Testing and Debugging

Abhishek Bhattacharya,IEM

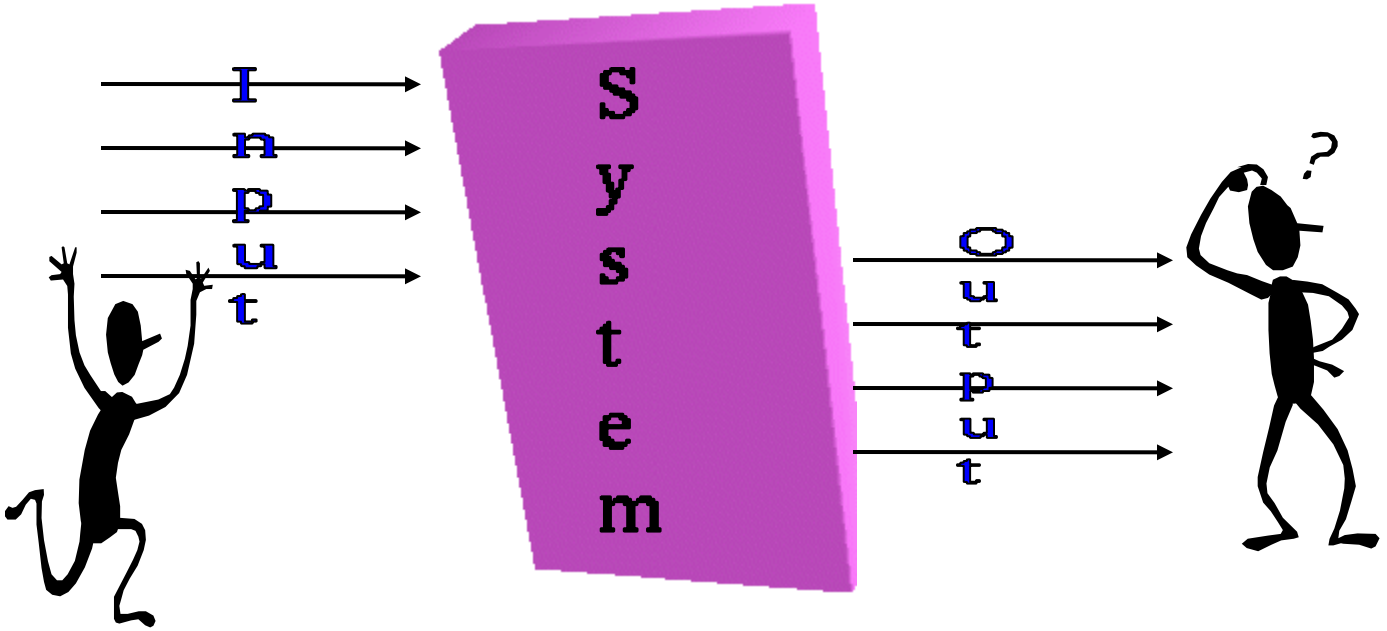
Organization of this lecture

- Important concepts in program testing
- Black-box testing:
  - equivalence partitioning
  - boundary value analysis
- White-box testing
- Debugging
- Unit, Integration, and System testing
- Summary

How do you test a program?

- Input test data to the program.
- Observe the output:
  - Check if the program behaved as expected.

How do you test a system?



How do you test a system?

- If the program does not behave as expected:
  - note the conditions under which it failed.
  - later debug and correct.

Error, Faults, and Failures

- A failure is a manifestation of an error (aka defect or bug).
  - mere presence of an error may not lead to a failure.

Error, Faults, and Failures

- A fault is an incorrect state entered during program execution:
  - a variable value is different from what it should be.
  - A fault may or may not not lead to a failure.

Test cases and Test suites

- Test a software using a set of carefully designed test cases:
  - the set of all test cases is called the test suite

Test cases and Test suites

- A test case is a triplet [I,S,O]
  - I is the data to be input to the system,
  - S is the state of the system at which the data will be input,
  - O is the expected output of the system.

Verification versus Validation

- Verification is the process of determining:
  - whether output of one phase of development conforms to its previous phase.
- Validation is the process of determining
  - whether a fully developed system conforms to its SRS document.

Verification versus Validation

- Verification is concerned with phase containment of errors,
  - whereas the aim of validation is that the final product be error free.

Design of Test Cases

- Exhaustive testing of any non-trivial system is impractical:
  - input data domain is extremely large.
- Design an optimal test suite:
  - of reasonable size and
  - uncovers as many errors as possible.

