Software Testing and Quality Assurance

Lecture 2:
Introduction to Software Testing II

Dr. Jameleddine HASSINE
ICS Department, KFUPM
jhassine@kfupm.edu.sa

Outline

- Validation & Verification
- Static Analysis
  - Inspection
  - Walkthroughs
  - Reviews
- Testing vs. Debugging vs. Quality Assurance
- Testing Taxonomy
  - Fault
  - Error
  - Failure
- Observability and Controllability
- Test Coverage Criteria

Validation & Verification (V &V) Process

- V & V takes place at each phase of software development life cycle.
  - Requirements
  - Design
  - Code
- Has two principal objectives
  - The discovery of defects in a system.
  - The assessment of whether or not the system is useful and useable in an operational situation.

Are we building the right product?

(Latin validus - healthy, sound, effective)

- The process of evaluating a system during and at the end of the development process to ensure compliance with intended usage (IEEE)

- The software should do what the user really requires.

Verification

Are we building the product right?

(Latin veritas - truth or integrity)

- The process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase (IEEE)
- The software should conform to its specification.

- Verification is usually a more technical activity that uses knowledge about the individual software artifacts, requirements, and specifications

- Validation usually depends on domain knowledge, that is the knowledge of the application for which the software is written. For example, the validation of "an airplane" requires knowledge from aerospace engineers and pilots.

- Ultimate goal
  - Software is **‘fit for the purpose’**.
- Software is not necessarily 100 % free of defects.
- Rather, it must be good enough
  - for its intended use; and
  - the type of use will determine the degree of confidence that is needed.

Independent V & V (IV&V)

- Process by which V&V is carried out by an organization that is neither the developer or the acquirer of the software
- Three types of Independence:

1. **Managerial independence**
   - Separate decision on which areas of the software to analyze and test and which techniques to use
   - Determines the schedule of tasks
2. **Financial independence**
   - Costs for V&V are funded separately
   - No risk of diverting resources that may cause delays
3. **Technical independence**
   - Different from the developers or analysts
   - Use of different tools

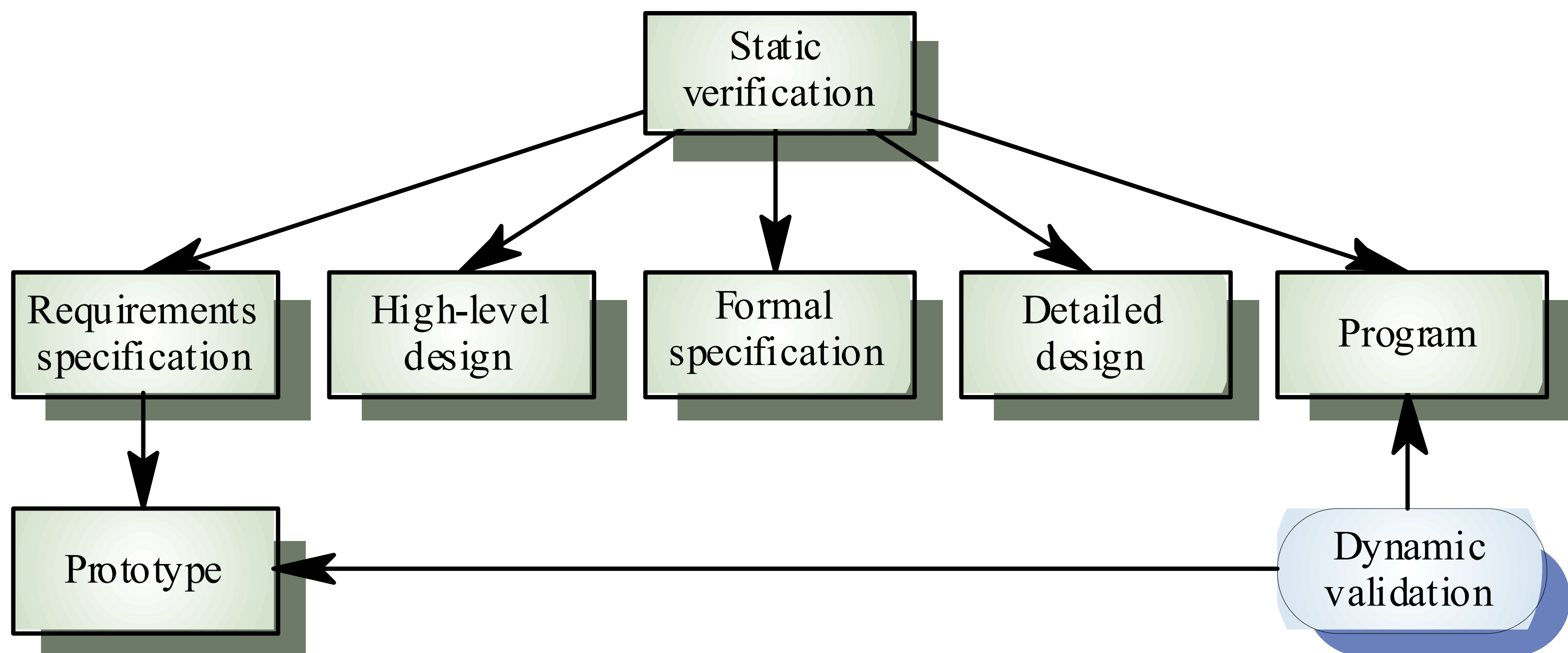
V & V Confidence

- Depends on system's purpose, user expectations and marketing environment
  - Software function
    - The level of confidence depends on how critical the software is to an organization.
  - User expectations
    - Users may have low expectations of certain kinds of software.
  - Marketing environment
    - Getting a product to market early may be more important than finding defects in the program.

