Lecture Notes for Lecture 14 of CS 5500 (Foundations of Software Engineering) for the Spring 2021 session at the Northeastern University San Francisco Bay Area Campuses.

Software Testing

Philip Gust,
Clinical Instructor
Department of Computer Science

Information and examples in this lecture are based on recommended readings.

http://www.ccis.northeastern.edu/home/pgust/classes/cs5500/2021/Spring/index.html

Software Testing

Review of Lecture 13

- In this lecture, we introduced the concepts of quality and software quality assurance (SQA), including its processes and characteristics. We also covered approaches to SQA, and metrics, and statistics.
- We learned that SQA and testing are recognized disciplines that easily merit their own courses. This introduction was meant to give software developers an idea of its goals, strategies, and practice.

Software Testing

Introduction

- In this lecture we will study software testing, its goals, methods, and ways to use the results of testing to improve the software development process.
- As we learned in in the previous lecture it is significantly less expensive to prevent errors from occurring during development than detecting and fixing them during testing.
- In the past, the only defense was the skill of individual developers. Modern design techniques and technical reviews are helping to systematically reduce the number of initial errors in code.

Software Testing

Introduction

- However even under ideal conditions, not all errors can be caught during development, so an efficient and thorough testing process is always a necessary part of software development.
- In fact, we learned in the previous lecture finding and fixing errors during testing is far more efficient and less expensive than once the software is deployed.

Software Testing

Introduction

- In the first part of the lecture, we examine software testing strategies that provide a roadmap for steps to be conducted as part of software testing, and the effort and time required.
- In the second part, we will look at some methods used to test software, including specific methods to test different kinds of software, and to identify and track defects and errors.

Testing Strategy
Software Test Plan
Testing Scope
Software Test Coverage

Software Testing

Software Testing Strategies

- The goal of software testing is to discover errors in software that were made when the software was being designed or during its construction.
- To be effective, software testing should focus on identifying the parts of the software most likely to have errors, and to perform tests that are likely to identify them.
- Software testing that is efficient attempts to reduce the amount of time and effort required to identify latent errors in the software being tested.
- Testing that is both effective and efficient requires tests to be carefully planned and systematically conducted, to ensure the greatest number of errors are found at the lowest cost.

Software Testing

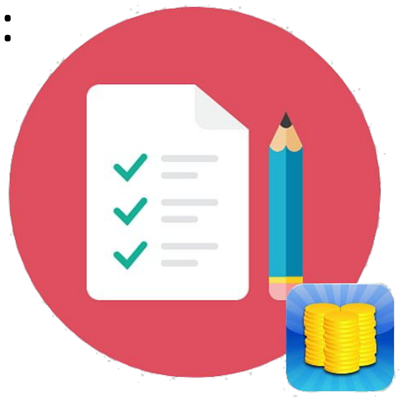
Software Testing Strategies

- A *software testing strategy* describes the planning required to effectively conduct software testing, the steps that should be performed, and the time and resources required.
- The strategy should be
    - flexible enough to promote a customized testing approach,
    - rigid enough to encourage reasonable planning and tracking as projects progress,
    - efficient enough to ensure that testing can be completed with the required level of certainty.

Software Testing

Software Testing Strategies

- The elements of a software testing strategy include:
  - the approach to be taken to conduct the tests
  - the scope of software testing to be performed
  - what will and will not be covered by the tests



Software Testing

Software Testing Strategies

- A *test plan* documents the approach to testing by defining a plan that describes a testing strategy, procedures that define specific testing steps, and the types of tests that will be conducted.
- Reviewing the test plan prior to testing enables the development team to assess the completeness of test cases and testing procedures and tasks.
- An effective test plan will lead to the orderly construction of the software and the discovery of errors at each stage in the construction process.

Software Testing

Software Testing Strategies

- Several software testing strategies are available. All provide a template for testing, and all have these characteristics:
  - To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
  - Testing begins at the component level and works “outward” toward the integration of the entire computer-based system.
  - Different testing techniques are appropriate for different software engineering approaches and at different points in time.
  - Testing is conducted by the developer of the software and (for large projects) an independent test group.

Software Testing

Software Testing Strategies

- A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source code segment has been correctly implemented.
- It must also accommodate high-level tests that validate major system functions against customer requirements.
- A strategy should provide guidance for the practitioner and a set of milestones for the manager.
- Because the steps of the test strategy often occur under the pressure of deadlines, progress must be measurable, and problems should surface as early as possible.

Software Testing

Testing verification and validation

- Software testing is one element of a broader topic that is often referred to as verification and validation.
- *Verification* refers to the set of tasks that ensure that software correctly implements a specific function. “Are we building the product right?”
- *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. “Are we building the right product?”

Software Testing

Testing verification and validation

- Note that there are several lines of thought about what types of testing are covered by the term “validation.”
- One is that all testing is verification, and that validation is conducted when requirements are reviewed and approved, and later, by the user when the system is operational.
- Another holds that unit and integration testing are verification, and that higher-order testing is validation.
- Whatever your line of thought, all these kinds of testing play an important roll in the SQA process.

Software Testing

Testing verification and validation

- Verification and validation includes a wide array of SQA activities. Although testing plays an extremely important role in verification and validation, many other activities are also necessary.
  - technical reviews,
  - quality and configuration audits,
  - performance monitoring,
  - simulation,
  - feasibility study,
  - documentation review,
  - database review,
  - algorithm analysis,
  - development testing,
  - usability testing,
  - qualification testing,
  - acceptance testing
  - installation testing

Software Testing

Testing verification and validation

- Testing provides an opportunity to assess quality and uncover errors, but testing should not be viewed as a safety net.
- As we learned in the last lecture, SQA is crucial throughout the software development process. If it is not there when testing starts, it will not be there when it finishes either.
- The proper application of methods and tools, effective technical reviews, and good management and measurement, and process improvement, all lead to quality that is affirmed during testing.
- The underlying motivation of program testing is to affirm software quality with methods that can be effectively and efficiently applied to any software system, no matter what the scale.

Software Testing

Software test organization

- As software testing begins, there is a kind of conflict of interest. The people who built the software now must test it. One might reasonably argue that no one knows the software better.
- However, developers naturally also have a vested interest in showing that their code is error-free, works according to customer requirements, and is completed on-time and on-budget.
- As well intentioned as the developers are, these interests mitigate against thorough testing.

Software Testing

Software test organization

- Psychologists describe software analysis and design (as well as coding) as *constructive tasks*. The developer analyzes, models, and then creates a computer program and documentation.
- The developer is naturally proud of his or her work. But the goal of the testing process is to uncover errors in what the developer has built – to “break” it, which is subconsciously seen as *destructive*.
- Consequently, the developer tends to run tests that confirm that the software works, rather to uncover errors. Psychologists refer to this as “confirmation bias.”
- If errors are present in the software, the customer rather than the developer will find them.

Software Testing

Software test organization

- One might reasonably draw several incorrect conclusions based on this discussion:
  - developers of software should not test their own software
  - the software should be “tossed over the wall” to strangers who will test it more thoroughly
  - testers should get involved with the project only when testing is about to begin

Software Testing

Software test organization

- A developer is responsible for testing her or his individual units (components) of the program, ensuring that each unit performs the function or exhibits the behavior for which it was designed.
- Often, the developer also performs integration testing—a step that ultimately leads to the completion and testing of the final software system.
- Once the software system is complete, then an independent test group (ITG) becomes involved.