Design of Test Cases

- If test cases are selected randomly:
  - many test cases would not contribute to the significance of the test suite,
  - would not detect errors not already being detected by other test cases in the suite.
- Number of test cases in a randomly selected test suite:
  - not an indication of effectiveness of testing.

Design of Test Cases

- Testing a system using a large number of randomly selected test cases:
  - does not mean that many errors in the system will be uncovered.
- Consider an example for finding the maximum of two integers x and y.

Design of Test Cases

- The code has a simple programming error:
- If (x>y) max = x;
  else max = x;
- test suite {(x=3,y=2);(x=2,y=3)} can detect the error,
- a larger test suite {(x=3,y=2);(x=4,y=3); (x=5,y=1)} does not detect the error.

Design of Test Cases

- Systematic approaches are required to design an optimal test suite:
  - each test case in the suite should detect different errors.

Design of Test Cases

- There are essentially two main approaches to design test cases:
  - Black-box approach
  - White-box (or glass-box) approach

Black-box Testing

- Test cases are designed using only functional specification of the software:
  - without any knowledge of the internal structure of the software.
- For this reason, black-box testing is also known as functional testing.

White-box Testing

- Designing white-box test cases:
  - requires knowledge about the internal structure of software.
  - white-box testing is also called structural testing.

Black-box Testing

- There are essentially two main approaches to design black box test cases:
  - Equivalence class partitioning
  - Boundary value analysis

Equivalence Class Partitioning

- Input values to a program are partitioned into equivalence classes.
- Partitioning is done such that:
  - program behaves in similar ways to every input value belonging to an equivalence class.

Why define equivalence classes?

- Test the code with just one representative value from each equivalence class:
  - as good as testing using any other values from the equivalence classes.

Equivalence Class Partitioning

- How do you determine the equivalence classes?
  - examine the input data.
  - few general guidelines for determining the equivalence classes can be given

Equivalence Class Partitioning

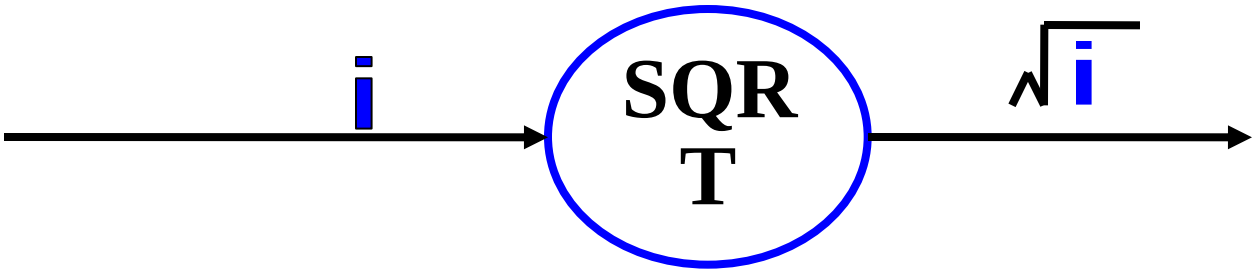
- If the input data to the program is specified by a range of values:
  - e.g. numbers between 1 to 5000.
  - one valid and two invalid equivalence classes are defined.

Equivalence Class Partitioning

- If input is an enumerated set of values:
  - e.g. {a,b,c}
  - one equivalence class for valid input values
  - another equivalence class for invalid input values should be defined.

Example

- A program reads an input value in the range of 1 and 5000:
  - computes the square root of the input number

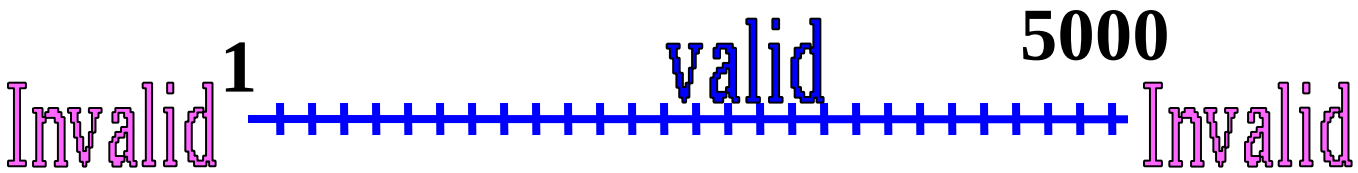


Example (cont.)

- There are three equivalence classes:
  - the set of negative integers,
  - set of integers in the range of 1 and 5000,
  - integers larger than 5000.

Example (cont.)

