



Testing and Inspections
(3C05/D22)


 **UCL** ComputerScience


Unit 11: Testing and Inspection

- Objectives
 - To introduce software testing and to develop its role within the software development process.
 - To introduce the use of formal inspections of design and code as a means of reducing errors in software development.

 **UCL** ComputerScience

What is Testing?



- software testing is the process of seeking errors
- this process is active - if we find no errors after performing a test we cannot passively assume that no errors exist
- we must eliminate all assumptions
- the goal is a product with "zero defects" 

 **UCL** ComputerScience

Questions



- why should we test
 - because the cost of not testing is much greater than the cost of testing
 - economics is both the limiting factor & the driving force
- who should do the testing
 - testing requires a rigorously objective view
- when should we test
 - not just program code, find errors as near source as possible, testing at each stage in the lifecycle

testing is directly linked to quality improvement!



When Should We Test?

<ul style="list-style-type: none"> • requirements 	<ul style="list-style-type: none"> • determine verification approach • determine adequacy of requirements • generate functional test data
<ul style="list-style-type: none"> • design 	<ul style="list-style-type: none"> • determine consistency of design with requirements • determine adequacy of design • generate structural and functional test data
<ul style="list-style-type: none"> • construction 	<ul style="list-style-type: none"> • determine consistency with design • determine adequacy of implementation • generate structural and functional test data • apply test data
<ul style="list-style-type: none"> • operation & maintenance 	<ul style="list-style-type: none"> • reverify, commensurate with the level of development


Requirements Stage

- test requirements documents by disciplined inspection and review
- developing scenarios of expected system use (aka use cases)
- preparation of test plan which should include:
 - specification
 - description of test procedures
 - test milestones
 - test schedule
 - test data reduction
 - evaluation criteria


Design Stage

- test design products by analysis, simulation, walkthroughs and inspection
- generate test data for functions
- generate test cases based on structure of system


UCL ComputerScience


Construction Stage

- actual execution of code with test data
- code walkthrough and inspection
- static analysis
- dynamic analysis
- construction of test drivers, harnesses and stubs




important

Control and management of test process is critical. All test sets, test results and test reports should be catalogued and stored.


UCL ComputerScience

Operation and Maintenance Stage


- Modifications require retesting
 - this is termed regression testing
- Changes at a given level will necessitate retesting at all levels below it.
- Beware! This is where the money goes.


UCL ComputerScience

Methods


- There are two basic methods for organising testing processes:
 - bottom-up testing
 - top-down testing

Testing is repetitive and lends itself to automation and tool support.

 **UCL** ComputerScience


The Myth of Exhaustive Testing


- Exhaustive testing is the only dynamic analysis technique which will guarantee validity.
 - Unfortunately it is not practical!
- The key problem for testing is how to derive an appropriate test data set (aka test set)

 **UCL** ComputerScience

Approaches

- Two basic approaches:
 - black box or "functional" analysis
 - white box or "structural" analysis



 **UCL** ComputerScience

Functional Testing

- boundary value analysis (stress testing)
 - Partition the input data, select data inside and at the boundary of each partition.
- design-based functional testing
 - Construct functional hierarchy, identify for each function at each level extremal, non-extremal and special value test data. Identify test data which will generate extremal, non-extremal and special output values.
- cause-effect graphing
 - Identify characteristic input stimuli (causes). Identify characteristic output classes (effects). Identify dependencies using specification. Present as directed graph, choose test cases to test dependencies. Can be partially automated.
- exhaustive testing (where applicable)



Structural Testing

- coverage-based testing
 - Represent program as control-flow graph. Identify paths. Choose data to maximise paths executed under test conditions. Paths not always finite. Some paths infeasible. Coverage metrics can be applied. Can be partially automated.
- complexity-based testing
 - Measure cyclomatic complexity. Determine paths actually executed by program running on test data set - actual complexity. Attempt to devise test set which will drive actual complexity closer to cyclomatic complexity.



Test Data Analysis


- "the goodness of the test data set"
- statistical analysis and error seeding
 - Seed known errors into code so that their placement is statistically similar to that of actual errors.
- mutation analysis
 - It is assumed that a set of test data that can uncover all simple faults in a program is also capable of detecting more complex faults. In mutation analysis a large number of simple faults, called mutations, are introduced in a program one at a time. The resulting changed versions of the test program are called mutants. Test data is then be constructed to cause these mutants to fail. The effectiveness of the test data set is measured by the percentage of mutants killed.




