CS608 - SOFTWARE VERIFICATION AND VALIDATION

BY BINT E ADAM

Week-01: Introduction to Software Quality Assurance

Quality popular view:

– Something "good" but not quantifiable

– Something luxury and classy

Quality professional view:

– Conformance to requirement

- The requirements are clearly stated, and the product must conform to it
- Any deviation from the requirements is regarded as a defect
- A good quality product contains fewer defects

– Fitness for use:

- Fit to user expectations: meet user's needs
- A good quality product provides better user satisfaction

What is quality?

- What you see of high quality others do not see it like that.
- I depend on your requirements.
- For example, a university perfect registration system is not necessarily suitable to a school.

ISO Definition of Quality

ISO 8402 definition of QUALITY:

The totality of features and characteristics of a product or a service that bear on its ability satisfy stated or implied needs

ISO 9216 Model:

Quality characteristics

1. Functionality
2. Reliability
3. Usability
4. Efficiency
5. Maintainability
6. Portability

Other definition and concept

• Four Absolutes:

– Quality Means Conformance to Requirements. Both functional and non-functional

- Quality Comes from Prevention

- Quality is never ending improvement

– Quality is Zero Defects

Benefits of software quality

- Decreased number of defects and errors in software
- Less rework as a result of less software defects
- Reduced development and maintenance cost
- Increased software reliability
- Increased customer satisfaction
- Happier software practitioners

Software Engineers and quality

- **Software engineers strive to control the**
  - Process applied: What is the best process (i.e SDLC) to be used for the development of software
  - Resources expended: Make sure software development is finished in expected time and also using the estimated budget
  - End product quality attributes: Make sure that the software it self is of a high quality and contains all features and requirements (functional and non functional)

Reasons for poor quality

1. Faulty requirements definition
2. Client-developer communication failures
3. Deliberate deviations from software requirements
4. Logical design errors
5. Coding errors
6. Non-compliance with documentation and coding instructions
7. Shortcomings of the testing process
8. User interface and procedure errors
9. Documentation errors

Software Quality: IEEE Definition

Software quality is:

(1) The degree to which a system, component, or process meets specified requirements.

(2) The degree to which a system, component, or process meets customer or user needs or expectations.

Software Quality Assurance

Software quality assurance is:

A systematic, planned set of actions necessary to provide adequate confidence that the software development process or the maintenance process of a software system product conforms to established functional technical requirements as well as with the managerial requirements of keeping the schedule and operating within the budgetary confines.

The objectives of SQA activities in software development

(1) Assuring an acceptable level of confidence that the software will conform to functional technical requirements.

(2) Assuring an acceptable level of confidence that the software will conform to managerial scheduling and budgetary requirements.

(3) Initiation and management of activities for the improvement and greater efficiency of software development and SQA activities.

Software quality perspectives

Software quality can be seen from different perspectives:

- Customer : Complete requirements (Functional and non functional)
- Project manager: Cost and schedule
- Maintenance engineer: Detection and correction times

Software quality should be considered in:

- Infrastructure and tools
- Staff
- Contract
- SDLC (Requirements, design, implementation, .. and etc
- Budget
- Schedule
- Maintenance

Prevention Versus Detection

- **Detection :**

  - Identify

- Correct

- **Prevention**
  - Train
  - Do it right from the first time

Summary

- **Introduction of software quality**
- **Causes of poor quality**
- **SQA**
- **Quality perspectives**

We want to be able to:

- 1. Identify the unique characteristics of software as a product and as process that justify separate treatment of its quality issues.
- 2. Recognize the characteristics of the software environment where professional software development and maintenance take place

The uniqueness / differences between a Software Product and an Industrial Product

- **High complexity**
  - The potential ways in which a software product can be used with different data / data paths reflecting different incoming data is almost infinite.
  - Manner in which industrial products can be used are usually well-defined.
  - Think about software: every loop with different values of data reflects a different opportunity to see software fail.

- **Invisibility of the product**
  - In an industrial product, missing parts are obvious.
    - Something missing? Easily identified.
  - Not so in software products.
    - May not be noticeable for years – if at all!
- **Opportunities to detect defects ("bugs")??**
- **Consider:**
  - Product Development
  - Product Planning
  - Product Manufacturing
    -
- **Product Development:**
  - Industrial: Designers and quality assurance (QA) staff check and test the product prototype, in order to detect its defects.
  - Computer Software: once prototype and system testing are concluded, product is ready for development.

Product Production Planning:

- **Industrial:** Often need new tooling approaches, assembly lines, new manufacturing processes.
  - Results in additional 'looks' at products
  - One could say that there is a better chance to discover defects
- **Computer Software**: Not required for the software production process
  - Manufacturing of software copies and printing of software manuals are conducted automatically.
  - No real chance to discover additional defects.

Product Manufacturing:

- **Industrial**: Usually defects uncovered here; easily fixed.
  - Typical burn-in problems; another view of product; stabilizes.
  - These represent additional opportunities to discover defects.

- **Computer Software:**
  - We merely copyright, print copies of software and manuals
  - No real chance for additional quality views
  - No real chance for discovering additional defects

SQA Environment

- **Being Contracted:**
  - Professional software development is almost always contracted.
    - A defined list of functional requirements that the developed software and its maintenance need to fulfil
    - Budget
    - Time schedule

- **Subject to Customer-Supplier Relationship**
  - **In professional software development, there is a constant oversight between customer and developer.**
    - Changes will occur;
    - Criticisms will arise.
    - Cooperation is critical to overall project success.
  - **Customer availability / relationship is essential and often problematic.**

- **Required Teamwork**
  - **We need teams due to**
    - Time required for development.
      - Workload is too much for a single person
    - A frequent variety of experts needed
      - Database; networking; algorithms; …
    - Wish to benefit from professional mutual support

- **Cooperation and Coordination with Other Software Teams**
  - Other software development teams in the same organization.
  - Hardware development teams in the same organization.
  - Software and hardware development teams of other suppliers.
  - Customer software and hardware development teams that take part in the project's development.
- **Interfaces with Other Systems**
  - Input interfaces, where other software systems transmit data to your software system.
  - Output interfaces, where your software system transmits processed data to other software systems.
- **Need to Continue Project despite Team Changes**
  - Team members leave, are hired, fired, take unexpected vacations, transferred within the company, and more.
  - Development must continue.
  - Timeline will not change.
- **Need to continue Software Maintenance for an Extended Period**
  - Customers expect to continue utilizing software for a long period.
  - SQA must address development, operations, and maintenance.

Summary

- **Difference between software and industry product**
- **Unique characteristics of software**
- **Unique characteristics of SQA environment**

Week-02: (SQA in SDLC, Verification and Validation)

Quality Assurance

- Quality Assurance is process oriented and focuses on defect prevention.
- Quality Assurance is a set of activities for ensuring quality in the processes by which products are developed.

Quality Control

- Quality Control is product oriented and focuses on defect identification.
- Quality Control is a set of activities for ensuring quality in products.
- The activities focus on identifying defects in the actual products produced.

Quality Assurance Focus

- Quality Assurance aims to prevent defects with a focus on the process used to make the product.
- It is a proactive quality process.
- It identifies weakness in processes to improve them.
- Quality control aims to identify and correct defects in the finished product.
- It is a reactive process.

Quality Assurance Goal

- The goal of Quality Assurance is to improve development and test processes so that defects do not arise when the product is being developed.
- Establish a good quality management system and the assessment of its adequacy.
- Periodic conformance audits of the operations of the system.
- Prevention of quality problems through planned and systematic activities including documentation.

Quality Control Goal

- The goal of Quality Control is to identify defects after a product is developed and before it's released.
- Finding & eliminating sources of quality problems through tools & equipment so that customer's requirements are continually met.
- The activities or techniques used to achieve and maintain the product quality, process and service.

Quality Assurance Examples

- A QA audit
- Process documentation
- Establishing standards
- Developing checklists
- A QC review
- Preforming testing

Summary

- **Difference b/w QA and QC**
  - **Focus**
  - **Goal**
  - **Examples**

SQA in SDLC

- Requirements
- Architectural design
- Detailed design
- Implementation
- Testing

Requirements Phase

- Senior QA/Manager ensures that the user/client requirements are captured correctly
- Find out the risks in the requirement and decide how the system will be tested.
- Properly expressed as functional, performance and interface requirements.
- Review the requirement document and other deliverables meeting the standard
- Prepare the formal test plan including the test tools are being used in the project.

Architectural Design Phase

- Ensure that architectural design meets standards as designated in the Project Plan
- Verify all captured requirement are allocated to software components
- Verify all the design documents are completed on time according to the project plan and kept in project repository (ER Diagram, Process diagram, Use Case, etc).
- Prepare the design test report and submit to the project manager.

Detailed Design Phase

- Prepare the test objectives from the requirement and design document created.
- Design a verification matrix or Check list and update on regular basis
- Send the test documents to project manager for approval and keep them in repository

Implementation Phase

- Verify the results of coding and design activities including the schedule available in the project plan
- Check the status of all deliverable items and verify that all are maintaining the standard.
- Getting updated with the tools and technologies used in the projects and provide the feedback to the team if any better solution is available.
- Complete writing the check list/ test cases to start testing.
- Verify that the components are ready to start test or not.

Testing Phase

- Start testing individual module and stat reporting bugs
- Verify that all tests are run according to test plans
- Verify all the bugs available in the bug tracking system are resolved.
- Compile the test reports and verify that the report is complete and correct
- Certify that testing is complete according to the plan
- Start creating the documentation and verify that all documents are ready for delivery

Verification & Validation (V&V)

- **Verification:**
  "Are we building the product right?" The software should conform to its specification.

- **Validation:**
  "Are we building the right product?" The software should do what the user really requires.

V&V Goals

- **Verification and validation should establish confidence that the software is fit for its purpose.**
  - **This does NOT mean completely free of defects.**
  - **Rather, it must be good enough for its intended use. The type of use will determine the degree of confidence that is needed.**

Static vs Dynamic V&V

- **Code and document inspections** - Concerned with the analysis of the static system representation to discover problems (*static v & v*)
  - May be supplement by tool-based document and code analysis
- **Software testing** - Concerned with exercising and observing product behaviour (*dynamic v & v*)
  - The system is executed with test data and its operational behaviour is observed

Summary

- **SQA activities in different SDLC phases**
  - **Requirements phase**
  - **Architectural design phase**
  - **Detailed design phase**
  - **Implementation phase**
  - **Testing phase**

Week-03: (Quality management, Principals, Practices and Standards)

SQA Principals

- Feedback
- Focus on Critical Factor
- Multiple Objectives
- Quality Control
- Motivation
- Evolution
- Process Improvement
- Persistence
- Different Effects of SQA
- Result-focused

Error, Defect, Failure

- Error: Human mistake cause error
- Defect: Improper program conditions generally result of Error
- Failure: difference from intended behavior

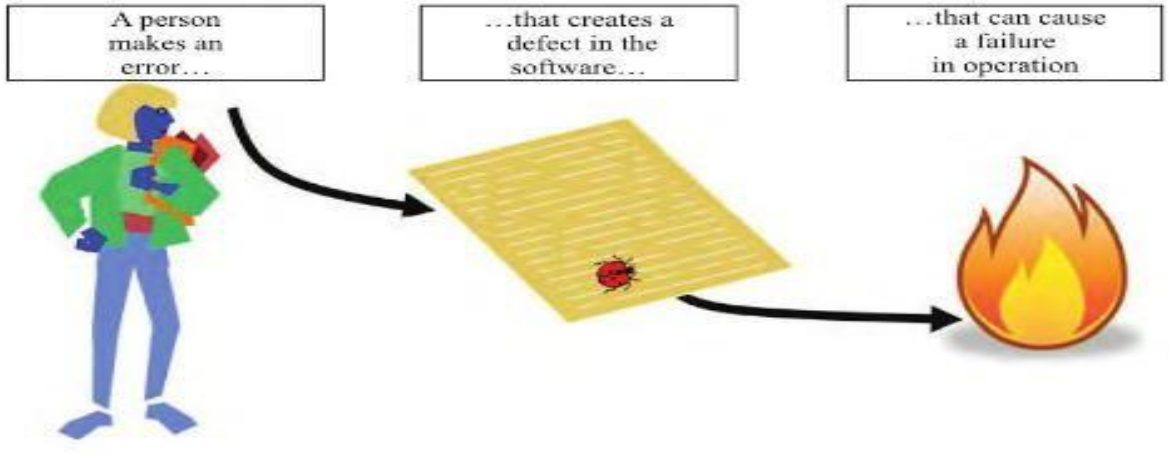


Figure from: SOFTWARE QUALITY ASSURANCE, TESTING AND METRICS by BASU, ANIRBAN

Defect Prevention and Detection

- Prevention: Lets the defect not arise
- Detection: If defect arises, lets detect and remove it

Inspection

- An inspection is a rigorous team review of a work product by peers of the producer of the work product
- The size of the team will vary with the characteristics of the work product being inspected; e.g., size, type
- Direct fault detection and removal

Testing

- The basic idea of testing involves the execution of software and the observation of its behavior or outcome
- If a failure is observed, the execution record is analyzed to locate and fix the faults that caused the failure
- Otherwise, we gain some confidence that the software under testing is more likely to fulfill its designated functions

SQA Group

- Every company, which wants to establish a reputation for producing high quality software, must establish a Software Quality Assurance (SQA) Group within the company
- This groups must be funded properly, and management must pay attention to the reports and presentations made by this group

Defect Quality Measurements

- Defect discovery points (i.e., inspections, tests, customer reports, etc.)
- Defect removal efficiency levels
- Normalized data (i.e., defects per function point or per KLOC)
- Causative factors (i.e., complexity, creeping requirements, etc.)
- Defect repair speeds or intervals from the first report to the release of the fix

Quality management

- SQM comprises of processes that ensure that the Software Project would reach its goals.
- In other words the Software Project would meet the clients expectations.
    - Quality planning
    - Quality assurance
    - Quality control

1. Quality planning

- Quality planning is the process of developing a quality plan for a project
- The starting point for the Planning process is the standards followed by the Organization.

    - **Inputs**
        - *Company's Quality Policy*
          b. Organization Standards
          c. Relevant Industry Standards
          d. Regulations
          e. Scope of Work
          f. Project Requirements
    - **Outputs**
    - *a. Standards defined for the Project*
          b. Quality Plan

- A quality plan sets out the desired product qualities and how these are assessed and define the most significant quality attributes
- It should define the quality assessment process
- It should set out which organisational standards should be applied and, if necessary, define new standards

Quality plan structure

- Product introduction
- Product plans
- Process descriptions
- Quality goals
- Risks and risk management

2. Quality assurance

- The Input to the Quality Assurance Processes is the Quality Plan created during Planning.
- Quality Audits and various other techniques are used to evaluate the performance.
- This helps us to ensure that the Project is following the Quality Management Plan.

3. Quality control

- Work done on deliverable is satisfactory or not?
- **Inputs:**
    - Quality Management Plan.
      2. Quality Standards for the Project.
      3. Actual Observations and Measurements of the work done or work in Progress.
- **Two approaches to quality control**
    - Quality reviews
    - Automated software assessment and software measurement

Quality reviews

- The principal method of validating the quality of a process or of a product
- Group examines part or all of system and its documentation to find potential problems

Quality assurance and standards

- Standards are the key to effective quality management
- They may be international, national, organizational or project standards
- Product standards define characteristics that all components should exhibit e.g. a common programming style
- Process standards define how the software process should be enacted

Importance of standards

- Encapsulation of best practice- avoids repetition of past mistakes
- Framework for quality assurance process - it involves checking standard compliance
- Provide continuity - new staff can understand the organisation by understanding the standards applied

Product and process standards

Product standards	Process standards
Design review form	Design review conduct
Document naming standards	Submission of documents to CM
Procedure header format	Version release process
Ada programming style standard	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

Problems with standards

- Not seen as relevant and up-to-date by software engineers
- Involve too much bureaucratic form filling
- Unsupported by software tools so tedious manual work is involved to maintain standards

Standards development

- Involve practitioners in development. Engineers should understand the rationale underlying a standard
- Review standards and their usage regularly.
  Standards can quickly become outdated and this reduces their credibility amongst practitioners
- Detailed standards should have associated tool support. Excessive clerical work is the most significant complaint against standards

Summary

- **SQA Principals/practices**
- **Quality Management**
- **QA and Standards**

Week-04: Software Testing

Overview of today's lecture

- **What is Software Testing?**
- **Software Testing Objectives/Principles.**
- **Successful Test**
- **Limitations of Testing.**
- **SDLC vs STLC**
- **Categories of testing techniques**

CS608 - SOFTWARE VERIFICATION AND VALIDATION

What is Software Testing?