V & V approaches

Within the V & V process, there are two complementary approaches to system analysis:

- Software Inspections, Reviews and Walkthroughs (Static analysis)
- Software Testing: Testing by executing the program with real inputs and observing its behavior (Dynamic analysis)



Static Analysis

- Testing without executing the program
  - This include inspections, reviews and walkthroughs, and some forms of analysis
  - Concerned with analysis of the static system representation to discover problems.
  - Review the documents and software system during different phases of development life-cycle.
  - Very effective at finding certain kinds of problems – especially "potential" faults, that is, problems that could lead to faults when the program is modified
  - Usually more cost-effective than testing for defect detection at the unit and module level
  - Allows defect detection to be combined with other quality checks
  - More than 60% of program errors can be detected by informal program inspections
- Supplement by tool-based document and code analysis.

Static Analysis Checks

Fault class	**Static analysis check**
Data faults	Variables used before initialisation Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter type mismatches Parameter number mismatches Non-usage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic

About Meetings

- "Walkthroughs, Reviews and Inspections" are a form of human-based testing that involves people working together cooperatively.
- We begin with a few basic axioms regarding meetings…

The Safety Axiom

- First, as we all know, meetings can be terrible…
  - Ever been to a really BAD meeting?
- In order for meetings to be effective, they need to be made safe…
  - safe to attend, and safe NOT to attend.

- One way to accomplish this, is to remove the uncertainty about what might be covered in a meeting:
  - Publish an agenda and stick to it.
  - Handle "emergency issues" in a way that will not hurt people who don't attend the meeting.
  - Be sure people who should attend are identified and explicitly invited in advance.
  - Gently confront those present who should not attend – preferably before the meeting starts.

Making Meetings Safe (cont'd)

- Establish ground rules for the conduct of meetings:
  - Establish a no-interruption policy, but also set time limits for individual speakers so that everyone will be able to participate.
  - Outlaw personal attacks and put-downs.
  - Finish on time, but schedule a continuation of the meeting if business isn't finished.
  - Use a related issues list and ensure follow-up for important off-topic matters that come up.

Other Meeting Axioms

- Meetings should be as small as possible, but no smaller.
- Keep the agenda short. (A meeting that tries to do too many things does none well.)
- Design meetings to have the appropriate structure and pace.
- Identify someone to act as a facilitator.
- Be prepared! (95% of meetings that fail do so because of inadequate preparation.)

Walkthroughs

- Usually done in a single meeting.
- Evaluate a software product to
  - Find anomalies & improve the software product.
  - Consider alternative implementations (at a detailed low-level).
  - Evaluate the conformance to standards and specifications .
- Rather informal.
- No formal training required beforehand.
- Success depends on experience and skills of the team members.
- Can be performed at any phase of the software development process.
- Can be performed on any artifact (SRS, Use Case Diagrams, Class diagrams, test cases, etc.)

Software Inspection

- Checklist-based formal approach to uncover errors.
- Intended explicitly for defect detection **(not correction).**
- Defects may be logical errors & anomalies in the code. For example:
  - An un-initialized variable.
  - Non-compliance with standards.
- Team members require formal training beforehand.
- Remove errors as near source as possible; hence reducing costs of rework.
- Its success depends on
  - The properness of the inspection process application,
  - Checks applied;
  - The diligence of the inspectors.
- Can be performed at any phase of the software development process
- Can be performed on any artifact (SRS, UC Diagrams, Class diagrams, test cases, etc.)

Example1: Checklist for inspections of Use Case models

1. Actors

1.1. Are there any actors that are not defined in the use case model, that is, will the system communicate with any other systems, hardware or human users that have not been described?

1.2. Are there any superfluous actors in the use case model, that is, human users or other systems that will not provide input to or receive output from the system?

1.3. Are all the actors clearly described, and do you agree with the descriptions?

1.4. Is it clear which actors are involved in which use cases, and can this be clearly seen from the use case diagram and textual descriptions? Are all the actors connected to the right use cases?

2. The use cases

2.1. Is there any missing functionality, that is, do the actors have goals that must be fulfilled, but that have not been described in use cases?

2.2. Are there any superfluous use cases, that is, use cases that are outside the boundary of the system, do not lead to the fulfillment of a goal for an actor or duplicate functionality described in other use cases?

2.3. Do all the use cases lead to the fulfillment of exactly one goal for an actor, and is it clear from the use case name what is the goal?

2.4. Are the descriptions of how the actor interacts with the system in the use cases consistent with the description of the actor?

2.5. Is it clear from the descriptions of the use cases how the goals are reached and do you agree with the descriptions?

Example 1: Checklist for inspections of Use Case model

3. The description of each use case

3.1. Is expected input and output correctly defined in each use case; is the output from the system defined for every input from the actor, both for normal flow of events and variations?

3.2. Does each event in the normal flow of events relate to the goal of its use case?

3.3. Is the flow of events described with concrete terms and measurable concepts and is it described at a suitable level of detail without details that restrict the user interface or the design of the system?

3.4. Are there any variants to the normal flow of events that have not been identified in the use cases, that is, are there any missing variations?

3.5. Are the triggers, starting conditions, for each use case described at the correct level of detail?

3.6. Are the pre- and post-conditions correctly described for all use cases, that is, are they described with the correct level of detail, do the pre- and post conditions match for each of the use cases and are they testable?

4. Relation between the use cases:

4.1. Do the use case diagram and the textual descriptions match?

4.2. Has the include-relation been used to factor out common behavior?

4.3. Does the behavior of a use case conflict with the behavior of other use cases?

4.4. Are all the use cases described at the same level of detail?

Example2: Code Inspection Checklist

1. Variable, Attribute, and Constant Declaration Defects

Are descriptive variable and constant names used in accord with naming conventions?
Are there variables or attributes with confusingly similar names?
Is every variable and attribute properly initialized?
Could any non-local variables be made local?
Are there literal constants that should be named constants?
Are there variables or attributes that should be constants?
Are there attributes that should be local variables?
Do all attributes have appropriate access modifiers (private, protected, public)?
Are there static attributes that should be non-static or vice-versa?