- The test suite must include:
  - representatives from each of the three equivalence classes:
  - a possible test suite can be: {-5,500,6000}.



Boundary Value Analysis

- Some typical programming errors occur:
  - at boundaries of equivalence classes
  - might be purely due to psychological factors.
- Programmers often fail to see:
  - special processing required at the boundaries of equivalence classes.

Boundary Value Analysis

- Programmers may improperly use < instead of <=
- Boundary value analysis:
  - select test cases at the boundaries of different equivalence classes.

Example

- For a function that computes the square root of an integer in the range of 1 and 5000:
  - test cases must include the values: {0,1,5000,5001}.

White-Box Testing

- There exist several popular white-box testing methodologies:
  - Statement coverage
  - branch coverage
  - path coverage
  - condition coverage
  - mutation testing
  - data flow-based testing

Statement Coverage

- Statement coverage methodology:
  - design test cases so that
    - every statement in a program is executed at least once.

Statement Coverage

- The principal idea:
  - unless a statement is executed,
  - we have no way of knowing if an error exists in that statement.

Statement coverage criterion

- Based on the observation:
  - an error in a program can not be discovered:
    - unless the part of the program containing the error is executed.

Statement coverage criterion

- Observing that a statement behaves properly for one input value:
  - no guarantee that it will behave correctly for all input values.

Example

```
int f1(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;       }
```

Euclid's GCD Algorithm

Euclid's GCD computation algorithm

- By choosing the test set {(x=3,y=3),(x=4,y=3), (x=3,y=4)}
  - all statements are executed at least once.

Branch Coverage

- Test cases are designed such that:
  - different branch conditions
    - given true and false values in turn.

Branch Coverage

- Branch testing guarantees statement coverage:
  - a stronger testing compared to the statement coverage-based testing.

Stronger testing

- Test cases are a superset of a weaker testing:
  - discovers at least as many errors as a weaker testing
  - contains at least as many significant test cases as a weaker test.

Example

```
int f1(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;        }
```

Example

- Test cases for branch coverage can be:
- {(x=3,y=3),(x=3,y=2), (x=4,y=3), (x=3,y=4)}

Condition Coverage

- Test cases are designed such that:
  - each component of a composite conditional expression
    - given both true and false values.

Example

- Consider the conditional expression
  - ((c1.and.c2).or.c3):
- Each of c1, c2, and c3 are exercised at least once,
  - i.e. given true and false values.

Branch testing

- Branch testing is the simplest condition testing strategy:
  - compound conditions appearing in different branch statements
    - are given true and false values.

Branch testing

- Condition testing
  - stronger testing than branch testing:
- Branch testing
  - stronger than statement coverage testing.

Condition coverage

- Consider a boolean expression having n components:
  - for condition coverage we require 2^n test cases.

Condition coverage

- Condition coverage-based testing technique:
  - practical only if n (the number of component conditions) is small.

Path Coverage

- Design test cases such that:
  - all linearly independent paths in the program are executed at least once.

Linearly independent paths

- Defined in terms of
  - control flow graph (CFG) of a program.

Path coverage-based testing

- To understand the path coverage-based testing:
  - we need to learn how to draw control flow graph of a program.

Control flow graph (CFG)

- A control flow graph (CFG) describes:
  - the sequence in which different instructions of a program get executed.
  - the way control flows through the program.

How to draw Control flow graph?

- Number all the statements of a program.
- Numbered statements:
  - represent nodes of the control flow graph.

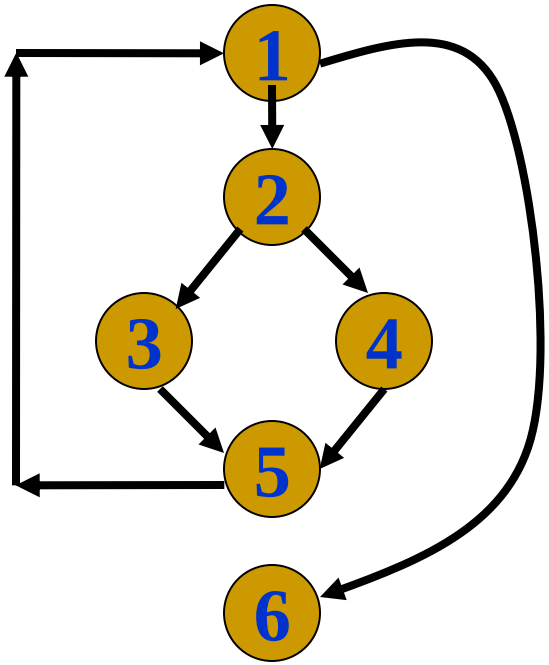
How to draw Control flow graph?

