WGU C857 Software QA

**Terms**

**CMMI** – Capability Maturity Model Integration

- If one were to take a high-level view of the CMMI® and its process areas without the interference of the staged or continuous representation that will be discussed later, it would not be hard to classify the CMMI® as project management, quality management, and engineering all glued together by process management.

**Reviews** - A review is an evaluation of a life-cycle work product(s) or project status to determine if there are any deviations from planned results and to recommend improvement. Peer reviews may be thought of as human-based testing as opposed to computer-based testing. An anomaly is any condition that deviates from expectations based on requirements specifications, design documents, standards, plans, and so on, or from someone's experiences. These anomalies are most often called defects.

**Project Manager Reviews** - The project manager review normally translates into the weekly project meeting that is called by the project manager. At a minimum, the project manager and development staff are in attendance. Representatives from systems engineering, QA, CM, and testing should also be in attendance. Items discussed at the weekly project meeting include tracking of the standard issues of technical activities, cost, and schedule performance against the plan. Staffing concerns may also be brought up. Resources planned for and consumed are reviewed. Risks are identified, prioritized, and contingency plans discussed. Necessary commitment changes are acknowledged, documented, and approved. Stakeholder involvement is discussed to ensure that the relevant stakeholders are involved at the time it was deemed necessary and the project is receiving the necessary results from their involvement. Corrective actions maybe necessary and all planning documents are updated as necessary.

**Milestone Reviews** - Formal milestone dates are documented in the project management plan, tracked, and reviewed. Milestones normally represent meaningful points in the project's schedule. This might signify the end of a phase or a major activity. It might be linked to customer involvement and even contract payments such as in the defense world of conducting preliminary design reviews (PDRs) and critical design reviews (CDRs).Typical milestone reviews include examination of technical progress, planned engineering activities, commitments, and plans. They normally address project and other engineering discipline risks. Action items from previous meetings are brought up to determine progress and closure and new action items are recorded, assigned, and reviewed as necessary. All decisions are documented with corresponding actions and resolutions.

**Peer Reviews** – Buddy Checks, Circulation reviews, Tech reviews, Inspections, Walkthroughs and Structured Walkthroughs

**Buddy Check** - A buddy check is normally thought of as an informal verification technique in which the life-cycle work product is examined by the author and one other person. The buddy check operates on basically the same principles as a walkthrough. The "buddy" may or may not prereview the life-cycle work product prior to the peer review. The author walks the "buddy" through the life-cycle work product and describes what is intended in each section. The "buddy" offers comments for improvement on technical correctness, style, order of presentation, clarity, and understandability. Most importantly the buddy decides on the "fit" to the described intent by the author. The author may or may not accept the buddy's suggestions for improvement.

**Circulation Reviews** - Circulation reviews take on attributes of both buddy checks and walkthroughs. Circulation reviews can be informal or follow strict rules. The life-cycle work product is circulated to each reviewer who reviews it and either attaches comments, questions, and recommendations directly on the life-cycle work product or places

them into a separate document. Circulation reviews are dependent on the goodwill of the reviewer. Normally, the author has the authority to accept the reviewers' comments and suggestions or ignore them.

**Inspections** - An inspection is a formal verification technique in which life-cycle work products are examined in detail by a group of peers for the explicit purpose of detecting and identifying defects. The process is led by a moderator or facilitator or inspection leader who is not the author and is impartial to the life-cycle work product under review. The author is not allowed to act as the moderator. Written action on all major defects is mandatory. Rework due to corrections of major defects is formally verified. Defect data is systematically collected and stored in an inspection database. This defect data is analyzed to improve the product, the process, and the effectiveness of the inspection process. Individuals holding management positions over any member of the inspection team are normally not permitted to participate in the inspection.

**Inspection Team** - An inspection team usually consists of four people. The first of the four plays the role of moderator, which in this context is tantamount to that of a quality-control engineer. The moderator is expected to be a competent programmer, but he or she is not the author of the program and need not be acquainted with the details of the program.

Moderator duties include:

- Distributing materials for, and scheduling, the inspection session.
- Leading the session.
- Recording all errors found.
- Ensuring that the errors are subsequently corrected.

The second team member is the programmer. The remaining team members usually are the program's designer (if different from the programmer) and a test specialist. The specialist should be well versed in software testing and familiar with the most common programming errors, which we discuss later in this lesson.

**Inspection Agenda -** Several days in advance of the inspection session, the moderator distributes the program's listing and design specification to the other participants. The participants are expected to familiarize themselves with the material prior to the session. During the session, two activities occur:

- The programmer narrates, statement by statement, the logic of the program. During the discourse, other participants should raise questions, which should be pursued to determine whether errors exist. It is likely that the programmer, rather than the other team members, will find many of the errors identified during this narration. In other words, the simple act of reading aloud a program to an audience seems to be a remarkably effective error-detection technique.
- The program is analyzed with respect to checklists of historically common programming errors (such a checklist is discussed in the next section).

The moderator is responsible for ensuring that the discussions proceed along productive lines and that the participants focus their attention on finding errors, not correcting them. (The programmer corrects errors after the inspection session.)

**Walkthrough** - Walkthroughs were designed to be a less formal verification technique in which life-cycle work products are examined by a group of peers for the purpose of finding defects, omissions, and contradictions. The walkthrough is normally led by the author or the producer of the material being reviewed. As the walkthrough progresses, errors, suggested changes, and improvement suggestions are noted and documented. The consolidated notes are taken by the author for review and revision as the author sees fit.

Like the inspection, the walkthrough is an uninterrupted meeting of one to two hours in duration. The walkthrough team consists of three to five people. One of these people plays a role similar to that of the moderator in the inspection process; another person plays the role of a secretary (a person who records all errors found); and a third person plays the role of a tester. Suggestions as to who the three to five people should be vary. Of course, the programmer is one of those people. Suggestions for the other participants include:

- A highly experienced programmer
- A programming-language expert
- A new programmer (to give a fresh, unbiased outlook)
- The person who will eventually maintain the program
- Someone from a different project
- Someone from the same programming team as the programmer

**Peer Ratings** - The last human review process is not associated with program testing (i.e., its objective is not to find errors). Nevertheless, we include this process here because it is related to the idea of code reading.

Peer rating is a technique of evaluating anonymous programs in terms of their overall quality, maintainability, extensibility, usability, and clarity. The purpose of the technique is to provide programmer self-evaluation.

