Advanced Software Engineering (CS6401)

Autumn Semester (2022-2023)

Dr. Judhistir Mahapatro
Department of Computer Science and
Engineering
National Institute of Technology Rourkela

Coding and Testing

Coding and Testing

- **Coding** is undertaken once the **design phase** is complete and the **design documents** have been successfully reviewed.

- After all the modules of a system have been coded and unit tested
  - The integration and system testing phase is undertaken

Coding and Testing

- Integration and testing of modules is carried out according to an ***integration plan***.

  - Integration of modules through a number of steps

  - During each step, a number of modules added to the partially integrated system and the resultant system is tested

Coding and Testing

- Full product takes shape only after all the modules have been integrated together

- System testing is conducted on the full product.

  - The full product is tested against its requirements as recorded in the SRS document

Coding and Testing

- Testing of professional software is carried out using a large number of test cases.

- Requires maximum effort among all the development phases

- Maximum number of software engineers can be found to be engaged in testing activities

- To reduce the testing time, different test cases can be executed in parallel by different team members

Coding and Testing

- Wrong impression that
  - Testing is a secondary activity
  - It is intellectually not stimulating as the activities associated with other development phases

- The general perception of testing as **monkeys** typing in random data and trying to **crash the system** has changed.

- Now testers are looked upon as masters of specialized concepts, techniques and tools.

Coding

- Design document is an input to the coding phase
  - Not only contains a structure chart, but also the detailed design
  - Detailed document usually documented in the form of module specifications
    - The data structures and algorithms for each module are specified

- In coding phase, modules are coded according their respective module specification

Coding

The objective of the coding phase is to transform the design of a system into code in a high-level language, and then to test this code

Coding

- Programmers to adhere to some well-defined and standard style of coding which is called their **coding standard**

- Advantages of adhering to a standard style of coding :

  - It gives a uniform appearance to the codes written by different engineers

  - It facilitates code understanding and code reuse

  - It promotes good programming practices

Coding

- A coding standard lists rules to be followed during coding
  - For example, the way variables are to be named, the way the code is to be laid out, etc.

- Compliance of their code to coding standards is verified during code inspection, otherwise, it will be rejected during code review

Coding

- **Coding guidelines** are also prescribed by software companies
  - They provide some general suggestions

- Coding standard is **mandatory** whereas leaves the implementation of Coding guideline to the **discretion** of the individual developers.

Coding

- After a module has been coded,
  - Code review is carried out to ensure that the coding standards are followed and detect as many errors as possible before testing

  - Reviews are an efficient way of removing errors from code as compared to removing using testing

Coding standards and Guidelines

- Organizations develop their own coding standards and guidelines
  - depending on what suits them
  - based on the specific types of software they develop

Coding standards

1. **Rules for limiting the use of global**
   - types of data can be declared global and what cannot

Coding standards

2. Standard headers for different modules:

- Header of different module should have standard format and information for ease of understanding and maintenance

  Ex: Name of the module, Date of creation

  Author's name, Modification history,

  Synopsis of the module,

  Different functions supported in module along

  with i/o parameter,

  Global variables accessed/modified by module

Coding standards

3. Naming conventions for global variables, local variables, and constant identifiers

- Global variable would always start with capital letter
- Local variable names would start with small letters
- Constant names should be formed using capital letters only

Coding standards

4. Conventions regarding error return values and exception handling mechanisms

- All functions while encountering an error condition should either return a 0 or 1 consistently
- This facilitates reuse and debugging

coding guidelines

1. Don't use a coding style that is too clever or too difficult to understand:

- easy to understand

- inexperienced engineers write cryptic and
  incomprehensible code

- clever coding can obscure meaning of the code
  and reduce code understandability; thereby
  making maintenance and debugging difficult
  and expensive

Representative coding guidelines

2. Avoid obscure side effects

- Side effect of a function call include modifications to the parameters passed by reference, modification of the global variable, I/O operations
- not obvious from a casual examination of the code
- Example: suppose a value of a global variable is changed or some file I/O is performed obscurely in a called module

Representative coding guidelines

3. Code should be well-documented:

  - There should be at least one comment line on the average for every three source lines of code

4. Length of any function should not exceed 10 source lines

  - difficult to understand

  - large number of variables and carries out many different types of computations

  - lengthy functions are likely to have disproportionately larger number of bugs

Representative coding guidelines

5. Don't use GO TO statement

   - makes program unstructured

   - difficult to understand, debug and maintain

6. Don't use an identifier for multiple purposes

   - use of variable for multiple purposes can lead to
   confusion and make it difficult for somebody trying
   to read and understand the code

   - makes future enhancement more difficult

Code review

- **Code Review** is an effective technique to remove defects from source code
  - **Testing** is an effective defect removal mechanism but applicable to only executable code

- Review techniques have become extremely popular and cost-effective

- Code review does not target to detect syntax errors in a module, but to detect logical, algorithmic and programming errors.

Code review

- Eliminating an error from code involves testing, debugging, and then correcting the errors.
  - Testing is to detect if the system fails to work satisfactorily for certain types of inputs and under certain circumstances.
  - Debugging is carried out to locate an error that causing the failure
    - The most laborious and time consuming activity

Code review: Code Walkthrough

- It is an informal code analysis technique

- Module is taken up for review after it has been coded and compiled successfully

- Members of the development team are given the code a couple of days before the walkthrough meeting

- Each member selects some test cases and simulates the execution of the code by hand and note down their findings for discussion in walkthrough meeting

Code review: Code Walkthrough

- Several guidelines have evolved based on personal experience, common sense, several other subjective factors

  - They should be considered as examples

Guidelines:

- Code walkthrough Team should not be either too big or too small. Ideally, three to seven members.