Software Testing

Software test organization

- The role of an *independent test group (ITG)* is to remove the problems associated with letting developers test the software they have built.
- Independent testing removes conflict of interest and confirmation bias that may otherwise be present. The ITG personnel are paid to find errors.
- However, developers can not just turn the software over to ITG and walk away. The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted.
- While the ITG is conducting the testing, the developer must be available to correct errors that are uncovered.

Software Testing

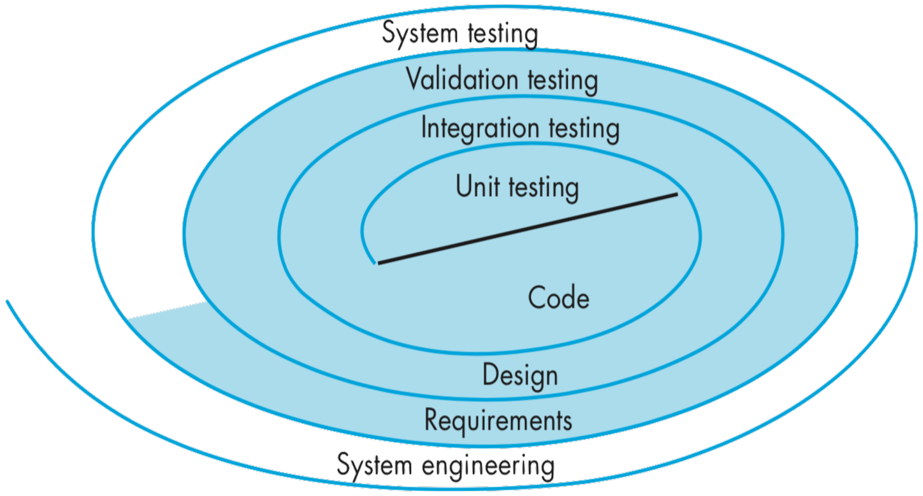
Software test organization

- The ITG is part of the software development project team. It becomes involved during analysis and design and stays involved by planning and specifying test procedures throughout a large project.
- In many cases the ITG reports to the SQA organization, achieving a degree of independence that might not be possible if it were a part of the software engineering team.

Software Testing

High-Level Perspective

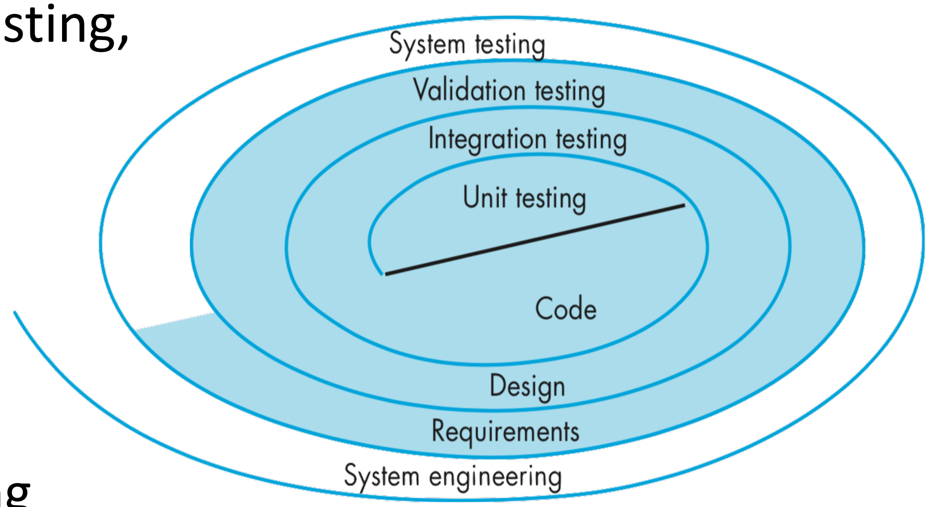
- The software process follows a spiral. Initially, the team defines the role of their software, leading to a software requirements analysis.
- Once that is complete, the information domain, function, behavior, performance, constraints, and validation criteria for the software are established.
- Moving inward along the spiral, you come to design and finally, to coding. As you spiral inward along the streamlines, that decreases the level of abstraction on each turn.



Software Testing

High-Level Perspective

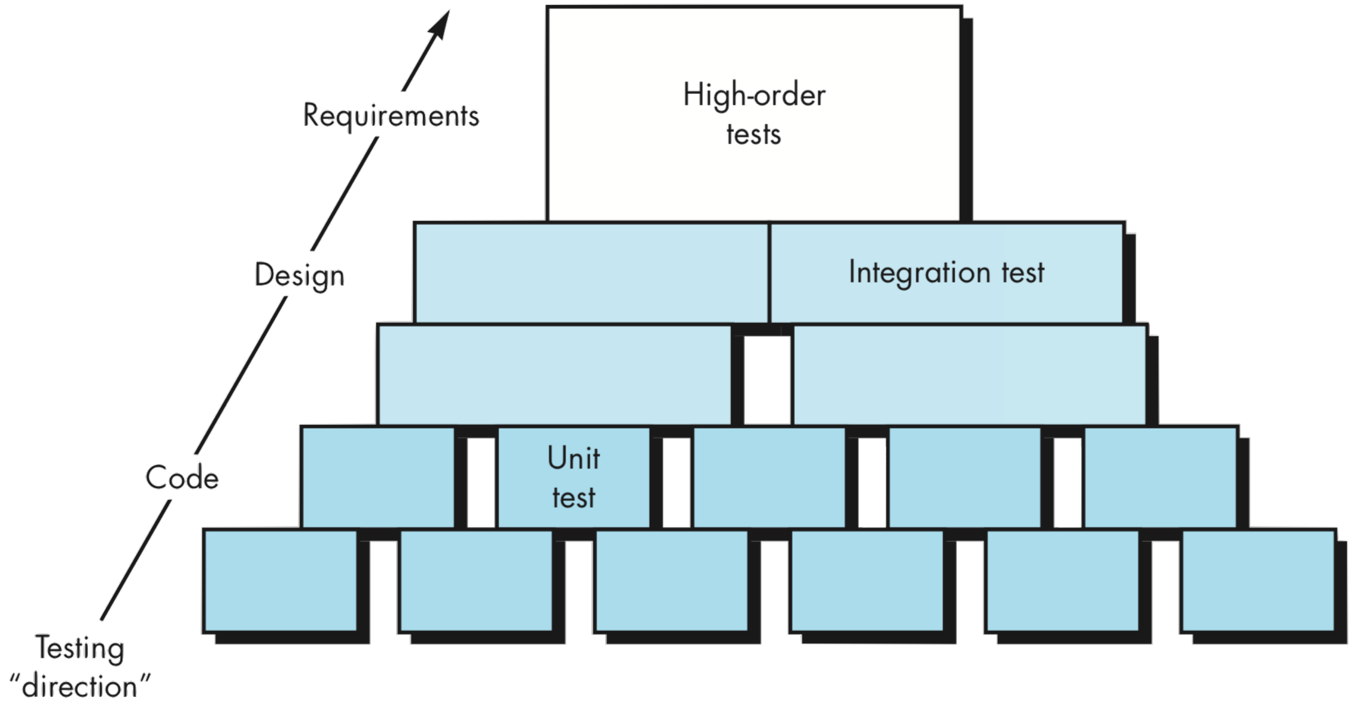
- A strategy for software testing also follows the spiral. Unit testing begins at the vortex with each unit of the software source code.
- Testing progresses outward along the spiral to integration testing, focusing is on design and the construction of the software system.
- The next turn is validation testing, where requirements from requirements modeling are validated against software.
- Finally, you arrive at system testing, where the software system elements are tested as a whole. Spiraling out along streamlines broadens the scope of testing with each turn.