2. Method Definition Defects

Are descriptive method names used in accord with naming conventions?
Do all methods have appropriate access modifiers (private, protected, public)?
Are there static methods that should be non-static or vice-versa?

3. Class Definition Defects

Does each class have appropriate constructors?
Do any subclasses have common members that should be in the superclass?

Example2: Code Inspection Checklist

4. Computation/Numeric Defects
Are there any computations with mixed data types?
Are parentheses used to avoid ambiguity?

5. Comment Defects
Does every method, class, and file have an appropriate header comment?
Does every attribute, variable, and constant declaration have a comment?
Is the underlying behavior of each method and class expressed in plain language?
Is the header comment for each method and class consistent with the behavior of the method or class?
Do the comments and code agree?
Do the comments help in understanding the code?
Are there enough comments in the code?
Are there too many comments in the code?

6. Layout and Packaging Defects
Is a standard indentation and layout format used consistently?
For each method: Is it no more than about 60 lines long?

7. Modularity Defects (MO)
Are the Java/C# class libraries used where and when appropriate?
Are there libraries imported but not used in a given class?

Activity:
Inspection of Sums Of Perfect Powers

A non-negative integer *n* is said to be a sum of two perfect powers if there exist two non-negative integers *a* and *b* such that:

$a^m + b^k = n$

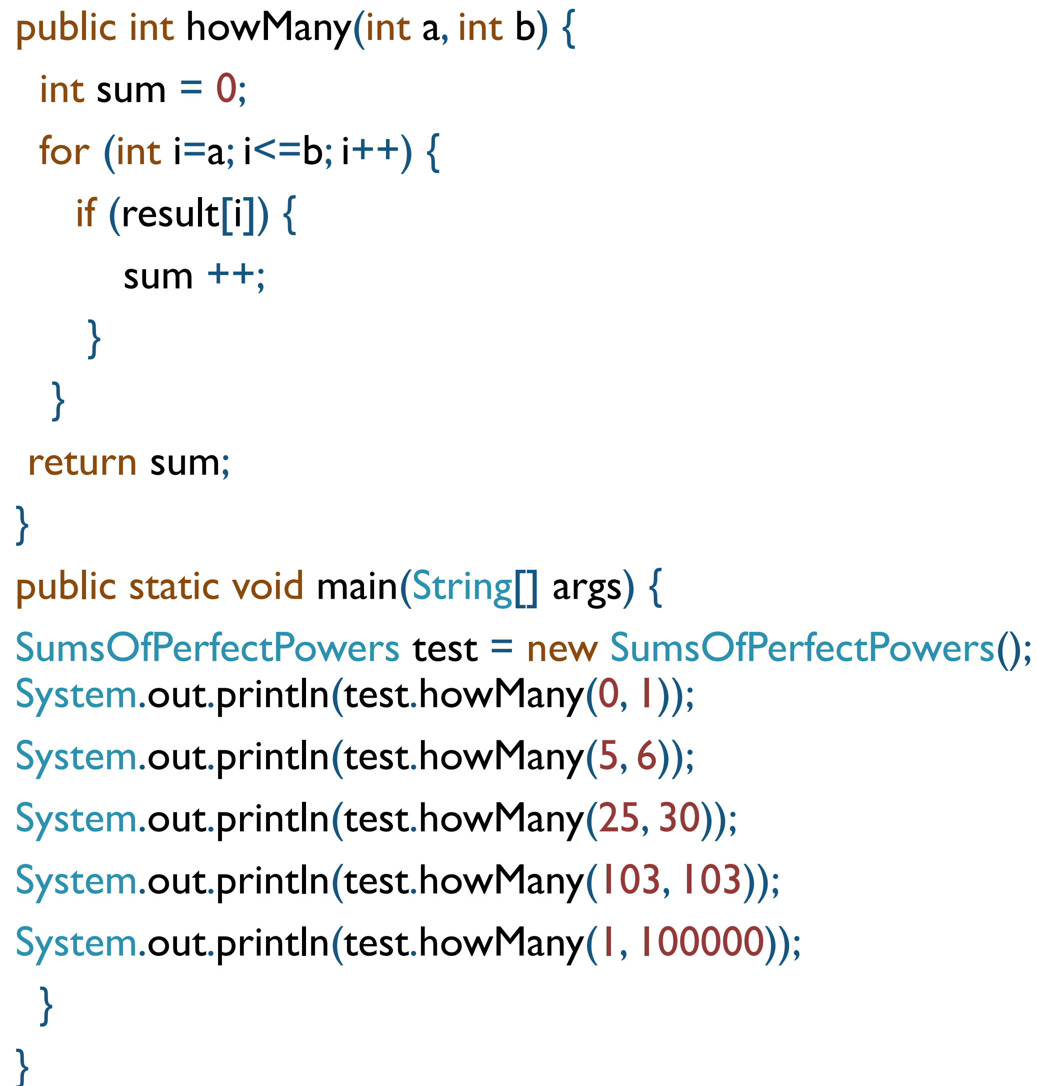
for some positive integers *m* and *k*, both greater than 1. Given two non-negative integers

lowerBound and upperBound, return the number of integers between lowerBound and upperBound, inclusive, that are sums of two perfect powers.

lowerBound will be between 0 and 5000000, inclusive.

upperBound will be between lowerBound and 5000000, inclusive.

```java
import java.util.ArrayList;
public class SumsOfPerfectPowers {
ArrayList<Long> numList = new ArrayList<Long>(5000001);
// status of whether a number is power number
boolean[] result = new boolean[5000001];
public SumsOfPerfectPowers() {
   numList.add((long) 0);
   numList.add((long) 1);
   for (int i = 2; i <= 2237; i++) {
       int j = 2;
       double value;
       while ((value = Math.pow(i, j)) <= 5000000) {
            numList.add((long) value);
            j++; }
  }
    int len = numList.size();
    int value;
    for (int i = 0; i < len; i++) {
        for (int j = 0; j < len; j++) {
          value = (int) (numList.get(i) + numList.get(j));
          if (value <= 5000001) {
               result[value] = true; }
        }
     }
}
```



```java
public int howMany(int a, int b) {
  int sum = 0;
  for (int i=a; i<=b; i++) {
     if (result[i]) {
         sum ++;
      }
   }
 return sum;
}
public static void main(String[] args) {
SumsOfPerfectPowers test = new SumsOfPerfectPowers();
System.out.println(test.howMany(0, 1));
System.out.println(test.howMany(5, 6));
System.out.println(test.howMany(25, 30));
System.out.println(test.howMany(103, 103));
System.out.println(test.howMany(1, 100000));
  }
}
```

Some found issues

1) Comments are missing.