- Discussions should focus on discovery of errors and avoid on how to fix the discovered errors.

- Managers should not attend the walkthrough meetings, otherwise, coders feel that they are being watched and evaluated.

Code review: Code Inspection

Principal aim:

- To check for the presence of some common types of programming errors that usually creep into code due to programmer mistakes and oversights

- To check whether coding standards have been adhered to.

Code review: Code Inspection

- Beneficial side effects such as programmer receives feedback on programming style, choice of algorithm, and programming techniques.

- Other participants gain by being exposed to another programmer's errors.

- **Example** of the type of errors detected during code inspection, the classic error of writing a procedure that modifies a formal parameter and then calls it with a constant actual parameter. DON'T REQUIRE HAND SIMULATION.

Commonly committed errors during code inspection

- Use of uninitialized variables
- Non-terminating loops
- Jumps into loops
- Incompatible assignments
- Array indices out of bounds
- Improper storage allocation and de-allocation
- Mismatch between actual and formal parameter in procedure calls

- Use of incorrect logical operators or incorrect precedence among operators
- Dangling reference caused when the referenced memory has not been allocated

Testing

- Aim of program testing is to help realize and identify all defects in a program

- After satisfactory completion of the testing phase, not possible to guarantee that a program is error free

Testing

- Input data domain of most programs is very large

- Not possible to test the program exhaustively with respect to each value that the input can assume

- Million testers for million years even cannot exhaustively test a function taking a floating point number as argument

    - each input data takes 1sec to type in

Testing

- Careful testing can expose a large percentage of the defects existing in a program
  - provides a practical way of reducing defects in a system

Basic concepts and terminology

- Testing a program involves
  - Executing the program with a set of test inputs and observing if the program behaves as expected.

  - The input data and the conditions under which it fails are noted for later debugging and error correction

Basic concepts and terminology

- Terminologies have been standardized by the IEEE Standard:

    - A mistake is essentially any programmer action that later shows up as an incorrect result during program execution

    - An error is the result of a mistake committed by a developer in any of the development activities.

        - **Example**: call made to a wrong function

    - Error, fault, bug, and defect are considered to be synonyms in the areas of program testing

Basic concepts and terminology

- A failure of a program essentially denotes an incorrect behavior exhibited by the program during its execution.

  - Every failure is caused by some bugs present in the program

- Some ways in which a program can fail, and randomly selected examples:

  - The result computed by a program is 0, when the correct result is 10

  - A program crashes on an input

  - A robot fails to avoid an obstacle and collides with it

Basic concepts and terminology

- Mere presence of an Error in a program code may not necessarily lead to a failure during its execution

- Program error that may not cause any failure:

int markList[1:10];

int *roll*;

if (*roll* > 0)

markList[*roll*] = mark;

else

markList[*roll*] = 0;

- If *roll* assumes zero or negative then an array index out of type or error would result

- roll value always assigned positive values, then no failure would result.

- does not show up the error even if an error is present

Basic concepts and terminology

- A test case is a triplet [I, S, R], where I is the data input to the program under test, S is the state of the program at which the data is to be input, and R is the result expected to be produced by the program.

- State of a program is also called its *execution mode*
  - Example: Text editor can at any time during its execution assume the following execution modes – edit, view, create and display
  - An example test case is – [**input**: "abc", **state**: edit, **result**: "abc" is displayed], which means that the input "abc" needs to be applied in the edit mode, result "abc" would be displayed

Basic concepts and terminology

- A **test scenario** is an abstract test case in the sense that it only identifies the aspects of the program that are to be tested without identifying the input, state, or output.

  - A test case can be said to be an implementation of a test scenario.

Basic concepts and terminology

- *A **test script** is an encoding of a test case as a short program.*

  - developed for automated execution of the test cases.

- A **positive test case** is designed to test whether the software correctly performs a required functionality.

- A **negative test case is** designed to test whether the software includes something that is not required

- *Example: manager login system*

  - Positive test case: validates a user with the correct username and password

  - Negative test case: admits user with wrong login username and password

Basic concepts and terminology

- A **test suit** is the set of all test that have been designed by a tester to test a given program.

Basic concepts and terminology

- **Testability** of a requirement denotes the extent to which it is possible to determine whether an implementation conforms to both functionality and performance.

- Example: **Two programs implements the same functionality. Which one is more testable?**

  - if it can be adequately tested with less number of

  test cases; A less complex program is more testable

  - A more testable one have a lower structural

  complexity metric such as # of decision statements

Basic concepts and terminology

- A **failure mode** of a software denotes that all failures that have similar observable symptoms

- **Example:** railway ticket booking software

  - *failing to book an available seat*

  - *incorrect seat booking*

  - *system crash*

- **Equivalent faults** denote two or more bugs that result in the system failing in the same failure mode.

  - Example: divide by zero and illegal memory access errors, two are equivalent faults as they both lead to system crash

Basic concepts and terminology

Verification versus validation:

- Both are designed to remove errors in a software. However, bug detection techniques and their applicability are very different

Basic concepts and terminology

- Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase; whereas validation is the process of determining whether a fully developed software conforms to its requirement specification.