Several Definitions:

- "Testing is the process of establishing confidence that a program or system does what it is supposed to." by Hetzel 1973
- "Testing is the process of executing a program or system with the intent of finding errors." by Myers 1979
- "Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results." by Hetzel 1983
- Software testing is the process of examining the software product against its requirements. Thus it is a process that involves verification of product with respect to its written requirements and conformance of requirements with user needs.
- From another perspective, software testing is the process of executing software product on test data and examining its output vis-à-vis the documented behavior.

Software Testing Objectives/Principles

- The correct approach to testing a scientific theory is not to try to verify it, but to seek to refute the theory. That is to prove that it has errors. (Popper 1965)
- The goal of testing is to expose latent defects in a software system before it is put to use.
- A software tester tries to break the system. The objective is to show the presence of a defect not the absence of it.
- Testing cannot show the absence of a defect. It only increases your confidence in the software.
- This is because exhaustive testing of software is not possible – it is simply too expansive and needs virtually infinite resources.

Successful Test

From the following sayings, a successful test can be defined;

- "If you think your task is to find problems then you will look harder for them than if you think your task is to verify that the program has none" – Myers 1979.
- "A test is said to be successful if it discovers an error" – doctor's analogy.

The success of a test depends upon;

- The ability to discover a bug not in the ability to prove that the software does not have one.
- Impossible to check all the different scenarios of a software application
- Our emphasis is on discovering all the major bugs that can be identified by running certain test scenarios.

Limitation of Testing

A function that compares two strings of characters stored in an array for equality.

bool isStringsEqual(char a[], char b[])

Inputs and Expected Outputs

A	B	Expected Result
“cat”	“dog”	False
“”	“”	True
“hen”	“hen”	True
“hen”	“heN”	False
“ “	“”	False
“”	“ball”	False
“cat”	“”	False
“HEN”	“hen”	False
“rat”	“door”	False
“ ”	“ “	True

Code of the Function

```
bool isStringsEqual(char a[], char b[])
{
        bool result = false;
        if (strlen(a) != strlen(b)) {
                result = false;
        }  else {
                for (int i =0; i < strlen(a); i++) {
                        if (a[i] == b[i])
                                result = true;
                        else
                                result = false;
                }
        }
        return result;
}
```

Analysis of Code

It passes all the designated tests but fails for two different strings of same length ending with the same character.

For example, “cat” and “rat” would results in true which is not correct.

Limitation of Testing

The above-mentioned defect signifies a clear limitation of the testing process in discovering a defect which is not very frequent.

However, it should be noted from this example that a tester cannot generate all possible combinations of test cases to test an application as the number of scenarios may become exhaustive.

In order to prove that a formula or hypothesis is incorrect all you have to do to show only one example in which you prove that the formula or theorem is not working.

To prove that it is correct any numbers of examples are insufficient. You have to give a formal proof of its correctness.

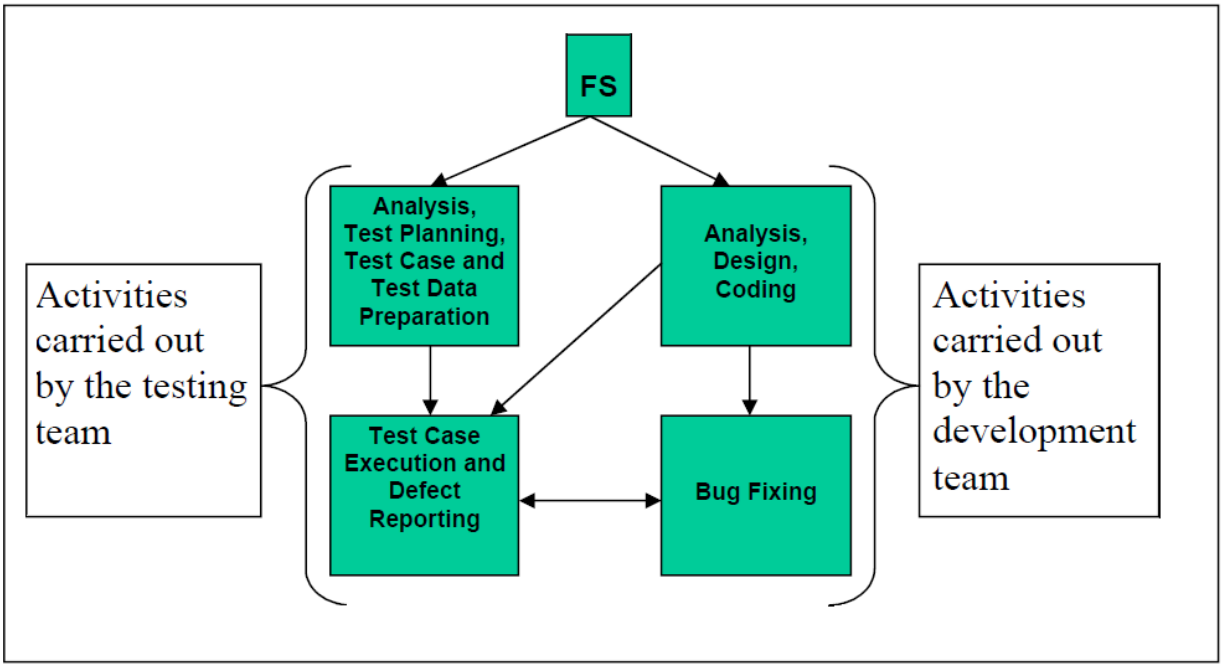
These examples only help you in coming up with a hypothesis but they are not proved by themselves.

They only enhance your comfort level in that particular hypothesis or in this particular case, in your piece of software.

Software Development Life Cycle vs Software Test Life Cycle

Development	Testing
Development is a creative activity	Testing is a destructive activity
Objective of development is to show that the program works	Objective of testing is to show that the program does not work

SDLC vs STLC



Description

- Functional specification document is the starting point, base document for both testing and the development
- Right side boxes describe the development, whereas, left side boxes explain the testing process
- Development team is involved into the analysis, design and coding activities.
- Whereas, testing team too is busy in analysis of requirements, for test planning, test cases and test data generation.
- System comes into testing after development is completed.
- Test cases are executed with test data and actual results (application behavior) are
- compared with the expected results,
- Upon discovering defects, tester generates the bug report and sends it to the development team for fixing.
- Development team runs the scenario as described in the bug report and try to reproduce the defect.
- If the defect is reproduced in the development environment, the development team identifies the root cause, fixes it and sends the patch to the testing team along with a bug resolution report.
- Testing team incorporates the fix (checking in), runs the same test case/scenario again and verifies the fix.
- If problem does not appear again testing team closes down the defect, otherwise, it is reported again.

Categories of testing techniques:

1. Specification based (Black Box) Testing
2. Structure based (White Box) Testing
3. Grey Box Testing
4. Others comprehensive software testing techniques for SDLC
5. Control flow oriented test construction techniques
6. Data flow oriented test construction techniques

Week-05: Black Box Testing

Specification based (Black Box) Testing

- **Definition of Black Box Testing**
- **Characteristics of Black Box Testing**
- **De/Merits**
- **Sub-Types**

Black Box Testing

- Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.
- Testing conducted to evaluate the compliance of a system or component with specified functional requirements.
- In this type of testing, a component or system is treated as a black box and it is tested for the required behavior.
- This type of testing is not concerned with how the inputs are transformed into outputs.
- As the system's internal implementation details are not visible to the tester. S/he gives inputs using an interface that the system provides and tests the output. If the outputs match with the expected results, system is fine otherwise a defect is found.

Characteristics of Black Box Testing

- Program is treated as a black box.
- Implementation details do not matter.
- Requires an end-user perspective.
- Criteria are not precise.
- Test planning can begin early.

Black Box Testing

- Testing without knowing the internal workings of the code
- WHAT a system does, rather than HOW it does it
- Typically used at System Test phase, although can be useful throughout the test lifecycle

- also known as specification based testing
- Applies for Functional and Non-Functional testing

Merits of Black Box Testing:

- Black box tests are reproducible.
- The environment the program is running is also tested.
- The invested effort can be used multiple times.
- More effective on larger units of code than glass box testing
- Tester needs no knowledge of implementation, including specific programming languages
- Tests are done from a user's point of view
- Will help to expose any ambiguities or inconsistencies in the specifications
- Efficient when used on Larger systems
- As the tester and developer are independent of each other, test is balanced and unbiased
- Tester can be non-technical.
- There is no need of having detailed functional knowledge of system to the tester.
- Tests will be done from an end user's point of view. Because end user should accept the system. (This is reason, sometimes this testing technique is also called as Acceptance testing)
- Testing helps to identify the vagueness and contradiction in functional specifications.
- Test cases can be designed as soon as the functional specifications are complete

De-Merits of Black Box Testing:

- The results are often overestimated.
- Not all properties of a software product can be tested
- The reason for a failure is not found.
- Only a small number of possible inputs can actually be tested, to test every possible input stream would take nearly forever

- Without clear and concise specifications, test cases are hard to design
- There may be unnecessary repetition of test inputs if the tester is not informed of test cases the programmer has already tried
- May leave many program paths untested
- Cannot be directed toward specific segments of code which may be very complex (and therefore more error prone)
- Most testing related research has been directed toward glass box testing
- Test cases are tough and challenging to design, without having clear functional specifications
- It is difficult to identify tricky inputs, if the test cases are not developed based on specifications.
- It is difficult to identify all possible inputs in limited testing time. So writing test cases is slow and difficult
- Chances of having unidentified paths during this testing
- Chances of having repetition of tests that are already done by programmer.

Sub-Types:

a) Equivalence Partitioning

b) Boundary Value Analysis

c) Decision Table Testing

d) State Transition Testing

e) Use Case Testing

f) Other black box test techniques

Equivalence Partitioning

- Aim is to treat groups of inputs as equivalent and to select one representative input to test them all
- Best shown in the following example….
- If we wanted to test the following IF statement:

- 'IF VALUE is between 1 and 100 (inclusive) (e.g. VALUE >=1 and VALUE <= 100) THEN …..'
- We could put a range of numbers as shown in the next slide through test cases
- If the tester chooses the right partitions, the testing will be accurate and efficient
- If the tester mistakenly thinks of two partitions as equivalent and they are not, a test situation will be missed
- Or on the other hand, if the tester thinks two objects are different and they are not, the tests will be redundant
- EP can be used for all Levels of Testing
- EP is used to achieve good input and output coverage, knowing exhaustive testing is often impossible
- It can be applied to human input, input via interfaces to a system, or interface parameters in integration testing

Boundary Value Analysis

- Boundary Value Analysis (BVA) uses the same analysis of partitions as EP and is usually used in conjunction with EP in test case design
- As with EP, it can be used for all Test levels
- BVA operates on the basis that experience shows us that errors are most likely to exist at the boundaries between partitions and in doing so incorporates a degree of negative testing into the test design
- BVA Test cases are designed to exercise the software on and at either side of boundary values

Decision Table Testing

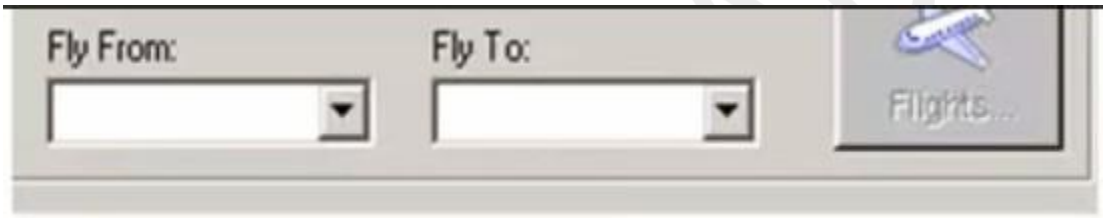
- Table based technique where
- Inputs to the system are recorded
- Outputs to the system are defined
- Inputs are usually defined in terms of actions which are Boolean (true or false)
- Outputs are recorded against each unique combination of inputs

- Using the Decision Table the relationships between the inputs and the possible outputs are mapped together
- Especially useful for complex business rules

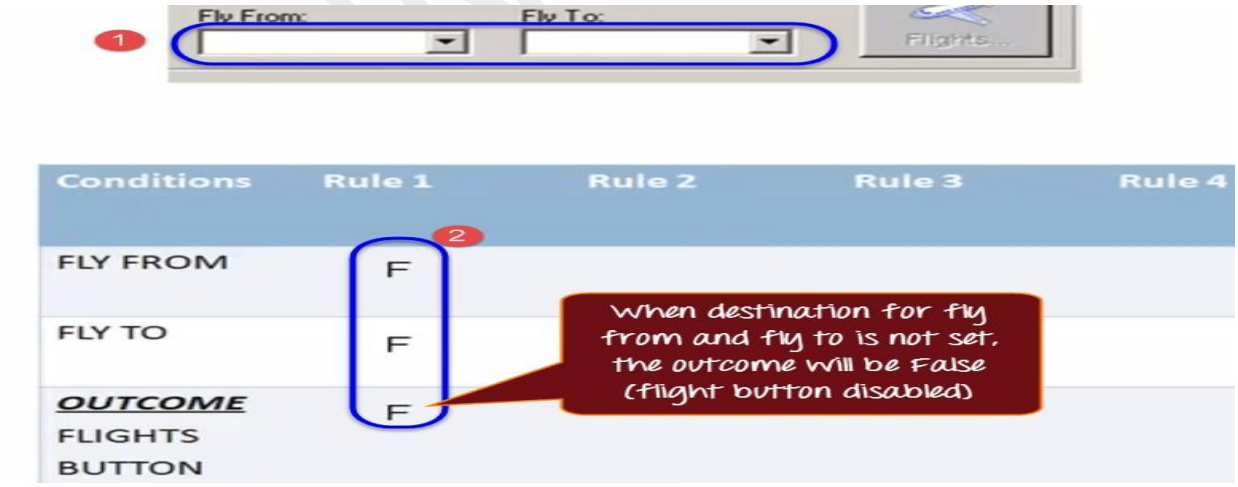
Decision table testing is black box test design technique to determine the test scenarios for complex business logic.

We can apply Equivalence Partitioning and Boundary Value Analysis techniques to only specific conditions or inputs. Although, if we have dissimilar inputs that result in different actions being taken or secondly we have a business rule to test that there are different combination of inputs which result in different actions. We use decision table to test these kinds of rules or logic.

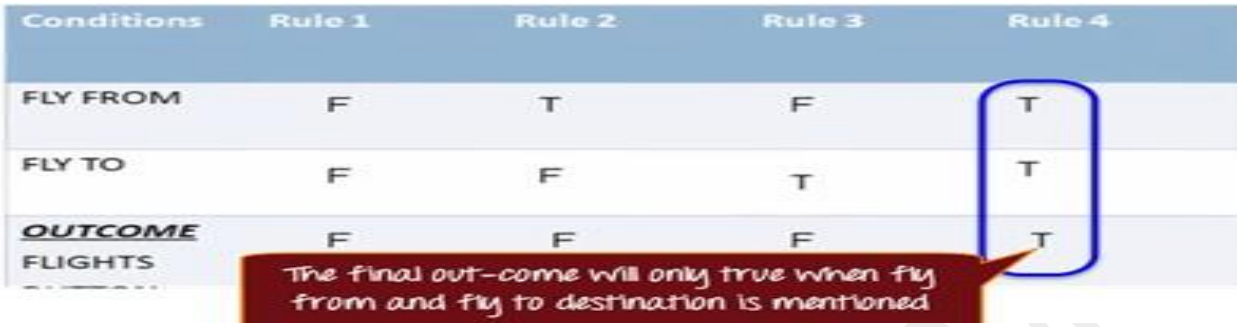
Example: let's consider the behavior of Flight Button for different combinations of Fly From & Fly To



Rule 1: When destination for both Fly From & Fly To are not set the Flight Icon is disabled.



Rule 2: When Fly From destination is set but Fly to is not set, Flight button is disabled.



State Transition Testing

State Transition testing, a black box testing technique, in which outputs are triggered by changes to the input conditions or changes to 'state' of the system. In other words, tests are designed to execute valid and invalid state transitions.

