Atria Institute of Technology

Department of Information Science and Engineering

Bengaluru-560024

ACADEMIC YEAR: 2021-2022

EVEN SEMESTER NOTES

Semester : 6th Semester

Subject Name : Software Testing

Subject Code : 18IS62

Faculty Name : Mrs. Uzma Taj

MODULE 1

Basics of Software Testing: Basic definitions, Software Quality , Requirements, Behaviour and Correctness, Correctness versus Reliability, Testing and Debugging, Test cases, Insights from a Venn diagram, Identifying test cases, Test-generation Strategies, Test Metrics, Error and fault taxonomies , Levels of testing, Testing and Verification, Static Testing. Problem Statements: Generalized pseudocode, the triangle problem, the NextDate function, the commission problem, the SATM (Simple Automatic Teller Machine) problem, the currency converter, Saturn windshield wiper

Basic definitions

What is Software Testing?

Software testing is defined as an activity to check whether the actual results match the expected results and to ensure that the software system is Defect free. It involves execution of a software component or system component to evaluate one or more properties of interest.

Software testing also helps to identify errors, gaps or missing requirements in contrary to the actual requirements. It can be either done manually or using automated tools. Some prefer saying Software testing as a White Box and Black Box Testing.

In simple terms, Software Testing means Verification of Application under Test (AUT).

Why is Software Testing Important?

Testing is important because software bugs could be expensive or even dangerous. Software bugs can potentially cause monetary and human loss, and history is full of such examples.

In April 2015, Bloomberg terminal in London crashed due to software glitch affected more than 300,000 traders on financial markets. It forced the government to postpone a 3bn pound debt sale.

Nissan cars have to recall over 1 million cars from the market due to software failure in the airbag sensory detectors. There has been reported two accident due to this software failure.

Starbucks was forced to close about 60 percent of stores in the U.S and Canada due to software failure in its POS system. At one point store served coffee for free as they unable to process the transaction.

Some of the Amazon‘s third party retailers saw their product price is reduced to 1p due to a software glitch. They were left with heavy losses.

Vulnerability in Window 10. This bug enables users to escape from security sandboxes through a flaw in the win32k system.

In 2015 fighter plane F-35 fell victim to a software bug, making it unable to detect targets correctly.

China Airlines Airbus A300 crashed due to a software bug on April 26, 1994, killing 264 innocent live

In 1985, Canada's Therac-25 radiation therapy machine malfunctioned due to software bug and delivered lethal radiation doses to patients, leaving 3 people dead and critically injuring 3 others.

In April of 1999, a software bug caused the failure of a $1.2 billion military satellite launch, the costliest accident in history

In may of 1996, a software bug caused the bank accounts of 823 customers of a major U.S. bank to be credited with 920 million US dollars.

Basic Definitions

Error
People make errors. A good synonym is ―mistake‖. When people make mistakes while coding, we call these mistakes ―bugs‖. Errors tend to propagate; a requirements error may be magnified during design, and amplified still more during coding.

Fault
A fault is the result of an error. It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, dataflow diagrams, hierarchy charts, source code, and so on. ―Defect‖ is a good synonym for fault; so is ―bug‖.

Failure
A failure occurs when a fault executes. Two subtleties arise here: one is that failures only occur in an executable representation, which is usually taken to be source code, or more precisely, loaded object code.

Incident
When a failure occurs, it may or may not be readily apparent to the user (or customer or tester). An incident is the symptom(s) associated with a failure that alerts the user to the occurrence of failure.

Test
Testing is obviously concerned with errors, faults, failures, and incidents. A test is the act ofexercising software with test cases. There are two distinct goals of a test: either to find failures, or to demonstrate correct execution.

Test Case
A test case has an identity, and is associated with a programbehaviour. A test case also has a set of inputs, a list of expected outputs.

Software Quality

SOFTWARE QUALITY is the degree of conformance to explicit or implicit requirements and expectations.

Explanation

- Explicit: clearly defined and documented
- Implicit: not clearly defined and documented but indirectly suggested
- Requirements: business/product/software requirements
- Expectations: mainly end-user expectations

The modern view of a quality associated with a software product several quality methods such as the following:

Portability: A software device is said to be portable, if it can be freely made to work in various operating system environments, in multiple machines, with other software products, etc.

