Different from previous project activity

Principles
Tests traceable to specifications (customer req.)

Tests (concepts) planned during previous project activity

Tests should begin:
- with components
- progress to large blocks

Exhaustive test isn’t possible

Test coordinated between designers & third party
Testability

⇒ Capability influenced by a number of factors
  ⇒ Operability
  ⇒ Observability
  ⇒ Controllability
  ⇒ Decomposability
  ⇒ Simplicity
  ⇒ Stability
  ⇒ Understandability

These are influenced by previous project activity: project definition, specs., preliminary & detailed design, coding, etc.
Test Case Design

⇒ **White Box**
  ⇒ tests designed with knowledge of internal structure
  ⇒ look for logic, process, condition, etc. errors
  ⇒ path and control structure testing

⇒ **Black Box**
  ⇒ tests designed without regard to internal structure
  ⇒ looks for inputs that cause error
  ⇒ looks at output that is incorrect or incomplete
Test Coordination between Developers & Outside Testers

Developers

Independent Testers

White-box

Interface

Black-box

Testing Techniques
Basic Path Testing

Approach - determine & execute every path

⇒ Flow Graph Analysis
⇒ Basics - sequence, if, while, until, switch, etc.
⇒ Cyclomatic Complexity
⇒ Edges, nodes, regions

Flow paths:
\[ V(g) = \text{No. of regions} \]
\[ V(g) = E-N+2 \]
\[ V(g) = P+1 \]
Basic Path Testing - Graph Matrices

Connection Matrix

<table>
<thead>
<tr>
<th>Node Connected To</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Cyclomatic complexity: \( V(g) = 3 + 1 = 4 \)

Attempt to execute every path at least once
Difficult in large programs due to large path count
Doesn't assure logic statements work over range or sequence block are error free
Control Structure Testing

⇒ Condition
  ⇒ Ej < relational operator > Ek

⇒ Error Conditions
  ⇒ Relational operator error
    ⇒ Boolean operator error (incorrect, missing, etc)
    ⇒ Boolean variable error
    ⇒ Boolean parenthesis error
  ⇒ Expression error

⇒ Path testing

⇒ Domain testing
Domain (branch & relational) Testing

⇒ Test conditions
  ⇒ Relational - 3; <, ==, >

⇒ Ej & Ek test and set over range and minimal relationship

⇒ C1: Bj & Bk - {(T,T), T,F), (F,T)}

⇒ For coverage a large number of tests are required
Data Flow Testing

- Test variables in the program - X
  - DEF(S) - statement S contains the definition of X
  - USE(S') - statement S' contains a use for X
  - DU chain - [X, S, S']

- DU chains strategies consider the logic and loop statements.

- Compiler error & warning can help
  - “Variable use without being set”
  - “Variable not defined”
Black Box Testing

Complimentary to white box testing

- Usually applied after white box testing

- Test questions:
  - How is functional validity tested?
  - How is performance & behavior tested?
  - What types of input for good test cases?
  - What features are sensitive to input parameters?
  - What are boundaries of the system?
  - What data rates & volume can tax the system?
  - What combinations of data can influence performance?
BB testing related to design models

- Transaction flow models
  - atomic - complete or rollback

- State transition models
  - actions which interface with externals - user interfaces

- Timing models
  - timings which interface with externals - real-time interfaces

- Data flow models

- Data entity models
  - volumetric situation which tax system
Other BB Testing Techniques

- Equivalence Partitioning
  - Classify the input domain into regions
  - Select tests in each region

- Boundary value & orthogonal testing

- Comparison testing
  - Same data - different time
  - Tests with same results
  - Test different versions
Special testing

⇒ Testing GUIs
⇒ Testing Client/Servers
⇒ Testing Documentation and Help
⇒ Testing Real-time Systems
⇒ Various tasks and behaviors
Testing is bottom-up

⇒ Approach
  ⇒ Start by developing test strategies
  ⇒ Begin with components outward to whole system
  ⇒ Types of test may differ
  ⇒ Test conducted by both developers & independents
  ⇒ Testing and some debugging are integrated

