Testing Principles

Lecture 8

The Purpose of Testing?

- Is looking for the (possible) presence of errors
- Testing can show the presence of an error, never the absence
- A successful test is one that reveals a defect
- Running a program is not testing (though it may be beta testing)
- Debugging is not testing
- The purpose of testing may not provide a quality guarantee

Actuality / Test Result	System has no Errors	System has Errors
Error Found	**False Alarm (Testing Error)**	**Correct**
No error Found	**Correct**	**Missed the Error**

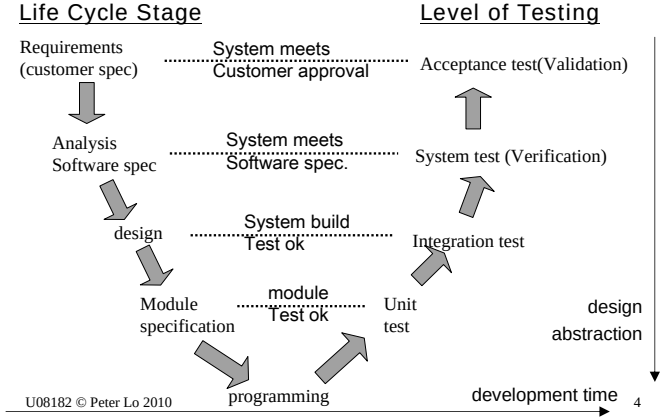
Activities in a typical software life cycle model may include the following:

- Quality Planning
- System Requirements Definition
- Detailed Software Requirements Specification
- Software Design Specification
- Construction or Coding
- Testing
- Installation
- Operation and Support
- Maintenance
- Retirement

Four Stage Systems Development Life Cycle Model

Stage	Activity	Deliverable	SQA
Requirements (Specify What)	Tender Feasibility Systems analysis	Tender Feasibility Requirement specification	Project set-up Requirements review
Design (Specify How)	System design	Test Plan Design specification	Design review
Implementation (Build)	Programming Testing Verification	User guide Release notes Software reference documentation	Release Validation Acceptance testing
Operation	Training Support Maintenance	Training request Enhancement request Bug report	Change request

Testing through the 'V' Life Cycle Stages



In the 'V' life cycle model the x-axis shows time and the y-axis shows the level of design abstraction.

- At the top is the customer view
- At the bottom the program code

Verification activities are generally on the V-model left hand side,

- E.g. confirming that design is based on requirements, then code is based on design etc while Validation activities are on the right hand side where the product is checked and tested.

Validation –

Does the System Do What the Users Want?

- Demonstrate that system satisfies users requirements
- Normally assessed using dynamic testing
  - Performance Testing
  - Regression Testing
  - Acceptance Testing
  - Usability Testing
  - Stress Testing
  - Security Testing
  - Recovery Testing
  - Alpha and Beta Testing

Software validation is a part of the design validation for a finished device. It ensures that the output of a life cycle stage (or more usually the complete system) is correct. I.e. that the system behavior, operation or output conforms to the customers specification. For example a complete system validation is commonly termed acceptance testing. Validation is also concerned that the system components and build procedures are traceable to the customers specification (Ince, 1994). Validation is therefore a check on the overall system quality. Further, validation could be applied incrementally, for example as part of a prototyping design methodology approach

Software validation can be defined as confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled.

In practice, software validation activities may occur both during, as well as at the end of the software development life cycle to ensure that all requirements have been fulfilled.

Since software is often part of a larger hardware system, the validation of software typically includes evidence that all software requirements have been implemented correctly and completely and are traceable to system requirements. A conclusion that software is validated is highly dependent upon comprehensive software testing, inspections, analyses, and other verification tasks performed at each stage of the software development life cycle. Testing of device software functionality in a simulated user environment, and user site testing are typically included as components of an overall design validation program for a software automated device.

Verification –

Are we Following the Design Process?

- Ensure component or system works (conforms) to its specification, i.e. the output conforms to its inputs (even if the specification is incorrect)
- Applies to **Complete System** (Product), or to **Separate Development Stages** (Process)
- Includes both static and dynamic testing activities
- Includes process traceability
- Not the same as demonstrating correctness, i.e. usually does not provide any check or proof as to the correctness of a system
- Program verification, which includes formal methods, may provide a high level of assurance.