- An edge from one node to another node exists:
  - if execution of the statement representing the first node
    - can result in transfer of control to the other node.

Example

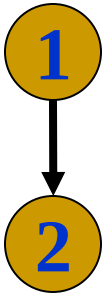
```
int f1(int x,int y){
1 while (x != y){
2    if (x>y) then
3         x=x-y;
4    else y=y-x;
5 }
6 return x;        }
```

Example Control Flow Graph



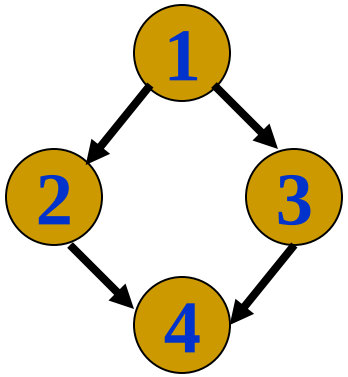
How to draw Control flow graph?

- Sequence:
  - 1 a=5;
  - 2 b=a*b-1;



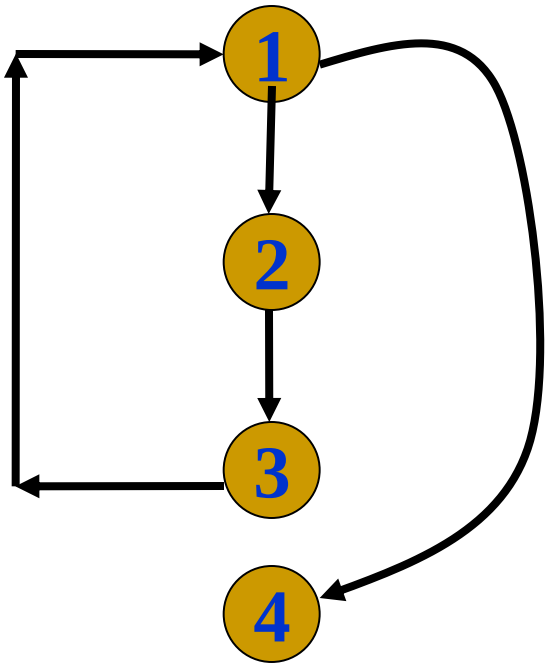
How to draw Control flow graph?

- Selection:
  - 1 if(a>b) then
  - 2 c=3;
  - 3 else c=5;
  - 4 c=c*c;



How to draw Control flow graph?

- Iteration:
  - 1 while(a>b){
  - 2 b=b*a;
  - 3 b=b-1;}
  - 4 c=b+d;



Path

- A path through a program:
  - a node and edge sequence from the starting node to a terminal node of the control flow graph.
  - There may be several terminal nodes for program.

Independent path

- Any path through the program:
- introducing at least one new node:
  - that is not included in any other independent paths.

Independent path

- **It is straight forward:**
  - **to identify linearly independent paths of simple programs.**
- **For complicated programs:**
  - **it is not so easy to determine the number of independent paths.**

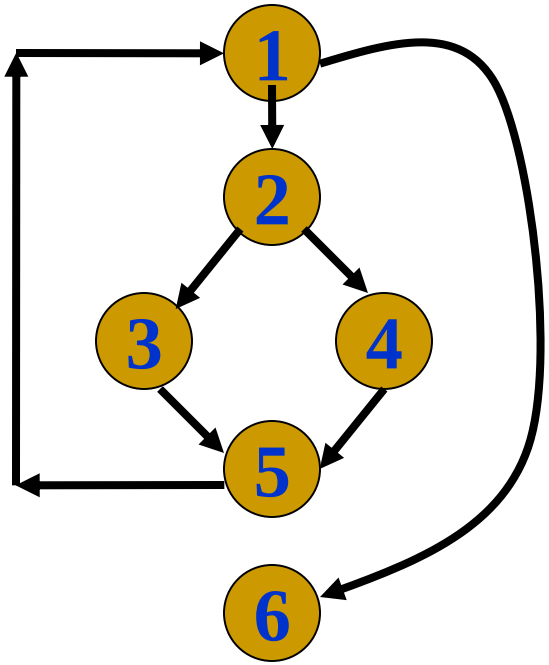
McCabe's cyclomatic metric

- **An upper bound:**
  - **for the number of linearly independent paths of a program**
- **Provides a practical way of determining:**
  - **the maximum number of linearly independent paths in a program.**

McCabe's cyclomatic metric

- Given a control flow graph G, cyclomatic complexity V(G):
  - V(G)= E-N+2
    - N is the number of nodes in G
    - E is the number of edges in G

Example Control Flow Graph



Example

- Cyclomatic complexity = 7-6+2 = 3.