A programmer is selected to serve as an administrator of the process. The administrator, in turn, selects approximately 6 to 20 participants (6 is the minimum to preserve anonymity). The participants are expected to have similar backgrounds (e.g., don't group Java application programmers with assembly language system programmers). Each participant is asked to select two of his or her own programs to be reviewed. One program should be representative of what the participant considers to be his or her finest work; the other should be a program that the programmer considers to be poorer in quality.

Once the programs have been collected, they are randomly distributed to the participants. Each participant is given four programs to review. Two of the programs are the "finest" programs and two are "poorer" programs, but the reviewer is not told which is which. Each participant spends 30 minutes reviewing each program and then completes an evaluation form. After reviewing all four programs, each participant rates the relative quality of the four programs. The evaluation form asks the reviewer to answer, on a scale from 1 to 10 (1 meaning definitely yes and 10 meaning definitely no), such questions as:

- Was the program easy to understand?
- Was the high-level design visible and reasonable?
- Was the low-level design visible and reasonable?
- Would it be easy for you to modify this program?
- Would you be proud to have written this program?
- The reviewer also is asked for general comments and suggested improvements.

After the review, the participants are given the anonymous evaluation forms for their two contributed programs. They also are given a statistical summary showing the overall and detailed ranking of their original programs across the entire set of programs, as well as an analysis of how their ratings of other programs compared with those ratings of other reviewers of the same program. The purpose of the process is to allow programmers to self-assess their programming skills. As such, the process appears to be useful in both industrial and classroom environments.

**Bug Record** - The bug record should provide clear and complete information about a bug, including details about the environment and specific steps that the developer can use to reproduce the issue. A complete bug record can assist the developer in moving towards a resolution faster and more efficiently. Knowing what a bug record should contain fosters an awareness so when information received is vague or incomplete, the developer is empowered to seek out additional information.

A bug record contains several important elements:

- An accurate description of the problem
- The browser/operating system being used
- Steps to take to repeat the problem
- What was expected vs. what actually happened
- Error messages (with screenshots), if applicable

**Black-box Testing** - To use this method, view the program as a black box. Your goal is to be completely unconcerned about the internal behavior and structure of the program. Instead, concentrate on finding circumstances in which the program does not behave according to its specifications.

- Boundary Value Analysis (BVA)
- Equivalence Class Partitioning
- Decision Table based testing
- Cause-Effect Graphing Technique
- Error Guessing

**Equivalence Partitioning** - The idea behind this technique is to divide a set of test conditions into groups or sets that can be considered the same. **Equivalence partitions** are also known as equivalence classes.

One way of locating this subset is to realize that a well-selected test case also should have two other properties:

1. It reduces, by more than a count of one, the number of other test cases that must be developed to achieve some predefined goal of "reasonable" testing.
2. It covers a large set of other possible test cases. That is, it tells us something about the presence or absence of errors over and above this specific set of input values.

**Boundary Value Analysis** - Experience shows that test cases that explore *boundary conditions* have a higher payoff than test cases that do not. Boundary conditions are those situations directly on, above, and beneath the edges of input equivalence classes and output equivalence classes. Boundary value analysis differs from equivalence partitioning in two respects:

1. Rather than selecting any element in an equivalence class as being representative, boundary value analysis requires that one or more elements be selected such that each edge of the equivalence class is the subject of a test.
2. Rather than just focusing attention on the input conditions (input space), test cases are also derived by considering the *result space* (output equivalence classes).

**Cause-Effect Graphing** - One weakness of boundary value analysis and equivalence partitioning is that they do not explore *combinations* of input circumstances. Cause-effect graphing aids in selecting, in a systematic way, a high-yield set of test cases. It has a beneficial side effect in pointing out incompleteness and ambiguities in the specification.

The following process is used to derive test cases:

1. The specification is divided into workable pieces. This is necessary because cause-effect graphing becomes unwieldy when used on large specifications. For instance, when testing an e-commerce system, a workable piece might be the specification for choosing and verifying a single item placed in a shopping cart. When testing a Web page design, you might test a single menu tree or even a less complex navigation sequence.

2. The causes and effects in the specification are identified. A cause is a distinct input condition or an equivalence class of input conditions. An effect is an output condition or a system transformation (a lingering effect that an input has on the state of the program or system). For instance, if a transaction causes a file or database record to be updated, the alteration is a system transformation; a confirmation message would be an output condition.
   You identify causes and effects by reading the specification word by word and underlining words or phrases that describe causes and effects. Once identified, each cause and effect is assigned a unique number.

3. The semantic content of the specification is analyzed and transformed into a Boolean graph linking the causes and effects. This is the cause-effect graph.

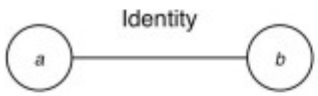
4. The graph is annotated with constraints describing combinations of causes and/or effects that are impossible because of syntactic or environmental constraints.

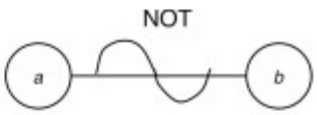
5. By methodically tracing state conditions in the graph, you convert the graph into a limited-entry decision table. Each column in the table represents a test case.

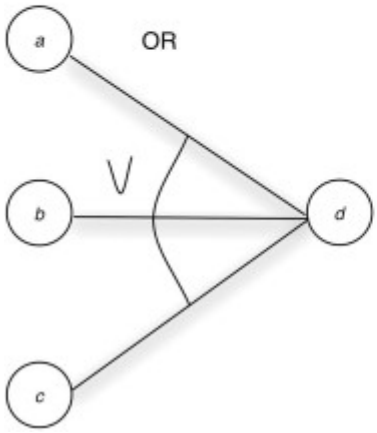
6. The columns in the decision table are converted into test cases.

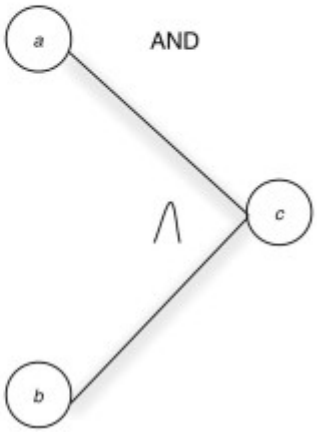
The basic notation for the graph is shown in Figure 4.5. Think of each node as having the value 0 or 1; 0 represents the "absent" state and 1 represents the "present" state.