Software verification provides objective evidence that the design outputs of a particular phase of the software development life cycle meet all of the specified requirements for that phase, i.e. verification is applied between stages as a check on the design process. Software verification looks for consistency, completeness, and correctness of the software and its supporting documentation, as it is being developed, and provides support for a subsequent conclusion that software is validated. Software testing is one of many verification activities intended to confirm that software development output meets its input requirements. Other verification activities include various static and dynamic analyses, code and document inspections, walkthroughs, and other techniques. Verification is achieved if the stage output meets the specification, even if the specification is incorrect. Verification therefore does not provide any check or proof as to the correctness of a system

Validation vs. Verification

Validation	Verification
Whether the developed system is what the user wanted.	*Whether the developed system is what have been specified*
◆ To prove that a system satisfies users' requirements	◆ To prove that a program is consistent with respect to the specification.
◆ Users' requirements may or may not be elicited	◆ A program formally proved to be consistent with respect to a specification can still fail to satisfy users' requirements
◆ Users' requirements may or may not be documented accurately and completely	◆ Consistency between software artefacts can be formally defined and proved
◆ Whether or not users' requirements are satisfied cannot be formally proved	◆ Program can be derived from from a specification
◆ Users' requirements change frequently	

Software verification and validation are difficult because a developer cannot test forever, and it is hard to know how much evidence is enough. In large measure, software validation is a matter of developing a "level of confidence" that the device meets all requirements and user expectations for the software automated functions and features of the device. Measures such as defects found in specifications documents, estimates of defects remaining, testing coverage, and other techniques are all used to develop an acceptable level of confidence before shipping the product. The level of confidence, and therefore the level of software validation, verification, and testing effort needed, will vary depending upon the safety risk (hazard) posed by the functions of the system.

Testing Basics

- Testing is the process used to verify or validate a specification, system or unit
- Testing is part of the design process (proactive), not a response to it (reactive), whereas debugging is a reactive activity
  - **Static Testing** – Applied without running or operating system
  - **Dynamic Testing** – Involves operating the real system

"Testing can be used to prove the presence of bugs, but not their absence"
Dijkstra

Testing is a proactive activity, whereas debugging is a reactive activity

Testing is a key element of any design methodology. In particular there should be a test plan or strategy for any system development and design for testability should be a consideration

The Test Plan is the testing procedure and documentation

- Static Testing – applied without running or operating system
- Dynamic Testing – involves operating the real system

Types of Testing

- Modelling: investigate behaviour of model of system
- Static and dynamic testing
- Module, integration and acceptance (validation) testing

Levels of testing

- Black box and white box testing
- Test software components in isolation
- Test integrated systems

Testing within the systems development lifecycle

- Static analysis
- Module Testing
- Coverage Measurement
- White box testing
- Integration testing
- System testing
- Acceptance testing

Black Box Testing & White Box Testing

- Black Box Testing
  - Test against the system functional specification
  - Can reveal errors in the design process
  - Generally less complex and costly to perform
  - Does not consider internal code paths
- White Box Testing
  - Considers the internal code structure, so capable of detecting subtle faults
  - Can be complex or even impractical for some programs
  - Has limited capability in detecting requirements and design errors

Often use a combination of black box and white box testing
• Black box to confirm external specification
• White box to provide test adequacy, e.g. by code path coverage

9

Black Box Testing

- Test against the system functional specification
- Can reveal errors in the design process (requirements, design, implementation)
- Generally less complex and costly to perform (than white box)
- Does not consider internal code paths – so may not provide test adequacy
  - E.g. in code path coverage.

White Box Testing

- Considers the internal code structure, so capable of detecting subtle faults
  - e.g. by testing all code paths
- Can be complex or even impractical for some programs
- Has limited capability in detecting requirements and design errors

Often use a combination of black box and white box testing

- Black box to confirm external specification
- White box to provide test adequacy, e.g. by code path coverage

Testing Adequacy

- Non functional requirement describing how thoroughly a system should be tested
- Is used to specify
  - When to stop testing
  - A measure of testing quality (e.g. % coverage)
  - How to generate tests (choose test values)
    - Specification based (generally black box)
    - Program based (generally white box)
  - Measure testing progress
  - Whether to add another test to a suite

 10

Is used to specify

- When to stop testing (e.g. safety requirement, testing costs)
- A measure of testing quality (e.g. % coverage)
- How to generate tests (choose test values)
  - Specification based (generally black box)
  - Program based (generally white box)
- Measure testing progress

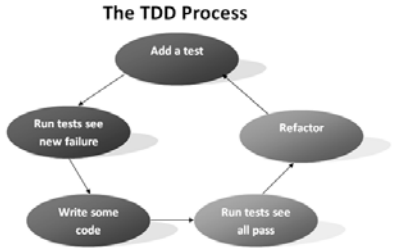
Whether to add another test to a suite

Techniques

- Test coverage measurement (e.g. lines, branch or path)
- Fault seeding (e.g. mutant score = % of planted faults detected)

Test Driven Development (TDD)

- A test first software development approach
- Popular technique with Extreme Programming (XP)