Usability: A software product has better usability if various categories of users can easily invoke the functions of the product.

Reusability: A software product has excellent reusability if different modules of the product can quickly be reused to develop new products.

Correctness: A software product is correct if various requirements as specified in the SRS document have been correctly implemented.

Maintainability: A software product is maintainable if bugs can be easily corrected as and when they show up, new tasks can be easily added to the product, and the functionalities of the product can be easily modified, etc.

Requirements

What is Software Requirement?

It's a primary requirement needed in the development of a software product. These requirements works as a base and is being used in developing a particular software product to perform specifically for a targeted group or audience and for the specific environment.

These requirements are of very much importance as any sort of compromise to them may produce undesirable final product and may fail to meet the needs & expectations of a client or a user. Therefore, there exists a separate phase in a SDLC to gather, study and analyse the software requirements so as to avoid such type of circumstances.

Types of Requirements In Software Testing

Business Requirements:

These requirements are specified from the business point of view. It generally involves the specified objectives and goals of a particular project that needs to be fulfilled. It provides an

abstract of a project. These requirements are not meant for specifying the functionalities or technicalities of a desired software product rather it outlines a general overview of a product, such as its primary use, why it is needed, its scope & vision, what business benefits will be gain, intended audience or users, etc. It generally involves the participation of the client, stakeholders, business and project managers for gathering and analyzing the business requirements.

Through business requirements, it is easy to assess the project cost, time required, business risks involved and many such things associated with a software development project.

System Requirements:
Requirements to be incorporated in a software product under development to make a software product perform and function in a specific manner to achieve a specific target and goal falls under the category of system requirements. These system requirements may be broadly classified in two types‘ functional requirements and non-functional requirements.

- Functional requirements:
  Requirements encompassing the functional attributes and behaviour of a software product are called functional requirements. These requirements reflect the working and functionalities of an intended software product.
  These requirements defines and describes the functions to be performed, and features to be possessed by a software product. What and how does a product supposed to perform on accepting inputs from the user, and what desirable output it should provide to the users. These requirements should be complete and clearly well-defined so as to meet all the specified feature and functionalities without misunderstanding or leaving the requirement so as to achieve a desirable quality product.

- Non-Functional Requirements:
  Requirements other than functional requirements which are essential and contribute towards the performance of a software product under variant type of conditions and multiple environments are commonly known as Non-functional requirements. These requirements are used to evaluate and assess the software product behaviour other than its specific or desired behaviour under unexpected conditions and environment, contrary to what is favourable for its functioning. It also covers the standards, rules and regulation that a software product must adhere and conform to it.

User Requirements:
Requirements generated from a user‘s point of view and scenarios of using a software product in a multiple manner under real environment by a targeted user to execute a particular task, specifies the user requirements. It defines the user's expectation from a software product. As user‘s exhaustive needs may not be covered under the domain of system requirement, it may be covered separately by business analysts through studying and analysing the user requirements.

These types of requirements are generally gathered and documented using use cases, user scenarios, and user stories. These requirements are documented in a user requirement document (URD) format by making use of narrative text and are usually signed off by the intended users.

Behaviour and Correctness

What is Correctness?

Correctness from software engineering perspective can be defined as the adherence to the specifications that determine how users can interact with the software and how the software should behave when it is used correctly.

If the software behaves incorrectly, it might take considerable amount of time to achieve the task or sometimes it is impossible to achieve it.

Important rules:

Below are some of the important rules for effective programming which are consequences of the program correctness theory.

- Defining the problem completely.
- Develop the algorithm and then the program logic.
- Reuse the proved models as much as possible.
- Prove the correctness of algorithms during the design phase.
- Developers should pay attention to the clarity and simplicity of your program.
- Verifying each part of a program as soon as it is developed.

Correctness versus Reliability

Correctness: The degree to which a system is free from [defects] in its specification, design, and implementation.

Reliability: The ability of a system to perform its requested functions under stated conditions whenever required - having a long mean time between failures.