2) Access modifiers of numList and result should be private:

```
private ArrayList<Long> numList = new ArrayList<Long>(5000001);
private boolean[] result = new boolean[5000001];
```

3) ArrayList<…> reference types should be simply List<…>

```
private List<Long> numList = new ArrayList<Long>(5000001);
```

4) 5000001 is a magic number. Use named constants instead of numbers like 50000 and 2237. This would make the code more readable and less fragile.

```
private static final int MAX = 5000000;
```

and use it everywhere, for example:

```
private boolean[] result = new boolean[MAX + 1];
```

As per 2237 you use the following:

```
final int maxSquare = (int) Math.ceil(Math.sqrt(MAX));
for (int i = 2; i <= maxSquare; i++) { ... }
```

Some found issues

5) numList only used in the constructor, so it could be a local variable there instead of a field. Try to minimize the scope of variables.

6) The variable *value* is used as int and as double.

7) Actually, rename numList to perfectPowers since it stores perfect powers.

8) In the first for loop, rename i to base and j to exponent.

9) Rename the parameters of the *howMany* method to lowerBound and upperBound.

10) The initial capacity of the list to 50000001

while it contains only about 2500 elements.

It's a huge memory waste. Use the default constructor of the ArrayList which uses less memory.

```
final List<Long> perfectPowers = new ArrayList<Long>();
```

Inspection Metrics

Many different metrics can be calculated during an inspection process:

- The number of major and minor defects found
- The number of major defects found to total found
- Total Defects Found: A + B – C

A=defects found by inspector 1,

B=defects found by inspector 2,

C=common defects found by 1 & 2

- Defect Density: Total Defects Found / Size

size=total number of pages or lines of code or some other measure

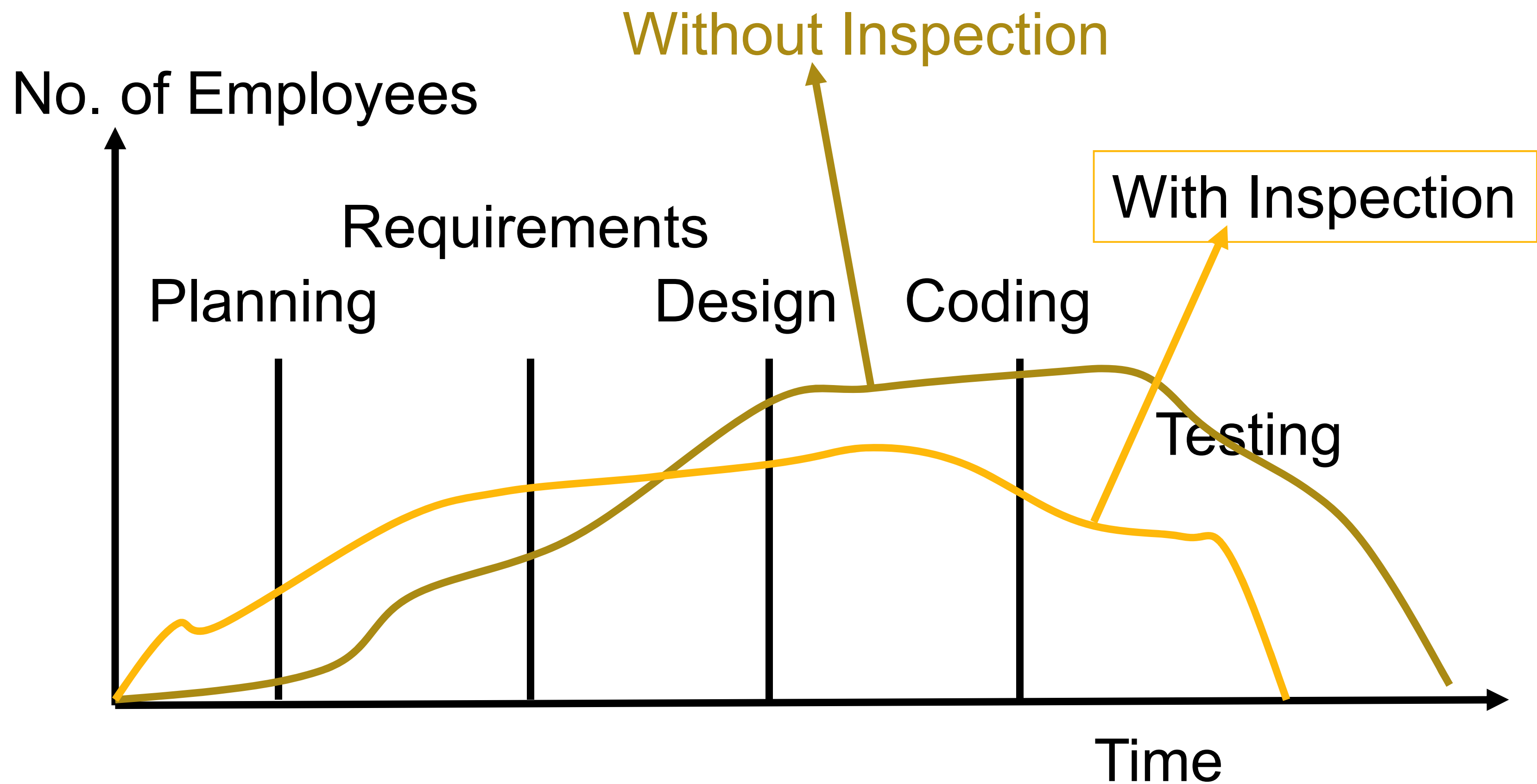
Inspection Metrics (continue…)

- Inspection Rate:
  - Size / Total Inspection Time

Total Inspection Time = the sum of the time of all reviewers plus the total person time spent in each meeting (e.g., 15 pages /hour).