The figure shows basic cause-effect graph symbols for four functions namely, Identity, NOT, OR, and AND. The identity function contains two nodes a and b connected to each other via a single path. The NOT function contains two nodes a and b connected via a zigzag path. The OR function contains four nodes namely, a, b, c, and d, where a, b, c are connected to d, forming two angles. The angle formed between paths a-d and b-d is showing a V-shaped symbol, referring the OR function. Lastly, the AND function contains nodes a and b connected to node c and the angle formed between paths a-c and b-c is represented with a reverse v-shaped symbol.




The identity function states that if a is 1, b is 1; else b is 0.

The not function states that if a is 1, b is 0, else b is 1.

The or function states that if a or b or c is 1, d is 1; else d is 0.

The and function states that if both a and b are 1, c is 1; else c is 0.

**Error Guessing** - It is difficult to give a procedure for the error-guessing technique since it is largely an intuitive and ad hoc process. The basic idea is to enumerate a list of possible errors or error-prone situations and then write test cases based on the list.

The test-case design methodologies discussed in this lesson can be combined into an overall strategy. The reason for combining them should be obvious by now: Each contributes a particular set of useful test cases, but none of them by itself contributes a thorough set of test cases. A reasonable strategy is as follows:

1. If the specification contains combinations of input conditions, start with cause-effect graphing.
2. In any event, use boundary value analysis. Remember that this is an analysis of input and output boundaries. The boundary value analysis yields a set of supplemental test conditions, but as noted in the section on cause-effect graphing, many or all of these can be incorporated into the cause-effect tests.
3. Identify the valid and invalid equivalence classes for the input and output, and supplement the test cases identified above, if necessary.
4. Use the error-guessing technique to add additional test cases.
5. Examine the program's logic with regard to the set of test cases. Use the decision coverage, condition coverage, decision/condition coverage, or multiple-condition coverage criterion (the last being the most complete). If the coverage criterion has not been met by the test cases identified in the prior four steps, and if meeting the criterion is not impossible (i.e., certain combinations of conditions may be impossible

to create because of the nature of the program), add sufficient test cases to cause the criterion to be satisfied.

Again, the use of this strategy will not guarantee that all errors will be found, but it has been found to represent a reasonable compromise. Also, it represents a considerable amount of hard work, but as we said at the beginning of this lesson, no one has ever claimed that program testing is easy.

- ***Logic coverage***. Tests that exercise all decision point outcomes at least once, and ensure that all statements or entry points are executed at least once.
- ***Equivalence partitioning***. Defines condition or error classes to help reduce the number of finite tests. Assumes that a test of a representative value within a class also tests all values or conditions within that class. The subset of all possible inputs is defined as equivalence classes. This supports a test approach to run a few test cases in a class of inputs to find all the areas across that class, which results in finding the most errors.
- ***Boundary value analysis***. Tests each edge condition of an equivalence class; also considers output equivalence classes as well as input classes.
- ***Cause-effect graphing***. Produces Boolean graphical representations of potential test case results to aid in selecting efficient and complete test cases. When there are specific combinations of inputs, it is best to view the inputs as the cause and test for the effect.
- ***Error guessing***. Produces test cases based on intuitive and expert knowledge of test team members to define potential software errors to facilitate efficient test case design. The error guessing approach is based on intuition and experience in knowing the errors that have been seen before. It is not strictly guessing; the errors may or may not currently exist, but are considered likely.

Extensive, in-depth testing is not easy; nor will the most extensive test case design assure that every error will be uncovered. That said, developers willing to go beyond cursory testing, who will dedicate sufficient time to test case design, analyze carefully the test results, and act decisively on the findings, will be rewarded with functional, reliable software that is reasonably error free.

**White-box Testing** - White-box testing is concerned with the degree to which test cases exercise or cover the logic (source code) of the program.

**Logic(Statement) Coverage Testing** - If you back completely away from path testing, it may seem that a worthy goal would be to execute every statement in the program at least once. Unfortunately, this is a weak criterion for a reasonable white-box test.

**Decision(Branch) Coverage Testing** - This criterion states that you must write enough test cases that each decision has a true and a false outcome at least once. In other words, each branch direction must be traversed at least once. Decision coverage usually can satisfy statement coverage. Since every statement is on some subpath emanating either from a branch statement or from the entry point of the program, every statement must be executed if every branch direction is executed. There are, however, at least three exceptions:

- Programs with no decisions.
- Programs or subroutines/methods with multiple entry points. A given statement might be executed only if the program is entered at a particular entry point.
- Statements within ***ON***-units. Traversing every branch direction will not necessarily cause all ***ON***-units to be executed.

Since we have deemed statement coverage to be a necessary condition, decision coverage, a seemingly better criterion, should be defined to include statement coverage. Hence, decision coverage requires that each decision have a true and a false outcome, and that each statement be executed at least once. An alternative and easier way of expressing it is that each decision has a true and a false outcome, and that each point of entry (including ***ON***-units) be invoked at least once. Decision coverage is a stronger criterion than statement coverage, but it still is rather weak.

Branch Testing = (number of decision outcomes tested/total number of decision outcomes)*100

**Condition Coverage** - In this case, you write enough test cases to ensure that each condition in a decision takes on all possible outcomes at least once. But, as with decision coverage, this does not always lead to the execution of each statement, so an addition to the criterion is that each point of entry to the program or subroutine, as well as ***ON***-units, be invoked at least once.

**Decision/Condition Coverage** - It requires sufficient test cases such that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each point of entry is invoked at least once. A weakness with decision/condition coverage is that although it may appear to exercise all outcomes of all conditions, it frequently does not, because certain conditions mask other conditions

**Multiple Condition Coverage** - This criterion requires that you write sufficient test cases such that all possible combinations of condition outcomes in each decision, and all points of entry, are invoked at least once.

**Module(Unit) Testing**

**Test-Case Design** - You need two types of information when designing test cases for a module test: a specification for the module and the module's source code. The specification typically defines the module's input and output parameters and its function.

**Module(Unit) Testing -** focuses on testing smaller units of the program first, rather than initially testing the whole program.

- Module specifications and its source code are required while designing the module test cases.
- Module's logic is analyzed using the white-box methods.
- Test cases are supplemented by applying black-box methods to the module's specification.

**Non-incremental(Big-Bang) Testing** - is a way of integration testing in which you test each module independently.

- Modules are tested independently and then combined to form a program
- Less machine time is involved

**Incremental Testing** - is a way of integration testing in which first you test each module of the software individually then continue testing by adding another module to it then another. Either, top-down, bottom-up or sandwich.

