the paths we select at the higher level is achievable.
- Paths within the low level components are also tested, so that there should not be any un-achievable path at higher level.
- Typically, we'll try to use the shortest entry/exit path that will do the job; avoid loops; avoid lower–level subroutine calls; avoid as much lower–level complexity as possible.
- Unit testing must be automated in such a way, that it must perform the testing at each level of integration.

(iii) Maintenance

- The maintenance situation is distinctly different.
- Path testing will be carried out on the modified components but called and corequisite components will be kept unchanged.
- If we have a configuration–controlled, automated, unit test suite, then path testing will be repeated entirely with such modifications as required to accommodate the changes.
- Otherwise, selected paths will be chosen in an attempt to achieve C2 over the changed code.
- As the maintenance methods are studied further a new methodology will be discovered, which will help us to achieve the desired coverage.

(iv) Rehosting

- Rehosting is a process of transforming the old software environment into a new more friendly environment in which rehosted software can run cost effectively.
- When used in conjunction with automatic or semiautomatic structural test generators, we get a very powerful, effective, rehosting process.
- The objective of rehosting is to change the operating environment and not the rehosted software.
- You cannot rehost the software, while performing changes in its environment i.e., the two things cannot be done simultaneously.
- Rehosting can be done in the following ways.

- First, a translator from the old to the new environment is created and tested as any piece of software would be. The bugs in the rehosting process, if any, will be in the translation algorithm and the translator, and the rehosting process is intended to catch those bugs .
- Second, a complete (C1 + C2) path test suite is created for the old software in the old environment.
- Components may be grouped to reduce total testing labor and to avoid a total buildup and reintegration, but C1 + C2 is not compromised.
- The suite is run on the old software in the old environment and all outcomes are recorded.
- These outcomes serve as a guideline for rehosted software. The outcomes and test cases are adapted by the new environment with the help of another interpreter.
- These adapted environment and software are integrated and retested.
- This approach might be even more costly than building the new software, but it provides us with an environment which suites the requirements of software there by providing stable and reliable software base without bothering about the issues pertaining to software security.

www.Jntufastupdates.com