⇒ Verification & validation
  ⇒ Validation - are we building the right product?
  ⇒ Verification - is the product we are building correct?
Logarithmic Poisson execution-time model

\[ f(t) = \frac{1}{p} \ln(p \cdot t + 1) \]

- \( p \) = exponential reduction rate
- \( l \) = initial software failure intensity at beginning of testing
- \( t \) = test time

\( f(t) \) = number of failure expected when software has been tested a certain test time, \( t \)
Integrated Testing

After unit testing is well along

⇒ General strategies
  ⇒ Top-down integration
  ⇒ Bottom-up integration
⇒ Regression testing
  ⇒ accumulate some older tests info and retest with then
    ⇒ can detect errors created in changing
⇒ Profiling - determine code where runs take most time
⇒ Smoke testing
  ⇒ at the limits of the system
⇒ Recovery testing
⇒ Validation testing
Debugging

Is an integral part of testing

⇒ Difficulties
  ⇒ Cause & effect may be miles apart

⇒ Symptoms may disappear without correction

⇒ Difficult to accurately reproduce

⇒ Tester must understand cause and effect
Debug features

Supported by most compilers

⇒ Many features
  ⇒ trace, step, break, watch, cast errors
  ⇒ assertions, inspect variables, memory set
  ⇒ various views
    ⇒ memory, CPU, registers, source, thread, process
    ⇒ inspect, local variables, set optimization, etc.
  ⇒ linking errors
  ⇒ turn on and off - compile with & without
⇒ Includes code logging during test runs
  ⇒ improper memory use
    ⇒ allocation, no deallocation, leaks, etc.
⇒ invalid pointers & handles
⇒ invalid arguments and returns
Profile:

- Coverage test
  - Record use - times a block was executed
  - Record timing - time used in block execution
- Support for version control
Technical Metrics for Software Testing

Provide a measure of quality

⇒ Product Operation
  ⇒ correctness, reliability, useability, integrity, efficiency

⇒ Product Revision
  ⇒ maintainability, flexibility, testability

⇒ Product Transition
  ⇒ portability, reuseability, interoperability

⇒ Can organize these into a checkoff Software Quality Table
  Figure 19.2

⇒ If you have all of these you have a great product
Software Metrics

Framework

⇒ Problem
  ⇒ difficult to find metrics for all quality measures
  ⇒ some measures are highly subjective
  ⇒ different meaning (or utility) to different users

⇒ Effective metric
  ⇒ Simple & computable (measurable)
  ⇒ Empirically and intuitively persuasive
  ⇒ Consistent & objective
  ⇒ Language independent
  ⇒ Feedback to software available
Function Point Metrics

⇒ Number of:
  ⇒ inputs, outputs, inquiries, files, external interfaces

⇒ Alternative
  ⇒ functional primitives
  ⇒ data elements
  ⇒ objects
  ⇒ states
  ⇒ transitions

⇒ Additionals -
  ⇒ external function primitives - functions outside program boundary
  ⇒ input & output data
  ⇒ retained data
  ⇒ data tokens (unparsed data primitives)
  ⇒ collaborations
Design Model Metrics

⇒ Structural complexity - \( S(i) = \text{fout}(i)^2 \)
  ⇒ \( \text{fout}(i) \) = fan-out of module \( i \)

⇒ Data complexity - \( D(i) = \frac{v(i)}{\text{fout}(i) + 1} \)
  ⇒ \( v(i) \) = I/O count for module \( i \)

⇒ High-level architectural complexity
  ⇒ HKM = \( \text{length}(i) \times [\text{fin}(i) + \text{fout}(i)]^2 \)
    ⇒ \( \text{fin} \) = fan-in
    ⇒ \( \text{length} \) = language statements in module \( i \)

⇒ Simple morphology
  ⇒ size = \( n + a \)
    ⇒ \( n \) = number of nodes; \( a \) = number of arcs
Design Model Metrics (cont’d)

⇒ US AF Systems Command measures
  ⇒ S1 = total modules defined in program
  ⇒ S2 = modules whose correct function depend upon input or produce data used elsewhere
  ⇒ S3 = modules depending upon prior processing
  ⇒ S4 = number of database items
  ⇒ S5 = number of uniques database items
  ⇒ S6 = number of database segments (different records or individual objects)
  ⇒ S7 = modules with single entry and exits