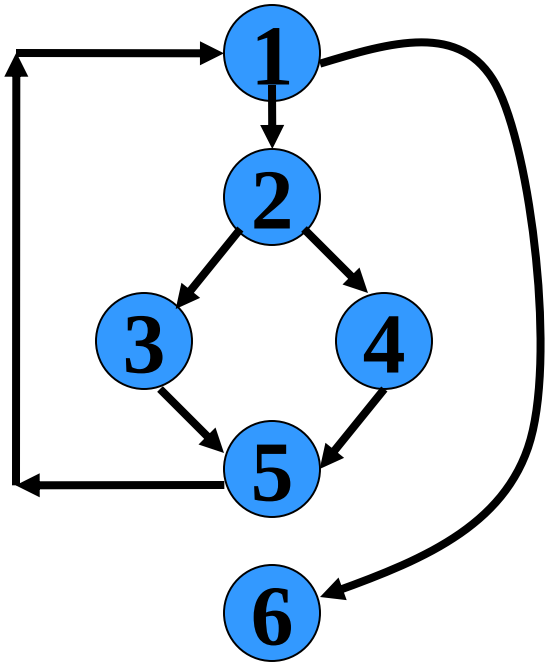
Cyclomatic complexity

- Another way of computing cyclomatic complexity:
  - inspect control flow graph
  - determine number of bounded areas in the graph
- V(G) = Total number of bounded areas + 1

Bounded area

- Any region enclosed by a nodes and edge sequence.

Example Control Flow Graph



Example

- From a visual examination of the CFG:
  - the number of bounded areas is 2.
  - cyclomatic complexity = 2+1=3.

Cyclomatic complexity

- **McCabe's metric provides:**
  - **a quantitative measure of testing difficulty and the ultimate reliability**
- **Intuitively,**
  - **number of bounded areas increases with the number of decision nodes and loops.**

Cyclomatic complexity

- **The first method of computing V(G) is amenable to automation:**
  - **you can write a program which determines the number of nodes and edges of a graph**
  - **applies the formula to find V(G).**

Cyclomatic complexity

- **The cyclomatic complexity of a program provides:**
  - **a lower bound on the number of test cases to be designed**
  - **to guarantee coverage of all linearly independent paths.**

Cyclomatic complexity

- **Defines the number of independent paths in a program.**
- **Provides a lower bound:**
  - **for the number of test cases for path coverage.**

Cyclomatic complexity

- **Knowing the number of test cases required:**
  - **does not make it any easier to derive the test cases,**
  - **only gives an indication of the minimum number of test cases required.**

Path testing

- **The tester proposes:**
  - **an initial set of test data using his experience and judgement.**

Path testing

- A dynamic program analyzer is used:
  - to indicate which parts of the program have been tested
  - the output of the dynamic analysis
    - used to guide the tester in selecting additional test cases.

Derivation of Test Cases

- Let us discuss the steps:
  - to derive path coverage-based test cases of a program.

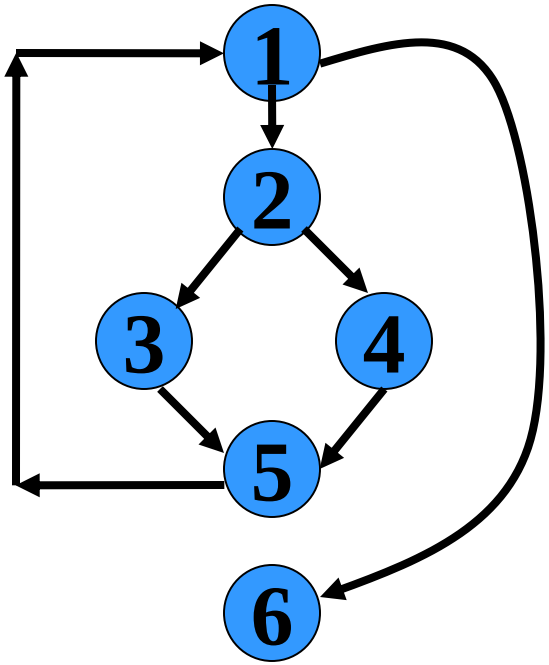
Derivation of Test Cases

- Draw control flow graph.
- Determine V(G).
- Determine the set of linearly independent paths.
- Prepare test cases:
  - to force execution along each path.