- Defect Detection Rate:
  - Total Defects Found / Total Inspection Time

Software Inspection Cost



Inspection teams

- Made up of at least 5 members
  - Author of the code being inspected
  - Reader who reads the code to the team
  - Inspector who finds errors, omissions and inconsistencies
  - Moderator who chairs the meeting and notes discovered errors
  - Scribe taking notes on the inspection process results

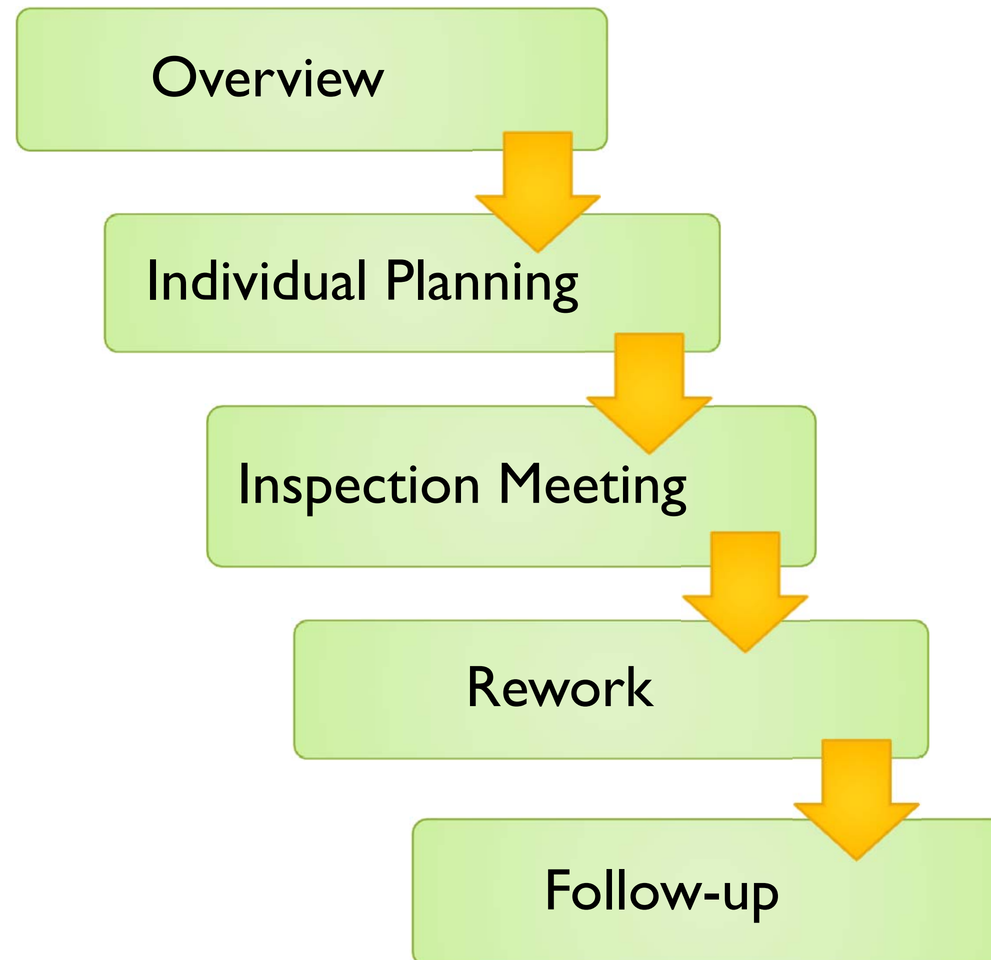
Inspection pre-conditions

- A precise specification must be available
- Static testing team members must be familiar with the organization standards
- Syntactically correct code must be available
- An error checklist should be prepared
- Management must accept that inspection will increase costs early in the software process

Inspection procedure

- System overview presented to inspection team
- Code and associated documents are distributed to inspection team in advance
- Inspection takes place and discovered errors are noted
- Modifications are made to repair discovered errors
- Re-inspection may or may not be required

Inspection Process



The Inspection Process

1. Overview (whole team)
   - What will be inspected?
   - Why are we spending time inspecting such artifact?
   - Designation of team roles.
2. Preparation (individual)
   - ranked distributions of error types
   - checklists of clues on finding errors
3. Inspection Meeting (whole team)
   - a "reader" is chosen by the moderator
   - every element of logic and every branch is considered
   - objective is to find errors
   - no specific solution hunting is permitted
   - moderator prepares written report within one day
4. Rework (owner)
5. Follow-up (moderator)
   - if > 5% of material has been reworked, the entire element is re-inspected

Inspecting *Modified* Code

- "Since most modifications are small…they are often erroneously regarded as trivially simple and handled accordingly; …However, *all* modifications are well worth inspecting…"
- "Human tendency is to consider the 'fix,' or correction, to a problem to be error-free itself. …The number of bad fixes can be reduced by some simple inspection after clean compilation of the fix."