Software Testing

High-Level Perspective

- Testing in the context of software engineering is a series of four steps that are implemented sequentially.



Software Testing

High-Level Perspective

- ***Unit testing.*** The initial series of tests focus on each component individually, ensuring that it functions properly as a unit. That is why this level is know as unit testing.
- Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection.
- The developer who executes the unit tests gets immediate feedback on the unit test results for the component, and any changes that resulted in errors being detected.
- Unit testing is often integrated with the software check-in process, so that the the corresponding unit tests are automatically run when a new components is checked-in or existing one is modified.

Software Testing

High-Level Perspective

- ***Integration testing.*** Integration testing addresses the issues associated with the dual problems of verification and program construction.
- Test-case design techniques that focus on inputs and outputs are more prevalent during the the process where lower-level components are integrated into high-level subsystems.
- Techniques that exercise specific program paths may also be used to ensure coverage of major control paths.

Software Testing

High-Level Perspective

- ***Higher-order testing.*** After the software has been integrated, a set of high-order tests is conducted. Validation criteria (established during requirements analysis) must be evaluated.
- Validation testing provides final assurance that software meets all functional, behavioral, and performance requirements.
- ***Systems testing.*** The final testing step falls outside the boundary of software engineering, into the context of system engineering.
- Software, once validated, must be combined with other elements (e.g., hardware, databases). Systems testing verifies that overall system function/performance is achieved.

Software Testing

Test completion criteria

- The classic question in software testing is: “When are we done testing—how do we know that we’ve tested enough?”
- There is no simple answer to this question, but there are a few pragmatic responses and early attempts at empirical guidance.
- One response to the question is: “You're never done testing; the burden simply shifts from the software engineer to the end user.”
- Every time the user executes a computer program, the program is being tested. This sobering fact underlines the importance of other software quality assurance activities.
- Another somewhat cynical but accurate response is: “You are done testing when you run out of time, or you run out of money.”

Software Testing

Test completion criteria

- Although these responses are ones that a practitioner might give, more rigorous criteria are needed for determining when sufficient testing has been conducted.
- The *cleanroom software engineering* approach suggests statistical techniques that execute tests derived from a statistical sample of all program executions by all users from a targeted population.
- Collecting metrics during software testing and making use of existing statistical models, it is possible to develop guidelines for answering the question: “When are we done testing?”

Software Testing

Strategic issues

- Software testing strategy will succeed only when software testers:
  - specify product requirements in a quantifiable manner long before testing commences
  - state testing objectives explicitly
  - understand the users of the software and develop a profile for each user category
  - develop a testing plan that emphasizes “rapid cycle testing”
  - build “robust” software that is designed to test itself (the concept of “antibugging”)
  - use effective technical reviews as a filter prior to testing
  - conduct technical reviews to assess the test strategy and test cases themselves
  - develop a continuous improvement approach for the testing process.

Software Testing

Strategic issues

- ***Rapid cycle testing*** is a technique that tests software in rapid cycles (2 percent of project effort) of customer-useful, at least field “triable” increments and/or quality improvements.
- The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategy.

Software Testing

Testing Fundamentals

- Now that we have examined software testing strategies, we will look how to execute the test plan by performing tests derived from the plan, and how to identify, classify, and track defects.
- Reviews and other SQA activities can and do uncover errors, but they are not sufficient. Every time the program is executed, the customer is testing it.
- That is why it is important to to run the program before it gets to the customer, with the intent of finding and removing all errors.
- To find the highest possible number of errors, systematic tests must be conducted systematically, and test cases must be designed using disciplined techniques.

Software Testing

Testing Fundamentals

- The goal of software testing is to design a series of test cases that have a high likelihood of finding errors. That’s where software testing techniques enter the picture.
- These techniques provide systematic guidance for designing tests that will
    - exercise the internal logic and interfaces of every software component
    - exercise the input and output domains of the program to uncover errors in program function, behavior, and performance.

Software Testing

Testability

- A good test is one that has a high probability of finding an error. Therefore, you should design and implement a computer-based system or product with *testability* in mind.
- At the same time, the tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.
- We will adopt a definition for testability provided by software testing authority James Bach:
  - Software testability is simply how easily a computer program can be tested.

Software Testing

Testing Fundamentals

- Test-case design focuses on techniques for creating test cases that meet overall testing objectives and the testing strategies described in the test plan.
- A set of test cases is designed to exercise internal logic, interfaces, component collaborations, and external requirements, expected results are defined, and actual results are recorded.

Software Testing

Testability

- The following are important characteristics for testable software.
  - ***Operability***. "The better it works, the more efficiently it can be tested."
    - If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests, allowing testing to progress without fits and starts.

Software Testing

Testability

- The following are important characteristics for testable software.
  - ***Observability.*** "What you see is what you test."
    - Inputs provided as part of testing produce distinct outputs.
    - System states and variables are visible or queryable during execution.
    - Incorrect output is easily identified.
    - Internal errors are automatically detected and reported.
    - Source code is accessible.

Software Testing

Testability

- The following are important characteristics for testable software.
  - ***Controllability.*** "The better we can control the software, the more the testing can be automated and optimized."
    - All possible outputs can be generated through some combination of input, and I/O formats are consistent and structured.
    - All code is executable through some combination of input.
    - Software and hardware states and variables can be controlled directly by the test engineer.
    - Tests can be conveniently specified, automated, and reproduced.

Software Testing

Testability

- The following are important characteristics for testable software.
  - ***Decomposability.*** "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."
    - The software system is built from independent modules that can be tested independently.

Software Testing

Testability

- The following are important characteristics for testable software.
  - ***Simplicity.*** “The less there is to test, the more quickly we can test it.” The program should exhibit
    - *functional simplicity* (e.g., the feature set is the minimum necessary to meet requirements);
    - *structural simplicity* (e.g., architecture is modularized to limit the propagation of faults)
    - *code simplicity* (e.g., a coding standard is adopted for ease of inspection and maintenance).

Software Testing

Testability

- The following are important characteristics for testable software.
  - ***Stability.*** "The fewer the changes, the fewer the disruptions to testing."
    - Changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests.
    - The software recovers well from failures.

Software Testing

Testability

- The following are important characteristics for testable software.
    - ***Understandability.*** "The more information we have, the smarter we will test."
        - The architectural design and the dependencies between internal, external, and shared components are well understood.
        - Technical documentation is instantly accessible, well organized, specific and detailed, and accurate.
        - Changes to the design are communicated to testers.

