UNIT-04

Coding standard and guidelines

UNIT-04/LECTURE-01

Software Coding standard: [RGPV/Jun 2014(7)]

Software coding standards are language-specific programming rules that greatly reduce the probability of introducing errors into your applications, regardless of which software development model (iterative, waterfall, extreme programming, and so on) is being used to create that application.

Software coding standards originated from the intensive study of industry experts who analyzed how bugs were generated when code was written and correlated these bugs to specific coding practices. They took these correlations between bugs and coding practices and came up with a set of rules that when used prevent coding errors from occurring. Coding standards offer incredible value to software development organizations because they are pre-packaged automated error prevention practices; they close the feedback loop between a bug and what must be done to prevent that bug from reoccurring. You don'tt have to write your own rules to get the benefit of coding standards – the experts have already done it for you.

In a team environment or group collaboration, coding standards ensure uniform coding practices, reducing oversight errors and the time spent in code reviews. When work is outsourced to a third-party contractor, having a set of coding standards in place ensures that the code produced by the contractor meets all quality guidelines mandated by the client company.

Coding Standards Are NOT merely a way of enforcing naming conventions on your code.

Coding Standards Enforcement IS static analysis of source code for:

- Certain rules and patterns to detect problems automatically
- Based on the knowledge collected over many years by industry experts
- Virtual code review or peer review by industry respected language experts – AUTOMATICALLY

Previous efforts at standards enforcement include SEI - CMM and ISO 9001. These efforts failed to deliver on their promise because they created stacks upon stacks of bureaucratic documents. There was no automation of processes– because of this the cost of implementation overwhelms the benefit of process implementation.

How Coding Standards are Classified

Software coding standards are classified by language, usage, and severity levels. Language specific rules and best coding practices are determined by industry experts in that particular language. Usage types and severity levels are set by the user.

Language

Parasoft provides coding standards for:

- C and C++ Testing
- Java
- .NET Languages (including C#, VB.NET, ASP.NET and Managed C++)
- SOA and Web (XML, HTML, CSS, JavaScript, JSP, WSDL, etc.)

How the Coding Standards Process is Automated

Coding standards are automated through:

1. Daily usage by developers. Each developer enforces rules every time a class is written and before the class is checked in to the source code repository.
2. Automated nightly builds. Coding standards are enforced upon all source code modified during the day by automatically running and testing the code in "batch mode".

Both of these methods verify that each developer adhered to the coding standards. In conjunction with Parasoft's reporting system, developers can send reports to management on the current status of their project. This closes the software development lifecycle feedback loop to ensure that the process is indeed in place and running properly.

Code review

Code review is systematic examination (often as peer review) of computer source code. It is intended to find and fix mistakes overlooked in the initial development phase, improving both the overall quality of software and the developers' skills. Reviews are done in various forms such as pair programming, informal walkthroughs, and formal inspections.

Inspection

Inspection in software engineering refers to peer review of any work product by trained individuals who look for defects using a well defined process. An inspection might also be referred to as a Fagan inspection after Michael Fagan, the creator of a very popular software inspection process.

The process

The inspection process was developed by Michael Fagan in the mid-1970s and it has later been extended and modified.

The process should have entry criteria that determine if the inspection process is ready to begin. This prevents unfinished work products from entering the inspection process. The entry criteria might be a checklist including items such as "The document has been spell-checked".

The stages in the inspections process are: Planning, Overview meeting, Preparation, Inspection meeting, Rework and Follow-up. The Preparation, Inspection meeting and Rework stages might be iterated.

- **Planning:** The inspection is planned by the moderator.
- **Overview meeting:** The author describes the background of the work product.
- **Preparation:** Each inspector examines the work product to identify possible defects.
- **Inspection meeting:** During this meeting the reader reads through the work product, part by part and the inspectors point out the defects for every part.
- **Rework:** The author makes changes to the work product according to the action plans from the inspection meeting.
- **Follow-up:** The changes by the author are checked to make sure everything is correct.

The process is ended by the moderator when it satisfies some predefined exit criteria.

Inspection roles

During an inspection the following roles are used.

- **Author:** The person who created the work product being inspected.
- **Moderator:** This is the leader of the inspection. The moderator plans the inspection and coordinates it.
- **Reader:** The person reading through the documents, one item at a time. The other inspectors then point out defects.
- **Recorder/Scribe:** The person that documents the defects that are found during the inspection.
- **Inspector:** The person that examines the work product to identify possible defects.

inspection types

Code review

A code review can be done as a special kind of inspection in which the team examines a sample

of code and fixes any defects in it. In a code review, a defect is a block of code which does not properly implement its requirements, which does not function as the programme intended, or which is not incorrect but could be improved (for example, it could be made more readable or its performance could be improved). In addition to helping teams find and fix bugs, code reviews are useful for both cross-training programmes on the code being reviewed and for helping junior developers learn new programming techniques.

Peer Reviews

Peer Reviews are considered an industry best-practice for detecting software defects early and learning about software artifacts. Peer Reviews are composed of software walkthroughs and software inspections and are integral to software product engineering activities. A collection of coordinated knowledge, skills, and behaviours facilitates the best possible practice of Peer Reviews. The elements of Peer Reviews include the structured review process, standard of excellence product checklists, defined roles of participants, and the forms and reports.

Software inspections are the most rigorous form of Peer Reviews and fully utilize these elements in detecting defects. Software walkthroughs draw selectively upon the elements in assisting the producer to obtain the deepest understanding of an artifact and reaching a consensus among participants. Measured results reveal that Peer Reviews produce an attractive return on investment obtained through accelerated learning and early defect detection. For best results, Peer Reviews are rolled out within an organization through a defined program of preparing a policy and procedure, training practitioners and managers, defining measurements and populating a database structure, and sustaining the roll out infrastructure.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain the coding standards of software engineering	Jun.2014	7

UNIT-04/LECTURE-02

coding conventions :

coding conventions are a set of guidelines for a specific programming language that recommend programming style, practices and methods for each aspect of a piece program written in this language. These conventions usually cover file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices, programming principles, programming rules of thumb, etc. Software programmers are highly recommended to follow these guidelines to help improve the readability of their source code and make software maintenance easier. Coding conventions are only applicable to the human maintainers and peer reviewers of a software project. Conventions may be formalized in a documented set of rules that an entire team or company follows, or may be as informal as the habitual coding practices of an individual. Coding conventions are not enforced by compilers. As a result, not following some or all of the rules has no impact on the executable programs created from the source code.

Common conventions

- Comment conventions
- Indent style conventions
- Naming conventions
- Programming practices
- Programming principles
- Programming rules of thumb
- Programming style conventions

Rapid application development (**RAD**)

Software prototyping is the activity of creating prototypes of software applications, i.e., incomplete versions of the software program being developed. It is an activity that can occur in software development and is comparable to prototyping as known from other fields, such as mechanical engineering or manufacturing.

A prototype typically simulates only a few aspects of, and may be completely different from, the final product.

Prototyping has several benefits: The software designer and implementer can get valuable

feedback from the users early in the project. The client and the contractor can compare if the software made matches the software specification, according to which the software program is built. It also allows the software engineer some insight into the accuracy of initial project estimates and whether the deadlines and milestones proposed can be successfully met.

It refers to a type of software development methodology that uses minimal planning in favor of rapid prototyping. The "planning" of software developed using RAD is interleaved with writing the software itself. The lack of extensive pre-planning generally allows software to be written much faster, and makes it easier to change requirements.

Rapid application development is a software development methodology that involves methods like iterative development and software prototyping. According to Whitten (2004), it is a merger of various structured techniques, especially data-driven Information Engineering, with prototyping techniques to accelerate software systems development.

In rapid application development, structured techniques and prototyping are especially used to define users' requirements and to design the final system. The development process starts with the development of preliminary data models and business process models using structured techniques. In the next stage, requirements are verified using prototyping, eventually to refine the data and process models. These stages are repeated iteratively; further development results in "a combined business requirements and technical design statement to be used for constructing new systems".

RAD approaches may entail compromises in functionality and performance in exchange for enabling faster development and facilitating application maintenance.

This table contains a high-level summary of some of the major types of RAD and their relative strengths and weaknesses.

Agile software development (Agile)	
Pros	Minimizes feature creep by developing in short intervals resulting in miniature softw releasing the product in mini-increments.
Cons	Short iteration may add too little functionality, leading to significant delays in final itera emphasizes real-time communication (preferably face-to-face), using it is problematic team distributed system development. Agile methods produce very little written doc require a significant amount of post-project documentation.