Testing and Debugging

Differences between Testing and Debugging

Testing:
Testing is the process of verifying and validating that a software or application is bug free, meets the technical requirements as guided by its design and development and meets the user requirements effectively and efficiently with handling all the exceptional and boundary cases.

Debugging:
Debugging is the process of fixing a bug in the software. It can defined as the identifying, analysing and removing errors. This activity begins after the software fails to execute properly and concludes by solving the problem and successfully testing the software. It is

considered to be an extremely complex and tedious task because errors need to be resolved at all stages of debugging.

TESTING	DEBUGGING
Testing is the process to find bugs and errors.	Debugging is the process to correct the bugs found during testing.
It is the process to identify the failure of implemented code.	It is the process to give the absolution to code failure.
Testing is the display of errors.	Debugging is a deductive process.
Testing is done by the tester.	Debugging is done by either programmer or developer.
There is no need of design knowledge in the testing process.	Debugging can‘t be done without proper design knowledge.
Testing can be done by insider as well as outsider.	Debugging is done only by insider. Outsider can‘t do debugging.
Testing can be manual or automated.	Debugging is always manual. Debugging can‘t be automated.
It is based on different testing levels i.e. unit testing, integration testing, system testing etc.	Debugging is based on different types of bugs.

Test Case

A TEST CASE is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly.
The process of developing test cases can also help find problems in the requirements or design of an application.
Test Case Template
A test case can have the following elements. Note, however, that a test management tool is normally used by companies and the format is determined by the tool used.

Test Suite ID	The ID of the test suite to which this test case belongs.
Test Case ID	The ID of the test case.
Test Case Summary	The summary / objective of the test case.
Related Requirement	The ID of the requirement this test case relates/traces to.
Prerequisites	Any prerequisites or preconditions that must be fulfilled prior to executing the test.
Test Procedure	Step-by-step procedure to execute the test.
Test Data	The test data, or links to the test data, that are to be used while conducting the test.
Expected Result	The expected result of the test.
Actual Result	The actual result of the test; to be filled after executing the test.
Status	Pass or Fail. Other statuses can be _Not Executed‘ if testing is not performed and _Blocked‘ if testing is blocked.

Remarks	Any comments on the test case or test execution.
Created By	The name of the author of the test case.
Date of Creation	The date of creation of the test case.
Executed By	The name of the person who executed the test.
Date of Execution	The date of execution of the test.
Test Environment	The environment (Hardware/Software/Network) in which the test was executed.

Test Case Example / Test Case Sample

Test Suite ID	TS001
Test Case ID	TC001
Test Case Summary	To verify that clicking the Generate Coin button generates coins.
Related Requirement	RS001
Prerequisites	User is authorized. Coin balance is available.
Test Procedure	Select the coin denomination in the Denomination field. Enter the number of coins in the Quantity field. Click Generate Coin.
Test Data	Denominations: 0.05, 0.10, 0.25, 0.50, 1, 2, 5
Quantities:	0, 1, 5, 10, 20
Expected Result	Coin of the specified denomination should be produced if the specified Quantity is valid (1, 5) A message _Please enter a valid quantity between 1 and 10' should be displayed if the specified quantity is invalid.
Actual Result	If the specified quantity is valid, the result is as expected. If the specified quantity is invalid, nothing happens; the expected message is not displayed
Status	Fail
Remarks	This is a sample test case.
Created By	John Doe
Date of Creation	01/14/2020
Executed By	Jane Roe
Date of Execution	02/16/2020
Test Environment	OS: Windows Y
Browser:	Chrome N