- Every new module is first combined with already tested module sets
- Incorrect assumption error among modules is detected earlier

**Top-Down** – The top-down strategy starts with the top, or initial, module in the program. After this, there is no single "right" procedure for selecting the next module to be incrementally tested; the only rule is that to be eligible

to be the next module, at least one of the module's subordinate (calling) modules must have been tested previously.

- Conducts testing from main module to sub module
- Finds difficulty in test case representation in stubs, before the I/O function addition
- Programming errors are detected earlier

**Bottom-Up** – For the most part, bottom-up testing is the opposite of top-down testing; thus, the advantages of top-down testing become the disadvantages of bottom-up testing, and the disadvantages of top-down testing become the advantages of bottom-up testing. A drawback of the bottom-up strategy is that there is no concept of an early skeletal program. In fact, the working program does not exist until the last module (module *A*) is added, and this working program is the complete program. Although the I/O functions can be tested before the whole program has been integrated, the advantages of the early skeletal program are not present.

- Conducts testing from sub module to main module
- Creates test conditions and concludes test results easily
- If independently tested, module testing will not see the errors of combining the modules until late in the process. Integration errors can be found earlier by building upon the modules and testing them as you go.

**Higher-Order Testing**

**Function Testing** - a process of attempting to find discrepancies between the program and the external specification. An external specification is a precise description of the program's behavior from the end-user point of view. To perform a function test, you analyze the specification to derive a set of test cases. The equivalence partitioning, boundary value analysis, cause-effect graphing, and error-guessing methods described in Lesson 4 are especially pertinent to function testing. External specifications define the exact representation of the program to the user. A function test would assess the functionality of the program and reveal any discrepancies between current functions and the original specifications.

**System Testing** - System testing is the most misunderstood and most difficult testing process. System testing is *not* a process of testing the functions of the complete system or program, because this would be redundant with the process of function testing. System testing has a particular purpose: to compare the system or program to its original objectives.

1. System testing is not limited to systems. If the product is a program, system testing is the process of attempting to demonstrate how the program, as a whole, fails to meet its objectives.
2. System testing, by definition, is impossible if there is no set of written, measurable objectives for the product.

| Category | Description |
|---|---|
| Facility | Ensure that the functionality in the objectives is implemented. |
| Volume | Subject the program to abnormally large volumes of data to process. |
| Stress | Subject the program to abnormally large loads, generally concurrent processing. |
| Usability | Determine how well the end user can interact with the program. |
| Security | Try to subvert the program's security measures. |
| Performance | Determine whether the program meets response and throughput requirements. |
| Storage | Ensure the program correctly manages its storage needs, both system and physical. |
| Configuration | Check that the program performs adequately on the recommended configurations. |
| Compatibility/Conversion | Determine whether new versions of the program are compatible with previous releases. |
| Installation | Ensure the installation methods work on all supported platforms. |
| Reliability | Determine whether the program meets reliability specifications such as uptime and MTBF. |
| Recovery | Test whether the system's recovery facilities work as designed. |
| Serviceability/Maintenance | Determine whether the application correctly provides mechanisms to yield data on events requiring technical support. |
| Documentation | Validate the accuracy of all user documentation. |
| Procedure | Determine the accuracy of special procedures required to use or maintain the program. |

- It is not a process of testing the functions of the complete system or program, because this would be redundant with the process of function testing.
- It focuses on translation errors made during the designing of external specification.
- It falls under the category of black box testing.
- It involves testing of the user's experience with the application.

**Acceptance Testing** - the process of comparing the program to its initial requirements and the current needs of its end users. It is an unusual type of test in that it usually is performed by the program's customer or end user and normally is not considered the responsibility of the development organization. In the case of a contracted program,

the contracting (user) organization performs the acceptance test by comparing the program's operation to the original contract. As is the case for other types of testing, the best way to do this is to devise test cases that attempt to show that the program does not meet the contract; if these test cases are unsuccessful, the program is accepted. In the case of a program product, such as a computer manufacturer's operating system, or a software company's database system, the sensible customer first performs an acceptance test to determine whether the product satisfies its needs.

- It compares the initial requirements of the program to its end user current needs.
- It is performed by the program's customer or end user.

**Installation Testing** - It is an unusual type of testing because its purpose is not to find software errors but to find errors that occur during the installation process.

Many events occur when installing software systems. A short list of examples includes the following:

- User must select a variety of options.
- Files and libraries must be allocated and loaded.
- Valid hardware configurations must be present.
- Programs may need network connectivity to connect to other programs.

The organization that produced the system should develop the installation tests, which should be delivered as part of the system, and run after the system is installed. Among other things, the test cases might check to ensure that a compatible set of options has been selected, that all parts of the system exist, that all files have been created and have the necessary contents, and that the hardware configuration is appropriate.

- It checks test cases to ensure that compatible set of options have been selected.
- Its test cases are developed by the organization that produced the system.

**Testing Planning and Control** –

The components of a good test plan are as follows:

1. *Objectives*: The objectives of each testing phase must be defined.
2. *Completion criteria*: Criteria must be designed to specify when each testing phase will be judged to be complete. This matter is discussed in the next section.
3. *Schedules*: Calendar time schedules are needed for each phase. They should indicate when test cases will be designed, written, and executed. Some software methodologies such as Extreme Programming (discussed in Lesson 9) require that you design the test cases and unit tests before application coding begins.
4. *Responsibilities*: For each phase, the people who will design, write, execute, and verify test cases, and the people who will repair discovered errors, should be identified. And, because in large projects disputes inevitably arise over whether particular test results represent errors, an arbitrator should be identified.
5. *Test case libraries and standards*: In a large project, systematic methods of identifying, writing, and storing test cases are necessary.
6. *Tools*: The required test tools must be identified, including a plan for who will develop or acquire them, how they will be used, and when they will be needed.

7. *Computer time*: This is a plan for the amount of computer time needed for each testing phase. It would include servers used for compiling applications, if required; desktop machines required for installation testing; Web servers for Web-based applications; networked devices, if required; and so forth.

8. *Hardware configuration*: If special hardware configurations or devices are needed, a plan is required that describes the requirements, how they will be met, and when they will be needed.

9. *Integration*: Part of the test plan is a definition of how the program will be pieced together (e.g., incremental top-down testing). A system containing major subsystems or programs might be pieced together incrementally, using the top-down or bottom-up approach, for instance, but where the building blocks are programs or subsystems, rather than modules. If this is the case, a system integration plan is necessary. The system integration plan defines the order of integration, the functional capability of each version of the system, and responsibilities for producing "scaffolding," code that simulates the function of nonexistent components.