⇒ USAF System Command metrics
  ⇒ Independence: D2 = 1 - \( \frac{S2}{S1} \)
  ⇒ Database size: D4 = 1 - \( \frac{S5}{S4} \)
  ⇒ Database compartmentalization: D5 = 1 - \( \frac{S6}{S4} \)
Software Reliability

Definitions

⇒ Informal
⇒ How well does the software preform for the user?
⇒ Software failures experienced by user
⇒ Depend upon failure
⇒ can’t carry out a particular function; can work around
⇒ bombs in critical part
⇒ completely unacceptable

⇒ Formal
⇒ Probability of failure free operation over a period of time
⇒ 99.99% reliable for 5 hour flight
⇒ one failure in 10,000 flights
Failure input-output mapping

Complex Relationship between observed software reliability and the software’s latent faults
Reliability Relationship Characteristics

Removal of large % of faults doesn’t make much difference between perceived reliability

- Program may have many faults but perceived reliable by user
  - Workarounds
  - Fault tolerance
  - Fault recovery

- Formal development methods
  - No guarantee that software will be more reliable
  - Specs may not reflect real user requirements
  - “Proofs” may contain errors
  - “Proofs” may not reflect user patterns
Reliability vs Efficiency Plus

Choose former

⇒ Computer hardware is fast and cheap

⇒ Unreliable software won’t be used

⇒ Failures may be extremely costly

⇒ Unreliable software is difficult & costly to repair

⇒ Inefficiency can be worked around
Reliability Metrics

Software failure characteristics

⇒ Normal - Mean Time to Failure (MTTF)
  ⇒ most commonly used - safety critical systems
  ⇒ time from placing in use to failure
⇒ Probability of Failure on Demand (POFOD)
  ⇒ when feature activated will if fail
  ⇒ operating systems
⇒ Rate of Failure Occurrence (ROFOC)
  ⇒ Inverse of MTTF
  ⇒ transactions
⇒ Availability (AVAIL)
  ⇒ Related to (ROFOC) - time system can be used
  ⇒ continuously running systems
Reliability metrics

Measures

⇒ Number of system failures given a number of system inputs (POFOD)

⇒ Time between system failures (MTTF - 1/ROFOC)

⇒ Elapsed repair time or restart time given system is "continuously" available (AVAIL)

⇒ What is a “failure”? Some faults such as transient and those without serious consequences - Are they failures?

Time is a measure in all reliability based metrics
Reliability Specifications

Need to define and discuss during project definition.

⇒ Many times specs are subjective, irrelevant and/or unmeasurable

⇒ Subsystems may have vastly different requirements

⇒ Types to consider
  ⇒ Transient - occurs in specific situations only
  ⇒ Permanent - occurs for all situations
  ⇒ Recoverable - can recover without operator intervention
  ⇒ Unrecoverable - needs operator intervention to recover
  ⇒ Corrupting - failure changes system state and/or data
  ⇒ Non-corrupting
Reliability Specs (cont’d)

⇒Steps
  ⇒For each subsystem identify and analysis different types of failures
  ⇒Partition types of failure into appropriate classes
  ⇒For each failure class, define reliability requirement using appropriate reliability metrics. Then define the characteristics for this subsystem

⇒Difficulties
  ⇒Operational uncertainty - software use in practice
  ⇒Operational profile - large number of inputs
  ⇒Statistical uncertainty
    ⇒Finding and correcting faults
Reliability growth model

Fault determination
Note: different reliability improvement

Fault repair adds new faults & decreases reliability

Curve fitted to data

Reliability Requirement

Time

Reliability
ROCOF
Fault Avoidance & Tolerance

⇒ Avoidance
  ⇒ Important strategy - design & implementation organized to enable

⇒ Tolerance
  ⇒ Assume faults remain - allow operation to continue after fault occurs
  ⇒ Proper use of exception handling very important here.

⇒ Detection
  ⇒ Testing and debugging - the very potential for enabling reliable systems

The objective of software engineering is embodied here.