Example

```
int f1(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;       }
```

Example Control Flow Diagram



Derivation of Test Cases

- Number of independent paths: 3
  - **1,6** **test case (x=1, y=1)**
  - **1,2,3,5,1,6** **test case(x=1, y=2)**
  - **1,2,4,5,1,6** **test case(x=2, y=1)**

An interesting application of cyclomatic complexity

- Relationship exists between:
  - **McCabe's metric**
  - **the number of errors existing in the code,**
  - **the time required to find and correct the errors.**

Cyclomatic complexity

- **Cyclomatic complexity of a program:**
  - **also indicates the psychological complexity of a program.**
  - **difficulty level of understanding the program.**

Cyclomatic complexity

- From maintenance perspective,
  - limit cyclomatic complexity
    - of modules to some reasonable value.
  - Good software development organizations:
    - restrict cyclomatic complexity of functions to a maximum of ten or so.

Testing Quotes-unknown

- A group of managers were given the assignment to measure the height of a flag pole. So they go to the flag pole with ladders and tape measures, and they're falling off the ladders, droping the tape measures-the whole thing is just a mess. A tester comes along and sees what they're trying to do, walks over, pulls the flag pole out of the ground, lays it flat, measures it from end to measurement to one manager and walks away.

Story Continues….

- After the tester has gone, one manager turns to another and laughs-”Isn’t that just like a tester ,we’re looking for the height and he gives us the length”.!!!!!!!!!!!

Summary

- Exhaustive testing of non-trivial systems is impractical:
  - we need to design an optimal set of test cases
    - should expose as many errors as possible.

Summary

- If we select test cases randomly:
  - many of the selected test cases do not add to the significance of the test set.

Summary

- There are two approaches to testing:
  - black-box testing and
  - white-box testing.

Summary

- Designing test cases for black box testing:
  - does not require any knowledge of how the functions have been designed and implemented.
  - Test cases can be designed by examining only SRS document.

Summary

- White box testing:
  - requires knowledge about internals of the software.
  - Design and code is required.

Summary

- We have discussed a few white-box test strategies.
  - Statement coverage
  - branch coverage
  - condition coverage
  - path coverage

Summary

- A stronger testing strategy:
  - provides more number of significant test cases than a weaker one.
  - Condition coverage is strongest among strategies we discussed.

Summary

- We discussed McCabe's Cyclomatic complexity metric:
  - provides an upper bound for linearly independent paths
  - correlates with understanding, testing, and debugging difficulty of a program.

Debugging

- Once errors are identified:
  - it is necessary identify the precise location of the errors and to fix them.
- Each debugging approach has its own advantages and disadvantages:
  - each is useful in appropriate circumstances.

Brute-force method

- This is the most common method of debugging:
  - least efficient method.
  - program is loaded with print statements
  - print the intermediate values
  - hope that some of printed values will help identify the error.

Symbolic Debugger

- Brute force approach becomes more systematic:
  - with the use of a symbolic debugger,
  - symbolic debuggers get their name for historical reasons
  - early debuggers let you only see values from a program dump:
    - determine which variable it corresponds to.

Symbolic Debugger

- Using a symbolic debugger:
  - values of different variables can be easily checked and modified
  - single stepping to execute one instruction at a time
  - break points and watch points can be set to test the values of variables.

Backtracking

- This is a fairly common approach.
- Beginning at the statement where an error symptom has been observed:
  - source code is traced backwards until the error is discovered.

Example

```
int main(){
 int i,j,s;
 i=1;
 while(i<=10){
s=s+i;
i++; j=j++;}
printf("%d",s);
}
```

Backtracking

- Unfortunately, as the number of source lines to be traced back increases,
  - the number of potential backward paths increases
  - becomes unmanageably large for complex programs.

Cause-elimination method

- Determine a list of causes:
  - which could possibly have contributed to the error symptom.
  - tests are conducted to eliminate each.
- A related technique of identifying error by examining error symptoms:
  - software fault tree analysis.

Program Slicing

- This technique is similar to back tracking.
- However, the search space is reduced by defining slices.
- A slice is defined for a particular variable at a particular statement:
  - set of source lines preceding this statement which can influence the value of the variable.

Example

```
int main(){
 int i,s;
 i=1; s=1;
 while(i<=10){
s=s+i;
i++;}
printf("%d",s);
printf("%d",i);
}
```

Debugging Guidelines

- Debugging usually requires a thorough understanding of the program design.
- Debugging may sometimes require full redesign of the system.
- A common mistake novice programmers often make:
  - not fixing the error but the error symptoms.