When to use?

When we have sequence of events that occur and associated conditions that apply to those events

When the proper handling of a particular event depends on the events and conditions that have occurred in the past

It is used for real time systems with various states and transitions involved

Deriving Test cases:

- Understand the various state and transition and mark each valid and invalid state
- Defining a sequence of an event that leads to an allowed test ending state
- Each one of those visited state and traversed transition should be noted down
- Steps 2 and 3 should be repeated until all states have been visited and all transitions traversed
- For test cases to have a good coverage, actual input values and the actual output values have to be generated

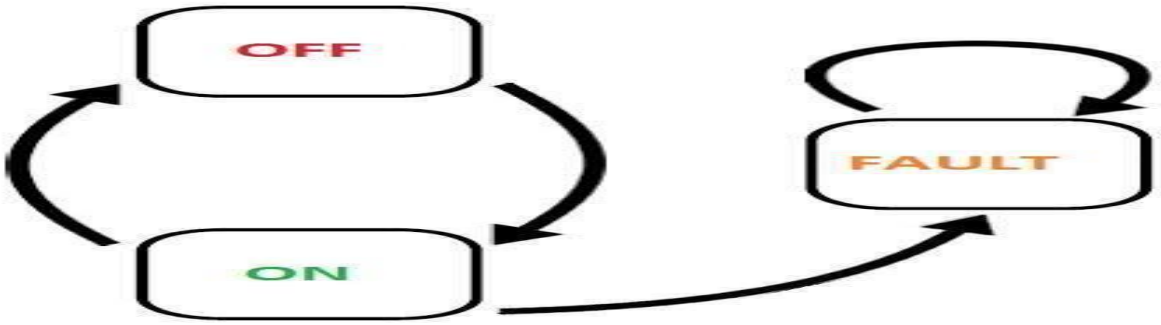
Advantages:

Allows testers to familiarize with the software design and enables them to design tests effectively.

It also enables testers to cover the unplanned or invalid states.

Example:

A System's transition is represented as shown in the below diagram:



Tests	Test 1	Test 2	Test 3
Start State	Off	On	On
Input	Switch ON	Switch Off	Switch off
Output	Light ON	Light Off	Fault
Finish State	ON	OFF	On

Use Case Testing

- Use Case Testing is a functional black box testing technique that helps testers to identify test scenarios that exercise the whole system on each transaction basis from start to finish.
- **Characteristics of Use Case Testing:**
- Use Cases capture the interactions between 'actors' and the 'system'.
- 'Actors' represents user and their interactions that each user takes part into.
- Test cases based on use cases and are referred as scenarios.
- Capability to identify gaps in the system which would not be found by testing individual components in isolation.

- Very effective in defining the scope of acceptance tests.

The main parts of a Use Case are:

Actors: Users of the systems

Pre conditions: What are the starting requirements for the use case

Post conditions: The state the system will end up in once completed

Other black box test techniques

- Syntax testing (critical item testing)
- test cases are prepared to exercise the rule governing the format of data in a system (e.g. a Zip or Postal Code, a telephone number)
- Random testing (using random generator)
- test cases are selected, possibly using a pseudo-random generation algorithm, to match an operational profile
- System testing of full application
- New function testing
- Lab testing
- Usability testing
- Customer acceptance testing
- Field (Beta) testing
- Clean-room statistical testing

Week-06: Software Quality Planning

Quality Planning

- A quality plan sets out the desired product qualities and how these are assessed and define the most significant quality attributes
- These are the activities that should be carried out before carrying out the regular QA activities
- Set specific quality goals.
- Form an overall QA strategy, which includes two sub-activities:
- ✓ Select appropriate QA activities to perform.
- ✓ Choose appropriate quality measurements and models to provide feedback, quality assessment and improvement.

Setting Quality Goals

- Identify quality views and attributes meaningful to target customers and users
- Select direct quality measures that can be used to measure the selected quality
- Quantify these quality measures

Setting QA Strategy

- Once specific quality goals were set, we can select appropriate QA alternatives as part of a QA strategy to achieve these goals.
    - *The influence of quality perspectives and attributes*
    - *The influence* of *different quality Levels*

Quality attributes

Maintainability	Maintainability is the ability of the system to undergo changes with a degree of ease.
Reusability	Reusability defines the capability for components and subsystems to be suitable for use in other applications and in other scenarios.
Availability	Availability defines the proportion of time that the system is functional and working.
Interoperability	Interoperability is the ability of a system or different systems to operate successfully by communicating and exchanging information with other external systems written and run by external parties.
Manageability	Manageability defines how easy it is for system administrators to manage the application.
Performance	Performance is an indication of the responsiveness of a system to execute any action within a given time interval.
Reliability	Reliability is the ability of a system to remain operational over time.
Scalability	Scalability is ability of a system to either handle increases in load without impact on the performance of the system.
Security	Security is the capability of a system to prevent malicious or accidental actions outside of the designed usage, and to prevent disclosure or loss of information.
Supportability	Supportability is the ability of the system to provide information helpful for identifying and resolving issues when it fails to work correctly.
Testability	Testability is a measure of how easy it is to create test criteria for the system and its components, and to execute these tests in order to determine if the criteria are met.
Usability	Usability defines how well the application meets the requirements of the user and consumer by being intuitive, easy to localize and globalize, providing good access for disabled users, and resulting in a good overall user experience.

Summary

- **QA Planning**
  - **Setting goals**
  - **Setting QA strategy**

SQA Plan (IEEE 730-2002)

SQA Plan

1. Purpose

This section shall describe specfic purpose and scope of the particular SQAP.

2. Reference Documents

This section shall provide a complete list of documents referenced elsewhere in the text of the SQAP.

3. Management

3.1 Organization

This section shall depict the organizational structure

3.2 Tasks

This section shall describe:

a) That portion of the software life cycle covered by the SQAP.

b) The tasks to be performed.

c) The entry and exit criteria for each task.

d) The relationships between these tasks and the planned major checkpoints.

3.3 Roles and responsibilities

This section shall identify the specific organizational element that is responsible for performing each task.

3.4 Quality assurance estimated resources

This section shall provide the estimate of resources and the costs to be expended on quality assurance and quality control tasks.

4. Documentation

4.1 Purpose

This section shall perform the following functions:

a. Identify the documentation governing the development, verification and validation, use, and maintenance of the software.

b. List which documents are to be reviewed or audited for adequacy.

4.2 Minimum documentation requirements

Minimum documentation to be maintained as part of development

The content of these documents may be included in other documents as long as traceability to the required information is maintained.

4.3 Other documentation

Identify other documents applicable to the software development project and software product that may be required.

5. Standards, practices, conventions, and metrics

This section shall:

a) Identify the standards, practices, conventions, statistical techniques to be used, quality requirements, and metrics to be applied e.g. Documentation standards, Design standards, Coding standards etc.

b) State how conformance with these items is to be monitored and assured.

6. Software reviews

This section shall:

a) Define the software reviews to be conducted.

b) Schedule for software reviews

c) How the software reviews shall be accomplished.

d) What further actions shall be required and how they shall be implemented and verified.

7. Test

This section shall identify all the tests not included in the software verification and validation plan for the software covered by the SQAP and shall state the methods to be used

8. Problem reporting and corrective action

This section shall:

- Describe the practices and procedures to be followed for reporting, tracking, and resolving problems.
- Specific organizational responsibilities concerned with their implementation.

9. Tools, techniques, and methodologies

This section shall identify the software tools, techniques, and methods used to support SQA processes.

10. Media control

The purpose of this section is to state the methods and facilities to be used.

- Media for each intermediate and deliverable computer work product
- Protection procedure of computer program physical media from unauthorized access etc.

11. Supplier control

This section shall state the provisions for assuring that software provided by suppliers meets established requirements.

12. Records collection, maintenance, and retention

This section shall identify the SQA documentation to be retained, shall state the methods and facilities to be used to assemble, file, safeguard, and maintain this documentation.

13. Training

This section shall identify the training activities necessary to meet the needs of the SQAP.

14. Risk management

This section shall specify the methods and procedures employed to identify, assess, monitor, and control areas of risk arising during the portion of the software life cycle covered by the SQAP.

15. Glossary

This section shall contain a glossary of terms unique to the SQAP.

16. SQAP change procedure and history

This section shall contain the procedures for modifying the SQAP and maintaining a history of the changes. It shall also contain a history of such modifications.

Summary

- **SQA Plan Template**
    - **IEEE 730-2002**

Week-07: Software Testing (Gray box testing White box testing)

Gray Box Testing

- Definition of Gray Box Testing
- Characteristics of Gray Box Testing
- De/Merits
- Sub-Types

Gray Box Testing

- Gray Box Testing is a combination of White Box testing and Black Box Testing.
- in which the tester has limited knowledge of the internal details of the program. A gray box is a device, program or system whose workings are partially understood.

Characteristics of Gray Box Testing

- Black box testers are unaware the internal structure of the system.
- White box testers do know the internal structure of the system.
- Gray Box testers partially know the internal structure and algorithms of defining test cases.
- Need overall and detailed description of documents of the application.

Merits of Gray Box Testing:

Offers combined benefits:

It serves advantages from both Black box and White box testing.

Non intrusive:

Based on functional specification, architectural view whereas not on source code of binaries which makes it invasive too.

Intelligent Test Authoring:

Tester handles intelligent test scenario, e.g. data type handling, communication protocol, exception handling etc.

Unbiased Testing:

It maintains boundary for testing between tester and developer.

De-Merits of Gray Box Testing:

Partial code coverage:

Source code or binaries are missing because of limited access to internal or structure of the applications which results in limited access for code path traversal.

Detect identification:

In distributed application, it is difficult to associate defect identification.

Sub-Types:

- Matrix Testing

  States the status report of the project.

- Regression Testing

  Rerunning of the test cases if new changes are made

- Pattern Testing

  Rerunning of the test cases if new changes are made

- Orthogonal Array Testing

  Used as subset of all possible combination

Structure based (White Box) Testing

- Definition of White Box Testing
- Characteristics of White Box Testing
- De/Merits
- Sub-Types

White Box Testing

- Testing based on an analysis of the internal structure of the component or system.
- testing based upon the structure of the code
- typically undertaken at Component and Component Integration Test phases by development teams
- also known as structural or glass box testing or structure based testing

As opposed to black box testing, in structural or white box testing we look inside the system and evaluate what it consists of and how is it implemented. The inner of a system consists of design, structure of code and its documentation etc. Therefore, in white box testing we analyze these internal structures of the program and devise test cases that can test these structures.

Characteristics of White Box Testing

The characteristics of white box testing limit its use to software modules of very high risk and very high cost of failure, where it is highly important to identify and fully correct as many of the software errors as possible.

Merits of White Box Testing:

- Forces test developer to reason carefully about implementation
- Approximates the partitioning done by execution equivalence
- Reveals errors in "hidden" code
- Beneficent side-effects
- As the knowledge of internal coding structure is prerequisite, it becomes very easy to find out which type of input/data can help in testing the application effectively.
- The other advantage of white box testing is that it helps in optimizing the code.
- It helps in removing the extra lines of code, which can bring in hidden defects.

De-Merits of White Box Testing:

- Expensive
- Miss cases omitted in the code
- As knowledge of code and internal structure is a prerequisite, a skilled tester is needed to carry out this type of testing, which increases the cost.
- And it is nearly impossible to look into every bit of code to find out hidden errors, which may create problems, resulting in failure of the application.
- Not looking at the code in a runtime environment. That's important for a number of reasons. Exploitation of vulnerability is dependent upon all aspects of the platform being targeted and source code is just of those components. The underlying operating system, the backend database being used, third party security tools, dependent libraries, etc. must all be taken into account when determining exploitability. A source code review is not able to take these factors into account.
- Very few white-box tests can be done without modifying the program, changing values to force different execution paths, or to generate a full range of inputs to test a particular function.

Sub-Types:

- Statement Testing
- Decision Testing
- Assessing Completeness (Coverage)
- Other White Box test techniques

Statement Testing

Statement coverage is a white box testing technique, which involves the execution of all the statements at least once in the source code. It is a metric, which is used to calculate and measure the number of statements in the source code which have been executed. Using this technique we can check what the source code is expected to do and what it should not. It can also be used to check the quality of the code and the flow of different paths in the program. The main drawback of this technique is that we cannot test the false condition in it.

Example:

Read A

Read B

if A>B

Print “A is greater than B”

else

Print "B is greater than A"

endif

Set1 :If A =5, B =2

No of statements Executed: 5

Total no of statements in the source code: 7

Statement coverage =5/7*100 = 71.00 %

Set1 :If A =2, B =5

No of statements Executed: 6

Total no of statements in the source code: 7

Statement coverage =6/7*100 = 85.20 %

This is purely a white box testing method. It tests the software’s internal coding and infrastructure and so the programmer is the one who should take the initiative to do this.This technique is very suitable for drupal programmers and other programmers.

Decision Testing

Decision coverage or Branch coverage is a testing method, which aims to ensure that each one of the possible branch from each decision point is executed at least once and thereby ensuring that all reachable code is executed.

That is, every decision is taken each way, true and false. It helps in validating all the branches in the code making sure that no branch leads to abnormal behavior of the application.

Read A Read B IF A+B > 10 THEN Print "A+B is Large" ENDIF If A > 5 THEN Print "A Large" ENDIF

To calculate Branch Coverage, one has to find out the minimum number of paths which will ensure that all the edges are covered. In this case there is no single path which will ensure

coverage of all the edges at once. The aim is to cover all possible true/false decisions. (1) 1A-2C-3D-E-4G-5H (2) 1A-2B-E-4F Hence Decision or Branch Coverage is 2.

Assessing Completeness (Coverage)

One of the basic truisms of software development is that it is not possible to fully test a program, so the basic question then becomes: how much testing is enough? In addition, for safety-critical systems, how can it be proved to the relevant authorities that enough testing has been performed on the software under development? The answer is software coverage analysis. While it has proven to be an effective metric for assessing test completeness, it only serves as an effective measure of test effectiveness when used within the framework of a disciplined test environment.

Other White Box test techniques

Subroutine testing

Unit testing

Viral protection testing

Stress or capacity testing

Performance testing

Security testing

Year 2000 testing

Week-08: (Software Quality)

Process Management

- Organization's software process assets include standard processes, SDLCs, Methodologies and tools, standards, tailoring guidelines for projects software process and a library of best practices.
- These are created, updated, maintained and made available to projects for use in developing, implementing and maintain the projects define software processes
- Software development is complex process.
- We have multiple options to choose from.
- Organizations should have defined software process.

- But every project has its own requirements
- So, we should have proper process management.
- Define a standard process
- Ensure that each project uses an appropriate version of the standard process
- Use results to improve the standard process

Standard Process Definition

- Organizations need to establish, document and maintain a standard software process.
- Staff and managers should participate in definition of standard process
- Carry out the definition activities in accordance with documented plan.

- **The process for standard process definition should address:**
  - Identify organization software process
  - Define/Change organization's software process
  - Document the process
  - Review and approve the process
  - Release, Distribution and Retirement of QMS document and tools.
  - Develop and maintain tailoring guidelines for projects
  - Guidelines for creating a release notice for announcing the new/modified process
  - Guidelines for reviewing new process documents
  - Guidelines on process release.
- The standard process describes and orders the software tasks that are common to all projects.
- It also contains guidelines for tailoring the standard process to meet needs of different projects.

- Each project has its own approved life cycle model that defines:
  - Required procedures, practices, methods and technologies
  - Applicable process and product standards
  - Responsibilities, authorities and staff interrelationships
  - Required tools and resources
  - Process dependencies and interfaces
  - Process outputs and completion criteria
  - Product and process measurements to be collected

Standard Process Measurements

- You cannot improve something if you cannot measure it.
- So it is essential to take measurements of the performance of standard software process.
- Measurements must be analyzed and process should be improved and find tuned.

- In developing a metrics program, following issues need to be resolved:
  - What should be measured
  - Why it should be measured
  - How it should be measured
  - Who should measure it
  - When and Where in the process it should be measure

- The selected metrics should:
  - Be linked to real customer requirements
  - Support the overall goals of measurement program
  - Support predefined analysis activities
  - Be consistent across all projects
  - Cover entire SDLC

- Examples:
  - Size, cost and schedule data
  - Productivity, effort, defect removal efficiency
  - Number of severity of defects in requirements, design and code

Defect Prevention

- Defect prevention is concerned with ensuring that that sources of defects that are inherent in the software process, or of defects that occur repeatedly are identified and eliminated.
- Defect prevention activities should be identified and implemented at both organizational and project level.