Concept

- Write the test first, before the code for each new or reworked feature
- Each test based on a use case
- Simple tests that each returns true (success) of false (fail)
- Ensure test fails with incorrect implementation but passes when correct
- Code complete when it passes its test
- Develop an automated test suite of all the tests = test suite ok if all tests ok
- Refactoring (M Fowler) – concept: make improvements to code implementation without changing the specification, using TDD to verify

Techniques

- test coverage criteria (e.g. lines, branch or path)
- fault seeding (e.g. mutant score = % of planted faults detected)

Static Testing

- Modelling
- Inspection
  - Formal Review
  - Code analysis
  - Trace table
  - Control flow analysis
  - Symbolic execution
  - Scenario based
- Formal Methods

Modelling (Approach to Testing)

- Investigate behaviour of an appropriate system model
  - E.g. a mathematical model
- Usually applied early in the system life-cycle
- Can be used to formulate a system specification, for subsequent systems design and validation
- Does not test actual system or program code
- Some properties cannot be tested by modelling,
  - E.g. reliability

Static Testing: Inspection

- A step-by-step reading of the software engineering product, with each step checked against a predetermined list of criteria, called check list.
- These criteria usually include checks for historically common errors, adherence to programming standards, and consistency with program specifications.
- Inspection requires a team of testers including the software developer.
- The developer narrates the reading of the product and finds many errors just by the simple review act of reading aloud.
- Other errors are determined as a result of discussion with team members and by applying the checklist.

Static Testing Methods: The Formal Review

- Applicability:
  - Designs, program codes and various documents produced at various stages of development
- The purpose:
  - To analyse internal consistency, satisfaction of requirements, and suitability for implementation
- Technique: Walk-through
  - Test data are selected and the software is simulated manually.
  - The test data are "walked through" the system, with intermediate results kept on a blackboard or a sheet of paper

Applicability:

- Designs, program codes and various documents produced at various stages of development

The purpose:

- To analyse internal consistency, satisfaction of requirements, and suitability for implementation

Technique: Walk-through

- Test data are selected and the software is simulated manually.
- The test data are "walked through" the system, with intermediate results kept on a blackboard or a sheet of paper

The key issues:

- Keep the test data simple
- Encourage discussion, not just to complete the simulation
- Most errors are discovered by questioning the developer's decisions, rather than by examining the test data

Life Cycle Structured Walkthrough – Principles

- Review of deliverables/products documentation at the end of a life cycle stage
- Objective is to identify errors, omissions and to initiate necessary corrective action
- Walkthrough types: formal or informal
  - **Formal Walkthrough** – Full review of work done in one stage, involving the client/customer
  - **Informal Walkthrough** – Internal to the development team

Life Cycle Structured Walkthrough – Principles

- Review team size: minimum 3, maximum 7
  - Team member roles: chair, secretary, document author, reviewers, optional user/customer/client, optional observer (from project team) and optional independent observer
- Objective:
  - To ensure document/product meets the requirements and conforms to the appropriate quality standards
  - Identify errors and omissions

Walkthroughs/ design reviews – Peer review involving the systematic investigation of documents, called a code walk-though. This requires an engineer to lead colleagues through the design or implementation of software and to convince them of its correctness.
Aim: To detect faults in some product of the development as soon as possible in the development cycle.
Software reviews are a "filter" for the software engineering process.
Reviews are applied at various points during software development and serve to uncover defects that can then be removed.
A person can often spot some of their own mistakes, but a group of people are more likely to spot more of the mistakes.

A **Formal Technical Review** (FTR) is a software quality assurance activity that is performed by software engineering practitioners. The objectives of the FTR are:

- To uncover errors in function, logic, or implementation for any representation of the product (hardware, software, safety).
- To verify that the product under review meets its requirements.
- To ensure that the product has been developed according to the predefined standards.
- To achieve a product that is developed in a uniform manner.
- To make projects more manageable.
- To ensure a projects quality.

Structured Walkthrough – 3 Stages

- **Preparation** – document author (presenter) passes the document to the chair (in good time) for distribution to the review team members, together with notification and agenda of the review meeting
- **Walkthrough meeting** – in which actions and comments are identified, agreed and recorded by the secretary. The overall review meeting outcome can be:
  - Accept and approve the document/product
  - Recommend minor revision, with no need for further review
  - Recommend major revisions and schedule a further review
- **Follow up** – address outcome of the review meeting

FTR review meeting procedure:

- Typically between 3 to 5 people in the review team.
- Advance preparation should occur with no more than 2 hours work per person.
- The duration of the review meeting should be less than 2 hours.
- The **review meeting records** should document:
  - What was reviewed, Who reviewed it, The findings and conclusions

An example of a Review Checklist for Coding

- Has the design properly been translated into code?
- Are there misspellings and typos?
- Has proper use of language conventions been made?
- Is there compliance with coding standards for language style, comments, module prologue?
- Are there incorrect or ambiguous comments?
- Are data types and data declarations accurate?
- Are physical constants correct?
- Have all the items on the design walkthrough checklist been reapplied as required?