Extreme Programming (XP)	
Pros	Lowers the cost of changes through quick spirals of new requirements. Most design activity occ incrementally and on the fly.
Cons	Programmers must work in pairs, which is difficult for some people. No up-front "detailed desig occurs, which can result in more redesign effort in the long term. The business champion attached to t project full time can potentially become a single point of failure for the project and a major source stress for a team.
Joint application design (JAD)	
Pros	Captures the voice of the customer by involving them in the design and development of the applicat through a series of collaborative workshops called JAD sessions.
Cons	The client may create an unrealistic product vision and request extensive gold-plating, leading a team over- or under-develop functionality.
Lean software development (LD)	
Pros	Creates minimalist solutions (i.e., needs determine technology) and delivers less functionality earlier; the policy that 80% today is better than 100% tomorrow.
Cons	Product may lose its competitive edge because of insufficient core functionality and may exhibit po overall quality.
Rapid application development (RAD)	
Pros	Promotes strong collaborative atmosphere and dynamic gathering of requirements. Business own actively participates in prototyping, writing test cases and performing unit testing.
Cons	Dependence on strong cohesive teams and individual commitment to the project. Decision making rel on the feature functionality team and a communal decision-making process with lesser degree centralized PM and engineering authority.
Scrum	
Pros	Improved productivity in teams previously paralyzed by heavy "process", ability to prioritize work, use backlog for completing items in a series of short iterations or sprints, daily measured progress a communications.
Cons	Reliance on facilitation by a master who may lack the political skills to remove impediments and deli the sprint goal. Due to relying on self-organizing teams and rejecting traditional centralized "proc

	control", internal power struggles can paralyze a team.

Table 1: Pros and Cons of various RAD types

Disadvantages of prototyping

Insufficient analysis: The focus on a limited prototype can distract developers from properly analyzing the complete project. This can lead to overlooking better solutions, preparation of incomplete specifications or the conversion of limited prototypes into poorly engineered final projects that are hard to maintain. Further, since a prototype is limited in functionality it may not scale well if the prototype is used as the basis of a final deliverable, which may not be noticed if developers are too focused on building a prototype as a model.

User confusion of prototype and finished system: Users can begin to think that a prototype, intended to be thrown away, is actually a final system that merely needs to be finished or polished. (They are, for example, often unaware of the effort needed to add error-checking and security features which a prototype may not have.) This can lead them to expect the prototype to accurately model the performance of the final system when this is not the intent of the developers. Users can also become attached to features that were included in a prototype for consideration and then removed from the specification for a final system. If users are able to require all proposed features be included in the final system this can lead to conflict.

Developer misunderstanding of user objectives: Developers may assume that users share their objectives (e.g. to deliver core functionality on time and within budget), without understanding wider commercial issues. For example, user representatives attending Enterprise software (e.g. PeopleSoft) events may have seen demonstrations of "transaction auditing" (where changes are logged and displayed in a difference grid view) without being told that this feature demands additional coding and often requires more hardware to handle extra database accesses. Users might believe they can demand auditing on every field, whereas developers might think this is feature creep because they have made assumptions about the extent of user requirements. If the developer has committed delivery before the user requirements were reviewed, developers are between a rock and a hard place, particularly if user management derives some advantage from their failure to implement requirements.

Developer attachment to prototype: Developers can also become attached to prototypes they

have spent a great deal of effort producing; this can lead to problems like attempting to convert a limited prototype into a final system when it does not have an appropriate underlying architecture. (This may suggest that throwaway prototyping, rather than evolutionary prototyping, should be used.)

Excessive development time of the prototype: A key property to prototyping is the fact that it is supposed to be done quickly. If the developers lose sight of this fact, they very well may try to develop a prototype that is too complex. When the prototype is thrown away the precisely developed requirements that it provides may not yield a sufficient increase in productivity to make up for the time spent developing the prototype. Users can become stuck in debates over details of the prototype, holding up the development team and delaying the final product.

Expense of implementing prototyping: the start up costs for building a development team focused on prototyping may be high. Many companies have development methodologies in place, and changing them can mean retraining, retooling, or both. Many companies tend to just jump into the prototyping without bothering to retrain their workers as much as they should.
A common problem with adopting prototyping technology is high expectations for productivity with insufficient effort behind the learning curve. In addition to training for the use of a prototyping technique, there is an often overlooked need for developing corporate and project specific underlying structure to support the technology. When this underlying structure is omitted, lower productivity can often result.

UNIT-04/LECTURE-03

Debugging

Debugging is that activity which is performed after executing a successful test case. Debugging consists of determining the exact nature and location of the suspected error and fixing the error. Debugging is probably the most difficult activity in software development from a psychological point of view for the following reasons:

- Debugging is done by the person who developed the software, and it is hard for that person to acknowledge that an error was made.
- Of all the software-development activities, debugging is the most mentally taxing because of the way in which most programs are designed and because of the nature of most programming languages (i.e., the location of any error is potentially any statement in the program).
- Debugging is usually performed under a tremendous amount of pressure to fix the suspected error as quickly as possible.
- Compared to the other software-development activities, comparatively little research, literature, and formal instruction exist on the process of debugging.

Of the two aspects of debugging, *locating the error represents about 95% of the activity.* Hence, the rest of this section concentrates on the process of finding the location of an error, given a suspicion that an error exists, based on the results of a successful test case.

Debugging by Brute Force

The most common and least effective method of program debugging is by "brute force". It requires little thought and is the least mentally taxing of all the methods. The brute-force methods are characterized by either debugging with a memory dump; scattering print statements throughout the program, or debugging with automated debugging tools.

Using a memory dump to try to find errors suffers from the following drawbacks:

- Establishing the correspondence between storage locations and the variables in the source program is difficult.
- Massive amounts of data, most of which is irrelevant, must be dealt with.
- A dump shows only the static state of the program at only one instant in time. The dynamics of the program (i.e., state changes over time) are needed to find most errors.

- The dump is rarely produced at the exact-time of the error. Hence the dump does not show the program's state at the time of the error.
- No formal procedure exists for finding the cause of an error analyzing a storage dump.

Scattering print statements throughout the program, although often superior to the use of a dump in that it displays the dynamics of a program and allows one to examine information that is easier to read, is not much better and exhibits the following shortcomings:

- It is still largely a hit-or-miss method.
- It often results in massive amounts of data to be analyzed.
- It requires changing the program, which can mask the error, alter critical timing or introduce new errors.
- It is often too costly or even infeasible for real-time software. Debugging with automated tools also exhibits the shortcomings of hit-or-miss and massive amounts of data which mist be analyzed. The problem of changing the program however is circumvented by the use of the automated debugging tool.

The biggest problem with the brute-force methods is that they ignore the most powerful debugging tool in existence, a well trained and disciplined human brain. Myers suggests that experimental evidence, both from students and experienced programmers, shows:

- Debugging aids do not assist the debugging processes.
- In terms of the speed and accuracy of finding the error, people who use their brains rather than a set of "aids" seem to exhibit superior performance.

Hence, the use of brute-force methods is recommended only when all other methods fail or as a supplement to (not a substitute for) the thought processes described in the subsequent sections.

Debugging by Induction

Many errors can be found by using a disciplined thought process without ever going near the computer. One such thought process is induction, where one proceeds from the particulars to the whole. By starting with the symptoms of the error, possibly in the result of one or more test cases, and looking for relationships among the symptoms, the error is often uncovered.

The induction process is illustrated in Figure 1 and described by Myers as follows:

- **Locate the pertinent data.** A major mistake made when debugging a program is failing to take account of all available data or symptoms about the problems. The first step is the enumeration of all that is known about what the program did correctly, and what it

did incorrectly (i.e., the symptoms that led one to believe that an error exists). Additional valuable clues are provided by similar, but different, test cases that do not cause the symptoms to appear.

- **Organize the data.** Remembering that induction implies that one is progressing from the particulars to the general, the second step is the structuring of the pertinent data to allow one to observe patterns, of particular importance is the search for contradictions (i.e., "the errors occurs only when the pilot perform a left turn while climbing"). A particularly useful organizational technique that can be used to structure the available data is shown in the following table. The "What" boxes list the general symptoms, the "Where" boxes describe where the symptoms were observed, the "When" boxes list anything that is known about the times that the symptoms occur, and the "To What Extent" boxes describes the scope and magnitude of the symptoms. Notice the "Is" and "Is Not" columns. They describe the contradictions that may eventually lead to a hypothesis about the error.

?	Is	Is Not
What		
Where		
When		
To What Extent		

- **Devise a hypothesis.** The next steps are to study the relationships among the clues and devise, using the patterns that might be visible in the structure of the clues, one or more hypotheses about the cause of the error. If one cannot devise a theory, more data are necessary, possibly obtained by devising and executing additional test cases. If multiple theories seem possible, the most probable one is selected first.
- **Prove the hypothesis.** A major mistake at this point, given the pressures under which debugging is usually performed, is skipping this step by jumping to conclusions and attempting to fix the problem. However, it is vital to prove the reasonableness of the hypothesis before proceeding. A failure to do this often results in the fixing of only a symptom of the problem, or only a portion of the problem. The hypothesis is proved by comparing it to the original clues or data, making sure that this hypothesis completely explains the existence of the clues. If it does not, either the hypothesis is invalid, the hypothesis is incomplete, or multiple errors are present.