- The objective of defect prevention at organizational level is to identify and prioritize defects which organization wide impact and prevent them from occurring again
- Identified from:
  - Project wind up reports
  - Organizational metrics analysis
  - Audit and Assessment reports
  - Other organizational level meetings
- Root cause analysis and preventive actions should be taken and preventive actions should be done.
- At project level it is essential to include defect prevention activities in the project development plan and schedule.
- One successful method is to arrange a meeting during development to identify the defects and analyze their root cause.

Technology innovation

- Technology innovation or technology change means introduction of new technologies that are likely to improve capability of Quality Management System.

- These changes:
    - Help achieve quality standards resulting decrease in defects
    - Empowering to reduce process cycle times and increase process effectiveness
    - Improve capability of organizations standard software process
- Appropriate technology should be implemented in organization.
- It is therefore necessary to develop and maintain a plan for technology innovation.
- Plan should define long term technical strategy for automating and improving software process activities.
- Feedback on the status of new technologies and results that have been achieved.

Process change management

- Sources of input for process change are:
    - Reports from audits and assessments
    - Lessons learnt from monitor process
    - Change proposal from staff itself
    - Process and product measurement data
- A system must be established for employees to communicate the process change request to process engineering group.
- Group will evaluate request and then accept/reject it.
- The accepted change request is closed after implementing it.
- New process is release then.
- Change request should include:
    - Requester's name
    - Requester's contact no
    - Reference to process in QMS
    - Description of change requested
    - Justification for the change suggested
- For successful process, changes should be considered and analyzed.

Some interesting observations

- Organizations try to project more than their capacity.
- There is always a tension between development team and QA team.
- Customers are shown sophisticated measures.
- QA is not given much importance and normally is pressurized.
- Exact figures about quality aspects are not given.

Summary

- **Organizational initiatives**
- **Some common practices**

Week-09: (Unit Testing)

Other comprehensive software testing techniques for SDLC:

- Component/Unit testing
- Integration testing
- System testing
- Acceptance testing
- Regression testing

Component/Unit testing

- This type of testing is performed by developers before the setup is handed over to the testing team to formally execute the test cases. Unit testing is performed by the respective developers on the individual units of source code assigned areas. The developers use test data that is different from the test data of the quality assurance team.
- The goal of unit testing is to isolate each part of the program and show that individual parts are correct in terms of requirements and functionality.

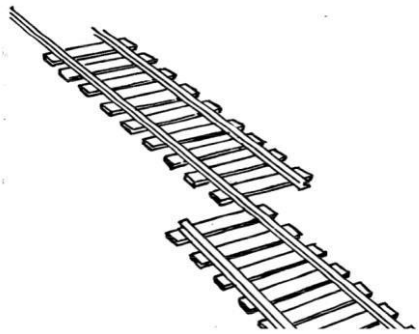
Why Unit Testing?

- **Reasons to bother unit testing:**
  - **Faster Debugging**
    - Unit tests smaller amounts of code, easily isolating points of error and narrowing the scope for errors.
  - **Faster Development**
    - Less time debugging with unit tests
    - Encourage aggressive refactoring resulting in better design and easier maintenance.
  - **Better Design**
    - As mentioned before, unit testing encouraging more refactoring.
    - Unit tests make developers focus more on the contracts of a class and those classes that might use it
  - **Reduce Future Cost**
    - Unit testing is an investment in the project. It takes time to ramp up, but in the long-term provides a useful basis for the project.

Terminology

- **Failure:** Any deviation of the observed behavior from the specified behavior
- **Erroneous state (error):** The system is in a state such that further processing by the system can lead to a failure
- **Fault:** The mechanical or algorithmic cause of an error ("bug")
- **Validation:** Activity of checking for deviations between the observed behavior of a system and its specification.

What is this?



A failure?

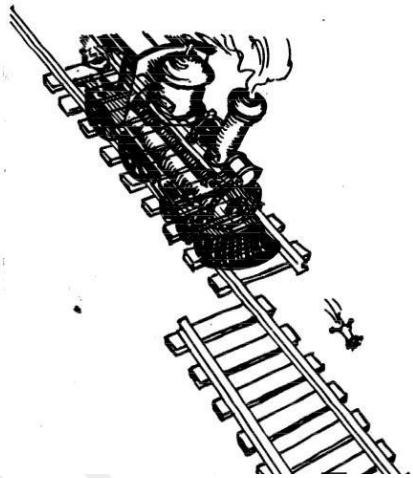
An error?

A fault?

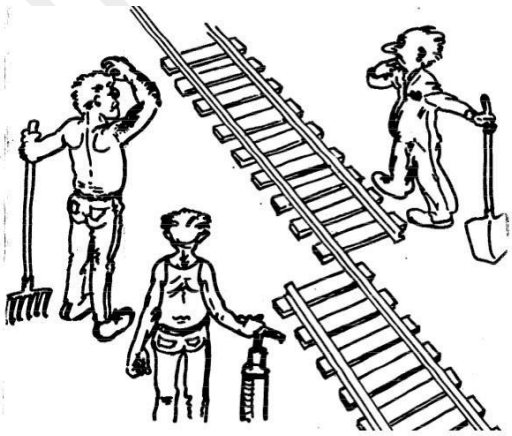
We need to describe specified

and desired behavior first!

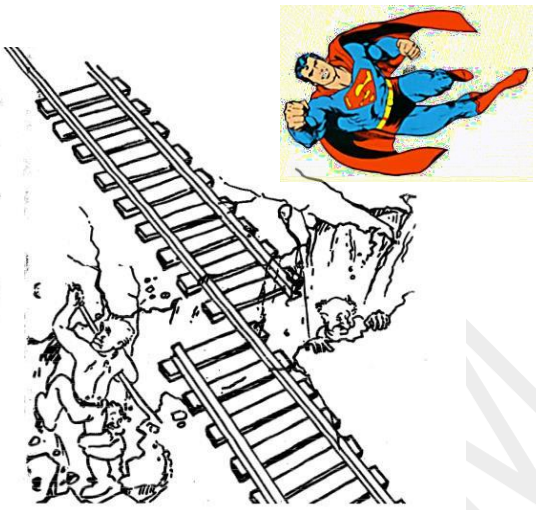
Erroneous State ("Error")



Algorithmic Fault

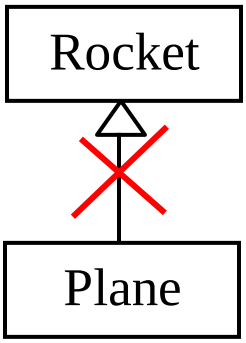


Mechanical Fault



F-16 Bug

- **What is the failure?**
- **What is the error?**
- **What is the fault?**
  - ❑ Bad use of implementation inheritance
  - ❑ A Plane is not a rocket.



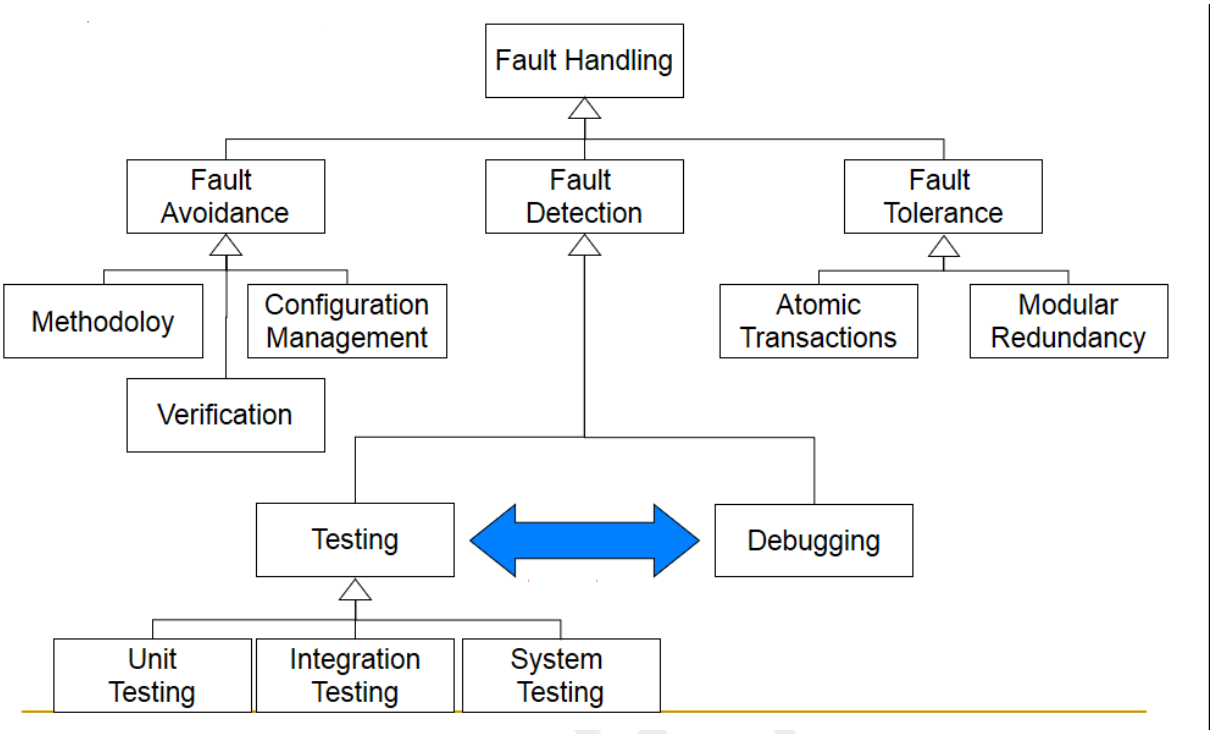
Examples of Faults and Errors

- **Faults in the Interface specification**
  - Mismatch between what the client needs and what the server offers
  - Mismatch between requirements and implementation
- **Algorithmic Faults**
  - Missing initialization
  - Incorrect branching condition
  - Missing test for null
- **Mechanical Faults (very hard to find)**
  - Operating temperature outside of equipment specification
- **Errors**
  - Null reference errors
  - Concurrency errors
  - Exceptions.

Another View on How to Deal with Faults

- **Fault avoidance**
  - Use methodology to reduce complexity
  - Use configuration management to prevent inconsistency
  - Apply verification to prevent algorithmic faults
  - Use Reviews
- **Fault detection**
  - Testing: Activity to provoke failures in a planned way
  - Debugging: Find and remove the cause (Faults) of an observed failure
  - Monitoring: Deliver information about state => Used during debugging
- **Fault tolerance**
  - Exception handling

Taxonomy for Fault Handling Techniques



Observations

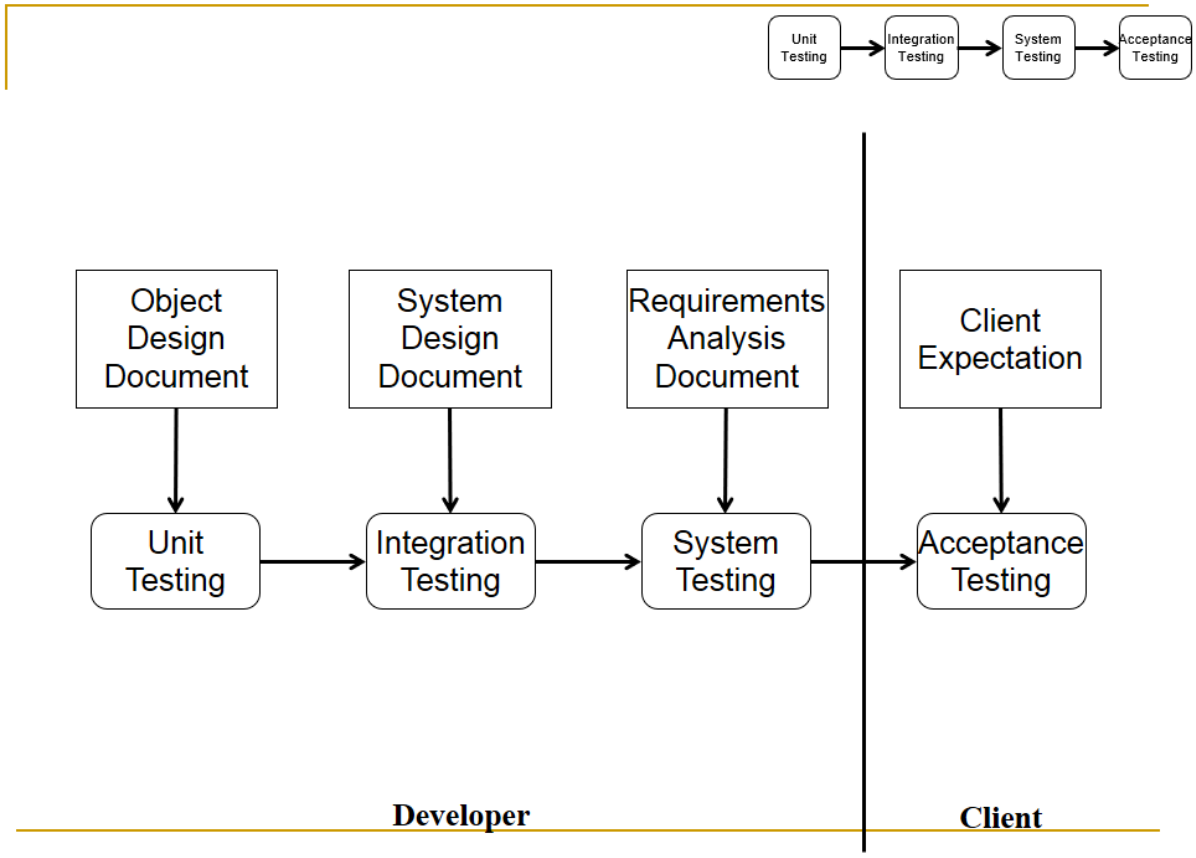
- **It is impossible to completely test any nontrivial module or system**
  - Practical limitations: Complete testing is prohibitive in time and cost
  - Theoretical limitations: e.g. Halting problem
- **"Testing can only show the presence of bugs, not their absence" (Dijkstra).**
- **Testing is not for free**

=> Define your goals and priorities

Testing takes creativity

- To develop an effective test, one must have:
  - ❑ Detailed understanding of the system
  - ❑ Application and solution domain knowledge
  - ❑ Knowledge of the testing techniques
  - ❑ Skill to apply these techniques
- Testing is done best by independent testers

Testing Activities



Types of Testing

- **Unit Testing**
  - Individual component (class or subsystem)
  - Carried out by developers
  - Goal: Confirm that the component or subsystem is correctly coded and carries out the intended functionality
- **Integration Testing**
  - Groups of subsystems (collection of subsystems) and eventually the entire system
  - Carried out by developers
  - Goal: Test the interfaces among the subsystems.
- **System Testing**
  - The entire system
  - Carried out by developers
  - Goal: Determine if the system meets the requirements (functional and nonfunctional)
- **Acceptance Testing**
  - Evaluates the system delivered by developers
  - Carried out by the client. May involve executing typical transactions on site on a trial basis
  - Goal: Demonstrate that the system meets the requirements and is ready to use.

When should you write a test?

- Traditionally after the source code is written
- In XP before the source code written
  - Test-Driven Development Cycle
    - Add a test
    - Run the automated tests

    => see the new one fail
    - Write some code

- Run the automated tests

  => see them succeed

- Refactor code.

Unit Testing

- **Static Testing (at compile time)**
  - Static Analysis
  - Review
    - Walk-through (informal)
    - Code inspection (formal)
  - **Dynamic Testing (at run time)**
  - Black-box testing
  - White-box testing.

Static Analysis with Eclipse

- **Compiler Warnings and Errors**
  - *Possibly uninitialized Variable*
  - *Undocumented empty block*
- **Checkstyle**
  - Check for code guideline violations
  - http://checkstyle.sourceforge.net
- **FindBugs**
  - Check for code anomalies
  - http://findbugs.sourceforge.net
- **Metrics**
  - Check for structural anomalies
  - http://metrics.sourceforge.net

Black-box testing

- **Focus: I/O behavior**
  - If for any given input, we can predict the output, then the component passes the test
  - Requires test oracle
- **Goal: Reduce number of test cases by equivalence partitioning:**
  - Divide input conditions into equivalence classes
  - Choose test cases for each equivalence class.