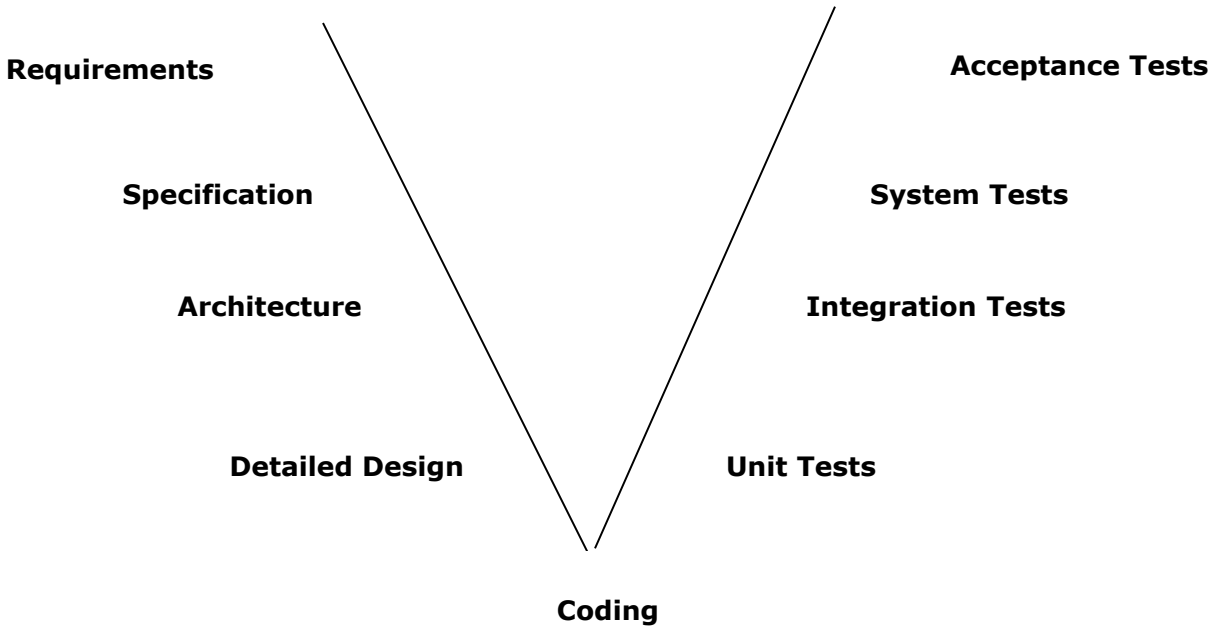
Debugging Guidelines

- Be aware of the possibility:
  - an error correction may introduce new errors.
- After every round of error-fixing:
  - regression testing must be carried out.

Testing

- Software products are tested at three levels:
  - Unit testing
  - Integration testing
  - System testing

The following diagram depicts the 'V' Model



The diagram is self-explanatory. For an easy understanding, look at the following table:

SDLC Phase	Test Phase
1. Requirements	1. Build Test Strategy. 2. Plan for Testing. 3. Acceptance Test Scenarios Identification.
2. Specification	1. System Test Case Generation.
3. Architecture	1. Integration Test Case Generation.
4. Detailed Design	1. Unit Test Case Generation

Unit testing

- During unit testing, modules are tested in isolation:
  - If all modules were to be tested together:
    - it may not be easy to determine which module has the error.

Unit testing

- Unit testing reduces debugging effort several folds.
  - Programmers carry out unit testing immediately after they complete the coding of a module.

Integration testing

- After different modules of a system have been coded and unit tested:
  - modules are integrated in steps according to an integration plan
  - partially integrated system is tested at each integration step.

Integration Testing

- Develop the integration plan by examining the structure chart :
  - big bang approach
  - top-down approach
  - bottom-up approach
  - mixed approach

Big bang Integration Testing

- Big bang approach is the simplest integration testing approach:
  - all the modules are simply put together and tested.
  - this technique is used only for very small systems.

Big bang Integration Testing

- Main problems with this approach:
  - if an error is found:
    - it is very difficult to localize the error
    - the error may potentially belong to any of the modules being integrated.
  - debugging errors found during big bang integration testing are very expensive to fix.

Bottom-up Integration Testing

- Integrate and test the bottom level modules first.
- A disadvantage of bottom-up testing:
  - when the system is made up of a large number of small subsystems.
  - This extreme case corresponds to the big bang approach.