Software Testing

Test characteristics

- Some of the attributes of a “good” test include:
  - *A good test has a high probability of finding an error.* The tester must understand the software and ways that the software might fail.
  - *A good test is not redundant.* Every test should have a different purpose (even if it is subtly different).
  - *A good test should be “best of breed”*. In a group of tests that have a similar intent, execute only those tests that has the highest likelihood of uncovering a whole class of errors.
  - *A good test should be neither too simple nor too complex.* In general, each test should be executed separately.

Software Testing

Internal and external views

- A software product can be tested in one of two ways:
  - Knowing the specified functions that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while searching for errors in each function.
  - Knowing the internals of a product, tests can be conducted to ensure internal operations are performed according to specifications and that internal components have been adequately exercised.
- The first test approach takes an external view and is called *black-box* (or *functional*) testing. The second requires an internal view and is called *white-box* (or *structural*) testing.

Software Testing

Internal and external views

- Black-box testing alludes to tests that are conducted at the software interface.
    - A black-box test examines fundamental aspect of a system without regard for the internal logical structure of the software.
    - Using black-box testing ensures that all externally defined functional aspects of the product have been tested.

Software Testing

Internal and external views

- White-box testing of software is predicated on close examination of procedural detail.
  - Logical paths through the software and collaborations between components are tested by exercising specific sets of conditions and/or loops.
  - Using white-box testing methods, you can derive test cases that
    - guarantee that all independent paths within a module have been exercised at least once
    - exercise all logical decisions on their true and false sides
    - execute all loops at their boundaries and within their operational bounds
    - exercise internal data structures to ensure their validity.

Software Testing

Internal and external views

- At first glance, white-box testing should lead to 100 percent correct programs. It simply requires defining all logical paths, developing tests to exercise them, and evaluating results.
- Unfortunately, exhaustive testing presents logistical problems. For even small programs, the number of logical paths can be very large.
- For example, consider a 100-line program with 2 nested loops that execute from 1-20 times each, and 4 if-then-else statements.
- This program has ~10^{14} paths. If one test case can be generated, run, and evaluated every 1ms, it would require ~3170 years to exhaustively test the program.

Software Testing

Internal and external views

- However, white-box testing can select and exercise just a limited number of important logical paths, and probe important data structures for validity.
- The software can provide *test points* for this purpose that provide access to key internal state through interfaces that are only available for testing.
- Test points are commonly provided in hardware designs but are not commonly incorporated into the design of software. This technique is critical for designing to support white-box testing.

Software Testing

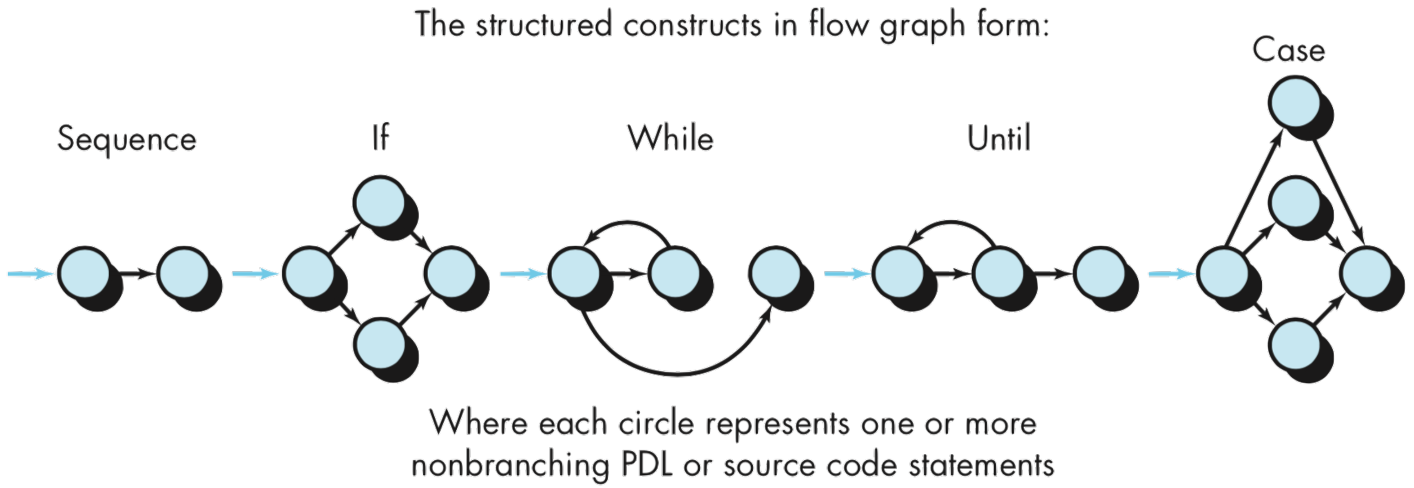
Basis path testing

- *Basis path testing* is a white-box testing technique first proposed in 1976 by software testing expert Tom McCabe.
- The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.
- Basis path testing uses a graphical notation called control flow graph (CFG) notation. This notation is used in many areas of computer science. We will briefly look at it next.

Software Testing

Control flow graph notation

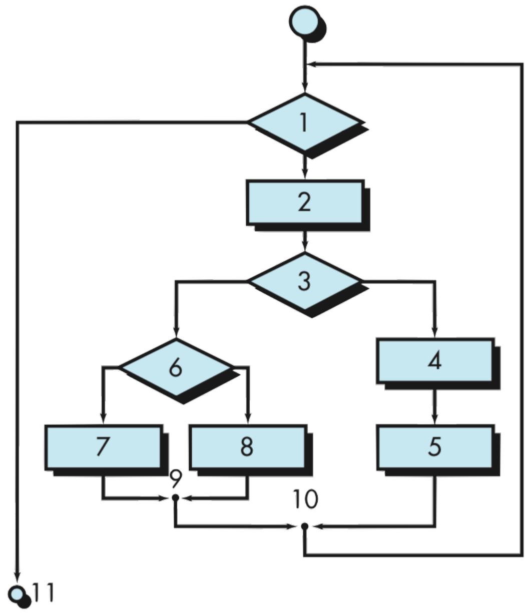
- The control flow graph (CFG) depicts logical control flow using a graphical notation. Each structured construct has a corresponding CFG symbol.