  - a verification step to check if the design documents produced after the design step conform to the requirements specification

  - validation step to check if it satisfies the customer's requirement

Cont…

- Techniques used for verification are
  - Informal verification (review, simulation, testing)
  - Formal verification usually involves use of theorem proving techniques or use of automated tools

- Validation techniques are primarily based on product testing

- Testing is categorized under both program verification and validation

Cont…

- Unit Testing and Integration Testing can be considered as verification steps

  - Verified whether the code is as per the module and module interface specifications

- System Testing can be considered as a validation step where it is determined whether the fully developed code is as per its requirements specification

Cont...

- Verification does not require execution of the software, whereas validation requires execution of the software

- Verification techniques can be viewed as an attempt to achieve phase containment of errors.

  - the principle of detecting errors as close to their

    points of commitment as possible is known as

    phase containment of errors.

  - Phase containment of errors can reduce the effort

    required for correcting bugs.

Cont…

- The aim of validation is to check whether the deliverable software is error free.

- Activities involved in these two bug detection techniques together are called "V and V" activities.

  Error detection techniques = verification techniques + validation techniques

- Highly reliable software can be developed using validation techniques alone but the development cost increases drastically.

Testing Activities

- **Test suit design:** The set of test cases using which a program is to be tested is designed possibly using several test case design techniques.

- **Running test cases and checking the results to detect failures:**

  - Each test case is run and the results are compared with the expected results.

    - Test cases for which the system fails are noted down for later debugging
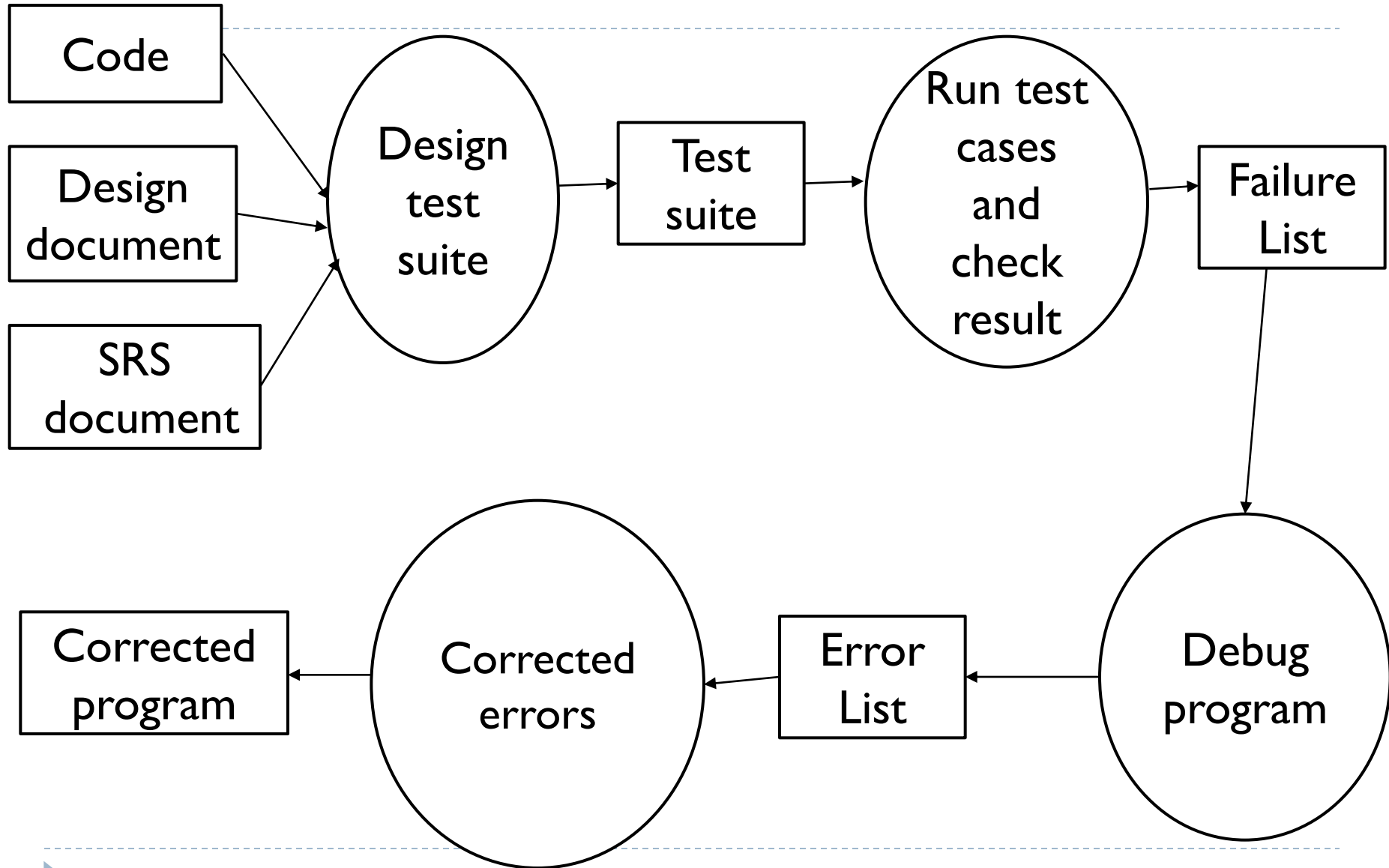
Testing Activities

- **Locate error**
  - The failure symptoms are analyzed to locate the errors.
  - The statement that are in error are identified

- **Error correction**
  - After the error is located during debugging, the code is appropriately changed to correct the error.