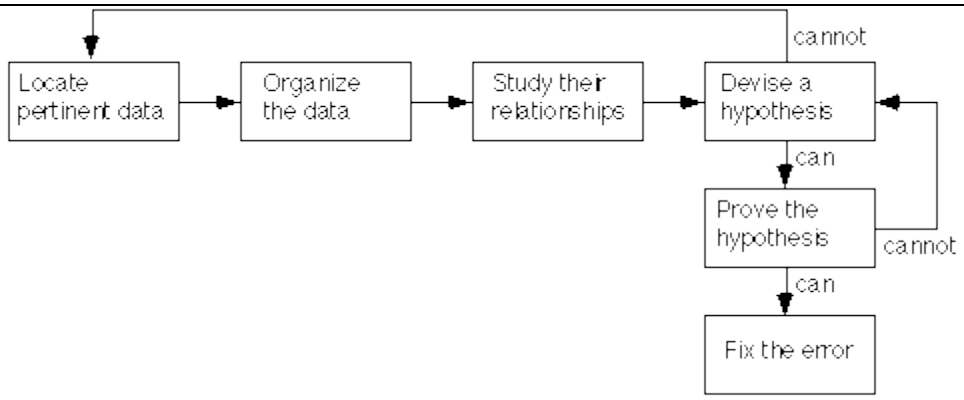


Figure 1. Inductive Debugging Process

Debugging By Deduction

An alternate thought process, that of deduction, is a process of proceeding from some general theories or premises, using the processes of elimination and refinement, to arrive at a conclusion. This process is illustrated in Figure 2 and also described by Myers as follows:

- **Enumerate the possible causes or hypotheses.** The first step is to develop a list of all conceivable causes of the error. They need not be complete explanations; they are merely theories through which one can structure and analyze the available data.
- **Use the data to eliminate possible causes.** By a careful analysis of the data, particularly by looking for contradictions (the previous table could be used here), one attempts to eliminate all but one of the possible causes. If all are eliminated, additional data are needed (e.g., by devising additional test cases) to devise new theories. If more than one possible cause remains, the most probable cause (the prime hypothesis) is selected first.
- **Refine the remaining hypothesis.** The possible cause at this point might be correct, but it is unlikely to he specific enough to pinpoint the error. Hence, the next step is to use the available clues to refine the theory to something more specific.
- **Prove the remaining hypothesis.** This vital step is identical to the fourth step in the induction method.

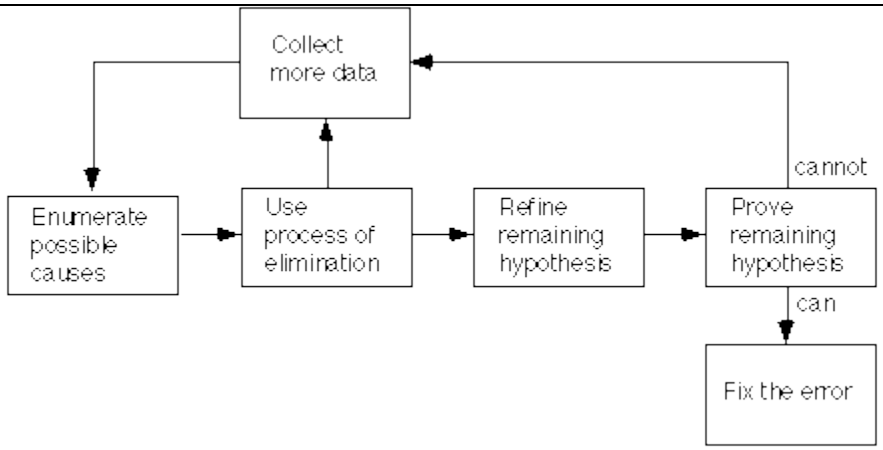


Figure 2. Deductive Debugging Process

Debugging by Backtracking

For small programs, the method of backtracking is often used effectively in locating errors. To use this method, start at the place in the program where an incorrect result was produced and go backwards in the program one step at a time, mentally executing the program in reverse order, to derive the state (or values of all variables) of the program at the previous step. Continuing in this fashion, the error is localized between the points where the state of the program was what was expected and the first point where the state was not what was expected.

Debugging by Testing

The use of additional test cases is another very powerful debugging method which is often used in conjunction with the induction method to obtain information needed to generate a hypothesis and/or to prove a hypothesis and with the deduction method to eliminate suspected causes, refine the remaining hypothesis, and/or prove a hypothesis.

The test cases for debugging differ from those used for integration and testing in that they are more specific and are designed to explore a particular input domain or internal state of the program. Test cases for integration and testing tend to cover many conditions in one test, whereas test cases for debugging tend to cover only one or a very few conditions. The former are designed to detect the error in the most efficient manner whereas the latter are designed to isolate the error most efficiently.

Debugging Guidelines (Error Locating)

As was the case for the testing guidelines, many of these debugging guidelines is intuitively obvious, yet they often forgotten or overlooked. The following guidelines are suggested by

Myers to assist in locating errors.

Debugging is a problem solving process. The most effective method of debugging is a mental analysis of the information associated with the error's symptoms. In efficient program debugger should be able to pinpoint most errors without going near a computer.

If you reach an impasse, sleep on it.

The human subconscious is a potent problem-solver. What we often refer to as inspiration is simply the subconscious mind working on a problem when the conscious mind is working on something else, such as eating, walking, or watching a movie. If you cannot locate an error in a reasonable amount of time (perhaps 30 minutes for a small program, a few hours for a large one), drop it and work on something else, since your thinking efficiency is about to collapse anyway. After "forgetting" about the problem for a while, either your subconscious mind will have solved the problem, or your conscious mind will be clear for a fresh examination of the symptoms.

If you reach an impasse, describe the problem to someone else.

By doing so, you will probably discover something new. In fact, it is often the case that by simply describing the problem to a good listener, you will suddenly see the solution without any assistance from the listener.

Use debugging tools only as a second resort.

And then, use them as an adjunct to, rather than as a substitute for, thinking. 15 noted earlier in this section, debugging tools, such as dumps and traces, represent a haphazard approach to debugging. Experiments show that people who shun such tools, even when they are debugging problems that are unfamiliar to them, tend to be more successful than people who use the tools.

Avoid experimentation.

Use it only as a last resort. The most common mistake made by novice debuggers is attempting to solve a problem by making experimental changes to the program. This totally haphazard approach cannot even be considered debugging; it represents an act of blind hope. Not only does it have a miniscule chance of success, but it often compounds the problem by adding new errors to the program.

Debugging Guidelines (Error Repairing)

The following guidelines for fixing or repairing the program after the error is located are also

suggested by Myers.

Where there is one bug, there is likely to be another.

When one finds an error in a section of a program, the probability of the existence of another error in that section is higher. When repairing an error, examine its immediate vicinity for anything else that looks suspicious.

Fix the error, not just a symptom of it.

Another common failing is repairing the symptoms of the error, or just one instance of the error, rather than the error itself. If the proposed correction does not match all the clues about the error, one may be fixing only a part of the error.

The probability of the fix being correct is not 100%.

Tell this to someone, and of course he would agree, but tell it to someone in the process of correcting an error, and one often gets a different reaction (e.g., "Yes, in most cases, but this correction is so minor that it just has to work"). Code that is added to a program to fix an error can never be assumed correct. Statement for statement, corrections are much more error prone than the original code in the program. One implication is that error corrections must be tested, perhaps more rigorously than the original program.

The probability of the fix being correct drops as the size of the program increases.

Experience has shown that the ratio of errors due to incorrect fixes versus original errors increases in large programs. In one widely used large program, one of every six new errors discovered was an error in a prior correction to the program.

Beware of the possibility that an error correction creates a new error.

Not only does one have to worry about incorrect corrections, but one has to worry about a seemingly valid correction having an undesirable side effect, thus introducing a new error. Not only is there a probability that a fix will be invalid, but there is also a real probability that a fix will introduce a new error. One implication is that not only does the error situation have to be tested after the correction is make, but one must also perform regression testing to determine if a new error has been introduced.

The process of error repair should put one back temporarily in the design phase.

One should realize that error correction is a form of program design. Given the error-prone nature of corrections, common sense says that whatever procedures, methodologies, and formalism were used in the design process should also apply to the error-correction process. For instance, if the project rationalized that code inspections were desirable, then it must be

doubly important that they be used after correcting an error.

Change the source code, not the object code.

When debugging large systems, particularly a system written in an assembly language, occasionally there is the tendency to correct an error by making an immediate change to the object code, with the intention of changing the source program later. Two problems associated with this approach are (l) it is usually a sign that "debugging by experimentation" is being practiced, and (2) the object code and source program are now out of synchronization, meaning that the error could easily surface again when the program is recompiled or reassembled.

UNIT-04/LECTURE-04

Software Testing Strategy: [RGPV/June 2014,2013(7),June 2012(10)]

A **test strategy** is an outline that describes the testing approach of the software development cycle. It is created to inform project managers, testers, and developers about some key issues of the testing process. This includes the testing objective, methods of testing new functions, total time and resources required for the project, and the testing environment.

Test strategies describe how the product risks of the stakeholders are mitigated at the test-level, which types of test are to be performed, and which entry and exit criteria apply. They are created based on development design documents. System design documents are primarily used and occasionally, conceptual design documents may be referred to. Design documents describe the functionality of the software to be enabled in the upcoming release. For every stage of development design, a corresponding test strategy should be created to test the new feature sets.