The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

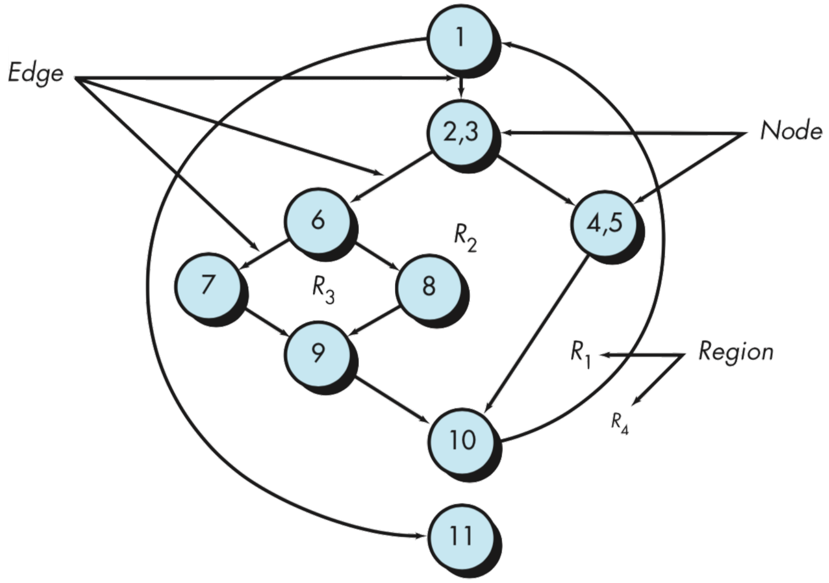
Software Testing

Control flow graph notation

- To illustrate a CFG, the illustration on the left is a *program design language (PDL)* flowchart that depicts a program control structure. The one on the right maps the flowchart into a corresponding CFG.



flowchart



control flow graph

Software Testing

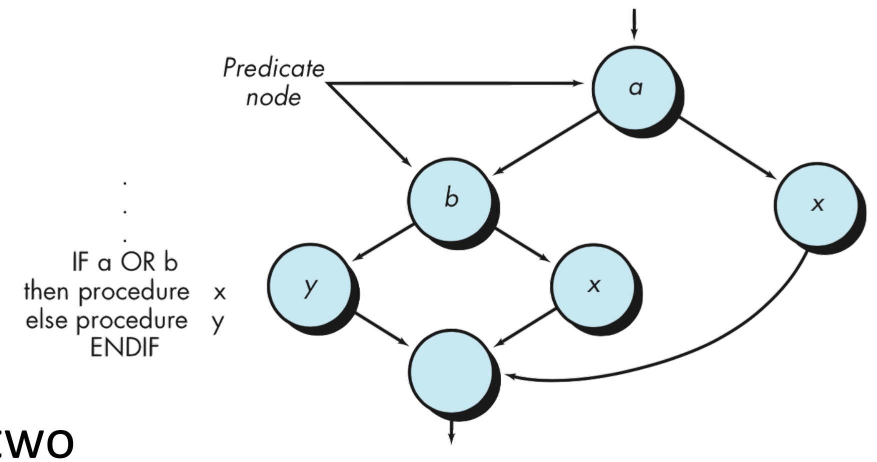
Control flow graph notation

- Each circle, called a *flow graph node*, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node.
- The arrows on the flow graph, called *edges* or *links*, represent flow of control and are analogous to flowchart arrows.
- An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the flow graph symbol for the if-then-else construct).
- Areas bounded by edges and nodes are called *regions*. When counting regions, we include the area outside the graph as a region.

Software Testing

Control flow graph notation

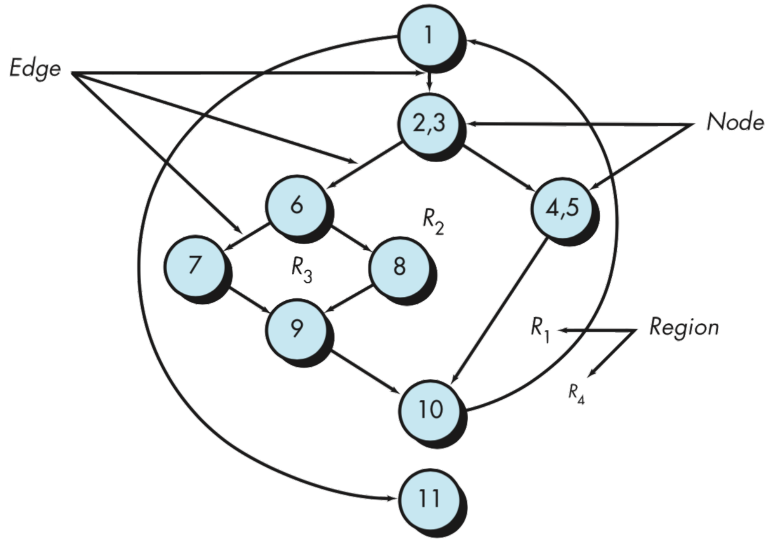
- When compound conditions occur in a procedural design, the flow graph is more complicated. A compound condition occurs when one or more Boolean operators is present.
- The program design language (PDL) segment translates into the flow graph shown.
- A separate node is created for each of the conditions in the statement IF a OR b.
- Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it.



Software Testing

Independent program path

- An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition.
- When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined.
- For example, a set of independent paths for this flow graph is
  - Path 1: 1-11
  - Path 2: 1-2-3-4-5-10-1-11
  - Path 3: 1-2-3-6-8-9-10-1-11
  - Path 4: 1-2-3-6-7-9-10-1-11



Software Testing

Independent program path

- The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not an independent path because it is simply a combination of already specified paths and does not traverse any new edges.
- Paths 1 through 4 constitute a *basis set* for the CFG.
- If you design tests to force execution of these paths (a basis set), every statement in the program is guaranteed to be executed at least once and every condition executed on its **true** and **false** sides.
- The basis set is not unique. In fact, several different basis sets can be derived for a given procedural design.

Software Testing

Independent program path

- How many paths are in a basis set for a CFG? The answer is given by computing cyclomatic complexity of the CFG.
- *Cyclomatic complexity* is a software metric that provides a quantitative measure of the logical complexity of a program.
- When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program.
- It also provides you with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

Software Testing

Independent program path

- Cyclomatic complexity has a foundation in graph theory and provides you with an extremely useful software metric. Complexity is computed in one of three ways:
  - The number of regions *R* of the flow graph corresponds to the cyclomatic complexity.
    - $V(G) = | R |$
  - Cyclomatic complexity *V(G)* for a CFG, *G* is defined as
    - $V(G) = E - N + 2$

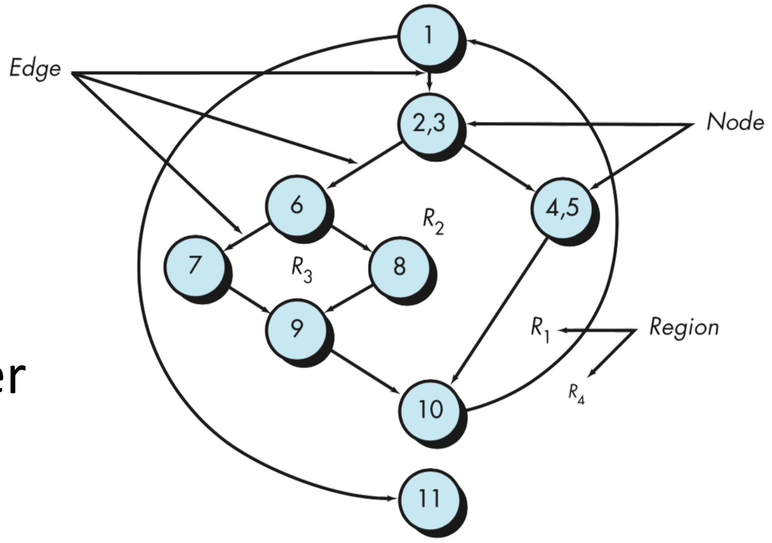
    where *E* is the number of CFG edges and *N* is the number of nodes.
  - Cyclomatic complexity V(G) for a CFG *G* is also defined as
    - $V(G) = P + 1$

    where *P* is the number of predicate nodes contained in the CFG *G*.