Testing Process

Why Design Test Cases?

- Why not be sufficient to test a software using a large number of random input value? Why design test cases?

  - When test cases are designed based on random input data, many of the test cases do not contribute to the significance of the test suite

  - They do not help detect any additional defects not already being detected by other test cases in the suite.

- A large collection of randomly selected test cases does not guarantee that all of the errors in the system will be uncovered

Why Design Test Cases?

- The number of test cases does not indicate of the effectiveness of testing

    if (x>y) max = x;

    else max = x ;

    {(x=3, y=2);(x=2, y=3)} can detect the error

    {(x=3, y=2);(x=4, y=3);(x=5, y=1)} does not detect

    the error

- Test suite should be carefully designed rather than picked randomly

Why Design Test Cases?

- The domain input data values is either extremely large or countably infinite, therefore, to satisfactorily test a software with minimum cost

  - Design a minimal test suite to uncover as many as existing errors.

  - Each test case of a minimal test suite helps detect different errors.

    - This is in contrast to testing using some random input values.

- Essentially two main approaches to systematically design test cases:

  - Black-box approach

  - White-box (or glass-box) approach

Why Design Test Cases?

- In black-box approach, test cases are designed using only the functional specification of the software.

- Test cases are designed based on an analysis of the input/output behavior and does not require any knowledge of the internal structure of a program.

- Black-box testing is also known as functional testing.

- Designing white-box test cases requires a thorough knowledge of the internal structure of a program

- White-box is also known as structural testing.

- These two test case design are complementary to each other.

- One testing using one approach does not substitute testing using the other.

Testing in the Large versus Testing in the Small

- Unit testing is referred to as testing in the small, whereas integration and system testing are referred to as testing in the large.

- There are two reasons for not testing the integrated set of modules once thoroughly?

  - While testing a module, other modules with which this module needs to interface may not be ready. This makes debugging easier.

  - If a failure is detected when an integrated set of modules is being tested, it would be difficult to determine which module exactly has the error.

Testing in the Large versus Testing in the Small

- Unit testing is carried out in the coding phase itself as soon as coding of the module is done.

- Integration and System testing are carried out during the testing phase.

Unit Testing

- Unit testing is undertaken after a module has been coded and reviewed

- Typically undertaken by the coder of the module himself in the coding phase

- Before carrying out unit testing
  - The unit test cases have to be designed
  - The test environment for the unit under test has to be developed

Unit Testing

Driver and stub modules

- To test a single module
  - we need a complete environment to provide all relevant code that is necessary for execution of the module
- The following are needed to test the module:
  - The procedure belonging to other modules that the module under test calls
  - Non-local data structures that the module access
  - A procedure to call the functions of the module under test with appropriate parameters

Unit Testing

- Modules required to provide necessary environment are usually not available until they too have been unit tested.

- *Stubs* and *drivers* are designed to provide the complete environment for a module so that testing can be carried out.

- A stub procedure is a dummy procedure that has the same I/O parameters as the function called by the unit under test but has a highly simplified behavior.
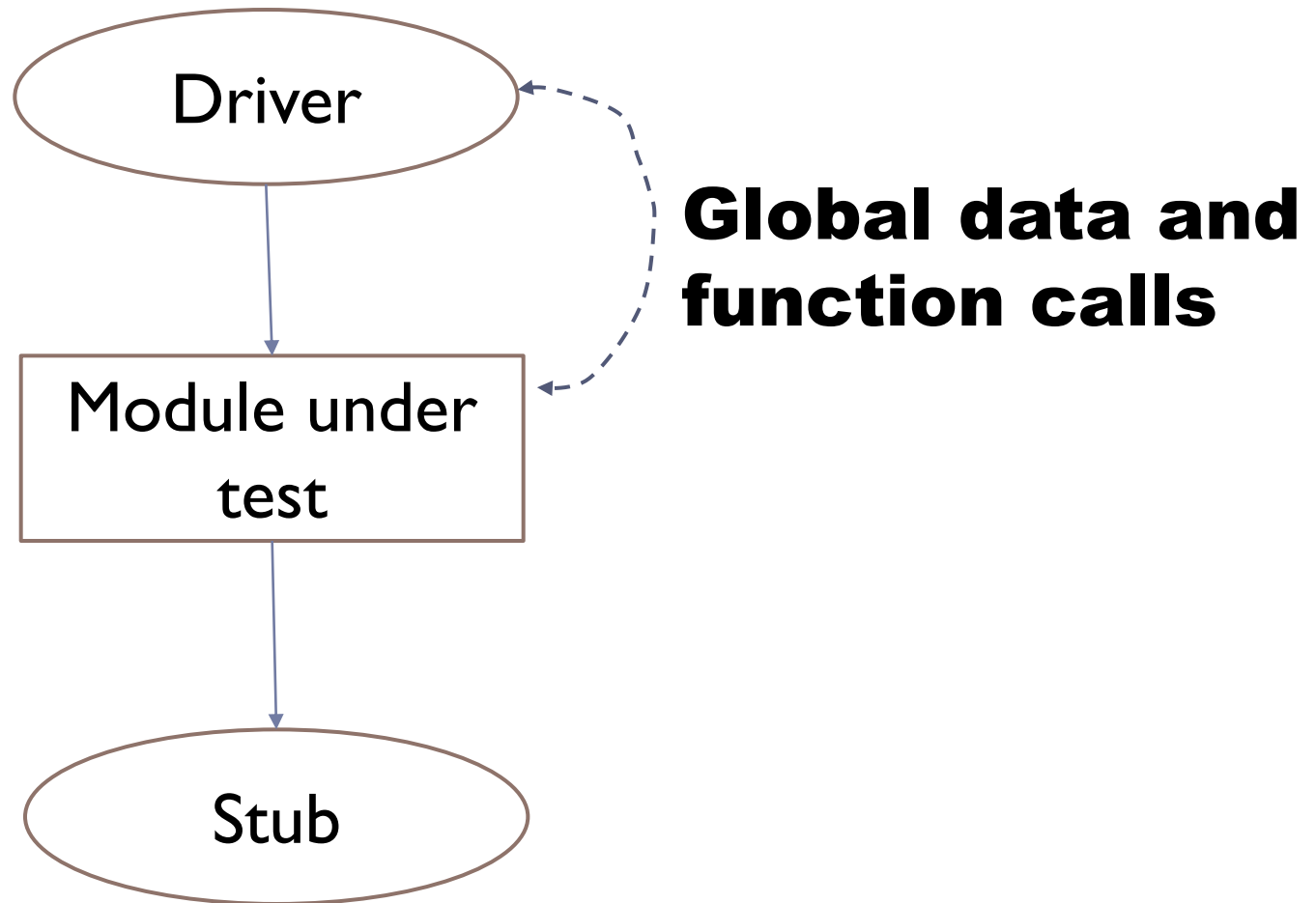
Unit Testing