Black-box testing: Test case selection

a) Input is valid across range of values

- Developer selects test cases from 3 equivalence classes:
  - Below the range
  - Within the range
  - Above the range

b) Input is only valid, if it is a member of a discrete set

- Developer selects test cases from 2 equivalence classes:
  - Valid discrete values
  - Invalid discrete values
  - No rules, only guidelines.

Black box testing: An example

```
public class MyCalendar {
    public int getNumDaysInMonth(int month, int year)
        throws InvalidMonthException
    { … }
}
```

Representation for month:

1: January, 2: February, …., 12: December

Representation for year:

1904, … 1999, 2000,…, 2006, …

How many test cases do we need for the black box testing of getNumDaysInMonth()?

White-box testing overview

- Code coverage
- Branch coverage
- Condition coverage
- Path coverage

Unit Testing Heuristics

1. Create unit tests when object design is completed

- ❑ Black-box test: Test the functional model
- ❑ White-box test: Test the dynamic model

2. Develop the test cases

- ❑ Goal: Find effective number of test cases

3. Cross-check the test cases to eliminate duplicates

- ❑ Don't waste your time!

4. Desk check your source code

- ❑ Sometimes reduces testing time

5. Describe the test oracle

- ❑ Often the result of the first successfully executed test

6. Execute the test cases

- ❑ Re-execute test whenever a change is made ("regression testing")

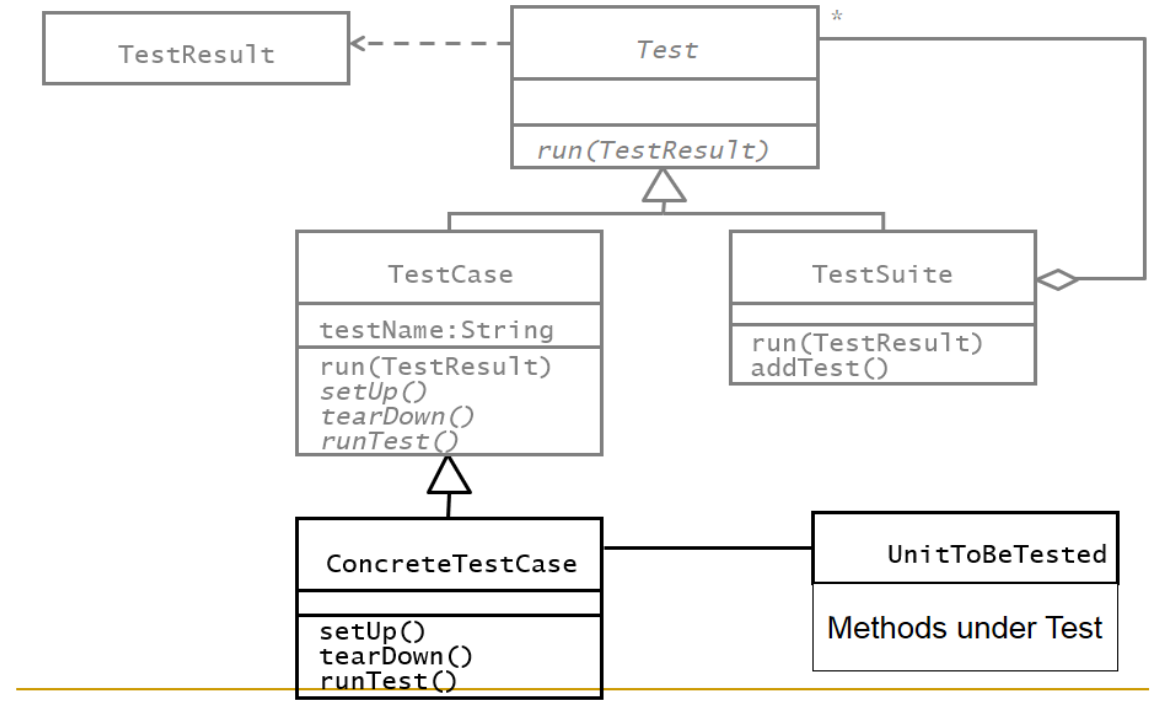
7. Compare the results of the test with the test oracle

- ❑ Automate this if possible.

JUnit: Overview

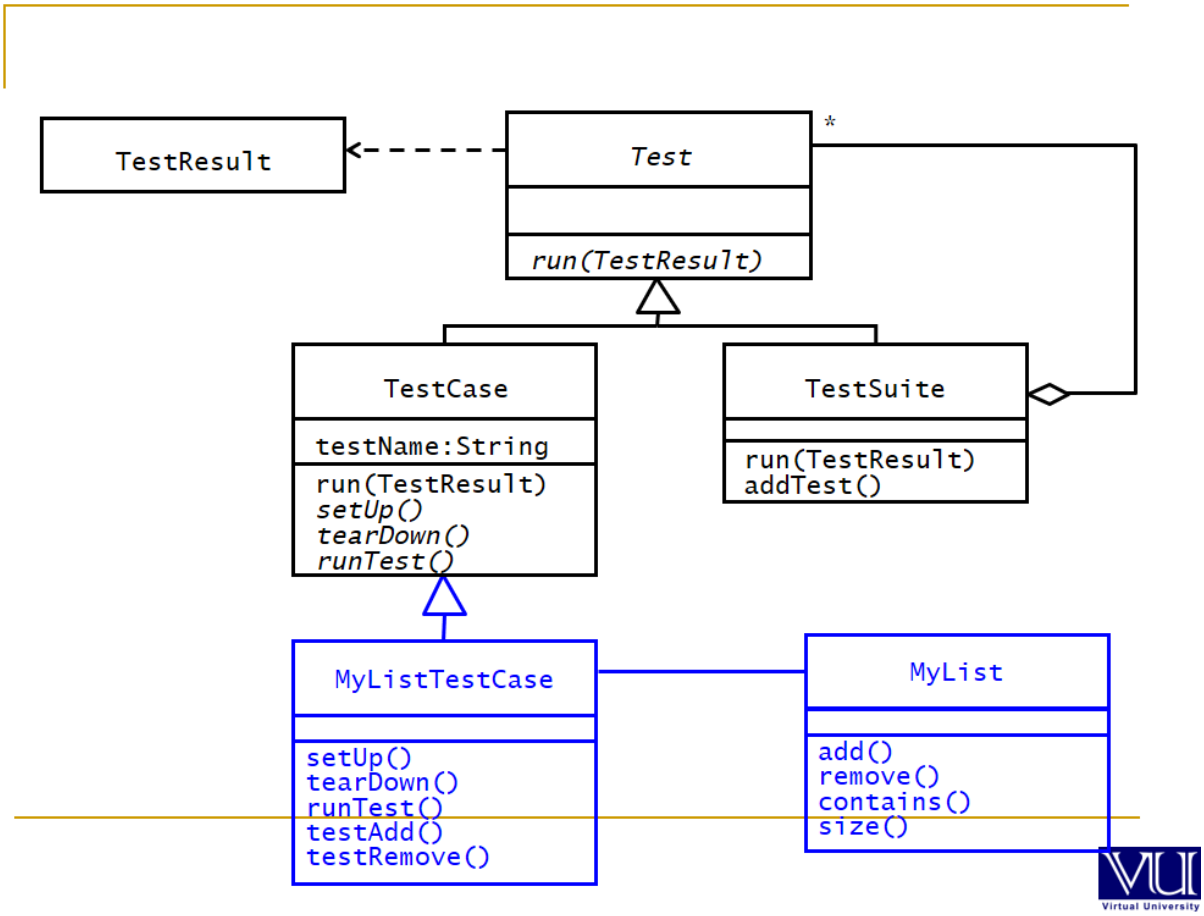
- **A Java framework for writing and running unit tests**
  - Test cases and fixtures
  - Test suites
  - Test runner
- **Written by Kent Beck and Erich Gamma**
- **Written with "test first" and pattern-based development in mind**
  - Tests written before code
  - Allows for regression testing
  - Facilitates refactoring
- **JUnit is Open Source**
  - www.junit.org
  - JUnit Version 4, released Mar 2006

JUnit Classes



An example: Testing MyList

- **Unit to be tested**
  - MyList
- **Methods under test**
  - add()
  - remove()
  - contains()
  - size()
- **Concrete Test case**
  - MyListTestCase



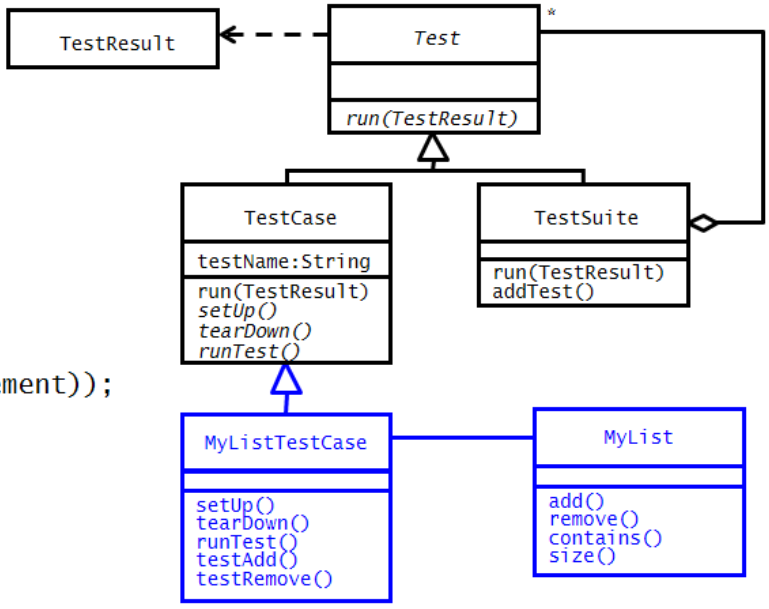
Writing TestCases in Junit

```
public class MyListTestCase extends TestCase {

public MyListTestCase(String name) {
    super(name);
}
public void testAdd() {
    // Set up the test
    List aList = new MyList();
    String anElement = "a string";

    // Perform the test
    aList.add(anElement);

    // Check if test succeeded
    assertTrue(aList.size() == 1);
    assertTrue(aList.contains(anElement));
}
protected void runTest() {
    testAdd();
}
}
```



Writing Fixtures and Test Cases

```
public class MyListTestCase extends TestCase {
// …
private MyList aList;
private String anElement;
public void setUp() {
    aList = new MyList();
    anElement = "a string";
}

public void testAdd() {
    aList.add(anElement);
    assertTrue(aList.size() == 1);
    assertTrue(aList.contains(anElement));
}
```

Writing Fixtures and Test Cases

```
public class MyListTestCase extends TestCase {
// …
private MyList aList;
private String anElement;
public void setUp() {
    aList = new MyList();
    anElement = "a string";
}

public void testAdd() {
    aList.add(anElement);
    assertTrue(aList.size() == 1);
    assertTrue(aList.contains(anElement));
}

public void testRemove() {
    aList.add(anElement);
    aList.remove(anElement);
    assertTrue(aList.size() == 0);
    assertFalse(aList.contains(anElement));
}
```

Test Fixture

Test Case

Test Case

Limitations of Unit Testing

- Testing cannot catch each and every bug in an application. It is impossible to evaluate every execution path in every software application. The same is the case with unit testing.
- There is a limit to the number of scenarios and test data that a developer can use to verify a source code. After having exhausted all the options, there is no choice but to stop unit testing and merge the code segment with other units.

Integration Testing

- Integration testing is defined as the testing of combined parts of an application to determine if they function correctly. Integration testing can be done in many ways: Bottom-up integration testing and Top-down integration testing.
- http://istqbexamcertification.com/what-is-integration-testing/

- **Bottom-up integration**

This testing begins with unit testing, followed by tests of progressively higher-level combinations of units called modules or builds.

- **Top-down integration**

In this testing, the highest-level modules are tested first and progressively, lower-level modules are tested thereafter.

In a comprehensive software development environment, bottom-up testing is usually done first, followed by top-down testing. The process concludes with multiple tests of the complete application, preferably in scenarios designed to mimic actual situations.

System Testing

- System testing tests the system as a whole. Once all the components are integrated, the application as a whole is tested rigorously to see that it meets the specified Quality Standards.
- This type of testing is performed by a specialized testing team.
- System testing is the first step in the Software Development Life Cycle, where the application is tested as a whole.
- https://www.tutorialspoint.com/software_testing_dictionary/system_testing.htm

Week-10: (Control Flow Testing)

Control-Flow Testing (Dependable Software Systems)

Control-Flow Testing

- Control-flow testing is a structural testing strategy that uses the program's control flow as a model.
- Control-flow testing techniques are based on judiciously selecting a set of test paths through the program.
- The set of paths chosen is used to achieve a certain measure of testing thoroughness.
- *E.g.*, pick enough paths to assure that every source statement is executed as least once.

Motivation

- Control-flow testing is most applicable to new software for unit testing.
- Control-flow testing assumptions:
  - specifications are correct
  - data is defined and accessed properly
  - there are no bugs other than those that affect control flow
- Structured and OO languages reduce the number of control-flow bugs.

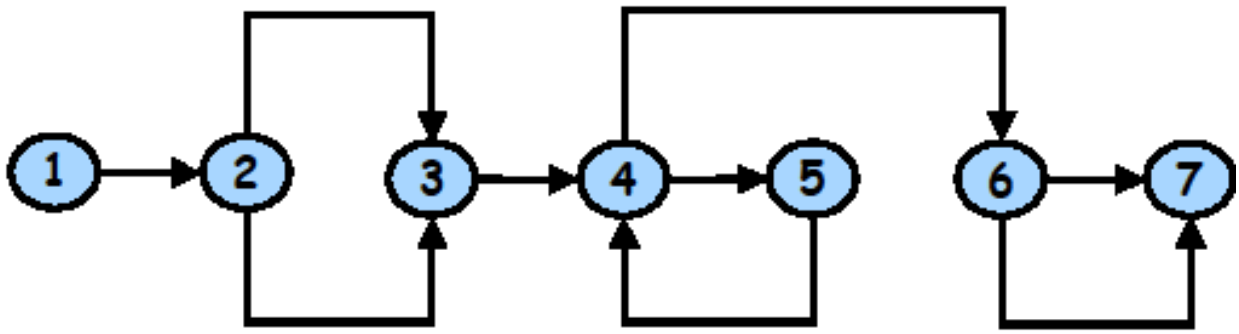
Control Flowgraphs

- The control flowgraph is a graphical representation of a program's control structure.
- Flowgraphs Consist of Three Primitives
  - A decision is a program point at which the control can diverge. (*e.g.*, if and case statements).
  - A junction is a program point where the control flow can merge. (*e.g.*, end if, end loop, goto label)
  - A process block is a sequence of program statements uninterrupted by either decisions or junctions. (*i.e.*, straight-line code).
    - A process has one entry and one exit.

- A program does not jump into or out of a process.