Software Testing

Independent program path

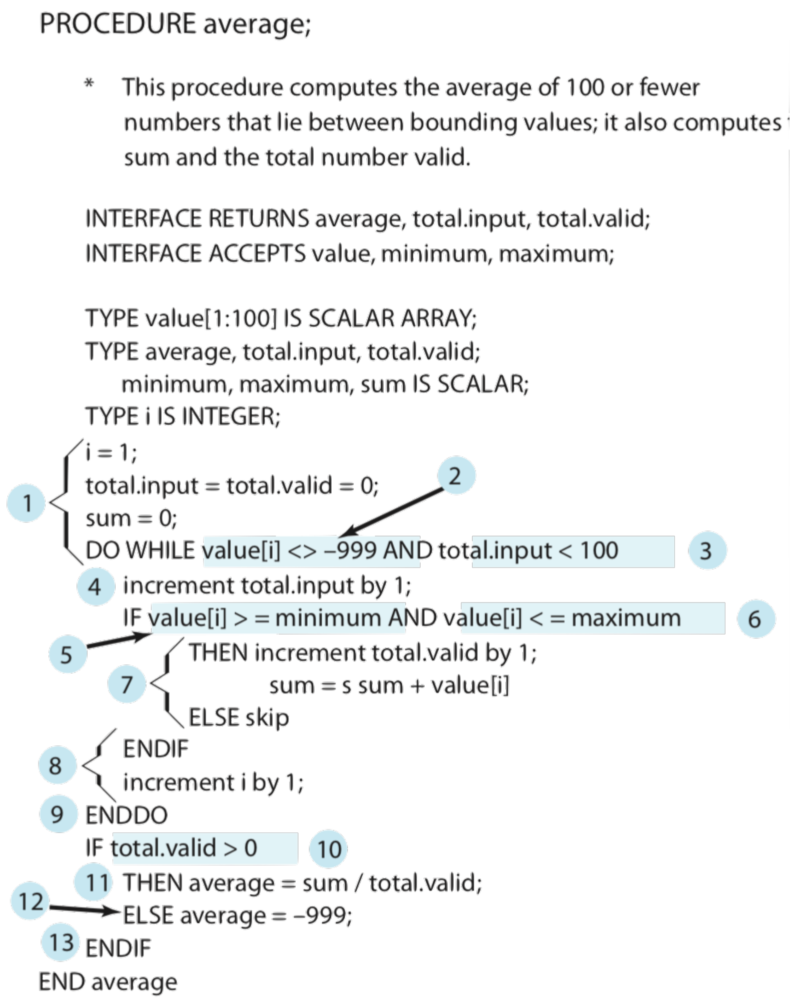
- The cyclomatic complexity of this CFG can be computed using each of the algorithms just noted:
  - The flow graph has four regions, R_1, R_2, R_3, R_4 so *V(G)* = 4
  - *V(G)* = 11 edges - 9 nodes + 2 = 4.
  - *V(G)* = 3 predicate nodes + 1 = 4.
- V(G) is a useful metric. It provides an upper bound for the number of tests to ensure all statements have been executed at least once.
- It also predicts those modules that are likely to be error prone. However a component with a low *V(G)* can also be error-prone.



Software Testing

Deriving test cases

- The basis path testing method can be applied to a procedural design or to source code. We will use the procedure *average*, depicted in PDL, as an example to illustrate each step in the test case design method.
- The first step is to number the PDL statements that will be mapped into corresponding flow graph nodes.

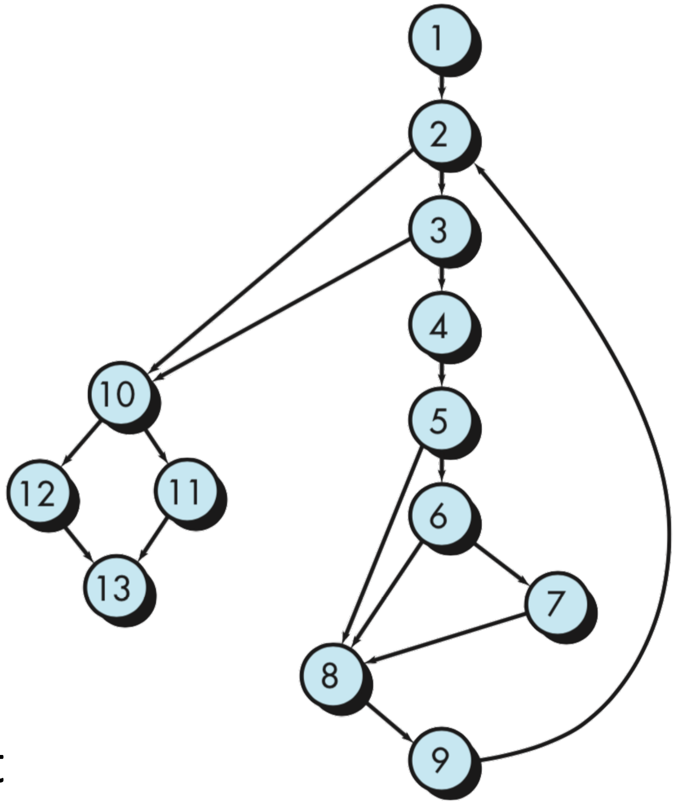


```
PROCEDURE average;

    *   This procedure computes the average of 100 or fewer
        numbers that lie between bounding values; it also computes
        sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
        minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
     ( i = 1;
  1 {  total.input = total.valid = 0;          2
     ( sum = 0;
       DO WHILE value[i] <> –999 AND total.input < 100     3
  4      increment total.input by 1;
         IF value[i] > = minimum AND value[i] < = maximum     6
  5 →
    7 {    THEN increment total.valid by 1;
                  sum = s sum + value[i]
           ELSE skip
  8 {    ENDIF
         increment i by 1;
  9    ENDDO
       IF total.valid > 0     10
  11     THEN average = sum / total.valid;
  12 →   ELSE average = –999;
  13   ENDIF
END average
```

Software Testing

Deriving test cases

- The next step is to create a CFG using the numbers in the PDL to create the graph.
- Now we determine cyclomatic complexity *V(G)* by applying the three formulae we saw earlier:
  - *V(G)* = 6 regions
  - *V(G)* = 17 edges - 13 nodes + 2 = 6
  - *V(G)* = 5 predicate nodes + 1 = 6
- Note that *V(G)* can be obtained without developing a CFG by counting conditional statements in the PDL (for procedure *average*, compound conditions count as two) and adding 1.


$$\begin{tikzcd}
 & & & 1 \arrow[d] & \\
 & & & 2 \arrow[d] \arrow[dlll] & \\
 & & & 3 \arrow[d] \arrow[dlll] & \\
10 \arrow[d] \arrow[dr] & & & 4 \arrow[d] & \\
12 \arrow[d] & 11 \arrow[dl] & & 5 \arrow[d] \arrow[ddl] & \\
13 & & & 6 \arrow[dl] \arrow[dr] & \\
 & & 8 \arrow[dr] & & 7 \arrow[ll] \\
 & & & 9 \arrow[uuuuuu, bend right] &
\end{tikzcd}$$

Software Testing

Deriving test cases

- The value of V(G) provides the number of linearly independent paths through the program control structure. In the case of procedure *average*, we expect a basis set of six paths:
    - Path 1: 1-2-10-11-13
    - Path 2: 1-2-10-12-13
    - Path 3: 1-2-3-10-11-13
    - Path 4: 1-2-3-4-5-8-9-2-. . .
    - Path 5: 1-2-3-4-5-6-8-9-2-. . .
    - Path 6: 1-2-3-4-5-6-7-8-9-2-. . .
- Ellipsis (. . .) following paths 4, 5, and 6 indicates any path through the remainder of the control structure is acceptable.
- It is useful to identify predicate nodes to aid in deriving test cases. In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes.