Inspections vs. Walkthroughs

- Inspections differ significantly from walkthroughs.
- An inspection is a five-step, formalized process. The inspection team uses the checklist approach for uncovering errors. A walkthrough is less formal, has fewer steps, and does not use a checklist to guide or a written report to document the team's work.
- Although the inspection process takes much longer than a walkthrough, the extra time is justified because an inspection is extremely effective for detecting faults early in the development process when they are easiest and least costly to correct

Inspections vs. Walkthroughs

Properties	Inspections	Walkthroughs
Formal moderator training	Yes	No
Definite participant roles	Yes	No
Who “drives” the process	Moderator	Owner
Use checklists?	Yes	No
Formal follow-up	Yes	No
Rigor level	Formal	Informal

Inspections and Walkthroughs vs. Reviews

- Inspections and walkthroughs concentrate **on assessing correctness**
- A review is also **an informal process** (**no formal training beforehand**). Success depends on the skills and the experience of the reviewers.
- Reviews seeks to ascertain that **tolerable levels of quality are being attained**.
- The review team is more concerned with design deficiencies and **deviations** from the conceptual model and **requirements**.
- Reviews **do not focus on discovering technical flaws** but on ensuring that the design and development fully and **accurately address the needs of the application.**

- Which technique is more of a validation process and which is more of a verification process?
  - Walkthroughs — Verification
  - Inspections — Verification
  - Reviews — Validation

- Testing by executing the program with real inputs and observing its behavior

Ultimate Goal of Testing

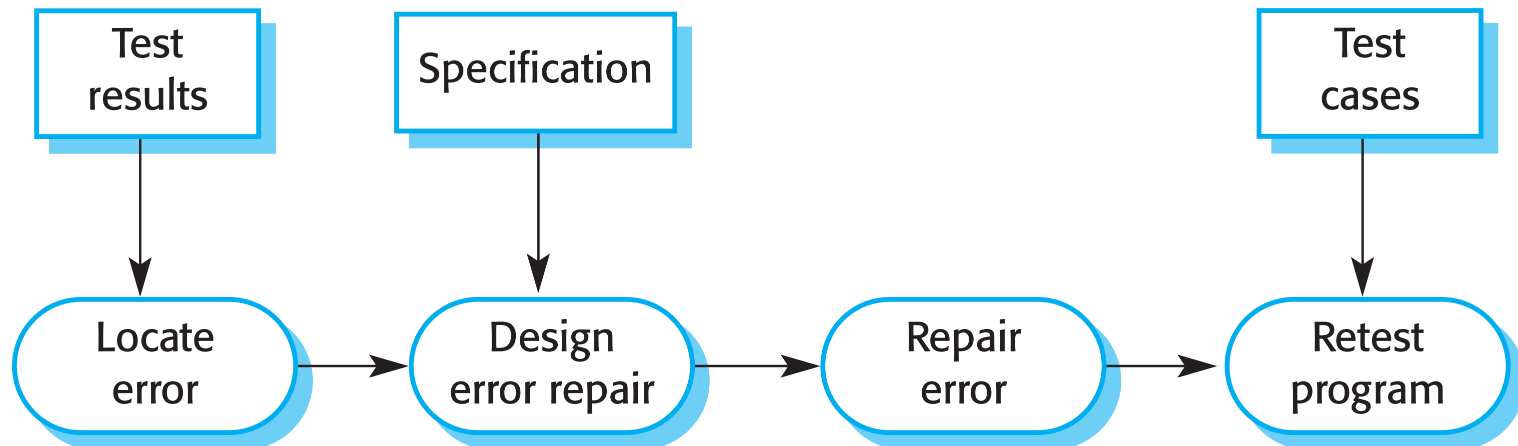
- Establishing confidence that a program **DOES** what it is supposed to do.
- Establishing confidence that a program **DOES NOT** do what it is **NOT** supposed to do.

Testing is not DEBUGGING

Testing is not Quality Assurance

Testing vs. Debugging

- Testing :The process of finding inputs that cause the software to fail (by dynamically exercising a software)
- Debugging :The process of finding a fault given a failure
- Debugging involves:
  - Locating the source code causing the bug
  - Fixing the bug
- Debugging happens **AFTER** testing
- After debugging, **MORE** testing is required
- The debugging process:



Testing vs. Quality Assurance (QA)

- Testing is necessary but not enough for QA process.
  - Testing contributes to improve quality by helping to identify problems.
- QA sets standards that project members (including testers) should follow in order to build a better software.

Testing Taxonomy (AMMANN and OFFUTT)

- Software Fault:

A static defect in the software

- Software Error:

An incorrect internal state that is the manifestation of some fault

- Failure:

External, incorrect behavior with respect to the requirements or other description of the expected behavior

Example

- Assume a program that uses the larger value resulting from an addition function and a square function

```
//…code……
r1 = add (a , b);
r2 = square (x) ;
//uses the larger of r1 and r2
//……rest of the code…
```

Example (cont'd)

```
int square (int x){
  if (x == 0 )
   return 0;
  else
   return x*2;  <------ fault
}
int add (int a, int b){
  return a+b;
}
```

- The fault here is the square of x was computed as 'x*2' instead of 'x*x'.
- A test case where the fault will not get executed is when 'x' = 0.
- A test case where the fault will be executed but will not result in an error is if 'x' = 2.

Example (cont'd)

```
int square (int x){
  if (x == 0 )
   return 0;
  else
   return x*2;  <------ fault
}
int add (int a, int b){
  return a+b;
}
```

- A test case where the fault will be executed and results in an error that DOES NOT result in a failure is when, for example, 'x' = 3 and 'a' and 'b' = 20.
- A test case where an error WILL result in a failure is if for example 'x' = 3 and 'a' and 'b' = 4.

Activity

```
public int findLast (int [] x, int y){
    //Effects: If x == null throw NullPointerException
   // else return the index of the LAST element in x
   // that equals y.
   // if no such element exists, return -1
   // Assume check for NullPointerException happens here
for (int i=x.length-1;i>0;i--){
        if (x[i] == y){
            return i;
        }
   }
   return -1;
}
```

Q1) Identify the fault

Q2) Indentify a test case that does not execute the fault.

Q3) Identify a test case that executes the fault but does not result in an error.

Q4) Identify a test case that results in an error but does not lead to a failure.