10. *Tracking procedures*: You must identify means to track various aspects of the testing progress, including the location of error-prone modules and estimation of progress with respect to the schedule, resources, and completion criteria.

11. *Debugging procedures*: You must define mechanisms for reporting detected errors, tracking the progress of corrections, and adding the corrections to the system. Schedules, responsibilities, tools, and computer time/resources also must be part of the debugging plan.

12. *Regression testing*: Regression testing is performed after making a functional improvement or repair to the program. Its purpose is to determine whether the change has regressed other aspects of the program. It usually is performed by rerunning some subset of the program's test cases. Regression testing is important because changes and error corrections tend to be much more error prone than the original program code (in much the same way that most typographical errors in newspapers are the result of last-minute editorial changes, rather than changes in the original copy). A plan for regression testing—who, how, when—also is necessary.

**Usability Testing** - Usability or user-based testing basically is a black-box testing technique. It involves actual users or customers of the product. It cannot be called user acceptance testing since it verifies that the deliverable meets the agreed upon requirements whereas usability testing seeks to verify an implementation's approach that works for the user base.

**Component Testing** - tests the interactive software parts for reasonable selection and user feedback.

**Usability Testing Process** - You should establish practical, real-world, repeatable exercises for each user to conduct. Design these testing scenarios to present the user with all aspects of the software, perhaps in various or random order. For example, among the processes you might test in a customer tracking application are:

- Locate an individual customer record and modify it.
- Locate a company record and modify it.

- Create a new company record.
- Delete a company record.
- Generate a list of all companies of a certain type.
- Print this list.
- Export a selected list of contacts to a text file or spreadsheet format.
- Import a text file or spreadsheet file of contacts from another application.
- Add a photograph to one or more records.
- Create and save a custom report.
- Customize the menu structure.

During each phase of the test, have observers document the user experience as they perform each task. When the test is complete, conduct an interview with the user or provide a written questionnaire to document other aspects of the user's experience, such as his or her perception of usage versus specification.

In addition, write down detailed instructions for user tests, to ensure that each user starts with the same information, presented in the same way. Otherwise, you risk coloring some of the tests if some users receive different instructions.

Usability tests are specifically designed to compare program performance to specifications. This is the only test with a purpose that specifically addresses the user.

**Test User Select** - A complete usability testing protocol usually involves multiple tests from the same users, as well as tests from multiple users.

**User Recall** - how much of what a user learns about software operation is retained from session to session.

**How Many Users** - Fortunately, significant research on usability has been conducted during the last 15 years. Based on the work of Jakob Nielsen, a usability testing expert, you may need fewer testers than you think. Nielsen's research found that the number of usability problems found in testing is:

$E=100*(1-(1-L)n)$

where: E = percent of errors found n = number of testers
L = percent of usability problems found by a tester

- It states that it is impossible to detect all of the usability errors in the application.
- It cautions that the precise number of testers depends on economic considerations by the user and on the type of system you are testing.
- It depicts that incurring extra cost and complexity of working with many testers for an application check is not required.

**Data-Gathering Methods -** Videotaping a user test and using a *think-aloud protocol* can provide excellent data on software usability and user perceptions about the application. A think-aloud protocol involves users speaking aloud their thoughts and observations while they are performing the assigned software testing tasks. Developers also may wish to conduct *remote user testing*, whereby the application is installed at the testing user's business where the software may ultimately be applied. Remote testing has the advantage of placing the user in a familiar environment, one in which the final application likely would be used, thus removing the potential for external

influences modifying test results. Of course, the disadvantage is that developers may not receive feedback as detailed as would be possible with a think-aloud protocol. Additional software can be installed with the application to be tested to gather user keystrokes and to capture time required for the user to complete each assigned task. A sophisticated but potentially useful data-gathering protocol is *eye tracking*. When we read a printed page, view a graphical presentation, or interact with a computer screen, our eyes move over the scanned material in particular patterns. Research data gathered on eye movement over more than 100 years shows that eye movement—particularly how long an observer pauses on certain visual elements—reflects at least to some degree the thought processes of the observer. Tracking this eye movement, which can be done with video systems and other technologies, shows researchers which visual elements attract the observers attention, in what order, and for how long. Such data is potentially useful in determining the efficiency of software screens presented to users.

**Hallway intercept** - testing involves testing random users for a software with a general target market.

**Usability Questionnaire -** As with the software testing procedure itself, a usability questionnaire should be carefully planned to return the information required from the associated test procedure. Although you may want to include some questions that elicit free-form comments from the user, in general you want to develop questionnaires that generate responses that can be counted and analyzed across the spectrum of testers. These fall into three general types:

- Yes/no answers
- True/false answers
- Agree/disagree on a scale

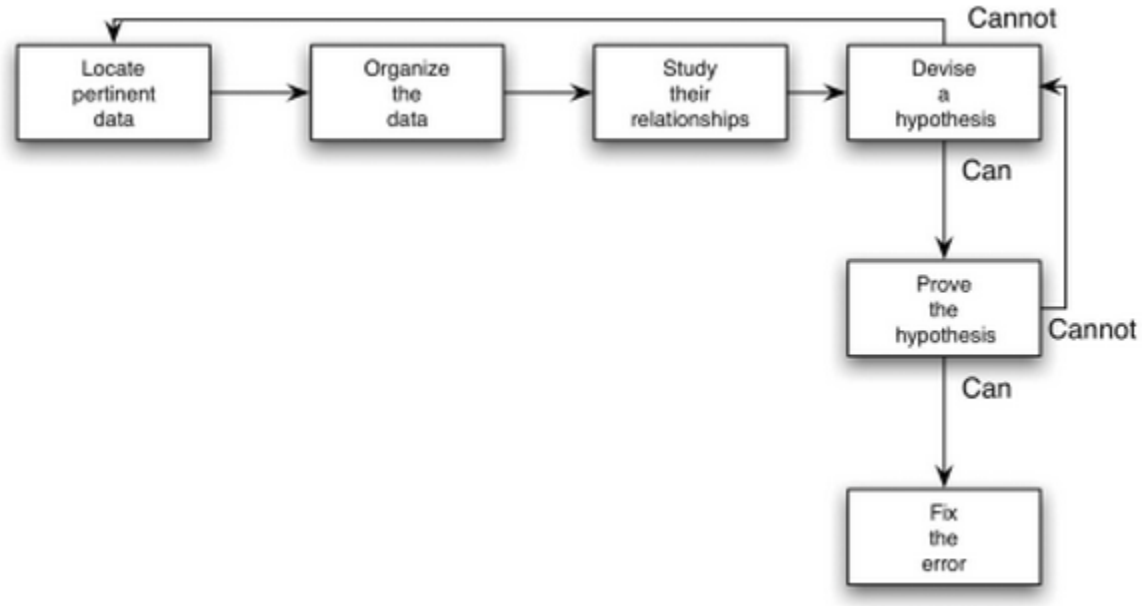
A user questionnaire is the only approach that provides information that can actually be counted. Allowing multiple answers to the same question results in statistical data that can support assumptions about the usability of a program.