The Fagan Inspection Method – 6 Stages

- **Planning** –inspection team appointed
- **Overview** – ensure the inspecting team understand the product for review relates to the overall system
- **Preparation** – team members familiarise themselves with the review documentation material. It is important that adequate time is allowed for this
- **Inspection** – the formal meeting where the document is inspected
- **Rework** – correcting errors and omissions recorded at the inspection meeting
- **Follow up** – checking that the rework has been performed adequately

Popular formal review/structured walkthrough technique

Inspections carried out on all major deliverables: requirement specification, program designs, code listing, test output etc

Aims to find errors and omissions in output from a systems development stage

Systematic approach using standardized checklist to assist fault finding

Inspections carried out using a predefined set of steps

Each inspection focuses on a small component of documentation

Inspection team: chair, minute taker, document author + 1 or more inspectors

Maximum meeting duration 2 hours

All types of defects noted - not just logical or function errors

Inspections can be carried out by colleagues at all levels, except the very top

Material is inspected at an optimal rate of about 100 lines an hour and Statistics are maintained so that effectiveness of the inspection process can be monitored

Static Testing – Code Analysis Techniques

- Boundary Value Analysis
- Control Flow Analysis
- Program Code Analysis
- Trace Table
- Symbolic Execution
- Cyclomatic Complexity

Boundary Value Analysis

- To examine behaviour at boundaries of input domain

Control Flow Analysis

- Code analysis using directed graphs to analyse program control structures. E.g. unreachable code

Program Code Analysis

- A post compiler for C to check that language rules are applied. E.g. LINT

Trace Table

- For manual check of program operation using data values

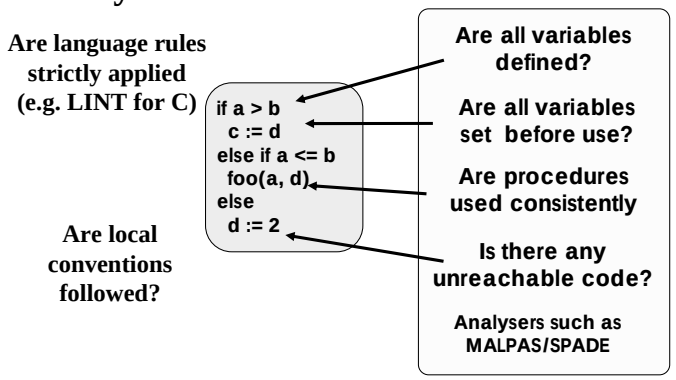
Symbolic Execution

- Program is traced manually or automatically (semantic analyser) using algebraic variables as data (unlike trace table where actual data is used) so that statements produce algebraic expressions.
- Resulting formulae for each code path then compared to its specification

Cyclomatic Complexity

- Complexity of design or code control structure. E.g. Halstead

Static Testing – Program Code Analysis



Code Analysis can cover many forms of testing. Some of the checks which can be made on source code will vary between languages. LINT for example is a program which cheeks for the strictest possible adherence to the C language. This reflects the fact that C allows programmers to do 'silly' things. Many of LINTs checks would be done by the compiler in another language. In some languages (e.g. BASIC, FORTRAN) variables do not have to be explicitly defined, accidentally misspelling a variable name can have a disastrous effect (one of the early US space rockets had to be blown up in the air because of a such a fault).

A number of proprietary products such as MALPAS & SPADE (aimed mostly at the real time market) will do much more rigorous checking of the logic of programs. Apart from checks which could be made by compilers (unreachable code, variables used before they have been given a value) such products provide detailed analysis of conditions for entry to and exit from modules and compare them to programmer provided specifications.

If local programming conventions are used (for example rules for naming variables, maximum allowed depth of nesting of if/while statements tic) then testing should also include ensuring that such conventions are followed. This may require specially written software.

Static Testing – Trace Table Example 1

- Trace through program statements

```
1 Program guess_the_number_game
2   output ‘Guess the number game’
3   answer=42 ! correct value
4   N:=1 ! number of guesses
5   Output 'Enter first guess: '
6   Input guess  ! first guess
7   While guess<>answer
8     N:=N+1  ! count the guesses
9     If guess<answer Then
10      Output 'Enter higher number'
11    Else
12      Output 'Enter lower number'
13    EndIf
14    Input guess  ! next guess
15  EndWhile
16  Output 'Correct in', N, guesses'
17End Program
```

Line	N	guess	output
2			Guess the number game
4	1		
5			Enter first guess
6		25	
8	2		
10			Enter higher number
14		50	
8	3		
12			Enter lower number
14		37	
8	4		
10			Enter higher number
14		42	
16			Correct in 4 guesses