- Driver module should contain the non-local data structures accessed by the module under test.

- Additionally, it should also have the code to call the different functions of the unit under test with appropriate parameter values.

Unit Testing

Global data and function calls

Unit testing with the help of driver and stub modules

Black-box testing

- Test cases are designed from an examination of the input/output values only
  - no knowledge of design or code is required

Equivalence class partitioning

- Domain of input values to the program under test is partitioned into a set of equivalence classes.

  - Every input data belonging to the same equivalence class, the program behaves similarly.

  - The main idea is that testing the code with any one value belonging to an equivalence class is as good as testing the code with any other value belonging to the same equivalence class

General guidelines for designing equivalence classes

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes need to be defined.

Example: if the equivalence class is the set of integers in the range [1, 10], then invalid equivalence classes are [-∞, 0] and [11, +∞]

General guidelines for designing equivalence classes

2. If input data assumes values from a set of discrete members of some domain, then one equivalence class for the valid input values and another equivalence class for the invalid input values should be defined.

Example: if valid equivalence class is {A, B, C}, then invalid class is U – {A, B, C}, where U is universe of possible inputs

Example: For a software that computes the square root of an input integer that can assume values in the range of 0 and 5000. Determine the equivalence classes and black box test suit.

- Three equivalence classes.
- Possible test suit is {-2, 400, 8000}

Boundary Value Analysis

- Programmers often fail to properly address the special processing required by the input values that lie at the boundary of the different equivalence classes.

  - Example: programmers may improperly use <

    instead of <=, or conversely <= for < etc.

- Boundary value analysis-based test suite design involves designing test cases using the values at the boundaries of different equivalence classes.

- To design boundary value test cases, it is required to examine the equivalence classes to check if any of the equivalence classes contains a range of values.

Cont…

- For those equivalence classes that consists of discrete collection of values, no boundary test cases can be defined

- For an equivalence class that is a range of values, the boundary values need to be included in the test suite.

Example: if an equivalence class contains the integers in the range 1 to 10, then the boundary value test suite is {0, 1, 10, 11}

Cont...

- Steps in the black-box test suite design approach:
  - Examine the input and output values of the program
  - Identify the equivalence classes
  - Design equivalence class test cases by picking one representative value from each equivalence class.
  - Design boundary value test cases.
- Most important step is the identification of the equivalence classes.
- Without practice, one may overlook many equivalence classes in the input data set.

White-Box Testing

- It is an important type of unit testing.

- A number of white-box testing strategies exist.

- Each strategy essentially designs test cases based on analysis of some aspect of source code and is based on some heuristic.

White-Box Testing

A white-box testing strategy can either be coverage-based or fault-based.

Fault-based Testing:

A fault-based testing strategy targets to detect certain types of faults. These faults that a test strategy focuses on constitutes the fault model of the strategy.

   Example: mutation testing

Coverage-based Testing:

It attempts to execute certain elements of a program.

   Examples: statement coverage, branch coverage, and path coverage

White-Box Testing

- Testing criterion for coverage-based testing
  - This strategy typically targets to execute certain program elements for discovering failures.
  - The set of specific program elements that a testing strategy targets to execute is called the testing criterion of the strategy.
- Stronger versus weaker testing
  - The stronger testing strategy covers all program elements covered by the weaker testing strategy, and the stronger strategy covers at least one program element that is not covered by the weaker strategy.