Strategic Approach to Software Testing

- Many software errors are eliminated before testing begins by conducting effective technical reviews
- Testing begins at the component level and works outward toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.
- The developer of the software conducts testing and may be assisted by independent test groups for large projects.
- Testing and debugging are different activities.
- Debugging must be accommodated in any testing strategy.

Verification and Validation: [RGPV/June2012,2011(10)]

- Make a distinction between verification (are we building the product right?) and validation (are we building the right product?)
- Software testing is only one element of Software Quality Assurance (SQA)
- Quality must be built in to the development process, you can't use testing to add quality after the fact

Organizing for Software Testing

- The role of the Independent Test Group (ITG) is to remove the conflict of interest inherent when the builder is testing his or her own product.
- Misconceptions regarding the use of independent testing teams
  - The developer should do no testing at all
  - Software is tossed “over the wall” to people to test it mercilessly
  - Testers are not involved with the project until it is time for it to be tested
- The developer and ITGC must work together throughout the software project to ensure that thorough tests will be conducted

Software Testing Strategy

- Unit Testing – makes heavy use of testing techniques that exercise specific control paths to detect errors in each software component individually
- Integration Testing – focuses on issues associated with verification and program construction as components begin interacting with one another
- Validation Testing – provides assurance that the software validation criteria (established during requirements analysis) meets all functional, behavioral, and performance requirements
- System Testing – verifies that all system elements mesh properly and that overall system function and performance has been achieved

Strategic Testing Issues

- Specify product requirements in a quantifiable manner before testing starts.
- Specify testing objectives explicitly.
- Identify categories of users for the software and develop a profile for each.
- Develop a test plan that emphasizes rapid cycle testing.
- Build robust software that is designed to test itself.
- Use effective formal reviews as a filter prior to testing.
- Conduct formal technical reviews to assess the test strategy and test cases.
- Develop a continuous improvement approach for the testing process.

Unit Testing[RGPV/ June 2011(10)]

- Module interfaces are tested for proper information flow.
- Local data are examined to ensure that integrity is maintained.
- Boundary conditions are tested.
- Basis (independent) path are tested.
- All error handling paths should be tested.
- Drivers and/or stubs need to be developed to test incomplete software.

Integration Testing: [RGPV/June 2013(7),June 2012(10)]

- Sandwich testing uses top-down tests for upper levels of program structure coupled with bottom-up tests for subordinate levels
- Testers should strive to indentify critical modules having the following requirements
- Overall plan for integration of software and the specific tests are documented in a test specification

Integration Testing Strategies

- Top-down integration testing

1. Main control module used as a test driver and stubs are substitutes for components directly subordinate to it.
2. Subordinate stubs are replaced one at a time with real components (following the depth-first or breadth-first approach).
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests and other stub is replaced with a real component.
5. Regression testing may be used to ensure that new errors not introduced.

- **Bottom-up integration testing**

1. Low level components are combined into clusters that perform a specific software function.
2. A driver (control program) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

- Regression testing – used to check for defects propagated to other modules by changes made to existing program

1. Representative sample of existing test cases is used to exercise all software functions.
2. Additional test cases focusing software functions likely to be affected by the change.
3. Tests cases that focus on the changed software components.

- Smoke testing

1. Software components already translated into code are integrated into a build.
2. A series of tests designed to expose errors that will keep the build from performing its functions are created.
3. The build is integrated with the other builds and the entire product is smoke tested daily (either top-down or bottom integration may be used).

General Software Test Criteria

- Interface integrity – internal and external module interfaces are tested as each module or cluster is added to the software
- Functional validity – test to uncover functional defects in the software
- Information content – test for errors in local or global data structures
- Performance – verify specified performance bounds are tested

Object-Oriented Test Strategies

- Unit Testing – components being tested are classes not modules
- Integration Testing – as classes are integrated into the architecture regression tests are run to uncover communication and collaboration errors between objects
- Systems Testing – the system as a whole is tested to uncover requirement errors

Object-Oriented Unit Testing

- smallest testable unit is the encapsulated class or object
- similar to system testing of conventional software
- do not test operations in isolation from one another
- driven by class operations and state behavior, not algorithmic detail and data flow across module interface

Object-Oriented Integration Testing

- focuses on groups of classes that collaborate or communicate in some manner
- integration of operations one at a time into classes is often meaningless
- **thread-based testing** – testing all classes required to respond to one system input or event
- **use-based testing** – begins by testing independent classes (classes that use very few server classes) first and the dependent classes that make use of them
- **cluster testing** – groups of collaborating classes are tested for interaction errors
- regression testing is important as each thread, cluster, or subsystem is added to the system

WebApp Testing Strategies

1. WebApp content model is reviewed to uncover errors.
2. Interface model is reviewed to ensure all use-cases are accommodated.
3. Design model for WebApp is reviewed to uncover navigation errors.
4. User interface is tested to uncover presentation errors and/or navigation mechanics problems.
5. Selected functional components are unit tested.
6. Navigation throughout the architecture is tested.
7. WebApp is implemented in a variety of different environmental configurations and the compatibility of WebApp with each is assessed.
8. Security tests are conducted.
9. Performance tests are conducted.
10. WebApp is tested by a controlled and monitored group of end-users (looking for content errors, navigation errors, usability concerns, compatibility issues, reliability, and performance).

Validation Testing

- Focuses on visible user actions and user recognizable outputs from the system
- Validation tests are based on the use-case scenarios, the behavior model, and the event flow diagram created in the analysis model
  - Must ensure that each function or performance characteristic conforms to its specification.
  - Deviations (deficiencies) must be negotiated with the customer to establish a means for resolving the errors.
- Configuration review or audit is used to ensure that all elements of the software

configuration have been properly developed, cataloged, and documented to allow its support during its maintenance phase.

Acceptance Testing

- Making sure the software works correctly for intended user in his or her normal work environment.
- Alpha test – version of the complete software is tested by customer under the supervision of the developer at the developer's site
- Beta test – version of the complete software is tested by customer at his or her own site without the developer being present

System Testing: [RGPV/June2013(7)]

- Series of tests whose purpose is to exercise a computer-based system
- The focus of these system tests cases identify interfacing errors
- Recovery testing – checks the system's ability to recover from failures
- Security testing – verifies that system protection mechanism prevent improper penetration or data alteration
- Stress testing – program is checked to see how well it deals with abnormal resource demands (i.e. quantity, frequency, or volume)
- Performance testing – designed to test the run-time performance of software, especially real-time software
- Deployment (or configuration) testing – exercises the software in each of the environment in which it is to operate

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	What are testing principles the software engineer must apply while performing the software testing?	Jun.2014, 2013	7
Q.2	Explain the integration testing process & System Testing processes & discuss their outcomes.	Jun.2013	7
Q.3	Discuss software testing strategies. Differentiate between verification & validation.	Jun.2012	10
Q.4	Describe verification & validation criteria for software.	Jun.2011	10
Q.5	Describe unit testing & integration testing. How test plans are generated?	June 2011	10

UNIT-04/LECTURE-05

Black Box Testing[RGPV/June 2014(7),June 2011(10)]

The technique of testing without having any knowledge of the interior workings of the application is Black Box testing. The tester is oblivious to the system architecture and does not have access to the source code. Typically, when performing a black box test, a tester will interact with the system's user interface by providing inputs and examining outputs without knowing how and where the inputs are worked upon.

Advantages

- Well suited and efficient for large code segments.
- Code Access not required.
- Clearly separates user's perspective from the developer's perspective through visibly defined roles.
- Large numbers of moderately skilled testers can test the application with no knowledge of implementation, programming language or operating systems.

Disadvantages

- Limited Coverage since only a selected number of test scenarios are actually performed.
- Inefficient testing, due to the fact that the tester only has limited knowledge about an application.
- Blind Coverage, since the tester cannot target specific code segments or error prone areas.
- The test cases are difficult to design.

White Box Testing

White box testing is the detailed investigation of internal logic and structure of the code. White box testing is also called glass testing or open box testing. In order to perform white box testing on an application, the tester needs to possess knowledge of the internal working of the code.

The tester needs to have a look inside the source code and find out which unit/chunk of the code is behaving inappropriately.

Advantages

- As the tester has knowledge of the source code, it becomes very easy to find out which type of data can help in testing the application effectively.
- It helps in optimizing the code.
- Extra lines of code can be removed which can bring in hidden defects.
- Due to the tester's knowledge about the code, maximum coverage is attained during test scenario writing.

Disadvantages

- Due to the fact that a skilled tester is needed to perform white box testing, the costs are increased.
- Sometimes it is impossible to look into every nook and corner to find out hidden errors that may create problems as many paths will go untested.
- It is difficult to maintain white box testing as the use of specialized tools like code analyzers and debugging tools are required.

Grey Box Testing

Grey Box testing is a technique to test the application with limited knowledge of the internal workings of an application. In software testing, the term the more you know the better carries a lot of weight when testing an application.