Software Testing

Deriving test cases

- Finally, we prepare test cases that force execution of each path in the basis set. Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested.
- Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.
- Some paths (e.g., Path 1) cannot be tested stand-alone. The data required to traverse the path cannot normally be achieved. In such cases, these paths are tested as part of another path test.
- Note that it is possibly to mechanically derive V(G) from source code or PDL and compute the basis paths for path basis testing.

Software Testing

Black box testing

- Black-box testing, also called behavioral testing or functional testing, focuses on the functional requirements of the software.
- Input conditions can be derived that will fully exercise all functional requirements for a program.
- Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

Software Testing

Black box testing

- Black-box testing attempts to find errors in the following categories:
  - incorrect or missing functions
  - interface errors
  - errors in data structures or external database access
  - behavior or performance errors
  - initialization and termination errors

Software Testing

Black box testing

- Because black-box testing purposely disregards control structure, attention is focused on the information domain. Tests are designed to answer the following questions:
    - How is functional validity tested?
    - How are system behavior and performance tested?
    - What classes of input will make good test cases?
    - Is the system particularly sensitive to certain input values?
    - How are the boundaries of a data class isolated?
    - What data rates and data volume can the system tolerate?
    - What effect will specific combinations of data have on system operation?

Software Testing

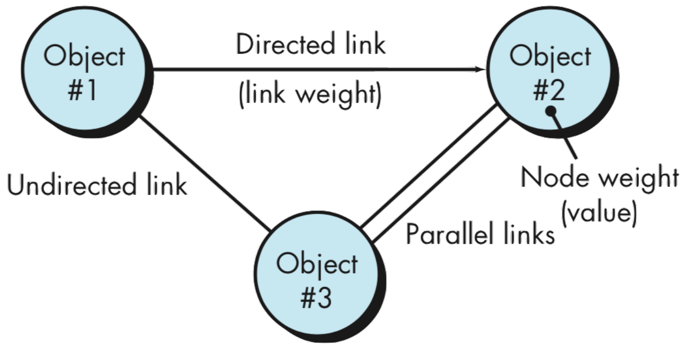
Black box testing

- Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing.
- By applying black-box methods, you derive a set of test cases that satisfy the following criteria:
  - test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing
  - test cases that tell you something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand

Software Testing

Black box testing methods

- There are several common black box testing methods.
  - **Graph-based testing models** use graph representations of “objects” being tested to record the relationships among the models. Then tests are created that verify all “objects” have expected relationships.
  - The graph represents the relationships between data objects and program objects, enabling us to derive test cases that search for errors associated with these relationships.



Software Testing

Black box testing methods

- There are several common black box testing methods.
    - **Equivalence partitioning** divides the input domain of a program into classes of data from which test cases can be derived.
    - An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of character data) that might otherwise require many test cases to be executed before the general error is observed.
    - Test-case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition.
    - If a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present.
    - By the guidelines for the derivation of equivalence classes, test cases for each input domain, data item can be developed so the largest number of attributes of an equivalence class are exercised at once.

Software Testing

Errors and Defects

- Although modern software development processes are designed to minimize errors, they will occur. Testing is an opportunity to discover errors and correct them before software is deployed.
- An *error* is a mistake made by a developer while creating the code. Some common classes of errors include:
    - ***user interface errors***: appear during user interaction
    - ***error handling errors:*** occur when error condition is not handled
    - ***syntactic errors:*** misspelled words, incorrect grammar
    - **calculation errors:** due to bad logic, incorrect formula wrong type
    - ***flow control errors:*** passing control in the incorrect direction
    - ***test errors:*** errors while creating or executing a test

Software Testing

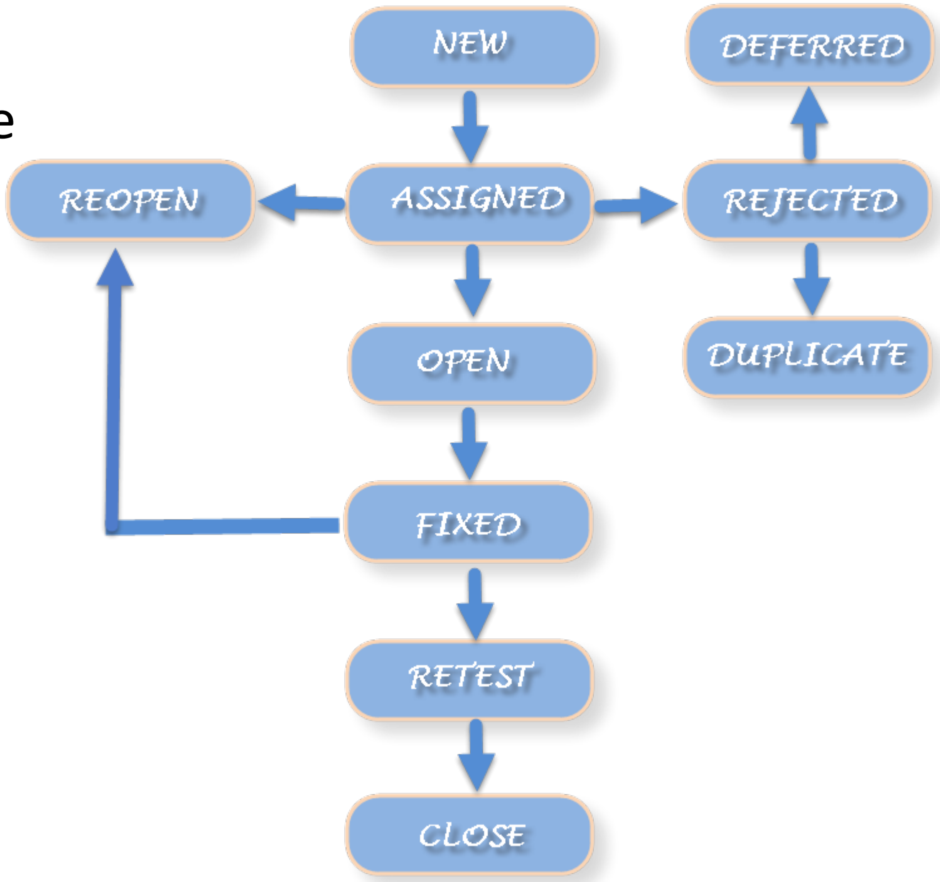
Errors and Defects

- A *defect* is an observed variance between expected and actual results as a result of an error made while designing, creating or integrating software.
- A defect reflects the inability or inefficiency to comply with the specified requirements and criteria and, subsequently, prevents the software from performing the desired and expected work.
- Defects discovered during testing must first be characterized and recorded. An online defect tracking system is used for this purpose. Atlassian’s Jira is an example of issue management software.