Cont…

- If a stronger testing has been performed, then a weaker testing need not be carried out.

- When none of two testing strategies fully covers the program elements exercised by the other, then the two are called complementary testing strategies.

- Coverage based testing is frequently used to check the quality of testing achieved by a test suite.

- It is hard to manually design a test suite to achieve a specific coverage for a non-trivial program.

Statement coverage

- The statement coverage strategy aims to design test cases so as to execute every statement in a program at least once.

int computeGCD (int x, int y) {

      while (x != y) {

         if (x > y) then

            x = x-y;

         else   y = y – x;

         }

        return x;

      }

> Test set
> {(x=3, y =3), (x=4, y =3), (x=3, y =4)}, all statements of the program would be executed at least once.

Branch Coverage

- A test suite satisfies branch coverage, if it makes each branch condition in the program to assume true and false values in turn.

- For branch coverage each branch in the CFG representation of the program must be taken at least once, when the test suite is executed.

- Branch testing is also known as edge testing.

- Test suite for the previous example

  {(x=3, y =3), (x=3, y =2), (x=4, y =3), (x=3, y =4)}
  achieves branch coverage

- Branch coverage ensures statement coverage.

Path coverage

- A test suite achieves path coverage if it executes each linearly independent paths (or basis paths) at least once.

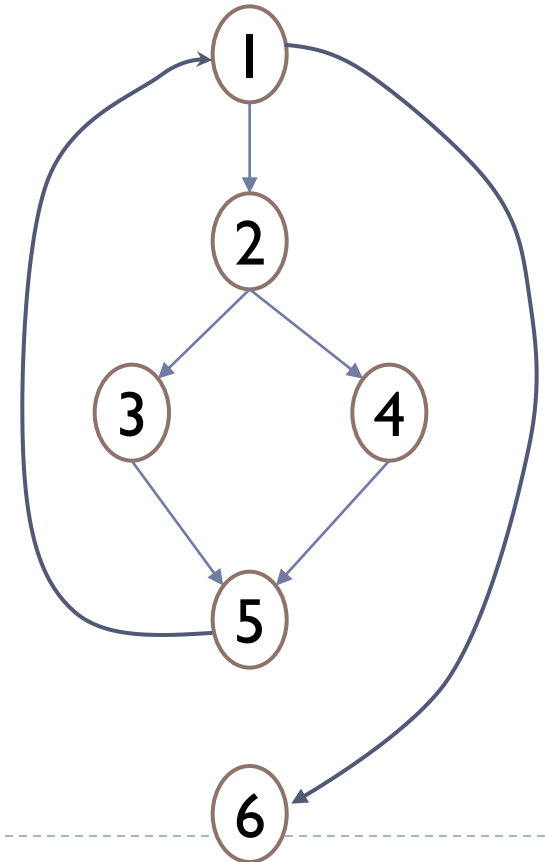- A linearly independent path can be defined in terms of the control flow graph of a program

Path coverage

A control flow graph describes the sequence in which the different instructions of a program get executed

```
int computeGCD (int x, int y)
        {
1          while (x != y) {
2                  if (x > y) then
3                          x = x-y;
4                  else   y  = y – x;
5              }
6          return x;
        }
```

Path coverage

- Path: A path through a program is any node and edge sequence from the start node to a terminal node of the control flow graph of a program.

- A program can have more than one terminal node (ex: multiple exit and return type of statements)

- Test cases to cover all paths is impossible because presence of infinite loops

Path coverage

- Path coverage testing does not try to cover all paths, but only a subset of independent paths.

- Linearly independent set of paths (or basis path set): It is called LIS if each path in the set introduces at least one new edge that is not included in any other path in the set.

- If a set of paths is linearly independent of each other, then no path in the set can be obtained through any linear operations (i.e., additions or subtractions) on the other paths in the set.

Path coverage

- Linearly independent set of paths (or basis path set), for any path in the set, its subpath cannot be a member of the set.

- Arbitrary path of a program can be synthesized by carrying out linear operations on the basis paths.

- It is straight forward to identify the Linearly independent paths for simple programs, for more complex programs it is not easy.

Path coverage

- McCabe's cyclomatic complexity metric is an important result that lets us compute the number of linearly independent paths for any arbitrary program.

- McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program.

McCabe's Cyclomatic Complexity Metric

McCabe obtained his results by applying graph-theoretic techniques to the control flow graph of a program.

Method 1: Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as:

$$V(G) = E - N + 2$$

N is the number of nodes of the control flow graph

E is the number of edges in CFG

McCabe's Cyclomatic Complexity Metric

Method 2: An alternative way of computing the cyclomatic complexity of a program is based on a visual inspection of the CFG is as follows –

$V(G)$ = Total number of non-overlapping bounded areas + 1

In CFG G, any region enclosed by nodes and edges can be called as a bounded area.

McCabe's Cyclomatic Complexity Metric

If the Graph G is not planar

CFG of structured programs always yields planar graphs.

But presence of GOTO's can easily add intersecting edges.

For non-structured programs, this way of computing McCabe's cyclomatic complexity does not apply.

McCabe's Cyclomatic Complexity Metric

Method 3: The cyclomatic complexity of a program can also be easily computed by computing the number of decision and loop statements of the program. If N is the number of decision and loop statements of a program, then McCabe's metric is equal to N+1.

Mutation Testing

- It is a fault-based testing

- Test cases are designed to find specific types of faults in a program.