Top-down integration testing

- Top-down integration testing starts with the main routine:
  - and one or two subordinate routines in the system.
- After the top-level 'skeleton' has been tested:
  - immediate subordinate modules of the 'skeleton' are combined with it and tested.

Mixed integration testing

- Mixed (or sandwiched) integration testing:
  - uses both top-down and bottom-up testing approaches.
  - Most common approach

Integration Testing

- In top-down approach:
  - testing waits till all top-level modules are coded and unit tested.
- In bottom-up approach:
  - testing can start only after bottom level modules are ready.

Smoke Testing

Smoke testing refers to physical tests made to closed systems of pipes to test for leaks. By metaphorical extension, the term is also used for the first test made after assembly or repairs to a system, to provide some assurance that the system under test will not catastrophically fail. After a *smoke test* proves that "the pipes will not leak, the keys seal properly, the circuit will not burn, or the software will not crash outright," the system is ready for more stressful testing.

Smoke Testing

- The plumbing industry started using the smoke test in 1875.
- Later this usage seems to have been forgotten, and the electronics industry believes it invented the term: "The phrase smoke test comes from [electronic] hardware testing. You plug in a new board and turn on the power. If you see smoke coming from the board, turn off the power. You don't have to do any more testing." !!!

Smoke Testing in Software

- For example, a smoke test may ask basic questions like "Does the program run?", "Does it open a window?", or "Does clicking the main button do anything?" The purpose is to determine whether the application is so badly broken that further testing is unnecessary.

Smoke Testing

- *"Smoke testing might be a characterized as a rolling integration strategy"*.
- Smoke testing is an integration testing approach that is commonly used when "shrink-wrapped" software products are being developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its project on a frequent basis.

Smoke Testing

- The smoke test should exercise the entire system from end to end. Smoke testing provides benefits such as:
- 1) Integration risk is minimized.
- 2) The quality of the end-product is improved.
- 3) Error diagnosis and correction are simplified.
- 4) Progress is easier to asses.

Smoke Testing

- Software components produced are integrated into a "build". A series of tests are designed to expose errors that will keep the build from properly performing its function. The intent is to uncover "Show stopper" errors that have the highest likelihood of throwing the software project behind schedule. Sometimes daily builds are made and subjected to smoke testing.

System Testing

- There are three main kinds of system testing:
  - Alpha Testing
  - Beta Testing
  - Acceptance Testing
  - Stress Testing.

Alpha Testing

- System testing is carried out by the test team within the developing organization.

Beta Testing

- System testing performed by a select group of friendly customers.

Beta Testing

- **A “beta version” is complete software that is “close” to done**
- Theoretically all defects have been corrected or are at least known to developers
- Idea is to give it to sample users and see if a huge problem emerges

Beta Testing

- **Purpose of beta test:**
- See if software is good enough for a friendly, small user community (limit risk)
- Find out if “representative” users are likely to stimulate failure modes missed by testing

Beta Testing

- **Approaches**
- This is almost all exploratory testing
- Assumption is that different users have different usage profiles
- Hope is that if small user community doesn't find problems, there won't be many
- important problems that slip through into full production

Acceptance Testing

- System testing performed by the customer himself:
  - to determine whether the system should be accepted or rejected.

Acceptance Testing

- **Acceptance tests ensure system provides all advertised functions**
- Testing performed by a customer or surrogate
- Might also involve a certification authority or independent test observer

Acceptance Testing

- **Purpose of acceptance test:**
- Does the system meet all requirements to be used by a customer?
- Usually the last checkpoint before shipping a system
- Might be performed on all systems to check for hardware
- defects/manufacturing defects, not just software design problems

Stress Testing

- Stress testing (aka 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

- Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. The following types of tests may be conducted during stress testing;
- Special tests may be designed that generate ten interrupts per second, when one or two is the average rate.
- Input data rates may be increases by an order of magnitude to determine how input functions will respond.
- Test Cases that require maximum memory or other resources.
- Test Cases that may cause excessive hunting for disk-resident data.

Regression Testing

- Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side affects.
- Regression may be conducted manually, by re-executing a subset of al test cases or using automated capture/playback tools.
- The Regression test suit contains three different classes of test cases:
- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

How many errors are still remaining?

- Seed the code with some known errors:
  - artificial errors are introduced 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)

Example

- 100 errors were introduced.
- 90 of these errors were found during testing
- 50 other errors were also found.
- Remaining errors=
  50 (100-90)/90 = 6

Error Seeding

- The kind 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 analyzing historical data from similar projects.

A clever person solves a problem.
A wise person avoids it.

-- ***Einstein***