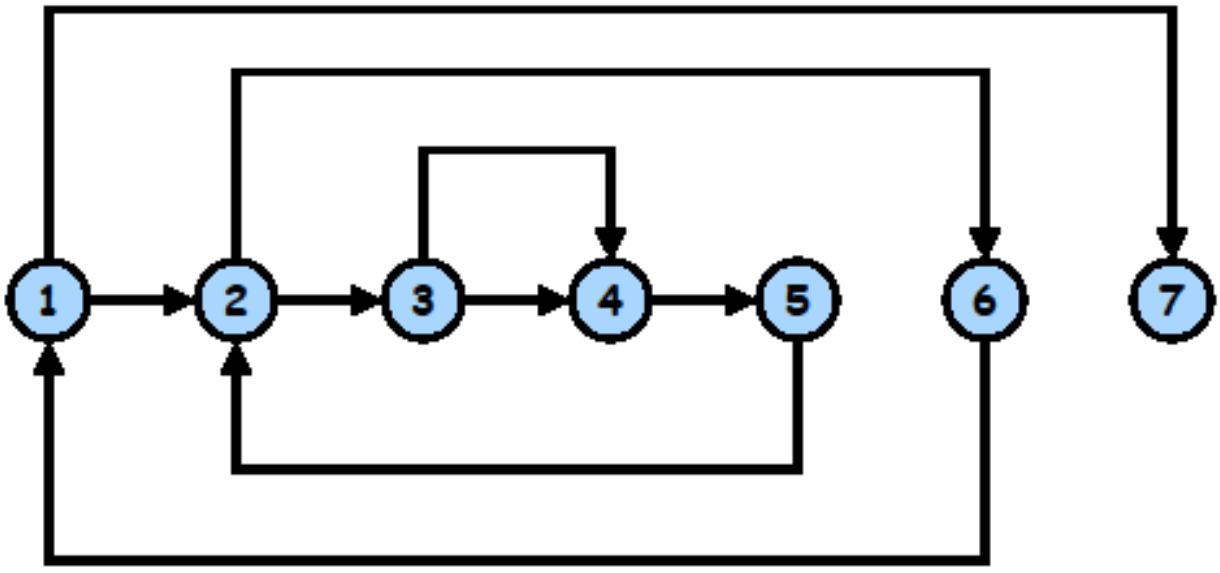
Exponentiation Algorithm

```
1       scanf("%d %d",&x, &y);
2       if (y < 0)
                pow = -y;
        else
                pow = y;
3       z = 1.0;
4       while (pow != 0) {
                z = z * x;
                pow = pow - 1;
5       }
6       if (y < 0)
                z = 1.0 / z;
7       printf ("%f",z);
```



Bubble Sort Algorithm

```
1       for (j=1; j<N; j++) {
                last = N - j + 1;
2               for (k=1; k<last; k++) {
3                   if (list[k] > list[k+1]) {
                        temp = list[k];
                        list[k] = list[k+1];
                        list[k+1] = temp;
4                   }
5               }
6       }
7       print("Done\n");
```



Control-flow Testing Criteria

- **3 testing criteria:**
  - **Path Testing:**
    - 100% path coverage
    - Execute all possible control flow paths through the program.
  - **Statement Testing:**
    - 100% statement coverage.
    - Execute all statements in a program at least once under some test.
  - **Branch Testing:**
    - 100% branch coverage.
    - Execute enough tests to assure that every branch alternative has been exercised at least once under some test.

Two Detailed Examples of Control-flow Testing

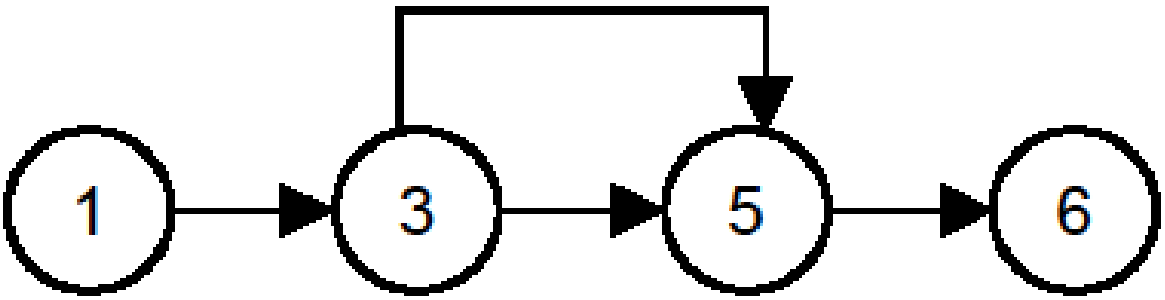
Using Control-flow Testing to Test Function ABS

- **Consider the following function:**

```
/* ABS
   This program function returns the absolute value of the integer
passed to the function as a parameter.
   INPUT: An integer.
   OUTPUT: The absolute value if the input integer.
*/
1     int ABS(int x)
2     {
3           if (x < 0)
4                 x = -x;
5           return x;
6     }
```

The Flowgraph for ABS

```
/* ABS
   This program function returns the absolute value of the integer
passed to the function as a parameter.
   INPUT: An integer.
   OUTPUT: The absolute value if the input integer.
*/
1       int ABS(int x)
2       {
3               if (x < 0)
4                       x = -x;
5               return x;
6       }
```



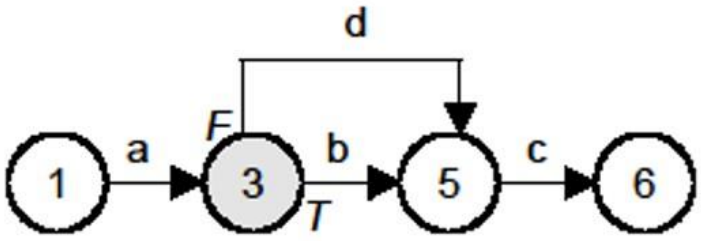
Test Cases to Satisfy Path Coverage for ABS

- *ABS* takes as its input any integer. There are many integers (depending on the maximum size of an integer for the language) that could be input to *ABS* making it impractical to test all possible inputs to ABS.

Test Cases to Satisfy Statement Testing Coverage for ABS

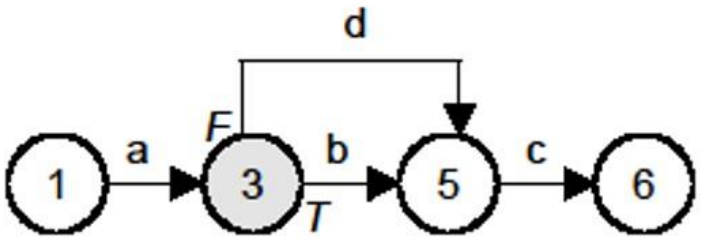
Test Cases to Satisfy **Statement Testing** Coverage for ABS

PATHS	PROCESS LINKS				TEST CASES	
	a	b	c	d	INPUT	OUTPUT
abc	√	√	√		A Negative Integer, x	-x
adc	√		√	√	A Positive Integer, x	x



Test Cases to Satisfy **Statement Testing** Coverage for ABS

PATHS	PROCESS LINKS				TEST CASES	
	a	b	c	d	INPUT	OUTPUT
abc	√	√	√		A Negative Integer, x	-x
adc	√		√	√	A Positive Integer, x	x



Example: Using Control-flow Testing to Test Program COUNT

- **Consider the following function:**

  /* COUNT

  This program counts the number of characters and lines in a text file.

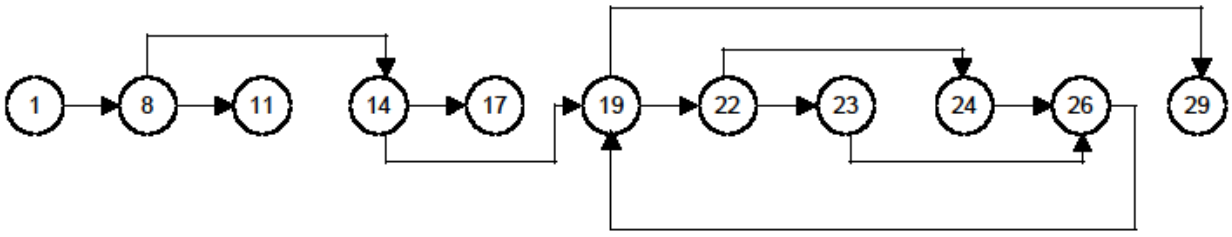
  INPUT: Text File

  OUTPUT: Number of characters and number of lines.

  ***/**

```
1 main(int argc, char *argv[])
2 {
3   int numChars = 0;
4   int numLines = 0;
5   char chr;
6   FILE *fp = NULL;
8   if (argc < 2)
9   {
10      printf("\nUsage: %s <filename>", argv[0]);
11      return (-1);
12  }
13  fp = fopen(argv[1], "r");
14  if (fp == NULL)
15  {
16      perror(argv[1]); /* display error message */
17      return (-2);
18  }
19 while (!feof(fp))
20  {
21      chr = getc(fp); /* read character */
22      if (chr == '\n') /* if carriage return */
23          ++numLines;
24      else
25          ++numChars;
26  }
27  printf("\nNumber of characters = %d", numChars);
28  printf("\nNumber of lines = %d", numLines);
29 }
```

The Flowgraph for COUNT

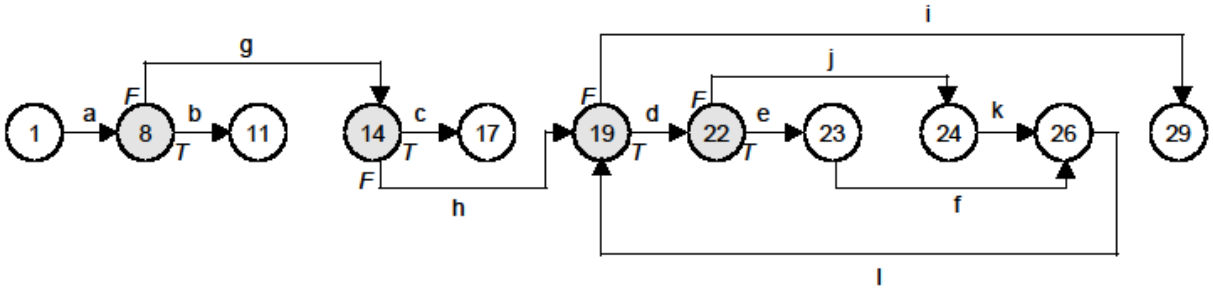


- The junction at line 12 and line 18 are not needed because if you are at these lines then you must also be at line 14 and 19 respectively.

Test Cases to Satisfy Path Coverage for COUNT

- Complete path testing of *COUNT* is impossible because there are an infinite number of distinct text files that may be used as inputs to *COUNT*.

Test Cases to Satisfy Statement Testing Coverage for COUNT



Test Cases to Satisfy **Statement Testing** Coverage for COUNT

PATHS	PROCESS LINKS												TEST CASES	
	a	b	c	d	e	f	g	h	i	j	k	l	INPUT	OUTPUT
ab	√	√											None	"Usage: COUNT <filename>"
agc	√		√				√						Invalid Input Filename	Error Message
aghdj kli	√			√			√	√	√	√	√	√	Input File with one character and no Carriage Return at the end of the line	Number of characters = 1 Number of lines = 0
aghd efli	√			√	√	√	√	√	√			√	Input file with no characters and one carriage return	Number of characters = 0 Number of lines = 1

Test Cases to Satisfy **Statement Testing** Coverage for COUNT

PATHS	PROCESS LINKS												TEST CASES	
	a	b	c	d	e	f	g	h	i	j	k	l	INPUT	OUTPUT
ab	√	√											None	"Usage: COUNT <filename>"
agc	√		√				√						Invalid Input Filename	Error Message
aghdj kli	√			√			√	√	√	√	√	√	Input File with one character and no Carriage Return at the end of the line	Number of characters = 1 Number of lines = 0
aghd efli	√			√	√	√	√	√	√			√	Input file with no characters and one carriage return	Number of characters = 0 Number of lines = 1

Week-11: User Interface Testing

Non functional Requirements Testing

Agenda

- **Previous Study**
- **Functional Requirements Testing**
- **White Box Testing**
- **Black Box Testing**
- **User Interface Testing (Usability Testing)**

Analytical Evaluation

- **Heuristic Evaluation**
- **Heuristic**
  - **Meaning:** guideline, general principle
  - Can guide a design decision
  - Can be used to critique a decision

- **Heuristic evaluation**

  - Method for structuring a critique of a system
  - Developed by Nielson and Molich
  - Flexible and relatively cheap approach
    - Uses a set of relatively simple and general heuristics
    - Useful for evaluating early designs
    - Can also be used on prototypes, storyboards, fully functioning systems

- **General idea**
  - Several evaluators independently critique a system
  - **Goal:** come up with potential usability problems
  - Number of evaluators needed [Nielson]
    - Between three and five is sufficient
    - Five usually result in ~75% of the overall usability problems being discovered

- Evaluators assess the severity of each problem (rating on a scale of 0-4)
  - 0 = no problem at all
  - 1 = cosmetic problem only (need not to be fixed)
  - 2 = minor usability problem (fixing has low priority)
  - 3 = major usability problem (important to fix)
  - 4 = usability catastrophe (imperative to fix)

- **Set of 10 heuristics [Nielson]**

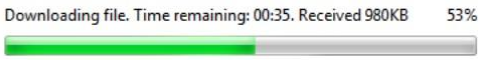
1. Visibility of system status
2. Match between system and real world
3. User control and freedom (e.g. support undo + redo)
4. Consistency and standards
5. Error prevention
6. Recognition rather than recall
7. Flexibility and efficiency of use
8. Aesthetic and minimalistic design
9. Help users recognize, diagnose and recover from errors
10. Help and documentation

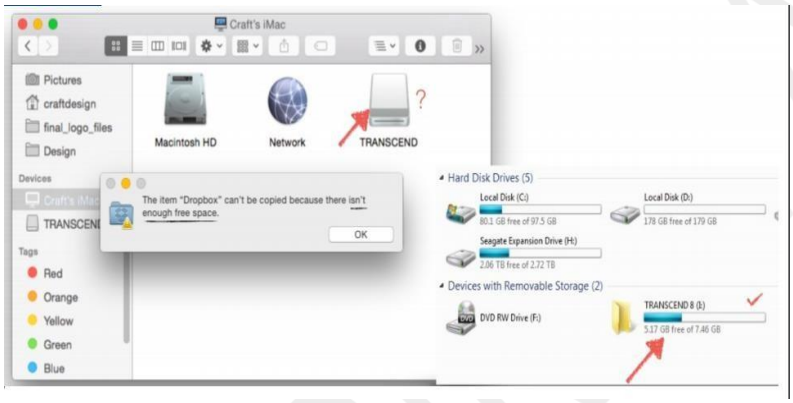
Nielson 10 Heuristics: Examples

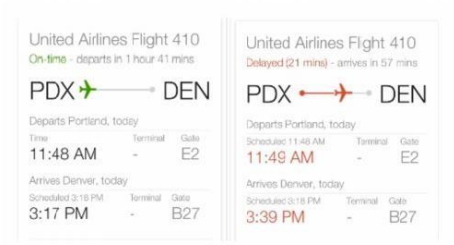
1. Visibility of system status

The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

- **0.1 sec:** Feels immediate to the user. No additional feedback needed.
- **1.0 sec:** Tolerable, but does not feel immediate. Some feedback needed.
- **10 sec:** Maximum duration for keeping user's focus on the action.
- **For longer delays,** use % done progress bars.

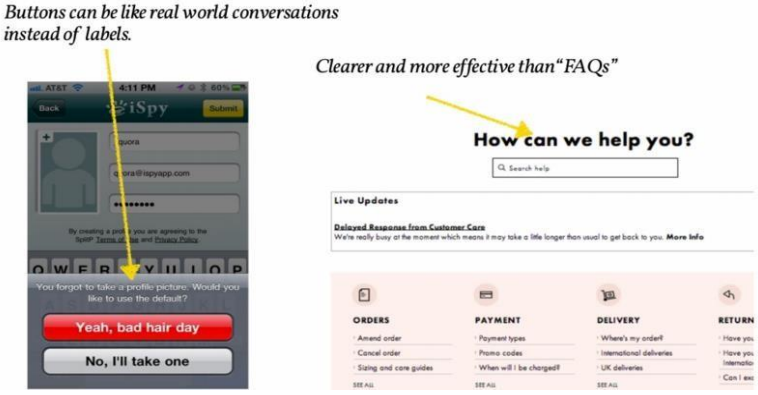






2. Match between system and real world

The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than systemoriented terms. Follow real-world conventions, making information appear in a natural and logical order.



Refrain from confusing people with system oriented language and design.

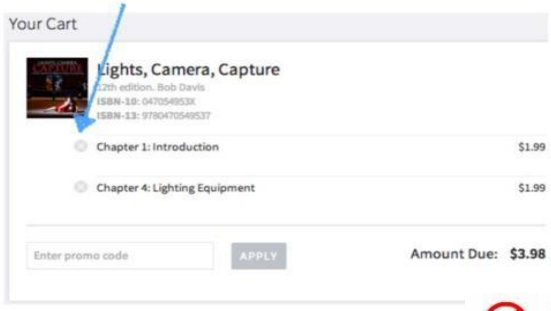
3. User control and freedom (e.g. support undo + redo)

Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

Ever felt the need for an undo button after sending an email to the wrong person?

It's a good thing Gmail allows that.