Trace through program statements

Static Testing – Trace Table Example 2 (Code)

- Analysing detailed execution can involve complex code lines involving multiple predicates

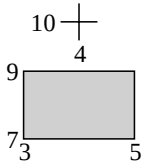
```
. . . .
struct coordinate { int x; int y;};
typedef struct coordinate point;
struct rectangle { point lo_left;  point up_right;};
typedef struct rectangle rect;
. . . .
  if(inside(r,p)) printf("Inside");
  else            printf("Outside");
. . . .
/* function returns TRUE if point is within rectangle
   and FALSE if point is on edge or outside rectangle */
int inside(rect R, point P)
Line { int result;
   1   result = (P.x > R.lo_left.x )
   2          && (P.x < R.up_right.x)
   3          && (P.y > R.lo_left.y )
   4          && (P.y < R.up_right.y);
   5   return result;
```

Analysing detailed execution can involve complex code lines involving multiple predicates

Static Testing – Trace Table Example 2 (Trace Table)

- Use of trace table to determine intermediate and overall results

```
P.x = 4      R.lower_left.x = 3
P.y = 10     R.upper_right.x = 5
             R.lower_left.y = 7
             R.upper_right.y = 9
```

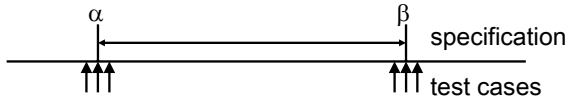


code line	P.x > R.lo_left.x	P.x < R.up_right.x	P.y > R.lo_left.y	P.y < R.up_right.y	result
1	4>3 (T)				
2		4<5 (T)			
3			10>7 (T)		
4				10<9 (F)	
5					False

Use of trace table to determine intermediate and overall results

Boundary Value Analysis

- Aim: To remove software errors occurring at parameter limits or boundaries
- If an input condition specifies a range bounded by values α and β, test cases should be designed with values α and β, just above and just below α and β, respectively.
- If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.



A greater number of errors tends to occur at the boundaries of the input domain.

Boundary Value Analysis (BVA) leads to a selection of test cases that exercise bounding values.

Boundary Check on Outputs

- The previous two guidelines also apply to output conditions.
- If the output is a sequence of data, e.g. a printed table, special attention should be paid to the first and last elements and to lists containing 0, 1 and 2 elements.

Care with first and last output value

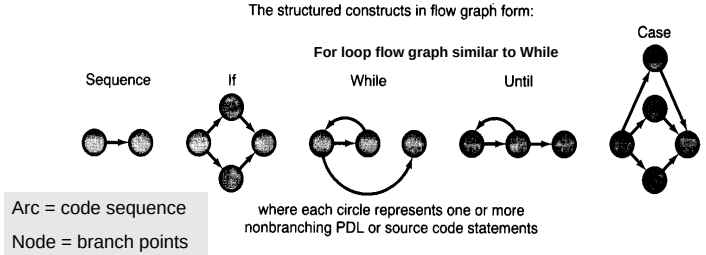
- If internal program data structures have prescribed boundaries (e.g. an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Care with internal data values

- The use of the value zero, in a direct as well as indirect translation, is often error-prone and demands special attention.
- E.g. zero divisor; blank ASCII characters; empty stack or list element; full matrix (array), zero table entry

Control Flow Analysis

- Control flow graphs of standard control structures for McCabe Cyclomatic Complexity
- McCabe Cyclomatic metric gives an indication of design and testing complexity



Control Flow Analysis is a static testing technique for finding suspect areas of code that do not follow good programming practice.

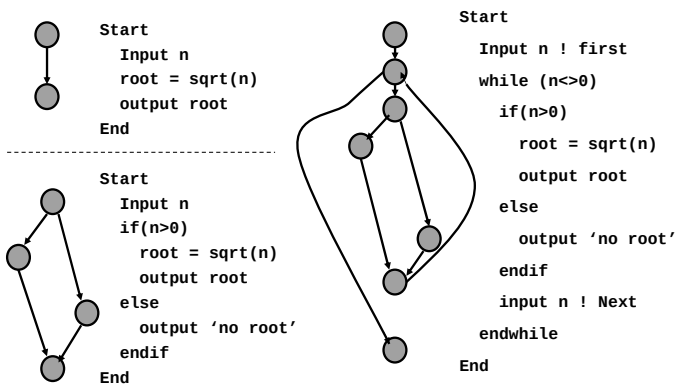
The aim is to detect poor and potentially incorrect program structures.

A directed graph represents the sequence of execution in a program module, in which nodes represent branching points or subprogram calls in a program, and arcs represent linear sequences of code.