Software Testing

Errors and Defects

- Defect management software enables tracking a defect in development, testing, and deployment phases of a product.
- The *defect life cycle* is the journey of a defect during its lifetime. The lifecycle varies by organization and by project as it is governed by the software testing process.



Software Testing

Errors and Defects

- A typical defect lifecycle includes the following stages.
  - ***New***: When a defect is logged and posted for the first time. Its state is given as new.
  - ***Assigned***: Once the defect is posted by the tester, the lead of the tester approves the defect and assigns the defect to developer team.
  - ***Open***: Its state when the developer starts analyzing and working on the defect fix.
  - ***Fixed***: When developer makes necessary code changes and verifies the changes then he/she can make bug status as 'Fixed'.
  - **Retest**: At this stage the tester do the retesting of the changed code which developer has given to him to check whether it was fixed
  - **Reopened**: If the defect still exists even after it is fixed by the developer, the tester changes the status to "reopened". The defect goes through the life cycle once again.

Software Testing

Errors and Defects

- A typical defect lifecycle includes the following stages.
  - ***Deferred***: The defect, changed to deferred state means it is expected to be fixed in next releases. Reasons for this state have many factors: priority may be low, lack of time for the release or no major effect.
  - ***Rejected***: If the developer feels that the defect is not genuine, the developer rejects it. The state is changed to "rejected".
  - ***Duplicate*** : If the defect is repeated or the two bugs mention the same concept, the recent/latest bug status is changed to "duplicate".
  - ***Closed***: Once the defect is fixed, it is tested by the tester. If the tester feels that it no longer exists in the software, tester changes the status to "closed". This state means that it is fixed, tested and approved.
  - ***Not a bug/Enhancement***: The state given if there is no change in the functionality of the application. For an example: If customer asks for a change in behavior, it is an enhancement request, not a defect.

Software Testing

Errors and Defects

- Once a defect has been characterized and recorded, it must now be classified. Three dimensions typically used to characterize a defect:
  - ***Severity***: defines the degree of impact
  - **Risk**: defines how likely it is to occur
  - **Priority**: defines how important it is to address
- We will look at each of these dimensions, and then discuss how they are typically applied.

Software Testing

Errors and Defects

- ***Severity Basis***. Severity reflects the degree or intensity of a defect to adversely impact a software product or its operation.
  - **Critical***:* Requires immediate attention and treatment. Directly affects the essential functionalities which can otherwise affect a software product or its large-scale functionality. For instance, failure of a feature/functionality.
  - **Major**: Affects the main functions of a software product. Although it does not result in failure of system, may disrupt use of primary functions.
  - **Minor**: Produces less impact and has no significant influence on the software product. Results are visible in the operation of the product but does not prevent executing the task -- can be carried out using some alternative.
  - **Trivial**: Has no impact on the operation of a product. Sometimes ignored or omitted. For example, spelling or grammatical errors.

Software Testing

Errors and Defects

- ***Risk Basis***. Probability reflects the likelihood of encountering the defect in regular product use.
  - **High**: Almost all users detect the presence of the defect in the application.
  - **Medium**: Some users detect the presence of the defect in the application.
  - **Low**: Hardly any users detects the presence of the defect in the application.

Software Testing

Errors and Defects

- ***Priority Basis***. Defects have a business perspective and dictate that some defects must be addressed sooner than others.
  - **High**: defines the most critical need to correct a defect. Should happen as soon as possible.
  - **Medium**: defines a normal business need to correct the defect. Can wait until next minor release of a product.
  - **Low**: does not need to be corrected individually. Can wait until the next major release of the product.

Software Testing

Triaging defects

- Once issues are created for defects, the project team next needs to determine which to work on, how soon, and in what order.
- Although severity is absolute and can be assigned by a tester, risk, priority, and schedule are more subjective and require discussion within the product team.
- To accomplish this, all or a subset of the product team meets regularly to review defects and enhancement requests. This is often referred to as a *triage meeting* or a *bug council*.

Software Testing

Triaging defects

- In a triage meeting, open issues are divided into categories. The most important distinction is between defects that will not be fixed in this release and those that will be.
- Triaging an issue involves:
  - Making sure the issue has enough information for the developers and makes sense
  - Making sure the issue is filed in the correct place
  - Making sure the issue has sensible "Severity," "Priority,” and “Risk” fields

Software Testing

Triaging defects

- In a triage meeting, the team will assign the priority of the fix based on the business perspective. They will check “How important is it to the business that we fix the bug?”
- In most of the times high severity bug becomes high priority bug, but it is not always. There are some cases where high Severity bugs will be low Priority and low Severity bugs will be high Priority.
- If the schedule draws closer to a release, even if severity is more based on technical perspective, the priority is given as low because the functionality mentioned in the bug is not critical to business.

Software Testing

Triaging defects

- Priority might change over time. Perhaps a bug initially deemed P1 becomes rated as P2 or even a P3 as the schedule draws closer to the release and as the test team finds even more heinous errors.
- Priority is a subjective evaluation of how important an issue is, given other tasks in the queue and the current schedule. It’s relative. It shifts over time. And it’s a business decision.
- By contrast, severity is absolute: it’s an assessment of the impact of the bug without regard to other work in the queue or the current schedule.
- The only reason severity should change is if we have new information that causes us to re-evaluate our assessment. If it was a high severity issue when I entered it, it’s still a high severity issue when it’s deferred to the next release. Only the priority changed.

Software Testing

Triaging defects

- Some people suggest using both severity and priority to produce a composite risk number. This intuitively sounds like a good way to resolve the priority-severity divide.
- Use such an approach with extreme caution. It is multiplying apples by oranges to quantify bananas. Risk is yet a third type of information.
- The risk associated with any bug depends on the likelihood that the user will run into it as well as the possible losses that might occur.

Software Testing

Triaging defects

- The ultimate lesson, regardless of the terms or levels you use to categorize your issues, is that any classification scheme will only be effective if everyone agrees on definitions.
- Perhaps that is the very first question to ask when an argument is brewing about severity, priority, or risk: “Help me understand exactly what information you’re using and how you’re using it?”

Software Testing

Summary

- In the first part of the lecture, we examined software testing strategies that provide a roadmap for steps to be conducted as part of software testing, and the effort and time required.
- In the second part, we looked at some methods used to test software, including specific methods to test different kinds of software, and to identify and track defects and errors.

Testing Strategy
Software Test Plan
Testing Scope
Software Test Coverage

Software Testing

Summary

- The goal of software testing is to discover errors in software that were made when the software was being designed or during its construction.
- To be effective, software testing should focus on identifying the parts of the software most likely to have errors, and to perform tests that are likely to identify them.
- Software testing that is efficient attempts to reduce the amount of time and effort required to identify latent errors in the software being tested.
- Testing that is both effective and efficient requires tests to be carefully planned and systematically conducted, to ensure the greatest number of errors are found at the lowest cost.