Mastering the domain of a system always gives the tester an edge over someone with limited domain knowledge. Unlike black box testing, where the tester only tests the application's user interface, in grey box testing, the tester has access to design documents and the database. Having this knowledge, the tester is able to better prepare test data and test scenarios when making the test plan.

Advantages

- Offers combined benefits of black box and white box testing wherever possible.
- Grey box testers don't rely on the source code; instead they rely on interface definition and functional specifications.
- Based on the limited information available, a grey box tester can design excellent test scenarios especially around communication protocols and data type handling.
- The test is done from the point of view of the user and not the designer.

Disadvantages

- Since the access to source code is not available, the ability to go over the code and test coverage is limited.
- The tests can be redundant if the software designer has already run a test case.
- Testing every possible input stream is unrealistic because it would take an unreasonable amount of time; therefore, many program paths will go untested.

Black Box vs. Grey Box vs. White Box

S.N.	Black Box Testing	Grey Box Testing	White Box Testing
1	The Internal Workings of an application are not required to be known	Somewhat knowledge of the internal workings are known	Tester has full knowledge of the Internal workings of the application
2	Also known as closed box testing, data driven testing and functional testing	Another term for grey box testing is translucent testing as the tester has limited knowledge of the insides of the application	Also known as clear box testing, structural testing or code based testing
3	Performed by end users and also by testers and developers	Performed by end users and also by testers and developers	Normally done by testers and developers
4	Testing is based on external expectations - Internal behavior of the application is unknown	Testing is done on the basis of high level database diagrams and data flow diagrams	Internal workings are fully known and the tester can design test data accordingly

5	This is the least time consuming and exhaustive	Partly time consuming and exhaustive	The most exhaustive and time consuming type of testing
6	Not suited to algorithm testing	Not suited to algorithm testing	Suited for algorithm testing
7	This can only be done by trial and error method	Data domains and Internal boundaries can be tested, if known	Data domains and Internal boundaries can be better tested

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	What is black box testing? It is necessary to perform this? Explain various test activities.	Jun.2014	7

UNIT-04/LECTURE-06

Cyclomatic complexity

is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of a basis path testing method, the value computed for Cyclomatic complexity defines the number for independent paths in the basis set of a program and provides us an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.

Computing Cyclomatic Complexity: Cyclomatic complexity has a foundation in graph theory and provides us with extremely useful software metric. Complexity is computed in one of the three ways:

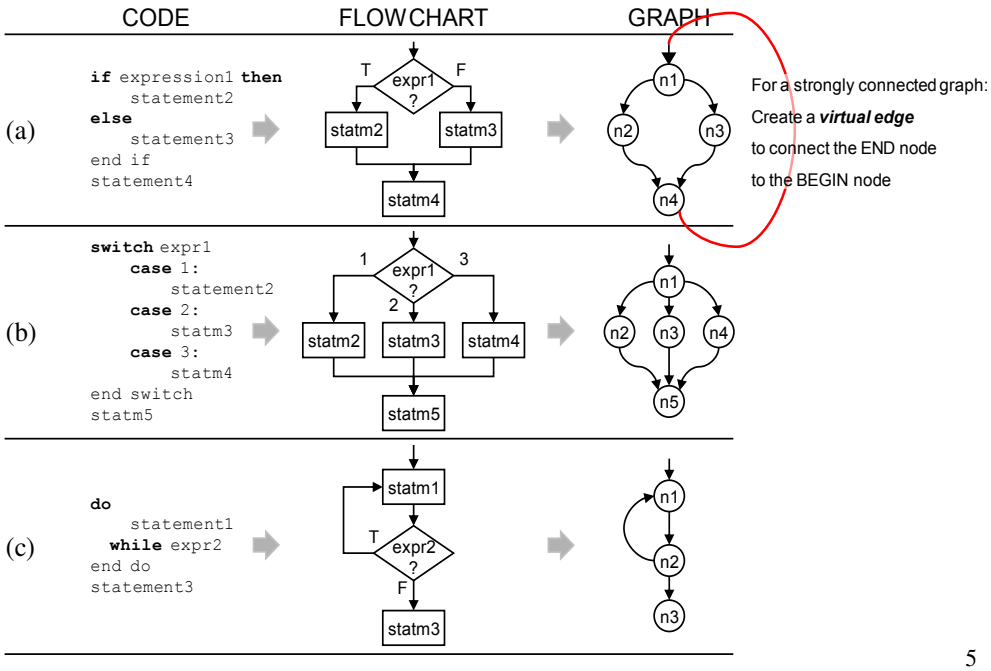
1. The number of regions of the flow graph corresponds to the Cyclomatic complexity.

2. Cyclomatic complexity, V(G), for a flow graph, G is defined as
V (G) = E-N+2P Where E, is the number of flow graph edges, N is the number of flow graph nodes, P is independent component.

3. Cyclomatic complexity, V (G) for a flow graph, G is also defined as:
V (G) = Pie+1 where Pie is the number of predicate nodes contained in the flow graph G.
4. Graph Matrices: The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. To develop a software tool that assists in basis path testing, a data structure, called a graph matrix can be quite useful.
A Graph Matrix is a square matrix whose size is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections between nodes. The connection matrix can also be used to find the cyclomatic complexity

Converting Code to Graph

Converting Code to Graph



5

Paths in Graphs

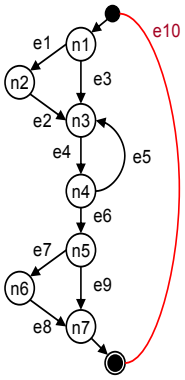
- A graph is **strongly connected** if for any two nodes x, y there is a path from x to y and vice versa
- A **path** is represented as an n-element vector where n is the number of edges <□, □, ..., □>
- The i-th position in the vector is the number of occurrences of edge i in the path

Example Paths

```
if expression1 then
    statement2
end if

do
    statement3
  while expr4
end do

if expression5 then
    statement6
end if
statement7
```



Paths:

	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
P1 = e1, e2, e4, e6, e7, e8	1,	1,	0,	1,	0,	1,	1,	1,	0,	0
P2 = e1, e2, e4, e5, e4, e6, e7, e8	1,	1,	0,	2,	1,	1,	1,	1,	0,	0
P3 = e3, e4, e6, e7, e8, e10	0,	0,	1,	1,	0,	1,	1,	1,	0,	1
P4 = e6, e7, e8, e10, e3, e4	0,	0,	1,	1,	0,	1,	1,	1,	0,	1
P5 = e1, e2, e4, e6, e9, e10	1,	1,	0,	1,	0,	1,	0,	0,	1,	1
P6 = e4, e5	0,	0,	0,	1,	1,	0,	0,	0,	0,	0
P7 = e3, e4, e6, e9, e10	0,	0,	1,	1,	0,	1,	0,	0,	1,	1
P8 = e1, e2, e4, e5, e4, e6, e9, e10	1,	1,	0,	2,	1,	1,	0,	0,	1,	1

NOTE: A path does not need to start in node n1 and does not need to begin and end at the same node.
E.g.,
- Path P4 starts (and ends) at node n4
- Path P1 starts at node n1 and ends at node n7

Paths in Graphs (2)

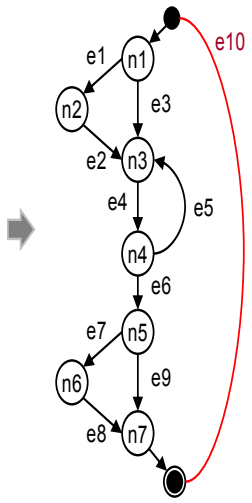
- A **circuit** is a path that begins and ends at the same node
  - e.g., P3 = <e3, e4, e6, e7, e8, e10> begins and ends at node n1
  - P6 = <e4, e5> begins and ends at node n3
- A **cycle** is a circuit with no node (other than the starting node) included more than once

Example Circuits & Cycles

```
if expression1 then
    statement2
end if

do
    statement3
  while expr4
end do

if expression5 then
    statement6
end if
statement7
```



Circuits:

	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
P3 = e3, e4, e6, e7, e8, e10	0,	0,	1,	1,	0,	1,	1,	1,	0,	1
P4 = e6, e7, e8, e10, e3, e4	0,	0,	1,	1,	0,	1,	1,	1,	0,	1
P5 = e1, e2, e4, e6, e9, e10	1,	1,	0,	1,	0,	1,	0,	0,	1,	1
P6 = e4, e5	0,	0,	0,	1,	1,	0,	0,	0,	0,	0
P7 = e3, e4, e6, e9, 10	0,	0,	1,	1,	0,	1,	0,	0,	1,	1
P8 = e1, e2, e4, e5, e4, e6, e9, e10	1,	1,	0,	2,	1,	1,	0,	0,	1,	1
P9 = e3, e4, e5, e4, e6, e9, 10	0,	0,	1,	2,	1,	1,	0,	0,	1,	1

Cycles:

P3 = e3, e4, e6, e7, e8, e10

P5 = e1, e2, e4, e6, e9, e10

P6 = e4, e5

P7 = e3, e4, e6, e9, 10

Linearly Independent Paths

- A path p is said to be a linear combination of paths p_1, …, p_n if there are integers a_1, …, a_n such that $p = \sum a_i \cdot p_i$
- A set of paths is linearly independent if no path in the set is a linear combination of any other paths in the set
- A basis set of cycles is a maximal linearly independent set of cycles
  - In a graph with e edges and n nodes, the basis has e – n + 1 cycles
- Every path is a linear combination of basis cycles

Baseline method for finding the basis set of cycles

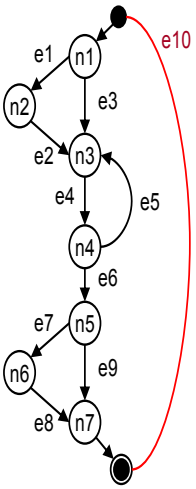
- Start at the source node
- Follow the leftmost path until the sink node is reached
- Repeatedly retrace this path from the source node, but change decisions at every node with out-degree ≥2, starting with the decision node lowest in the path

Linearly Independent Paths

```
if expression1 then
    statement2
end if

do
    statement3
  while expr4
end do

if expression5 then
    statement6
end if
statement7
```



Paths:

	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
P1 = e1, e2, e4, e6, e7, e8	1,	1,	0,	1,	0,	1,	1,	1,	0,	0
P2 = e1, e2, e4, e5, e4, e6, e7, e8	1,	1,	0,	2,	1,	1,	1,	1,	0,	0
P3 = e3, e4, e6, e7, e8, e10	0,	0,	1,	1,	0,	1,	1,	1,	0,	1
P4 = e6, e7, e8, e10, e3, e4	0,	0,	1,	1,	0,	1,	1,	1,	0,	1
P5 = e1, e2, e4, e6, e9, e10	1,	1,	0,	1,	0,	1,	0,	0,	1,	1
P6 = e4, e5	0,	0,	0,	1,	1,	0,	0,	0,	0,	0
P7 = e3, e4, e6, e9, 10	0,	0,	1,	1,	0,	1,	0,	0,	1,	1
P8 = e1, e2, e4, e5, e4, e6, e9, e10	1,	1,	0,	2,	1,	1,	0,	0,	1,	1

V(G) = e – n + 2 = 9 – 7 + 2 = 4

Or, if we count e10, then e – n + 1 = 10 – 7 + 1 = 4

Cycles:

P3 = e3, e4, e6, e7, e8, e10

P5 = e1, e2, e4, e6, e9, e10

P6 = e4, e5

P7 = e3, e4, e6, e9, 10

EXAMPLE #1: P5 + P6 = P8

P5	{1, 1, 0, 1, 0, 1, 0, 0, 1, 1}
+ P6	{0, 0, 0, 1, 1, 0, 0, 0, 0, 0}
= P8	{1, 1, 0, 2, 1, 1, 0, 0, 1, 1}

EXAMPLE #2: 2×P3 – P5 + P6 =

2×P3	{ 0, 0, 2, 2, 0, 2, 2, 2, 0, 2}
– P5	{ 1, 1, 0, 1, 0, 1, 0, 0, 1, 1}
	{-1,-1, 2, 1, 0, 1, 2, 2,-1, 1}
+ P6	{ 0, 0, 0, 1, 1, 0, 0, 0, 0, 0}
= P?	{-1,-1, 2, 2, 1, 1, 2, 2,-1, 1}

Unit Testing: Path Coverage

- Finds the number of distinct paths through the program to be traversed at least once

- Minimum number of tests necessary to cover all edges is equal to the number of

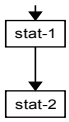
independent paths through the control-flow graph

Issues (1)

Single statement:

= CC =

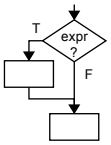
Two (or more) statements:



Cyclomatic complexity (CC) remains the same for a linear sequence of statements regardless of the sequence length
—insensitive to complexity contributed by the multitude of statements

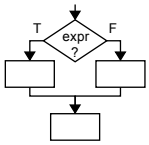
Issues (2)

Optional action:



= CC =

Alternative choices:

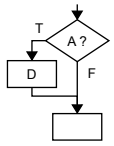


Optional action versus alternative choices —
the latter is psychologically more difficult

Issues (3)

Simple condition:

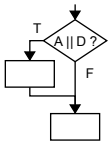
```
if (A) then D;
```



= CC =

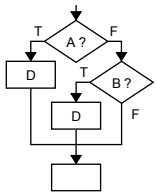
Compound condition:

```
if (A OR B) then D;
```



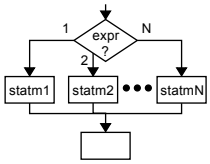
BUT, compound condition can be written as a nested IF:

```
if (A) then D;
else if (B) then D;
```



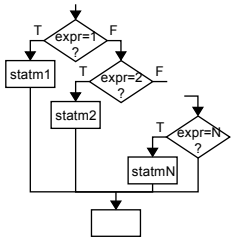
Issues (4)

Switch/Case statement:



= CC =

N–1 predicates:



Counting a **switch** statement:

—as a single decision

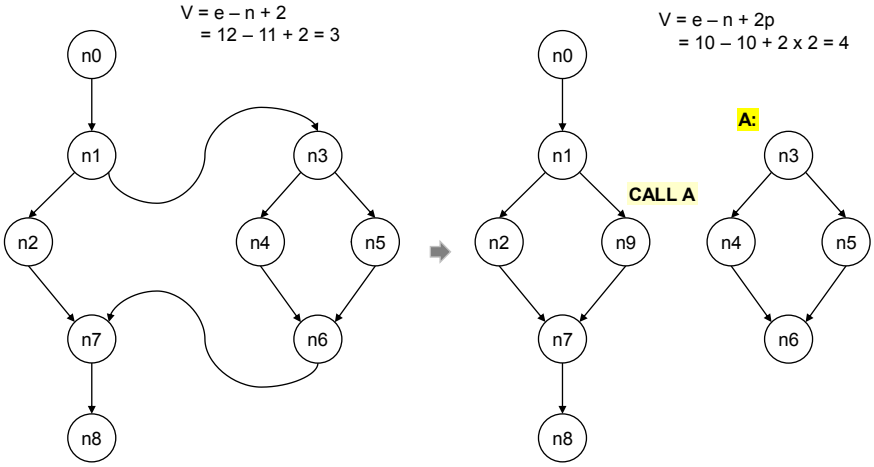
proposed by W. J. Hansen, "Measurement of program complexity by the pair (cyclomatic number, operator count)," *SIGPLAN Notices*, vol.13, no.3, pp.29-33, March 1978.

—as $\log_2(N)$ relationship

proposed by V. Basili and R. Reiter, "Evaluating automatable measures for software development," *Proceedings of the IEEE Workshop on Quantitative Software Models for Reliability, Complexity and Cost*, pp.107-116, October 1979.

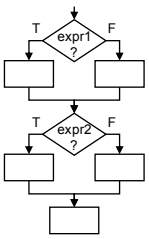
CC for Modular Programs (2)

Intuitive expectation:
Modularization should not increase complexity



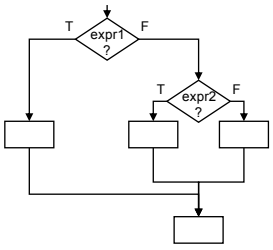
Issues (5)

Two sequential decisions:



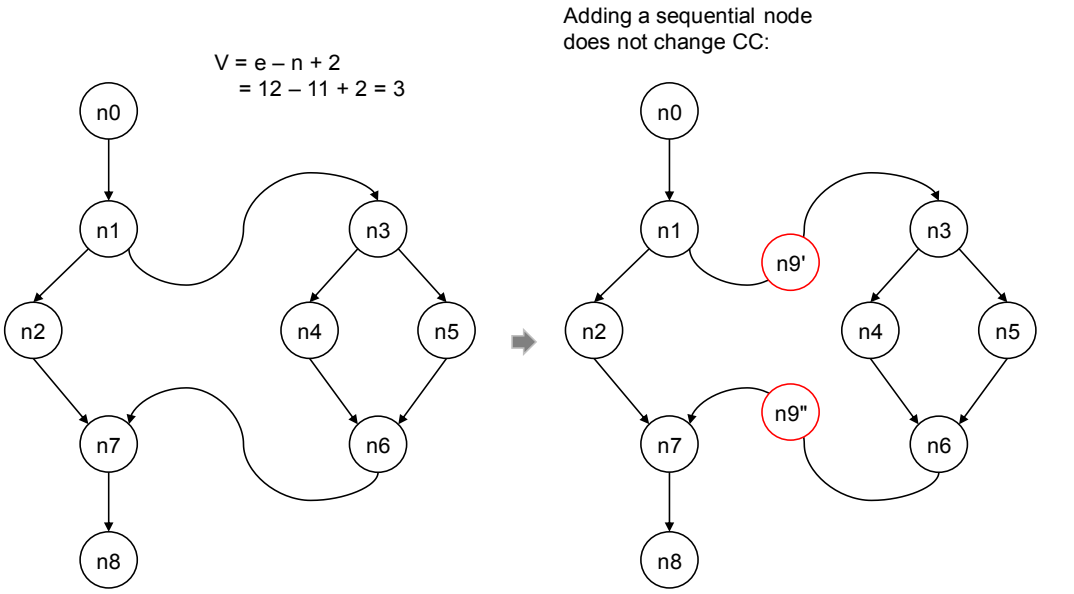
= CC =

Two nested decisions:



But, it is known that people find nested decisions more difficult …

CC for Modular Programs (1)



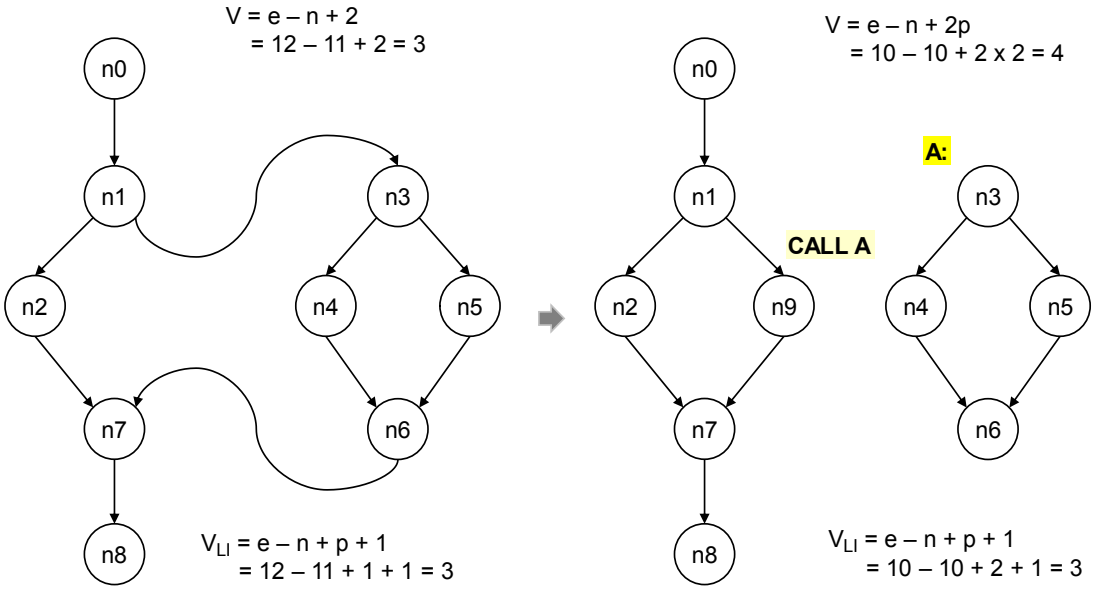
19

Alternative CC Measures

- Given **p** connected components of a graph:
  - V(G) = e – n + 2p (1)
  - $V_{LI}(G)$ = e – n + p + 1 (2)
  - Eq. (2) is known as linearly-independent cyclomatic complexity
  - V_{LI} does not change when program is modularized into p modules

CC for Modular Programs (3)

Intuitive expectation:
Modularization should not increase complexity



UNIT-04/LECTURE-07

Example of Test Case Design

Initial Functional Test Cases for Example ATM System

The following initial test cases can be identified early in the design process as a vehicle for checking that the implementation is basically correct. No attempt has been made at this point to do thorough testing, including all possible errors and boundary cases. That needs to come later. These cases represent an initial check that the functionality specified by the use cases is present.

Some writers would argue for developing test cases like these in place of use cases. Here, they are presented as a vehicle for "fleshing out" the use cases, not as a substitute for them.

Use Case	Function Being Tested	Initial System State	Input	Expected Output
System Startup	System is started when the switch is turned "on"	System is off	Activate the "on" switch	System requests initial cash amount
System Startup	System accepts initial cash amount	System is requesting cash amount	Enter a legitimate amount	System is on
System Startup	Connection to the bank is established	System has just been turned on	Perform a legitimate inquiry transaction	System output should demonstrate that a connection has been established to the Bank
System Shutdown	System is shut down when the switch is turned "off"	System is on and not servicing a customer	Activate the "off" switch	System is off
System Shutdown	Connection to the Bank is terminated when the system is shut down	System has just been shut down		Verify from the bank side that a connection to the ATM no longer exists

Session	System reads a customer's ATM card	System is on and not servicing a customer	Insert a readable card	Card is accepted; System asks for entry of PIN
Session	System rejects an unreadable card	System is on and not servicing a customer	Insert an unreadable card	Card is ejected; System displays an error screen; System is ready to start a new session
Session	System accepts customer's PIN	System is asking for entry of PIN	Enter a PIN	System displays a menu of transaction types
Session	System allows customer to perform a transaction	System is displaying menu of transaction types	Perform a transaction	System asks whether customer wants another transaction
Session	System allows multiple transactions in one session	System is asking whether customer wants another transaction	Answer yes	System displays a menu of transaction types
Session	Session ends when customer chooses not to do another transaction	System is asking whether customer wants another transaction	Answer no	System ejects card and is ready to start a new session
Transaction	Individual types of transaction will be tested below			
Transaction	System handles an invalid PIN properly	A readable card has been entered	Enter an incorrect PIN and then attempt a	The Invalid PIN Extension is performed

			transaction	
Withdrawal	System asks customer to choose an account to withdraw from	Menu of transaction types is being displayed	Choose Withdrawal transaction	System displays a menu of account types
Withdrawal	System asks customer to choose a dollar amount to withdraw	Menu of account types is being displayed	Choose checking account	System displays a menu of possible withdrawal amounts
Withdrawal	System performs a legitimate withdrawal transaction properly	System is displaying the menu of withdrawal amounts	Choose an amount that the system currently has and which is not greater than the account balance	System dispenses this amount of cash; System prints a correct receipt showing amount and correct updated balance; System records transaction correctly in the log (showing both message to the bank and approval back)
Withdrawal	System verifies that it has sufficient cash on hand to fulfill the request	System has been started up with less than the maximum withdrawal amount in cash on hand; System is requesting a withdrawal	Choose an amount greater than what the system currently has	System displays an appropriate message and asks customer to choose a different amount

		amount		
Withdrawal	System verifies that customer's balance is sufficient to fulfill the request	System is requesting a withdrawal ammount	Choose an amount that the system currently has but which is greater than the account balance	System displays an appropriate message and offers customer the option of choosing to do another transaction or not.
Withdrawal	A withdrawal transaction can be cancelled by the customer any time prior to choosing the dollar amount	System is displaying menu of account types	Press "Cancel" key	System displays an appropriate message and offers customer the option of choosing to do another transaction or not.
Withdrawal	A withdrawal transaction can be cancelled by the customer any time prior to choosing the dollar amount	System is displaying menu of dollar amounts	Press "Cancel" key	System displays an appropriate message and offers customer the option of choosing to do another transaction or not.
Deposit	System asks customer to choose an account to deposit to	Menu of transaction types is being displayed	Choose Deposit transaction	System displays a menu of account types
Deposit	System asks customer to enter a dollar amount to deposit	Menu of account types is being displayed	Choose checking account	System displays a request for the customer to type a dollar amount
Deposit	System asks	System is	Enter a legitimate	System requests that

	customer to insert an envelope	displaying a request for the customer to type a dollar amount	dollar amount	customer insert an envelope
Deposit	System performs a legitimate deposit transaction properly	System is requesting that customer insert an envelope	Insert an envelope	System accepts envelope; System prints a correct receipt showing amount and correct updated balance; System records transaction correctly in the log (showing message to the bank, approval back, and acceptance of the envelope)
Deposit	A deposit transaction can be cancelled by the customer any time prior to inserting an envelope	System is displaying menu of account types	Press "Cancel" key	System displays an appropriate message and offers customer the option of choosing to do another transaction or not.
Deposit	A deposit transaction can be cancelled by the customer any time prior to inserting an envelope	System is requesting customer to enter a dollar amount	Press "Cancel" key	System displays an appropriate message and offers customer the option of choosing to do another transaction or not.