- In mutation testing, a program is first tested using an initial test suite designed by using various white-box testing strategies.

- After initial testing is complete, mutation testing can be taken up.

Mutation Testing

- Make a few arbitrary changes to a program at a time. Each time the program is changed, called as Mutated program and the change effected is called mutant.

- Underlying assumption is that all programming errors can be expressed as a combination of simple errors.

- Mutation operator makes specific changes to a program.

- For example, one mutation operator may delete a random statement. A mutant may or may not cause an error in the program. If no error is introduced then the mutated program and original program are called equivalent programs.

Mutation Testing

- Mutated program is tested against the test suite of the program.

- If there exists at least one test case for which mutated program yields incorrect result, then mutant is dead. Otherwise it is Alive.

- Test cases are enhanced to kill the mutant.

- Remember that there is a possibility of a mutated program to be an equivalent program. When this is the case, it is futile to try to design a test case that would identify error.

Mutation Testing

Advantages is that it can be automated to a great extent. By certain primitive changes we can generate mutants.

Deleting variable definition, deleting a statement are examples of primitive changes.

Disadvantage is computationally very expensive, since large number of mutation.

Each mutant is tested against full test suite. It is not suitable for manual testing. There are test tools, which automatically generate the mutants.

Integration Testing

- It is carried out after all (or at least some of) the modules have been unit tested.

- The objective of integration testing is to check whether the different modules of a program interface with each other properly.

- Any one (or mixture of) of the following approaches can be used to develop the test plan:

    - Big-bang approach
    - Top-down approach
    - Bottom-up approach
    - Mixed (also called sandwiched) approach

Integration Testing

- Big-bang approach:
    - all modules making up a system are integrated in a single step
    - difficult to localize error as the error may potentially lie in any of the modules
- Top-down approach
    - It starts with the root module in the structure chart and one or two
    - subordinate modules of the root module
    - use of program stubs to simulate the effect of lower-level routines

Integration Testing

- Bottom-up approach
  - first the modules for the each subsystem are integrated
  - integration testing a subsystem is to test whether the interfaces among
  - various modules making up the subsystem work satisfactorily
  - test drivers are required
- Mixed (also called sandwiched) approach
  - combination of Top-down and bottom-up

System Testing

- All units of a program have been integrated together and tested, system testing is undertaken

- System tests are designed to validate a fully developed system to assure that it meets its requirement.

- Test cases are designed based on the SRS document.

System Testing

Three kinds of testing:

- Alpha Testing
  - System testing carried out by the test team with the developing organization.

- Beta Testing
  - Testing performed by a select group of friendly customers.

- Acceptance Testing
  - It is performed by the customer to determine whether to accept the delivery of the system

System Testing

- System Test cases are classified into functionality and performance test cases.

- Before a fully integrated system is accepted for system testing, smoke testing is performed.

Smoke Testing

- Initiated before system testing ensure that system testing would be meaningful or whether many parts of the software would fail.

- A few test cases are designed to test whether the basic functionalities are working.

Performance Testing

- Addresses non-functional requirements.

  - May sometimes involve testing hardware and software together.

  - There are several categories of performance testing.

Stress Testing

- Stress testing (also called endurance testing):

  - Impose abnormal input to stress the capabilities of the software.

  - Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity.

Stress Testing

- If the requirements is to handle a specified number of users, or devices:
  - Stress testing evaluates system performance when all users or devices are busy simultaneously.

Stress Testing

- If an operating system is supposed to support 15 multiprogrammed jobs,

  - The system is stressed by attempting to run 15 or more jobs simultaneously.

- A real-time system might be tested

  - To determine the effect of simultaneous arrival of several high-priority interrupts.

Stress Testing

- Stress testing usually involves an element of time or size,

    - Such as the number of records transferred per unit time,

    - The maximum number of users active at any time, input data size, etc.

- Therefore stress testing may not be applicable to many types of systems.

Volume Testing

- Addresses handling large amounts of data in the system:

  - Whether data structures (e.g. queues, stacks, arrays, etc.) are large enough to handle all possible situations.

  - Fields, records, and files are stressed to check if their size can accommodate all possible data volumes.

  - Example: Symbol Table in Compiler

Configuration Testing

- Analyze system behaviour:

  - in various hardware and software configurations specified in the requirements

  - sometimes systems are built in various configurations for different users

  - for instance, a minimal system may serve a single user,

    - other configurations for additional users.

Compatibility Testing

- These tests are needed when the system interfaces with other systems:
  - Check whether the interface functions as required.

Compatibility testing Example

- If a system is to communicate with a large database system to retrieve information:
  - A compatibility test examines speed and accuracy of retrieval.

Recovery Testing

- These tests check response to:

  - Presence of faults or to the loss of data, power, devices, or services

  - Subject to loss of system resources

    - Check if the system recovers properly.

Maintenance Testing

- Diagnostic tools and procedures:
    - help find source of problems.
    - It may be required to supply
        - memory maps
        - diagnostic programs
        - traces of transactions,
        - circuit diagrams, etc.

Maintenance Testing

- Verify that:
    - all required artifacts for maintenance exist
    - they function properly

Documentation tests

- Check that required documents exist and are consistent:

  - user guides,

  - maintenance guides,

  - technical documents

Documentation tests

- Sometimes requirements specify:
    - Format and audience of specific documents
    - Documents are evaluated for compliance

Latent Errors: How Many Errors are Still Remaining?