Q5) Identify a test case that results in a error which leads to a failure.

Exercise

```
public int findLast (int [] x, int y){
    //Effects: If x == null throw NullPointerException
   // else return the index of the LAST element in x
   // that equals y.
   // if no such element exists, return -1
   // Assume check for NullPointerException happens here
for (int i=x.length-1;i>0;i--){
       if (x[i] == y){
           return i;
       }
   }
  return -1;
}
```

Q1) Identify the fault

Answer: Counter setup in for loop is incorrect (i>0). Should be i>=0.

Q2) Indentify a test case that does not execute the fault.

Answer: x=null. Result expected is a NullPointerException thrown.

Exercise

```
public int findLast (int [] x, int y){
for (int i=x.length-1;i>0;i--){
       if (x[i] == y){
          return i;
       }
   }
  return -1;
}
```

Q3) Identify a test case that executes the fault but does not result in an error.

- Answer: *For any input where y appears in the second or later position, there is no error. Also, if* x *is empty, there is no error.* x = [0, 1 ,2] and y= 2. Expected result :is '2'.

Q4) Identify a test case that results in an error but does not lead to a failure.

- Answer: *For an input where y is not in x, the missing path (i.e. an incorrect PC on the final loop that is not taken) is an error, but there is no failure.* x = [1 ,1 ,1] and y = 2. Expected result is '-1'.

Exercise

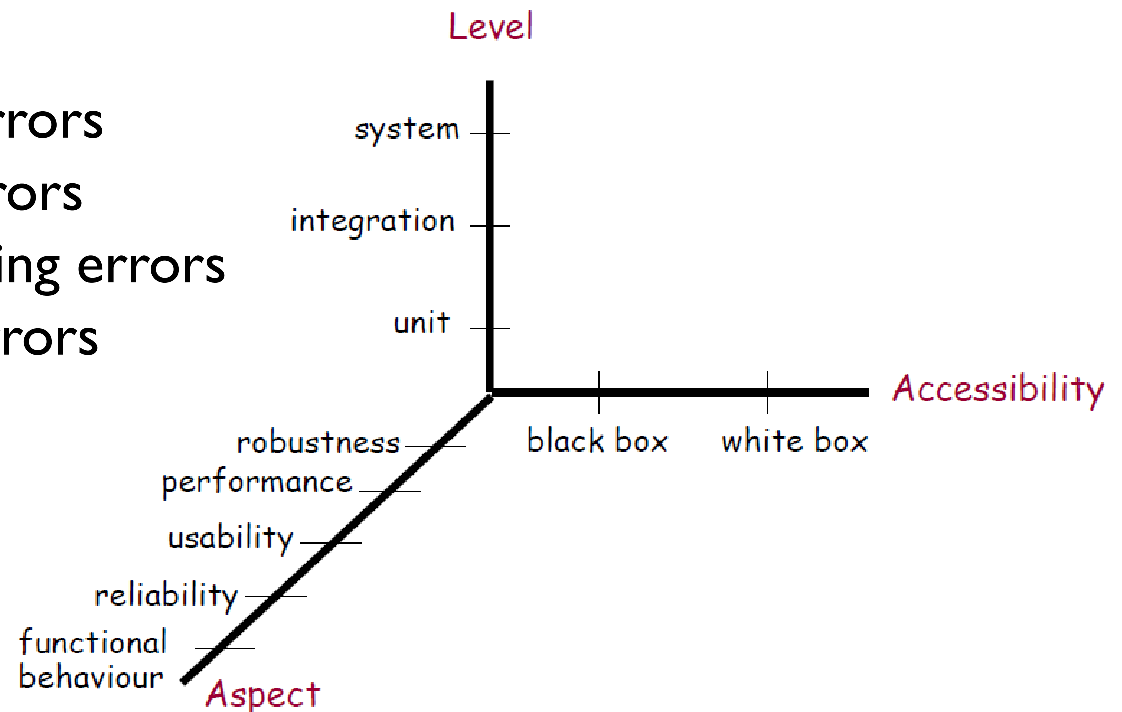
```
public int findLast (int [] x, int y){

for (int i=x.length-1;i>0;i--){
        if (x[i] == y){
            return i;
        }
    }
   return -1;
}
```

Q5) Identify a test case that results in a error which leads to a failure.