Deposit	A deposit transaction can be cancelled by the customer any time prior to inserting an envelope	System is requesting customer to insert an envelope	Press "Cancel" key	System displays an appropriate message and offers customer the option of choosing to do another transaction or not.
Deposit	A deposit transaction is cancelled automatically if an envelope is not inserted within a reasonable time	System is requesting customer to insert an envelope	Wait for the request to time out	System displays an appropriate message and offers customer the option of choosing to do another transaction or not.
Transfer	System asks customer to choose an account to transfer from	Menu of transaction types is being displayed	Choose Transfer transaction	System displays a menu of account types specifying transfer from
Transfer	System asks customer to choose an account to transfer to	Menu of account types to transfer from is being displayed	Choose checking account	System displays a menu of account types specifying transfer to
Transfer	System asks customer to enter a dollar amount to transfer	Menu of account types to transfer to is being displayed	Choose savings account	System displays a request for the customer to type a dollar amount
Transfer	System performs a legitimate transfer transaction properly	System is displaying a request for the customer to type a dollar amount	Enter a legitimate dollar amount	System prints a correct receipt showing amount and correct updated balance;

				System records transaction correctly in the log (showing both message to the bank and approval back)
Transfer	A transfer transaction can be cancelled by the customer any time prior to entering dollar amount	System is displaying menu of account types specifying transfer from	Press "Cancel" key	System displays an appropriate message and offers customer the option of choosing to do another transaction or not.
Transfer	A transfer transaction can be cancelled by the customer any time prior to entering dollar amount	System is displaying menu of account types specifying transfer to	Press "Cancel" key	System displays an appropriate message and offers customer the option of choosing to do another transaction or not.
Transfer	A transfer transaction can be cancelled by the customer any time prior to entering dollar amount	System is requesting customer to enter a dollar amount	Press "Cancel" key	System displays an appropriate message and offers customer the option of choosing to do another transaction or not.
Inquiry	System asks customer to choose an account to inquire about	Menu of transaction types is being displayed	Choose Inquiry transaction	System displays a menu of account types

Inquiry	System performs a legitimate inquiry transaction properly	System is displaying menu of account types	Choose checking account	System prints a correct receipt showing correct balance; System records transaction correctly in the log (showing both message to the bank and approval back)
Inquiry	An inquiry transaction can be cancelled by the customer any time prior to choosing an account	System is displaying menu of account types	Press "Cancel" key	System displays an appropriate message and offers customer the option of choosing to do another transaction or not.
Invalid PIN Extension	Customer is asked to re-enter PIN		Enter an incorrect PIN; Attempt an inquiry transaction on the customer's checking account	Customer is asked to re-enter PIN
Invalid PIN Extension	Correct re-entry of PIN is accepted	Request to re-enter PIN is being displayed	Enter correct PIN	Original transaction completes successfully
Invalid PIN Extension	A correctly re-entered PIN is used for subsequent transactions	An incorrect PIN has been re-entered and transaction completed	Perform another transaction	This transaction completes successfully as well

		normally		
Invalid PIN Extension	Incorrect re-entry of PIN is not accepted	Request to re-enter PIN is being displayed	Enter incorrect PIN	An appropriate message is displayed and re-entry of the PIN is again requested
Invalid PIN Extension	Correct re-entry of PIN on the second try is accepted	Request to re-enter PIN is being displayed	Enter incorrect PIN the first time, then correct PIN the second time	Original transaction completes successfully
Invalid PIN Extension	Correct re-entry of PIN on the third try is accepted	Request to re-enter PIN is being displayed	Enter incorrect PIN the first time and second times, then correct PIN the third time	Original transaction completes successfully
Invalid PIN Extension	Three incorrect re-entries of PIN result in retaining card and aborting transaction	Request to re-enter PIN is being displayed	Enter incorrect PIN three times	An appropriate message is displayed; Card is retained by machine; Session is terminated

UNIT-04/LECTURE-08

Software Evolution

- It is impossible to produce system of any size which do not need to be changed. Once software is put into use, new requirements emerge and existing requirements changes as the business running that software changes.
- Parts of the software may have to be modified to correct errors that are found in operation, improve its performance or other non-functional characteristics.
- All of this means that, after delivery, software systems always evolve in response to demand for change.

Program Evolution Dynamic

- Program evolution dynamic is the study of system change. The majority of work in this area has been carried out by Lehman and Belady. From these studies , they proposed a sets of laws concerning system change.

Law	Description
Continuing change	A program that is used in real-world environment necessarily must change or become progressively less useful in that environment.
Increasing complexity	As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplify the structure.

Law	Description
Large program evolution	Program evolution is self-regulation process. System attributes such as size, time between release and the number of report errors are approximately invariant for each system release
Organizational stability	Over a program' s lifetime, its rate of development is approximately constant and independent of the resources devoted to the system development
Conservation of familiarity	Over the lifetime of system, the incremental change in each release is approximately constant.

Software Evolution Approaches

- There are a number of different strategies for software change.[SOM2004]
  - Software maintenance
  - Architectural transformation
  - Software re-engineering.
- Software maintenance

– Changes to the software are made in response to changed requirements but the fundamental structure of the software remains stable. This is most common approach used to system change.

UNIT-04/LECTURE-09

Maintenance activities[RGPV/June 2014(7)]

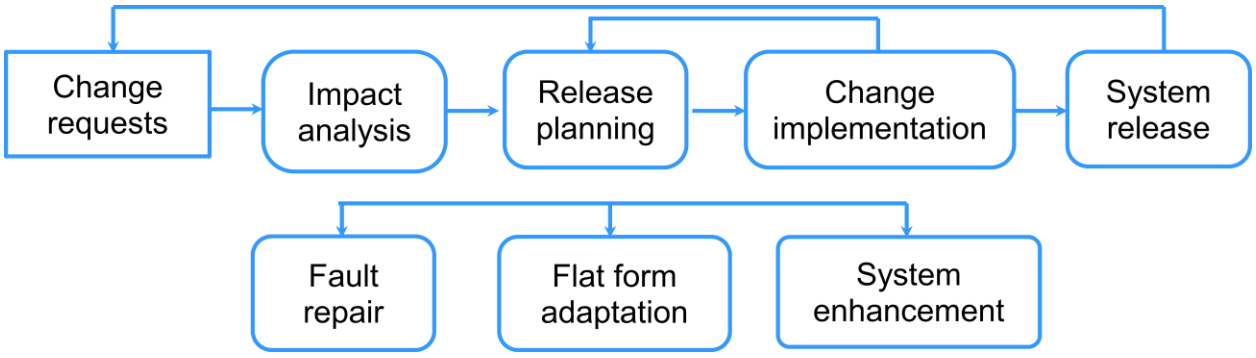
- Software maintenance is the general process of changing a system after it has been diverted.
- The change may be simple changes to correct coding errors, more extensive changes to correct design errors or significant enhancement to correct specification error or accommodate new requirements.

Maintenance Characteristics

- We need to look at maintenance from three different viewpoints: [PRE2004]
  - the activities required to accomplish the maintenance phase and the impact of a software engineering approach (or lack thereof) on the usefulness of such activities
  - the costs associated with the maintenance phase
  - the problems that are frequently encountered when software maintenance is undertaken

The Maintenance Process

- Maintenance process vary considerably depending on the types of software being maintained, the development processes used in an organization and people involved in the process.



Types of Maintenance

Corrective maintenance

Corrective maintenance deals with the repair of faults or defects found in day-today system

functions. A defect can result due to errors in software design, logic and coding. Design errors occur when changes made to the software are incorrect, incomplete, wrongly communicated, or the change request is misunderstood. Logical errors result from invalid tests and conclusions, incorrect implementation of design specifications, faulty logic flow, or incomplete test of data. All these errors, referred to as residual errors, prevent the software from conforming to its agreed specifications. Note that the need for corrective maintenance is usually initiated by bug reports drawn by the users.

In the event of a system failure due to an error, actions are taken to restore the operation of the software system. The approach in corrective maintenance is to locate the original specifications in order to determine what the system was originally designed to do. However, due to pressure from management, the maintenance team sometimes resorts to emergency fixes known as patching. Corrective maintenance accounts for 20% of all the maintenance activities.

Adaptive Maintenance

Adaptive maintenance is the implementation of changes in a part of the system, which has been affected by a change that occurred in some other part of the system. Adaptive maintenance consists of adapting software to changes in the environment such as the hardware or the operating system. The term environment in this context refers to the conditions and the influences which act (from outside) on the system. For example, business rules, work patterns, and government policies have a significant impact on the software system.

For instance, a government policy to use a single 'European currency' will have a significant effect on the software system. An acceptance of this change will require banks in various member countries to make significant changes in their software systems to accommodate this currency. Adaptive maintenance accounts for 25% of all the maintenance activities.

Perfective Maintenance

Perfective maintenance mainly deals with implementing new or changed user requirements. Perfective maintenance involves making functional enhancements to the system in addition to the activities to increase the system's performance even when the changes have not been suggested by faults. This includes enhancing both the function and efficiency of the code and changing the functionalities of the system as per the users' changing needs.

Examples of perfective maintenance include modifying the payroll program to incorporate a new union settlement and adding a new report in the sales analysis system. Perfective maintenance accounts for 50%, that is, the largest of all the maintenance activities.

Preventive Maintenance

Preventive maintenance involves performing activities to prevent the occurrence of errors. It tends to reduce the software complexity thereby improving program understandability and increasing software maintainability. It comprises documentation updating, code optimization, and code restructuring. Documentation updating involves modifying the documents affected by the changes in order to correspond to the present state of the system. Code optimization involves modifying the programs for faster execution or efficient use of storage space. Code restructuring involves transforming the program structure for reducing the complexity in source code and making it easier to understand.

Preventive maintenance is limited to the maintenance organization only and no external requests are acquired for this type of maintenance. Preventive maintenance accounts for only 5% of all the maintenance activities.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Discuss briefly on software maintenance activities & how do you estimate the cost involved. Explain.	Jun.2014	7

REFERENCCE

BOOK	AUTHOR	PRIORITY
Software Engineering	P,S. Pressman	1
Software Engineering	Pankaj jalote	2