- Make a few arbitrary changes to the program:
  - Artificial errors are seeded into the program.
  - Check how many of the seeded errors are detected during testing.

Error Seeding

Let:

- N be the total number of errors in the system
- n of these errors be found by testing.
- S be the total number of seeded errors,
- s of the seeded errors be found during testing.

Error Seeding

n/N = s/S

N = S (n/s)

remaining defects:

$$N - n = n ((S - s)/ s)$$

Error Seeding

- The kinds of seeded errors should match closely with existing errors:

  - However, it is difficult to predict the types of errors that exist.

- Categories of remaining errors:

  - Can be estimated by analysing historical data from similar projects.

What is regression testing?

- Regression testing is testing done to check that a system update does not cause new errors or re-introduce errors that have been corrected earlier.
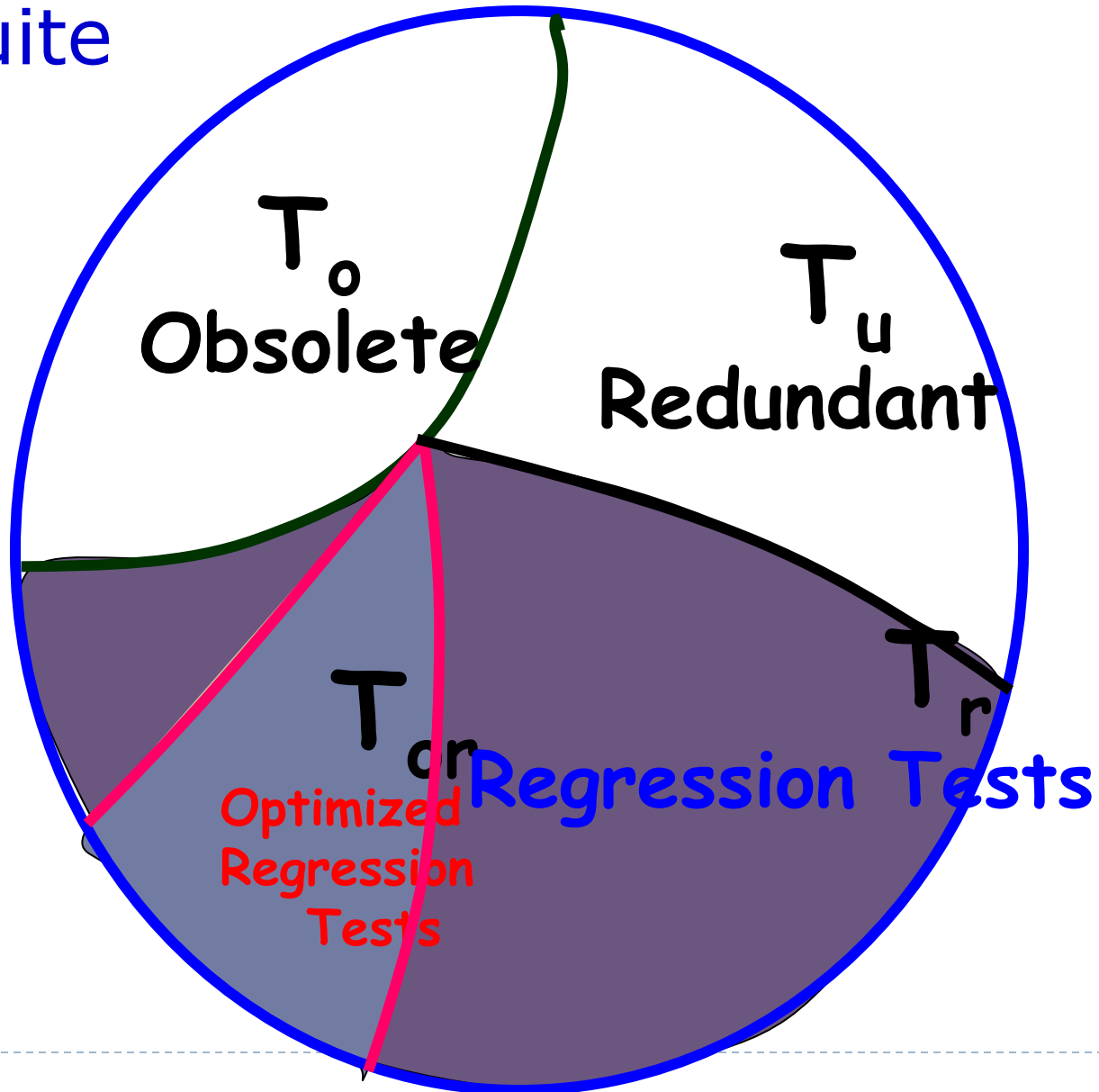
Need for Regression Testing

- Any system during use undergoes frequent code changes.

  - Corrective, Adaptive, and Perfective changes.

- Regression testing needed after every change:

  - Ensures unchanged features continue to work fine.

Partitions of an Existing Test Suite

Major Regression Testing Tasks

Test revalidation (RTV):

- Check which tests remain valid

Test selection (RTS):

- Identify tests that execute modified portions.

Test minimization (RTM):

- Remove redundant tests.

Test prioritization (RTP):

- Prioritize tests based on certain criteria.

Automating regression testing