From the control-flow graph an analysis can show:

- The structure of the program
- Starts and ends of program segments
- Unreachable code and dynamic halts
- Branches from within loops
- Entry and exit points for loops
- Paths through the program, that can be separately tested

Control Flow Analysis – Examples



Construct flow graph from pseudocode/PDL

Arc = control flow

Node = sequence of code

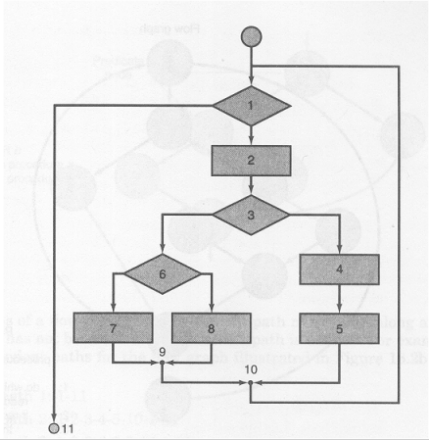
Example: Program Design Language (Pseuodcode)

```
PDL/pseuodcode - procedure: sort

1:      while records remain do
2:          read record
3:          if record field 1 = 0 then
4:                  process record
5:                  store in buffer and
                    increment counter
6:          else
                    if record field 2 = 0 then
7:                          reset counter
8:                  else
                            process record
                            store in file
9:                  end if
10:         end if
11:     end while
```

27

Example: Control Structure in Flowchart

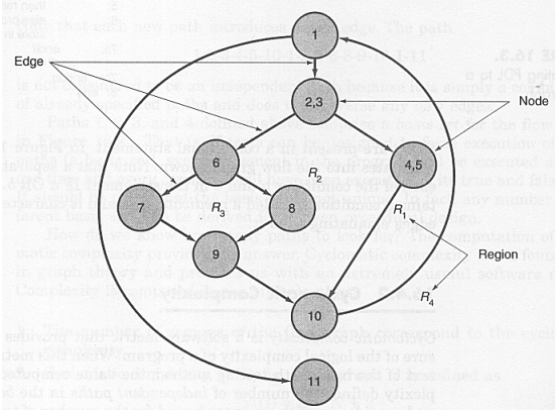


Node – control statement

Edge /arc – control flow

28

Example : Control Flow Graph



Example : McCabe Cyclomatic Complexity Calculation

- McCabe Cyclomatic complexity – is a software metric that provides a Quantitative measure of the logical complexity of a program.
  - The flow has 4 regions (marked as R1, R2, R3 and R4); hence V(G) = 4
  - V(G) = E - N + 2 = 11 edges - 9 nodes + 2 = 4
  - V(G) = P + 1 = 3 predicate nodes + 1 = 4. Predicate nodes are: node (2,3), node (1), and node (6).

Example: Independent Path

- Thus, the value of V(G) provides us with an upper bound for the number of independent paths ....(etc...)
- The set of independent paths are:
  - Path 1: 1-11
  - Path 2: 1-2-3-4-5-10-1-11
  - Path 3: 1-2-3-6-8-9-10-1-11
  - Path 4: 1-2-3-6-7-9-10-1-11

Control Flow Analysis Approaches

- These approaches can test a program statically:
  - **Statement Testing** (Coverage Testing) – Every statement in the program is executed at least once (finding a set of paths whose union contains all the nodes of the graph). Design individual tests by setting values of appropriate program variables;
  - **Branch Testing** – For every decision point in the program, each branch is chosen at least once (finding a set of paths whose union covers all the edges of the graph);
  - **Path Testing** (Structural Testing) – Every distinct path through the program is executed at least once (need to find all possible paths through the graph).

Control flow analysis can reveal each control path allowing full structural testing to take place

Dynamic testing should attempt to cover the same tests.

Symbolic Execution

- A technique in which the input variables of a program are assigned symbolic values rather than literal values.
- The aim is show the agreement between source code and its specification.
- It is a Algebraic method.

Symbolic Execution – A technique in which the input variables of a program are assigned symbolic values rather than literal values.

- Aim: To show the agreement between source code and its specification.
- Algebraic method.
- A program is analyzed by propagating the symbolic values of the inputs into the operands in expressions.
- The resulting symbolic expressions are simplified at each step in the computation so that all intermediate computations and decisions are always expressed in terms of symbolic inputs.
- As a result all computations and decisions are expressed as symbolic values of the inputs.

Symbolic Evaluation of Program

A program is "executed" using symbols rather than actual values.