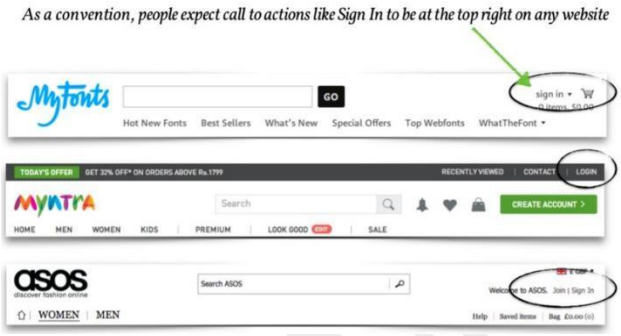
Allow people to change or delete items in a shopping cart as seen in this example from Inkling. It's also useful to allow them to continue shopping.

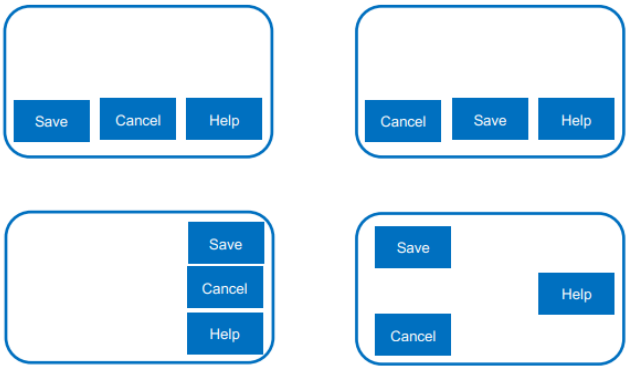


There is no way to undo a text edit on Android!
Avoid pissing the user off by giving him no choice to revert to an earlier state.

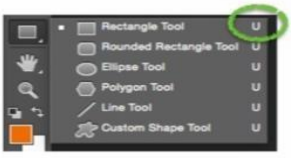
4. Consistency and standards

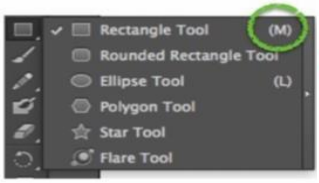
Users should not have to wonder whether different words, situations, or actions mean the same thing.





Similar functions have different shortcuts in Adobe's various software.





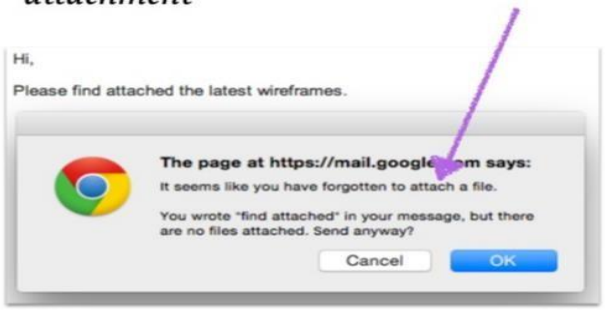
One would expect the knob to be the volume control, but it's not.

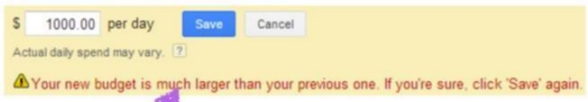
HCI – 3.3.15

5. Error prevention

Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

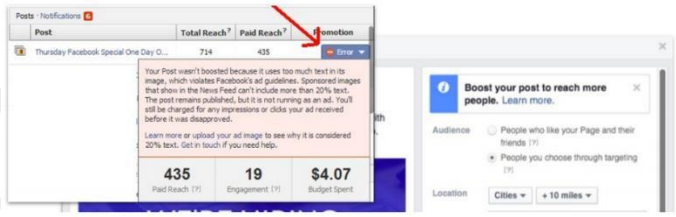
Gmail prompts you when you forget to insert an attachment





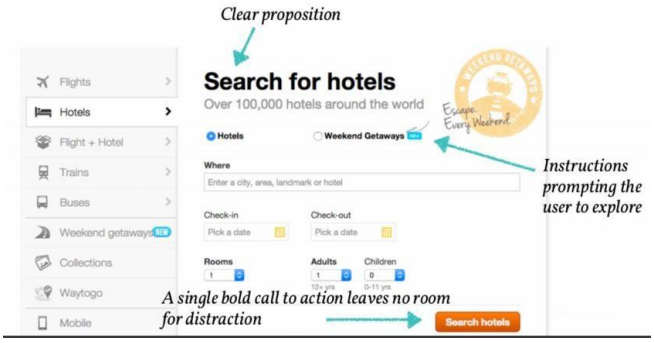
On Google Ads, if you input a budget much larger than your previous one, the system checks with you if you really meant to do so.

Facebook does not try and prevent you from posting an ad that is against its guidelines. e.g. There is no way for a first time user to assume that his ad might get pulled off mid-campaign because Facebook might later find it inappropriate.

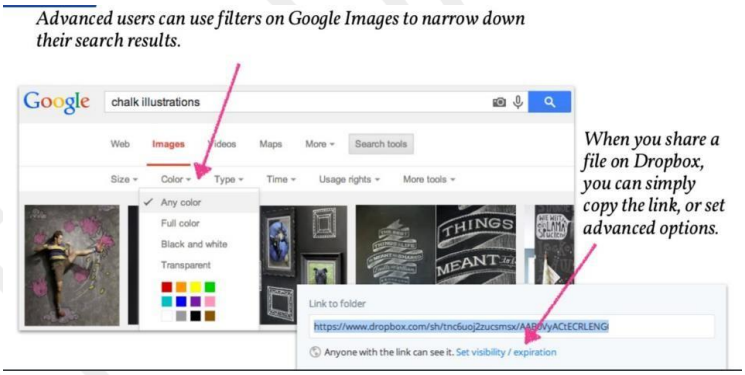


6. Recognition rather than recall

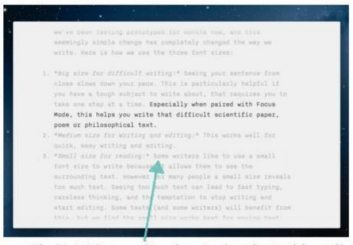
Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.



7. **Flexibility and efficiency of use**
   Accelerators -- unseen by the novice user -- may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.



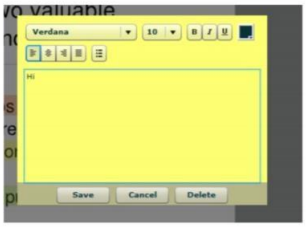
8. **Aesthetic and minimalistic design**
   Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.



The iA Writer app is a clean typing sheet with no distractions in the interface. It allows you to focus on what you're writing and hides everything else.

Dieter Ram's designs reflect this principle. "Less, but better – because it concentrates on the essential aspects, and the products are not burdened with non-essentials."

Avoid overloading the interface with features. Examine every element and ask - is this really needed?



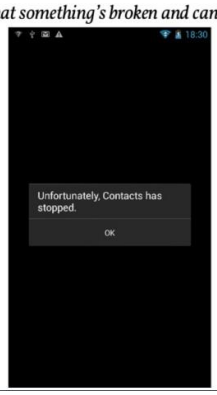
9. **Help users recognize, diagnose and recover from errors**

Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

A funny message keeps the audience engaged, while relevant links make sure they stay on your website.

An assuring error message on Dropbox

Don't tell people that something's broken and can't be fixed.

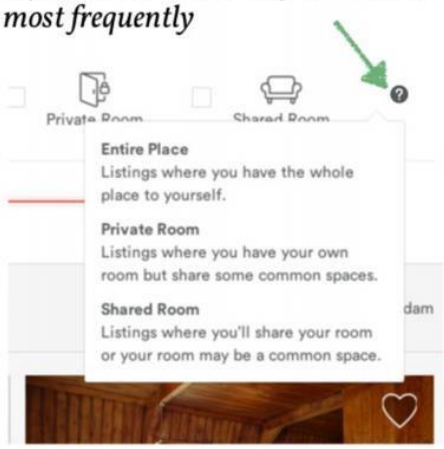


10. Help and documentation

Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

The Kindle comes with an instruction card tucked inside the box flap, instead of a long boring manual

Contextual help on AirBnB provides explanations where they are needed most frequently



If only diapers came with an easy tutorial!

Heuristic Evaluation (cont.)

- **Styles**
  - A single inspector
  - Multiple inspectors
  - Individuals vs. teams
  - Self guided vs. scenario exploration

Average over six case studies

- **A single inspector**
  - 35% of all usability problems
  - 42% of the major problems
  - 32% of the minor problems
  - Not great, but finding some problems with one evaluator is much better than finding no problems with no evaluators!

- **Multiple inspectors**
  - 3-5 evaluators find 66-75% of usability problems
  - Different people find different usability problems
  - Only modest overlap between the sets of problems found

- **Individual vs. teams**
  - Nielsen recommends individual evaluators
  - Not influenced by others
  - Independent and unbiased
  - No overhead of group meetings

- **Self guided vs. scenario exploration**
- **Self guided:**
  - Open-ended exploration and not necessary task directed
  - Good for exploring diverse aspects of the interface and to follow potential pitfalls

- **Scenario exploration:**
  - Step through the interface using representative end-user tasks
  - Ensures problems identified in relevant portions of the interface
  - Ensures that specific features of interest are evaluated

Week-12: Product Quality and Process Quality

Product Quality and Process Quality

- Product and process are two separate things.
- It is important to distinguish between product quality and process quality.
- Software engineering institute (SEI) has done extensive work in the area of process improvement.
- Similarly models for product quality are also developed.

Difference b/w software product and others

- Software is developed or Engineered. It is not manufactured.
- Software does not "wear out".
- Mostly software is assembled rather than developing from scratch.

Product Quality

- The infrastructure that you provide to develop software provides framework through which quality is "built in" the software product.
- But this framework is the "process" aspect of system.
- "Attributes" or characteristics provide measure of quality of end product.

Standards

- According to IEEE STD 983-1986:

"Standards are Mandatory requirements employed and enforced to prescribe a disciplined uniform approach to software development, i.e. mandatory conventions and practices are in fact standards"

    - Quality "gates" must be identified.
- Without measuring conformance to standards, practices and conventions, quality of a software product cannot be measured.

Product Quality

- Several researchers have decomposed the notion of "Software Product Quality" into a number of features or characteristics, which typically include:
- **Reliability**
- **Usability**
- **Maintainability**
- **Correctness**
- **Portability**
- **Testability**
- **Efficiency**

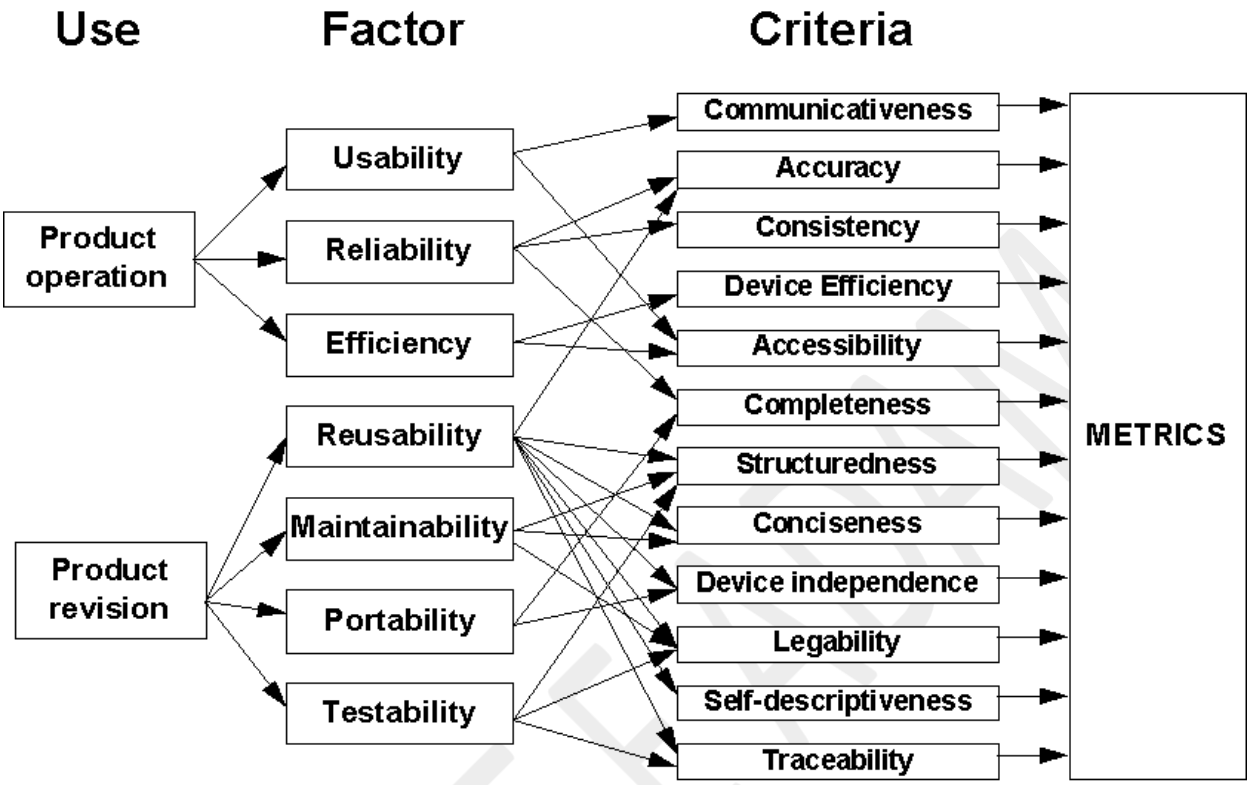
Models for software product quality

- **McCall's Factor-Criteria-Metric Model**
- **ISO 9126 Standard Quality Model**
- **Goal-Question-Metric Model**
- **…**

McCall's Factor-Criteria-Metric Model

- There are number of high-level quality factors (e.g. modifiability) which can be defined in terms of criteria (e.g. modularity). At lowest level there are metrics which indicate or measure the criteria.

Models for software product quality



ISO 9126 Standard Quality Model

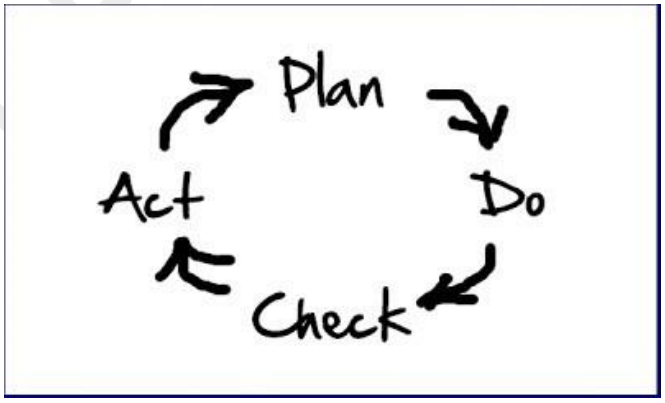
- According to this standard, quality can be defined as:
    - "The totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs".
- **Functionality**
- **Reliability**
- **Efficiency**
- **Usability**
- **Maintainability**
- **Portability**

Process Quality

- A quality process leads to a quality product.
- A process must be tangible to evaluate its quality.
- One way of doing this is to based process on standards or model against which conformance can be measured.
- ISO 9001 (Europe)
- Maturity model (USA)

ISO 9001

- Quality management standards provide a baseline for an adequate production process by subjecting the processes to fulfill certain key requirements.
- This requires organizational commitment
- The leading international Quality Management Standard (QMS) for software development is the ISO's generic "Quality Systems" series of standards, ISO 9000 to 9004
- ISO 9001 makes reference to the process approach to managing an organization.
- In applying ISO 9001, it has to be recognized that software differs in a number of ways from other industrial products.
- Processes used to produce software are not typical industrial processes
- PDCA (Plan, Do, Check, Act)



- ISO 9000-3 helps address some of these differences with references to the software lifecycle and supporting activities.
- The standard assumes a lifecycle model of some type is used but does not prescribe any particular one
- A number of plans are required by the standard:
    - Development plan
    - Quality plan
    - Test plan
    - Maintenance plan

Maturity models for process quality

- Software Engineering Institute (SEI), USA, has developed a model of software development process known as "Capability Maturity Model (CMM).
- Used as basis for process improvement and evaluation.
- Many documents related to public domain are available for download.
- SEI framework identifies five maturity levels:
- **Initial level**: processes are disorganized, even chaotic. Success is likely to depend on individual efforts.
- **Repeatable level**: basic project management techniques are established, and successes could be repeated, because the requisite processes would have been made established, defined, and documented.
- SEI framework identifies five maturity levels:
- **Defined level**: an organization has developed its own standard software process through greater attention to documentation, standardization, and integration.
- **Managed level**: An organization monitors and controls its own processes through data collection and analysis.
- SEI framework identifies five maturity levels:
- **Optimizing level**: Processes are constantly being improved through monitoring feedback from current processes and introducing innovative processes to better serve the organization's particular needs.