Static Analysis

- flow analysis
 - Construct data flow graph. Trace behaviour of program variables.
- symbolic execution
 - Instead of executing with actual data values, the variable names that hold input values are used as input values. All branches are taken and tree constructed which can be used to identify control paths and hence test sets.
- instrumentation
 - eg Insertion of counters or turnstiles in code.


Methods can be combined






Inspections

- formal inspection of system development products is a very effective means of reducing errors in software development
- we will be looking at "Fagan Inspections" the most well known set of inspection techniques
- goal - remove errors as near source as possible hence reducing costs of rework.




Prerequisite

- Describe the software development process in terms of operations, and define exit criteria which must be satisfied for completion of each operation.
- Inspections at each exit point. Key inspections to be carried out:
 - I0 Design Architecture
 - I1 Design Complete
 - IT1 Test Plan
 - IT2 Test Cases
 - I2 Code




People

- the people involved
 - moderator
 - designer
 - coder/implementor
 - tester
- four people constitute a reasonable size for an inspection team

 **UCL** ComputerScience


Process

- overview (whole team)
- preparation (individual)
- inspection (whole team)
- rework
- follow-up

 **UCL** ComputerScience

Steps

- steps in inspection
 - reader paraphrases design, describing how it will be implemented
 - questions raised during discourse, only pursued until error recognised
 - error noted (but not solution) and classified by severity
 - written report of inspection prepared
- important to inspect modified products (reinspection to avoid "bad fix" problem)

 **UCL** ComputerScience

Rate of Progress

<i>process operations</i>	<i>rate of progress (loc/hr)</i>		<i>objectives of operation</i>
	<i>Design</i>	<i>Code</i>	
overview	500	not required	communication
preparation	100	125	education
inspection	130	150	find errors
rework	20 (hrs/KNCSS)	16 (hrs/KNCSS)	resolve errors
follow-up	-	-	ensure resolution of errors

UCL
Computer Science

Scheduling

- Inspections should be scheduled with care
- The result of postponing inspection is usually lengthening of the overall schedule and increased product cost!
- Error detection efficiency tends to dwindle after 2 hours
- Two sessions of 2 hours are acceptable in a day

UCL
Computer Science

Finding Errors

- finding errors is difficult...
 - condition people to seek high occurrence, high cost error types
 - take representative sample of code; obtain suitable quantity of errors; analyse by type, origin, cause and salient indicative feature
 - use this to prepare inspection specification to guide process

UCL
Computer Science

Feedback

- inspections provide detailed real-time feedback to developers success is due to:
 - management attitude
 - conduct of trained moderator
- process control using inspection (identification of error prone modules and distribution of error types)
 - leading to process reengineering

under no circumstances should inspections be used for programmer performance appraisal

Walkthroughs

- inspections may be contrasted with walkthroughs
 - formality
 - regularity
 - feedback
 - feedforward
 - self-improvement

a common misunderstanding

Inspection Overview

The diagram illustrates the inspection process flow:

- operation 1** leads to **Analysis**.
- Analysis** leads to **Rework** and **operation 2**.
- Rework** leads back to **operation 1**.
- Analysis** also leads to **feedback** (left) and **feedforward** (right).
- Feedback** includes:
 - fix process holes
 - fix short term problems
 - error feedback for learning to programmer
 - special rework or rewrite recommendations
- Feedforward** includes:
 - error prone modules - ranked
 - error types distribution - ranked
 - number of errors/KLOC compared to average
 - learning input for inspector & moderators
 - what error types to look for
 - better ways to find each error type
 - detail error follow-up
 - number of errors/inspection hr.
 - number of LOC inspected/hr

or special attention

For Details

- Fagan, M.E. (1976); Design and Code Inspections to Reduce Errors in Program Development; IBM Systems Journal; 15, 3, pp 182-211.
- Fagan, M.E. (1986); Advances in Software Inspections IEEE Transactions on Software Engineering, Vol SE12 No 7, p744, July 1986.



Key Points

- Testing is a key aspect of verification and is vital to ensure software quality. Testing should be a concern throughout the life-cycle and should be carefully planned and managed. There are a range of techniques to help improve the effectiveness of testing. Many of these are supported by tools.
- Define exit criteria for development operations. Focus objectives of inspection process. Identify which types of errors to spend time looking for. Analyse inspection results and use for process improvement.