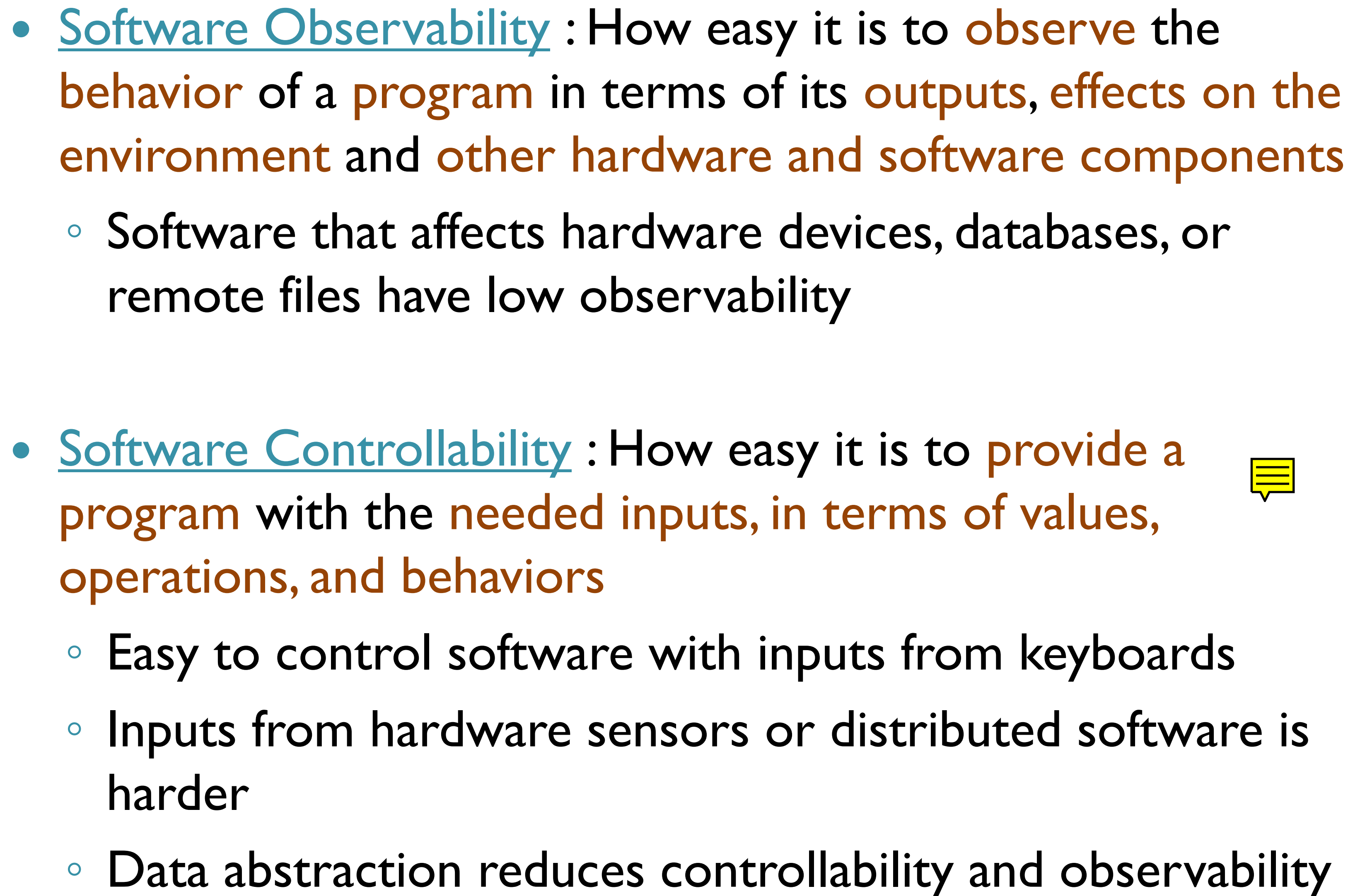
Answer: x = [2, 1 ,1] and y = 2. Result expected is '0'.

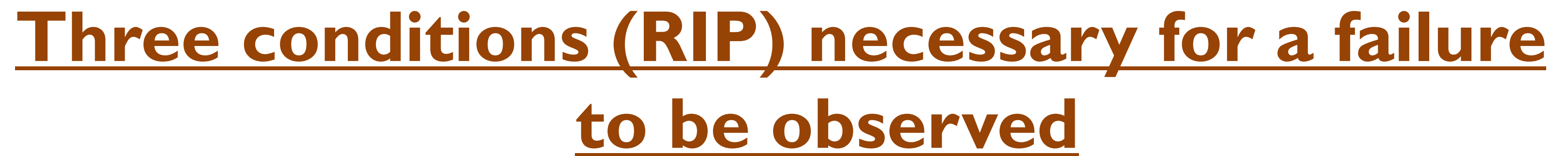
Types of Errors

- User interface error
- Boundary related errors
- Calculation/Data errors
- Initial and later state errors
- Control flow errors
- Errors handling
- Race conditions errors
- Load condition errors
- Hardware interfacing errors
- Documentation errors



Observability and Controllability





Three conditions (RIP) necessary for a failure to be observed

1. **Reachability** : The location or locations in the program that contain the fault must be reached
2. **Infection** : The state of the program must be incorrect
3. **Propagation** : The infected state must propagate to cause some output of the program to be incorrect

Inputs to affect controllability and observability

- **Prefix Values**:
  - Any inputs necessary to put the software into the appropriate state to receive the test case values.
- **Post-Fix Values**:
  - Any inputs that need to be sent to the software after the test cases values.
  - Two Types of postfix values
    - Verification Values: Values necessary to see the results of the test case values.
    - Exit Commands: Values needed to terminate the program or otherwise return it to a stable state.

Test Cases and Test Sets

- All types of test boil down to creating test cases and executing them.
- Test Case: is composed of the test case values, expected results, prefix values, postfix values necessary for a complete execution and evaluation of the software under test.
- Test Sets: is simply a set of tests.

Test Coverage Criteria

- Define a model of the software, then find ways to cover it.
- Test requirements: Specific things that must be satisfied or covered during testing.
- Test criteria: A collection of rules and a process that define test requirements.
- Coverage: Given a set of test requirements ***TR*** for a coverage criterion ***C***, a test set ***T*** satisfies ***C*** if and only if for every test requirement ***tr*** in ***TR***, at least one test ***t*** in ***T*** exists such that ***t*** satisfies ***tr***.
- Coverage level: Given a set of test requirements ***TR*** and a test set ***T***, the coverage level is simply the ratio of the number of test requirements satisfied by ***T*** and the size of ***TR***.

Test Coverage Criteria Example

Player 1	Player 2	Player 3	Player 4
A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3
A4	B4	C4	D4
A5	B5	C5	D5
A6	B6	C6	D6
A7	B7	C7	D7
A8	B8	C8	D8
A9	B9	C9	D9
A10	B10	C10	D10
A11	B11	C11	D11
A12	B12	C12	D12
A13	B13	C13	D13

Coverage Criteria Example (cont'd)

- Coverage Criteria requires each patch of the field to be covered by a player's route.
- Therefore there will be a test requirement for each patch of the field to be covered at least once (13 x 4 = 52).
- What is the coverage level if:
  - Only player 1 ran? (Answer = 16/52→ 30.7%)
  - Only players 2 and 3 ran? (Answer = 27/52 → 51.9%)
  - Only players 1 and 3 ran? (Answer = 30/52 → 57.6%)
  - All players ran? (Answer = 42/52 → 80.7%)