Week-13: Product and Process metrics

If you can't measure it, you can't manage it
Tom DeMarco, 1982

What to measure

- **Process**

  Measure the efficacy of processes. What works, what doesn't.

- **Project**

  Assess the status of projects. Track risk. Identify problem areas. Adjust work flow.

- **Product**

  Measure predefined product attributes

- **Process**

  Measure the efficacy of processes. What works, what doesn't.

- **Code quality**
- **Programmer productivity**
- **Software engineer productivity**
  - Requirements,
  - design,
  - testing
  - and all other tasks done by software engineers
- **Software**
  - Maintainability
  - Usability
  - And all other quality factors
- **Management**
  - Cost estimation
  - Schedule estimation, Duration, time
  - Staffing

Process Metrics

- Process metrics are measures of the software development process, such as
  - Overall development time
  - Type of methodology used
- Process metrics are collected across all projects and over long periods of time.
- Their intent is to provide indicators that lead to long-term software process improvement.

Project Metrics

- Project Metrics are the measures of Software Project and are used to monitor and control the project. Project metrics usually show how project manager is able to estimate schedule and cost
- They enable a software project manager to:
  - Minimize the **development time** by making the adjustments necessary to avoid delays and potential problems and risks.
  - Assess **product cost** on an ongoing basis & modify the technical approach to improve cost estimation.

Product metrics

- Product metrics are measures of the software product at any stage of its development, from requirements to installed system. Product metrics may measure:
  - How easy is the software to use
  - How easy is the user to maintain
  - The quality of software documentation
  - And more ..

Why do we measure?

- Determine quality of piece of software or documentation
- Determine the quality work of people such software engineers, programmers, database admin, and most importantly MANAGERS
- Improve quality of a product/project/ process

Why Do We Measure?

- To assess the benefits derived from new software engineering methods and tools
- To close the gap of any problems (E.g training)
- To help justify requests for new tools or additional training

Examples of Metrics Usage

- Measure estimation skills of project managers (Schedule/ Budget)
- Measure software engineers requirements/analysis/design skills
- Measure Programmers work quality
- Measure testing quality

And much more …

IEEE definitions of software quality metrics

- A quantitative measure of the degree to which an item possesses a given quality attribute.
- A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute.

Main objectives of software quality metrics

- Facilitate management control, planning and managerial intervention.
  Based on:
  - Deviations of actual from planned performance
  - Deviations of actual timetable and budget performance from planned.
- Identify situations for development or maintenance process improvement (preventive or corrective actions). Based on:
  - Accumulation of metrics information regarding the performance of teams, units, etc.

Error density metrics

Error density metrics

Code	Name	Calculation formula
CED	Code Error Density	$CED = \frac{NCE}{KLOC}$
DED	Development Error Density	$DED = \frac{NDE}{KLOC}$
WCED	Weighted Code Error Density	$WCDE = \frac{WCE}{KLOC}$
WDED	Weighted Development Error Density	$WDED = \frac{WDE}{KLOC}$
WCEF	Weighted Code Errors per Function Point	$WCEF = \frac{WCE}{NFP}$
WDEF	Weighted Development Errors per Function Point	$WDEF = \frac{WDE}{NFP}$

NCE = The number of code errors detected by code inspections and testing.
NDE = total number of development (design and code) errors) detected in the development process.
WCE = weighted total code errors detected by code inspections and testing.
WDE = total weighted development (design and code) errors detected in development process.

Error severity metrics

Code	Name	Calculation formula
ASCE	Average Severity of Code Errors	$ASCE = \frac{WCE}{NCE}$
ASDE	Average Severity of Development Errors	$ASDE = \frac{WDE}{NDE}$

NCE = The number of code errors detected by code inspections and testing.
NDE = total number of development (design and code) errors detected in the development process.
WCE = weighted total code errors detected by code inspections and testing.
WDE = total weighted development (design and code) errors detected in development process.

Software process timetable metrics

Code	Name	Calculation formula
TTO	Time Table Observance	$TTO = \frac{MSOT}{MS}$
ADMC	Average Delay of Milestone Completion	$ADMC = \frac{TCDAM}{MS}$

MSOT = Milestones completed on time.
MS = Total number of milestones.
TCDAM = Total Completion Delays (days, weeks, etc.) for all milestones.

Error removal effectiveness metrics

Code	Name	Calculation formula
DERE	Development Errors Removal Effectiveness	$DERE = \frac{NDE}{NDE + NYF}$
DWERE	Development Weighted Errors Removal Effectiveness	$DWERE = \frac{WDE}{WDE+WYF}$

NDE = total number of development (design and code) errors) detected in the development process.
WCE = weighted total code errors detected by code inspections and testing.
WDE = total weighted development (design and code) errors detected in development process.
NYF = number software failures detected during a year of maintenance service.
WYF = weighted number of software failures detected during a year of maintenance service.

HD calls density metrics

Code	Name	Calculation Formula
HDD	HD calls density	$HDD = \frac{NHYC}{KLMC}$
WHDD	Weighted HD calls density	$WHYC = \frac{WHYC}{KLMC}$

NHYC = the number of HD calls during a year of service.
KLMC = Thousands of lines of maintained software code.
WHYC = weighted HD calls received during one year of service.
NMFP = number of function points to be maintained.

Severity of HD calls metrics

Code	Name	Calculation Formula
ASHC	Average severity of HD calls	$ASHC = \frac{WHYC}{NHYC}$

NHYC = the number of HD calls during a year of service.

WHYC = weighted HD calls received during one year of service.

HD success metrics

Code	Name	Calculation Formula
HDS	HD service success	$HDS = \frac{NHYOT}{NHYC}$

NHYNOT = Number of yearly HD calls completed on time during one year of service.
NHYC = the number of HD calls during a year of service.

Summary

- **Metrics**
- **Importance**
- **Product/Process metrics**

Week-14: Software Reviews

Software Reviews

What is software reviews

- Software reviews are a "quality improvement processes for written material".
- The process of examine the software product by project personnel's managers users and customers is called software review process.
- The term software product means any technical document which is developed to fulfill the needs of end user.
- Examine the system as per the need of customer.
- To ensure that the resultant product is up to the standards and satisfy all the goals described by end user.

- The review process guarantee that all the requirements of the end user are available as per performing.

Objectives of Review Process

- The basic objective of review process to provide help support to different phases of system like.
  - Verification and validation
  - Configuration management
  - Quality assurance
  - Project management
  - Systems engineering

Cycle of Review Process

- Software review process is applicable on all phase of software development.

- Project plans
- Budget
- Requirement documents
- Specifications
- Design
- Source code
- Testing phases
- Support and maintenance.

Examples of Software Products

- Software design descriptions
- Release notes
- Software requirements specifications
- Source code
- Contracts

- Installation plans
- Progress reports

Review as Quality Improvement

- Reviews can find 60-100% of all defects.
- Reviews are technical, not management.
- Review data can assess/improve quality of:
- Work product.
- Software development process.
- Review process itself.
- Reviews reduce total project cost
- Early defect removal is 10-100 times cheaper
- Reviews distribute domain knowledge, development skills, and corporate culture.

Types of Review

- Management Reviews
- Technical Reviews
- Inspections (Formal Peer Review)
- Walk-throughs
- Audits

Management Reviews Overview

- Performed by those directly responsible for the system
- Monitor progress
- Determine status of plans and schedules
- Confirm requirements and their system allocation

Management Reviews Outputs

Documented evidence that identifies:

- Project under review
- Review team members
- Review objects
- Software product reviewed
- Inputs to the review
- Action item status
- List of defects identified by the review team

.

Technical Reviews Overview

Confirms that product

- Conforms to specifications
- Adheres to regulations, standards, guidelines, plans
- Changes are properly implemented
- Changes affect only those system areas identified by the change specification

Technical Reviews Roles

The roles established for the technical review

- Decision maker
- Review leader
- Recorder
- Technical staff

Technical Reviews Outputs

Outputs, documented evidence that identifies:

- Project under review
- Review team members

- Software product reviewed
- Inputs to the review
- Review objectives and status
- List of resolved and unresolved software defects
- List of unresolved system or hardware defects
- List of management issues
- Action item status

.

Inspection (Formal Peer Reviews)

Confirms that the software product satisfies

- Specifications
- Specified quality attributes
- regulations, standards, guidelines, plans
- Identifies deviations from standard and specification

Inspections Roles

The roles established for the Inspection

- Moderator – Coordinates the inspection and leads the discussion
- Producer – Responsible for the work being inspected
- Reader – Paraphrases the work inspected
- Inspector – Inspects the product
- Recorder – Records problems discussed
- Manager – Supervises the producer

Requirements Inspections

"If you can only afford to do one inspection on a project, you will get the biggest return on investment from a requirements inspection. A requirements inspection should be the one inspection that is never skipped."

- Steven R. Rakitin

Why are Requirements Inspections Important?

- Requirements are the most common source of problems in the development process
- Requirements are written in English by people who typically have little or no training in writing software requirements
- The English language is imprecise, ambiguous, and nondeterministic

Attributes of Good Requirements Specifications

- **Unambiguous**
- **Complete**
- **Verifiable**
- **Consistent**
- **Modifiable**
- **Traceable**
- **Usable**

Requirements Inspection Objectives

- Make sure each requirement in the Software Requirements Specification (SRS) is consistent and traceable to the document that preceded the SRS
- Make sure each requirement in the SRS is clear, concise, internally consistent, unambiguous, and testable

Requirements Inspection Prerequisites

- All inspection team members must receive appropriate training
- The document(s) that preceded the SRS must have been reviewed and approved
- The SRS must have been internally reviewed
- A requirements inspection checklist must be available
- Guidelines for a good SRS must be available

A Sample Inspection Process

- Planning
- Overview Meeting (optional)
- Preparation
- Inspection Meeting
- Follow-up

Objectives of Planning

- Determine which work products need to be inspected
- Determine whether a work product is ready for inspection
- Identify the inspection team
- Determine whether an overview meeting is necessary
- Schedule overview and inspection meetings

Objective of Overview Meeting

- Educate the inspection team on the work product being inspected and discuss the review material

Objective of Preparation

- To be prepared for the inspection meeting by critically reviewing the review materials and the work product

Objective of the Inspection Meeting

- Identify errors and defects in the work product being inspected

An **error** is a problem in which the software or documentation does not meet defined requirements and is found at the point of origin

A **defect** is a problem in which the software or its documentation does not meet defined requirements and is found beyond the point of origin.

Objective of the Follow-Up

- Assure that appropriate action has been taken to correct problems found during an inspection


INSPECTION PROBLEM REPORT

Report No. _____

Inspector

Item Information: Date _____

Item inspected: _____ Inspector: _____

Defect description: Defect location: _____

Recorder

Meeting Decisions:

- ❑ Accepted-Planned Resolution date: _____
- ❑ Duplicate of Problem Report No. _____
- ❑ Rejected-Reason: _____
- ❑ Deferred-Reason: _____

Impact:	Category:	Type:	Origin:
❑ Local	❑ Missing	❑ Procedure/logic	❑ Requirements
❑ External	❑ Wrong	❑ Interface	❑ Code
	❑ Extra	❑ Data definition	❑ Design
	❑ Unclear	❑ Documentation	❑ Test
	❑ Suggestion	❑ Other: _____	❑ Other: _____

Producer

Resolution: Date: _____

Description: _____

Items changed: _____

Verifier

Verification: Date: _____

Verified by: _____

Items checked: _____

Comments: _____

Inspections Outputs

Outputs, documented evidence that identifies:

- Project under inspection
- Inspection team members
- Inspection meeting duration
- Software product inspected
- Size of the materials inspected
- Inputs to inspection
- Inspection objectives and status
- Defect list (detail)
- Defect summary list

Walk-throughs

- Evaluate a software product
- Sometimes used for educating an audience
- **Major objectives:**
  - Find anomalies
  - Improve the software product
  - Consider alternative implementations
  - Evaluate performance to standards and specs

Walk-throughs Roles

The roles established for Walk-throughs

- Walk-through leader
- Recorder
- Author
- Team member

Walk-throughs Outputs

The outputs of the walk-through

- Walk-through team members
- Software product being evaluated
- Statement of objectives and their status
- Recommendations made regarding each anomaly
- List of actions, due-dates, responsible parties
- Recommendations how to dispose of unresolved anomalies

Audits

The purpose of an audit is to provide an independent evaluation of conformance of software products and processes to applicable;

- Regulations
- Standards
- Guidelines
- Plans
- Procedures

Week-15: Inspection Process

Review & Inspection Process

Review Materials

- Source Document
- Checklist
- Supporting Documents
- Invitation
- Master Plan
- Issue/Defect Log
- Data Summary

Review Methods

Synchronous

- Traditional Approach
- Meeting-based

Asynchronous

- Relatively new area
- Meeting replaced with email (or other electronic) communication

Synchronous Review

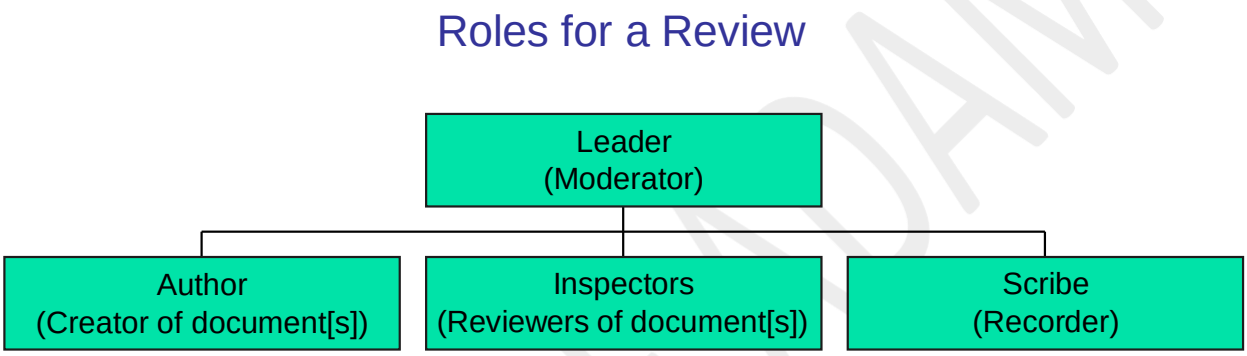
- Review is separated into 5/6 phases

1. (Planning)
2. Overview
3. Preparation
4. Inspection
5. Rework
6. Follow-up

Planning/Overview

1. Reviewers are selected
2. Roles are assigned
3. Documents are distributed
4. General review task is discussed

Review Roles



Roles: Leader

1. Manages inspection
2. Acts as moderator
3. Determines document worthiness
4. Identifies/invites reviewers
5. Assigns roles
6. Distributes documents
7. Schedules meeting times/locations

Roles: Author

1. Creates the document for review
2. Assists with answering questions
3. Typically not directly involved in review
4. Makes corrections to document if necessary

Roles: Inspector/Reviewer

1. Complete familiarization of document on time
2. Review document(s) for defects
3. Look for assigned defects (if appropriate)
4. Make use of checklists or other supporting documents
5. Contact leader early if problems arise or if the review might be a waste of time

Roles: Scribe/Recorder

1. Records issues as they are raised
2. Ideally not the moderator or reviewer
3. Record information legibly

Inspection/Review Meeting

1. Review team attempts to locate defects
2. Defects are not fixed at this point
3. Meeting < 2 hours long!
4. Round-robin approach or Reader approach
5. Scribe records all issues
   - Where defect was located
   - Why is it a defect (cite requirement or checklist)
   - Suggested severity level (Major, minor)
   - Do Not record names of reviewers with defect
   - Try to make visible to all participants (avoid duplication)

Rework

1. Author receives defect log
2. Identifies true defects vs. “false positives”
3. Fixes defects, provides justification for false positive

Follow-Up

- Leader verifies all defects have been addressed
- Decides if document passes review or if another review is necessary

Review Pitfalls

1. Insufficient Preparation
2. Moderator Domination
3. Incorrect Review Rate
4. Ego-involvement and Personality Conflict
5. Issue Resolution and Meeting Digression
6. Recording Difficulties and Clerical Overhead

THE END

BY BINT E ADAM