**Debugging -** a two-step process that begins with determination of the nature and location of the suspected error and then fixing it.

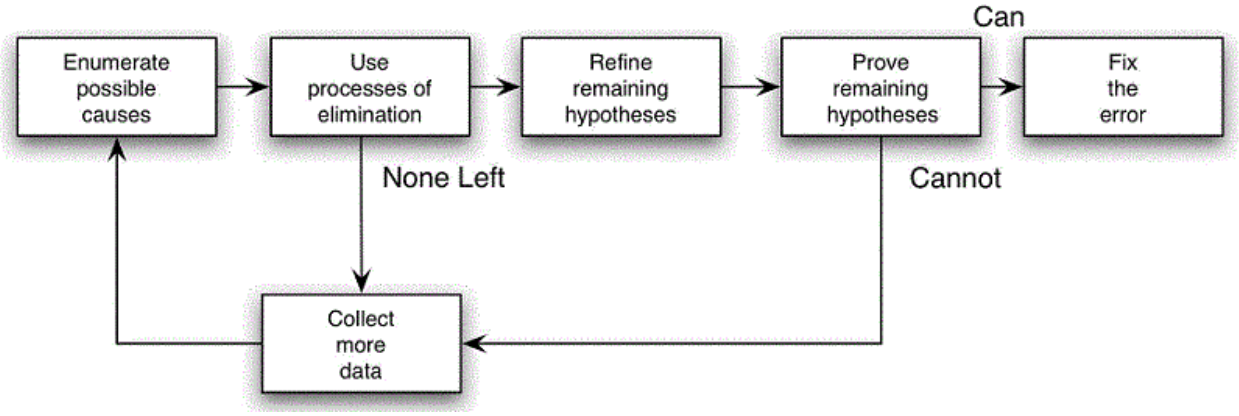
**Debugging by Brute Force -** It is popular because it requires little thought and is the least mentally taxing of the methods; unfortunately, it is inefficient and generally unsuccessful.
Brute-force methods can be partitioned into at least three categories:

- Debugging with a storage dump.
- Debugging according to the common suggestion to "scatter print statements throughout your program."
- Debugging with automated debugging tools.

**Debugging by Induction** - It should be obvious that careful thought will find most errors without the debugger even going near the computer. One particular thought process is *induction*, where you move from the particulars of a situation to the whole. That is, start with the clues (the symptoms of the error and possibly the results of one or more test cases) and look for relationships among the clues. The basic idea behind induction is to find clues that will eventually lead to a possible solution to the problem.




**Debugging by Deduction** - The process of *deduction* proceeds from some general theories or premises, using the processes of elimination and refinement, to arrive at a conclusion.




**Debugging by Backtracking** - An effective method for locating errors in small programs is to backtrack the incorrect results through the logic of the program until you find the point where the logic went astray. In other words, start

at the point where the program gives the incorrect result—such as where incorrect data were printed. Here, you deduce from the observed output what the values of the program's variables must have been. By performing a mental reverse execution of the program from this point and repeatedly applying the if-then logic that states "if this was the state of the program at this point, then this must have been the state of the program up here," you can quickly pinpoint the error. You're looking for the location in the program between the point where the state of the program was what it was expected to be and the first point where the state of the program was not what it was expected to be.

**Debug by Testing** - The last "thinking type" debugging method is the use of test cases. This probably sounds a bit peculiar since, at the beginning of this lesson, we distinguished debugging from testing. However, consider two types of test cases: *test cases for testing*, whose purpose is to expose a previously undetected error, and *test cases for debugging*, whose purpose is to provide information useful in locating a suspected error. The difference between the two is that test cases for testing tend to be "fat," in that you are trying to cover many conditions in a small number of test cases. Test cases for debugging, on the other hand, are "slim," because you want to cover only a single condition or a few conditions in each test case.

**Error analysis -** focuses on judging the exact location of the error, the developer of code, the preventive measures taken to avoid those errors in future, and so on.

**Agile Development**

**Features of Agile Development -** Agile development promotes iterative and incremental development, with significant testing, that is customer-centric and welcomes change during the process. All attributes of traditional software development approaches neglect or minimize the importance of the customer. Although Agile methodologies incorporate flexibility into their processes, the main emphasis is on customer satisfaction. The customer is a key component of the process; simply put, without customer involvement, the Agile method fails. And knowing their interaction is welcomed helps customers build satisfaction and confidence in the end product and development team. If the customer is not committed, then more traditional processes may be a better development choice.

Ironically, Agile development has no single development methodology or process; many rapid development approaches may be considered Agile. These approaches do, however, share three common threads: They rely on customer involvement, mandate significant testing, and have short, iterative development cycles. It is beyond the scope of this course to cover each methodology in detail, but in Table 9.1 we identify the methodologies considered Agile and give a brief description of each. (We urge you to learn more about them because they represent the essence of the Agile philosophy.) In addition, we cover Extreme Programming, one of the more popular Agile methodologies, in greater detail later in this lesson, and offer a practical example.

| Methodology | Description |
|---|---|
| Agile Modeling | Not so much a single modeling methodology, but a collection of principles and practices for modeling and documenting software systems. Used to support other methods such as Extreme Programming and Scrum. |
| Agile Unified Process | Simplified version of the Rational Unified Process (RUP) tailored for Agile development. |
| Dynamic Systems Development Method | Based on rapid application development approaches, this methodology relies on continuous customer involvement and uses an iterative and incremental approach, with the goal of delivering software on time and within budget. |
| Essential Unified Process (EssUP) | An adaptation of RUP in which you choose the practices, (e.g. use cases or team programming) that fit your project. RUP generally uses all practices, whether needed or not. |
| Extreme Programming | Another iterative and incremental approach that relies heavily on unit and acceptance testing. Probably the best known of the Agile methodologies. |
| Feature Driven Development | A methodology that uses industry best practices, such as regular builds, domain object modeling, and feature teams, that are driven by the customer's feature set. |
| Open Unified Process | An Agile approach to implementing standard Unified practices that allows a software team to rapidly develop their product. |
| Scrum | An iterative and incremental project management approach that supports many Agile methodologies. |
| Velocity Tracking | Applies to all Agile development methodologies. It attempts to measure the rate, or "velocity," at which the development process is moving. |

**Extreme Programming** – As mentioned, XP is a software process that helps developers create high-quality code, rapidly. Here, we define "quality" as a code base that meets the design specification and customer expectation. XP focuses on:

- Implementing simple designs.
- Communicating between developers and customers.
- Continually testing the code base.
- Refactoring, to accommodate specification changes.
- Seeking customer feedback.