- Expression
  - Substitute the symbolic (algebraic) value of each variable into the expression
  - Simplify the result expression by application of algebraic laws
- Assignment
  - The resulting symbolic value of the right-hand side expression becomes the new symbolic value of the variable on the left-hand side
- Conditional branching
  - The predicate becomes a constraint on the symbolic value
- Output
  - The symbolic value of the variables is the result

Example of Symbolic Execution

Program Example;

```
Begin
     Var x, y: Integer;
     Input (x);
     Input(y);
     x := x+x;
     y := y*y;
     IF x > y
        THEN x := x – y
        ELSE x:= x + y
     END;
     Output(x)
End
```

Symbolic execution

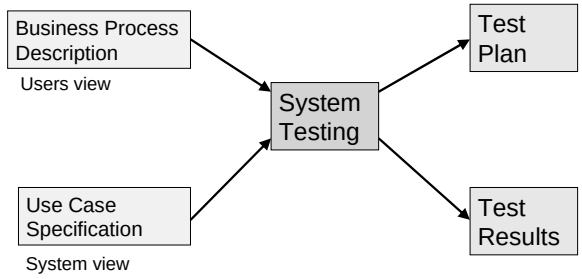
```
x = a          y = b
x = a+a=2a
                y = b*b=b^2
2a>b^2
x=2a-b^2  where 2a>b^2
(x=2a+b^2 where 2a≤b^2)

Output: 2a-b^2 where 2a>b^2
(Output: 2a+b^2 where 2a≤b^2)
```

A Black Box Testing Approach

- Produce test cases + test data from software specification
- Check correctness by test execution, comparing actual against expected output/behaviour
- Provide adequate test cases to cover all system functions in all scenarios

Scenario (Use Case) Testing



Scenario – a specific sequence of actions and interactions between actors and the system

Use cases can provide basis for deriving tests for a system

Generally a Black Box test approach is taken

Static Testing – each scenario can be separately reviewed

Dynamic Testing – each scenario can be separately exercised

Realistic and comprehensive set of operations

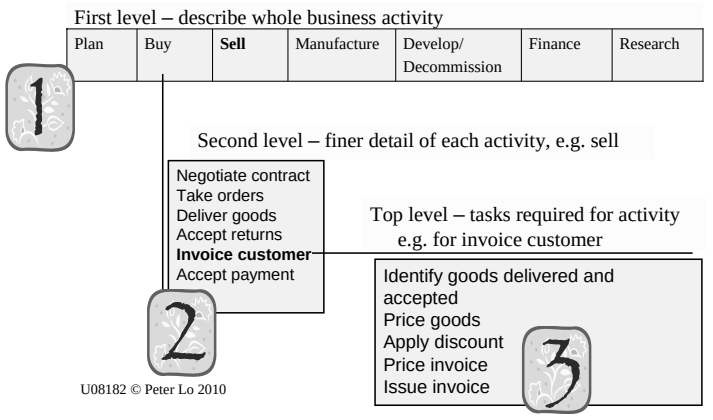
Creating Scenarios considerations:

- Object life history
- All possible users and their interests
- Likely events and special events

Business Process Definition

- Collection of related structural activities that are required by the organisation
- Specify how a business operates, or how the system will work
- Often constrained by standard business practice interfacing with other parties.
  - E.g. service provision, transaction processing

Business Process Definition - ICANDO Chemicals Example (Lunn, 2003)



Use Case Specification

- UML systems development documentation
- Collections of related scenarios describing actor involvement
- Documentation to support use case diagrams giving fine detail if the user interaction
- Provides a link between user specification (what the user expects) and system specification (what the system must do)
- Sequence diagrams useful to show sequence of events
- Can be formalised: precondition, operation, post-condition - ideal for test specification
- Provides a basis for acceptance testing

Use Cases and Testing

- As each Use Case evolves
  - Capture requirements
  - Provide development input for design and implementation
  - Provide basis for test cases
  - Provide basis for user documentation
  - Provides a basis for project management
  - Provide documentation for business customer

Test Case

- As each Use Case evolves
  - Define set of conditions for a use case
  - Testing will determine if a requirement is satisfied in a specification, design or system implementation
  - Must be at least one test per test case (see adequacy later)
  - Documentation: test case id, pre-conditions, inputs, test procedure, expected output or post-conditions
  - Test result may need to be evaluated (if not simple pass / fail)
  - Test cases compiled into test suites

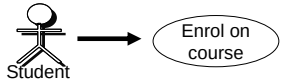
Synonyms of 'test case': test condition, test script, test scenario, test specification, test assertion

Use Case Template

Use case name – unique identifier
Status – development state
Description - overview
Precondition – conditions required to initiate use case
Trigger – condition/s which initiated a use case
Basic course of events – primary scenario main steps
Alternative paths – dealing with unusual or exceptional events
Post condition - state after completing use case
Business process rules – how business operated in relation to use case
Notes – any additional information
History record of changes - author, date, version