XP tends to work well for small to medium-size development efforts in environments that have frequent specification changes, and where near- instant communication is possible.

XP differs from traditional development processes in several ways. First, it avoids the large-scale project syndrome in which the customer and the programming team meet to design every detail of the application before coding

begins. Project managers know this approach has its drawbacks, not the least of which is that customer specifications and requirements constantly change to reflect new business rules or marketplace conditions. For example, the finance department may want payroll reports sorted by processed date instead of check numbers; or the marketing department may determine that consumers will not buy product XYZ if it doesn't send an e-mail after website registration. In contrast, XP planning sessions focus on collecting *general application requirements*, not narrowing in on every detail.

Another difference with the XP methodology is that it avoids coding unneeded functionality. If your customer thinks the feature is needed but not required, it generally is left out of the release. Thus, you can focus on the task at hand, adding value to a software product. Concentrating only on the required functionality helps you produce quality software in short time frames.

But the primary difference of XP compared to traditional methodologies is its approach to testing. After an all-inclusive design phase, traditional software development models suggest you code first and create testing interfaces later. In XP, you *must* create the unit tests first, and then write the code to pass the tests. You design unit tests in an XP environment by following the concepts discussed in Lesson 5.

The XP development model has 12 core practices that drive the process, summarized in Table 9.2. In a nutshell, you can group the 12 core XP practices into four concepts:

1. Listening to the customer and other programmers.
2. Collaborating with the customer to develop the application's specification and test cases.
3. Coding with a programming partner.
4. Testing, and retesting, the code base.

**XP Planning**

A successful planning phase lays the foundation of the XP process. The planning phase in XP differs from that in traditional development models, which often combine requirements gathering and application design. Planning in XP focuses on identifying your customer's application requirements and designing user stories (or case stories) that meet them. You gain significant insight into the application's purpose and requirements by creating user stories. In addition, the customer employs the user stories when performing acceptance tests at the end of a release cycle. Finally, an intangible benefit of the planning phase is that the customer gains ownership and confidence in the application by participating intimately in it.

**XP Testing**

Continuous testing is central to the success of a XP-based effort. Although acceptance testing falls under this principle, unit testing occupies the bulk of the effort. Unit tests are devised to make the software fail. Only by ensuring that your tests detect errors can you begin correcting the code so it passes the tests. Assuring that your unit tests catch failures is key to the testing process—and to a developer's confidence. At this point, the developer can experiment with different implementations, knowing that the unit tests will catch any mistakes.

You want to ensure that any code changes improve the application and do not introduce bugs. The continuous testing principle also supports refactoring efforts used to optimize and streamline the code base. Constant testing also leads to that intangible benefit already mentioned: confidence. The programming team gains confidence in the code base because you constantly validate it with unit tests. In addition, your customers' confidence in their investment soars because they know the code base passes unit tests every day.

**Example XP Project Flow**

Now that we've presented the 12 practices of the XP process, you may be wondering, how does a typical XP project flow? Here is a quick example of what you might experience if you worked on an XP-based project:

1) Programmers meet with the customer to determine the product requirements and build user stories.

2) Programmers meet without the customer to divide the requirements into independent tasks and estimate the time to complete each task.

3) Programmers present the customer with the task list and with time estimates, and ask them to generate a priority list of features.

4) The programming team assigns tasks to pairs of programmers, based on their skill sets.

5) Each pair creates unit tests for their programming task using the application's specification.

6) Each pair works on their task with the goal of creating a code base that passes the unit tests.

7) Each pair fixes, then retests their code until all unit tests have passed.

8) All pairs gather every day to integrate their code bases.

9) The team releases a preproduction version of the application.

10) Customers run acceptance tests and either approve the application or produce a report identifying the bugs/deficiencies.

11) Upon successful acceptance tests, programmers release a version into production.

12) Programmers update time estimates based on latest experience.

Although compelling, XP is not for every project or every organization. Proponents of XP conclude that if a programming team fully implements the 12 practices, then the chances of successful application development increase dramatically. Detractors say that because XP is a process, you must do all or nothing; if you skip a practice, then you are not properly implementing XP, and your program quality may suffer. Detractors also claim that the cost of changing a program in the future to add more features is higher than the cost of initially anticipating and coding the requirement. Finally, some programmers find working in pairs very cumbersome and invasive; therefore, they do not embrace the XP philosophy.

The XP development model has 12 core practices that drive the process, summarized in Table 9.2. In a nutshell, you can group the 12 core XP practices into four concepts: next page

1. Listening to the customer and other programmers.
2. Collaborating with the customer to develop the application's specification and test cases.
3. Coding with a programming partner.
4. Testing, and retesting, the code base.

Here are the targets of extreme programming:

- Communication between developers and customers
- Continuous testing of the code base
- Implementation of simple designs
- Accommodate specification changes

| Practice | Comment |
| --- | --- |
| 1. Planning and requirements | Marketing and business development personnel work together to identify the maximum business value of each software feature.<br><br>Each major software feature is written as a user story.<br><br>Programmers provide time estimates to complete each user story.<br><br>The customer chooses the software features based on time estimates and business value. |
| 2. Small, incremental releases | Strive to add small, tangible, value-added features and release a new code base often. |
| 3. System metaphors | Your programming team identifies an organizing metaphor to help with naming conventions and program flow. |
| 4. Simple designs | Implement the simplest design that allows your code to pass its unit tests. Assume change will come, so don't spend a lot of time designing; just implement. |
| 5. Continuous testing | Write unit tests before writing the code module. Each unit is not complete until it passes its unit test. Further, the program is not complete until it passes all unit tests, and acceptance tests are complete. |
| 6. Refactoring | Clean up and streamline your code base. Unit tests help ensure that you do not destroy the functionality in the process. You must rerun all unit tests after any refactoring. |
| 7. Pair programming | You and another programmer work together, at the same machine, to create the code base. This allows for real-time code review, which dramatically facilitates bug detection and resolution. |
| 8. Collective ownership of the code | All code is owned by all programmers.<br>No single programmer is dedicated to a specific code base. |
| 9. Continuous integration | Every day, integrate all changes; after the code passes the unit tests, add it back into the code base. |
| 10. Forty-hour workweek | No overtime is allowed. If you work with dedication for 40 hours per week, overtime will not be needed. The exception is the week before a major release. |
| 11. On-site customer presence | You and your programming team have unlimited access to the customer, to enable you to resolve questions quickly and decisively, which keeps the development process from stalling. |
| 12. Coding standards | All code should look the same. Developing a system metaphor helps meet this principle. |

Here are the testing types:

- **Extreme unit:**
    - All code modules must have primary tests before coding begins.
    - The primary tests must be defined and created before coding the module.
- **Extreme acceptance:**
    - It determines whether the application meets its functional and usable requirements.
    - Customers creates test cases during the design/planning phases.