Ref: Wikipedia

Test Case Testing – Example



Use Case diagram

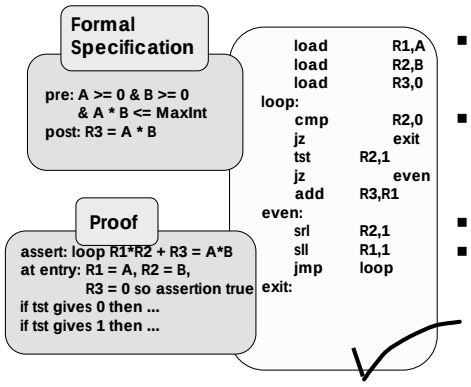
Use Case text

Main Flow	
Student (Actor) Action	**System Response**
Enrol on course	
	Get pre-requisites for course
	Validate student qualification and conditions
	Validate course state
	Save the enrolment entry
	Inform student
Alternative Flows	
	Student does not meet course conditions
	The course is full
	The course is closed

GUI Testing

- Traditionally labour intensive since difficult to automate user interaction, so expensive and unreliable
- Capture/replay/playback tools useful to automate user scenarios
- Generally event driven programming
- Testing aspects:
  - Navigation (hierarchy)
  - Application equivalence and boundary states
  - Desktop integration and synchronization
  - Non-functional, e.g. interoperability

Static Testing: Formal Proof



- Feasible for small sections of critical code only requires formal spec
- Semi-experimental status (except in some safety critical and similar systems)
- Intensive & expensive
- Proof could contain bugs

This form of static testing is used to prove mathematically that a program conforms to its specification. Static testing generally involves identifying and analysing individual test cases whereas formal proof uses mathematical techniques (e.g. equality, induction etc) to show or prove that a system meets its specification

Formal proof requires a program to be fully specified using a formal specification language.

Specify a module in terms of:

- signature (arguments)
- pre, post conditions
- module specification, e.g. using Z, CSP or AMN

While this method has been used for some applications it still presents problems:

- ***It is only feasible on small sections of code*** – a proof of the correctness of a program soon becomes incredibly long.
- ***It requires a formal specification*** – a program can be proved to behave according to its specification only if the specification is written in a sufficiently precise form (essentially using a mathematical notation). Formal specification languages have a number of disadvantages not least that relatively few people can understand them. Even if system designers understand formal specifications most users don't and so they may lack confidence in the specification.
- ***It is still semi-experimental*** – an increasing amount of safety critical systems work uses formal specification and proof but there is little experience of using it in mainstream computing areas.
- ***It is expensive and intensive*** – not least as the number of experienced practitioners is so low.
- ***Proofs could contain bugs*** – a proof may itself contain errors. One possibility is automatic proof (or proof checking) but this is still in its research stage.

Despite the problems, the use of formal specifications, where the expense can be justified, does seem to reduce the number of bugs simply by requiring that specifications are written in a mathematically precise way (even if no proofs are attempted).

Formal Specification

- Used for critical code sections, e.g. OS deadlock avoidance
- Conventional informal systems development:
- Limitations of conventional approach
- Formal specification
  - Based on mathematics and logic
  - Precise and unambiguous
  - Easier to modify system
  - Potential for automated checking

Conventional informal systems development:

- Requirement specification (what) Design specification (how) and Test plan: show that implementation satisfies the specification

Limitations of conventional approach

- Ambiguous (semantics, completeness, etc) of natural language or pseudocode specification

Formal specification

- Based on mathematics and logic
- Precise and unambiguous
- Easier to modify system
- Potential for automated checking

Algebraic: system is described in terms of operations and their relationships e.g. OBJ (sequential), Lotos (concurrent)

Model Based: system model constructed using mathematics e.g. Z, VDM (sequential), CSP, Abstract Machine Notation(AMN), Petri Nets (concurrent)

- Components of a Model based Formal Specification:
- The operation inputs
- The precondition, which must be true for the operation to be applicable
- The post condition showing the relationship of the system before and after applying the operation

Model Based Formal Specification Example 1

- Operation to add a customer at a bank to a queue

```
Operation: arrive(customer: account_number)
Precondition:
     customer account_number belongs to bank
     customer does not belong to queue
Postcondition
     queue = queue append[customer]
```


Model Based Formal Specification Example 2

- Operation to search an array for a given element value

```
Operation:
     search(A:array[1..N], X)
Precondition:
     N>0
Postcondition
     X = A[I] AND 1<=I<=N
     OR
     I=0 AND (A[J]<>X For All 1<=J<=N)
```

Model Based Formal Specification Example 3

- Abstract Machine Notation (AMN) example

```
.
.
PermitPassing =
PRE
  train=approachPermitted  AND
  Signal=red  AND
  Barrier=closed  AND
  Communication_interference=FALSE
THEN
  train :=  passingPermitted
END
.
.
```