Here are the functions:

- assertFalse(): It checks whether the parameter supplied causes the method to return an incorrect Boolean value.
- primeCheck(): It checks the input value against a calculated list of numbers divisible only by itself and 1.
- checkArgs(): It asserts that the input value is a positive integer.
- main(): It provides the entry point into the application.

**Scrum** - is a short team meeting to discuss progress and work. It is common across all agile methodologies.

Rapid code development requires a clear goal and a quality definition that are simple and achievable. The design specifications and customer expectations establish these requirements.

**Automated testing** - Rapid development requires immediate feedback. The best way to provide this is to have automated tests ready to run.

**Extreme program testing** - With an agile approach, the team is responsive to change and adaptation. In order for this to work, everyone must be involved throughout the entire process.

**Testing Principles** –

| Principle Number | Principle |
|---|---|
| 1 | A necessary part of a test case is a definition of the expected output or result. |
| 2 | A programmer should avoid attempting to test his or her own program. |
| 3 | A programming organization should not test its own programs. |
| 4 | Any testing process should include a thorough inspection of the results of each test. |
| 5 | Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected. |
| 6 | Examining a program to see if it *does not do what it is supposed to do* is only half the battle; the other half is seeing whether the program *does what it is not supposed to do*. |
| 7 | Avoid throwaway test cases unless the program is truly a throwaway program. |
| 8 | Do not plan a testing effort under the tacit assumption that no errors will be found. |
| 9 | The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section. |
| 10 | Testing is an extremely creative and intellectually challenging task. |

**Code Inspections** - A code inspection is a set of procedures and error-detection techniques for group code reading. Most discussions of code inspections focus on the procedures, forms to be filled out, and so on. Here, after a short summary of the general procedure, we will focus on the actual error-detection techniques.

**Inspection Team -** An inspection team usually consists of four people. The first of the four plays the role of moderator, which in this context is tantamount to that of a quality-control engineer. The moderator is expected to be a competent programmer, but he or she is not the author of the program and need not be acquainted with the details of the program. Moderator duties include:

- Distributing materials for, and scheduling, the inspection session.
- Leading the session.
- Recording all errors found.
- Ensuring that the errors are subsequently corrected.

The second team member is the programmer. The remaining team members usually are the program's designer (if different from the programmer) and a test specialist. The specialist should be well versed in software testing and familiar with the most common programming errors, which we discuss later in this lesson.

**Error Checklist**

| Data Reference | Computation |
|---|---|
| 1. Unset variable used? | 1. Computations on nonarithmetic variables? |
| 2. Subscripts within bounds? | 2. Mixed-mode computations? |
| 3. Noninteger subscripts? | 3. Computations on variables of different lengths? |
| 4. Dangling references? | 4. Target size less than size of assigned value? |
| 5. Correct attributes when aliasing? | 5. Intermediate result overflow or underflow? |
| 6. Record and structure attributes match? | 6. Division by zero? |
| 7. Computing addresses of bit strings? Passing bit-string arguments? | 7. Base-2 inaccuracies? |
| 8. Based storage attributes correct? | 8. Variable's value outside of meaningful range? |
| 9. Structure definitions match across procedures? | 9. Operator precedence understood? |
| 10. Off-by-one errors in indexing or subscripting operations? | 10. Integer divisions correct? |
| 11. Inheritance requirements met? | |
| **Data Declaration** | **Comparison** |
| 1. All variables declared? | 1. Comparisons between inconsistent variables? |
| 2. Default attributes understood? | 2. Mixed-mode comparisons? |
| 3. Arrays and strings initialized properly? | 3. Comparison relationships correct? |
| 4. Correct lengths, types, and storage classes assigned? | 4. Boolean expressions correct? |
| 5. Initialization consistent with storage class? | 5. Comparison and Boolean expressions mixed? |
| 6. Any variables with similar names? | 6. Comparisons of base-2 fractional values? |
| | 7. Operator precedence understood? |
| | 8. Compiler evaluation of Boolean expressions understood? |

| Control Flow | Input/Output |
|---|---|
| 1. Multiway branches exceeded? | 1. File attributes correct? |
| 2. Will each loop terminate? | 2. `OPEN` statements correct? |
| 3. Will program terminate? | 3. Format specification matches I/O statement? |
| 4. Any loop bypasses because of entry conditions? | 4. Buffer size matches record size? |
| 5. Possible loop fall-throughs correct? | 5. Files opened before use? |
| 6. Off-by-one iteration errors? | 6. Files closed after use? |
| 7. `DO/END` statements match? | 7. End-of-file conditions handled? |
| 8. Any nonexhaustive decisions? | 8. I/O errors handled? |
| 9. Any textual or grammatical errors in output information? | |

| Interfaces | Other Checks |
|---|---|
| 1. Number of input parameters equal to number of arguments? | 1. Any unreferenced variables in cross-reference listing? |
| 2. Parameter and argument attributes match? | 2. Attribute list what was expected? |
| 3. Parameter and argument units system match? | 3. Any warning or informational messages? |
| 4. Number of arguments transmitted to called modules equal to number of parameters? | 4. Input checked for validity? |
| 5. Attributes of arguments transmitted to called modules equal to attributes of parameters? | 5. Missing function? |
| 6. Units system of arguments transmitted to called modules equal to units system of parameters? | |
| 7. Number, attributes, and order of arguments to built-in functions correct? | |
| 8. Any references to parameters not associated with current point of entry? | |
| 9. Input-only arguments altered? | |
| 10. Global variable definitions consistent across modules? | |
| 11. Constants passed as arguments? | |