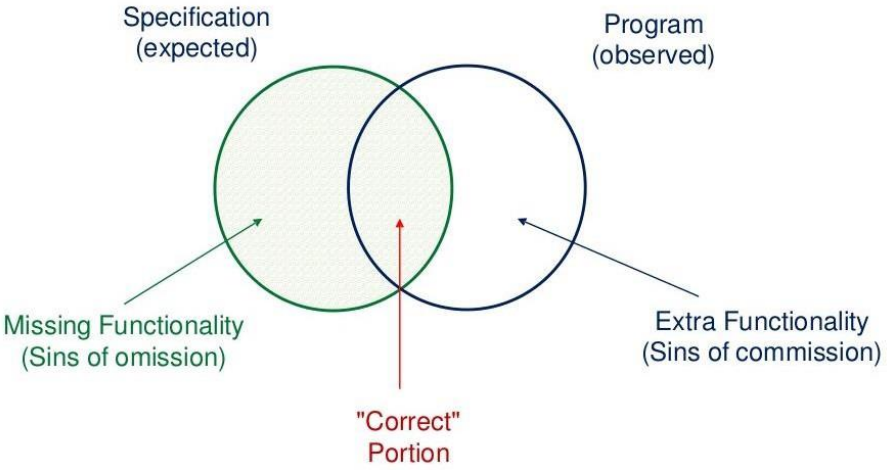
Insights from a Venn diagram

Insights from a Venn Diagram

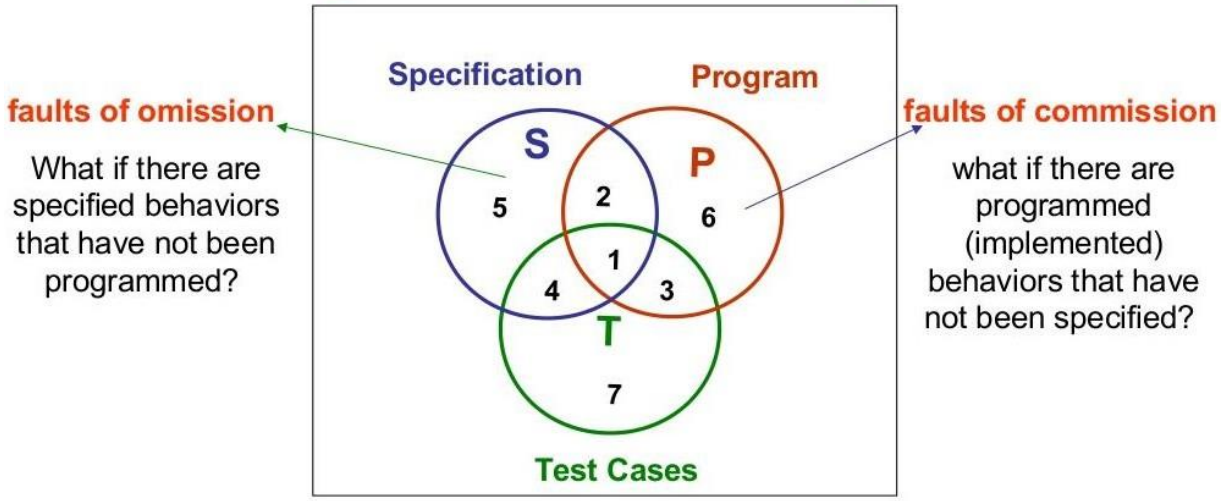
Testing is fundamentally concerned with behavior, and behavior is orthogonal to the code-based view common to software (and system) developers

A quick distinction is that: – The code-based view focuses on what it is – The behavioral view considers what it does

Program Behaviors



Insights from a Venn Diagram



- A very good view of testing is that it is the determination of the extent of program behavior that is both specified and implemented
- What can a tester do to make the region where these sets all intersect be as large as possible?
- How the test cases in the set T are identified?.

Identifying Test Cases

There are two fundamental approaches to identifying test cases; these are known as functional and structural testing. Each of these approaches has several distinct test case identification methods, more commonly called testing methods.

What is Structural Testing ?

Structural testing, also known as glass box testing or white box testing is an approach where the tests are derived from the knowledge of the software's structure or internal implementation.

The other names of structural testing includes clear box testing, open box testing, logic driven testing or path driven testing.

Structural Testing Techniques:

- **Statement Coverage -** This technique is aimed at exercising all programming statements with minimal tests.
- **Branch Coverage -** This technique is running a series of tests to ensure that all branches are tested at least once.
- **Path Coverage -** This technique corresponds to testing all possible paths which means that each statement and branch are covered.

Advantages of Structural Testing:

- Forces test developer to reason carefully about implementation
- Reveals errors in "hidden" code
- Spots the Dead Code or other issues with respect to best programming practices.

Disadvantages of Structural Box Testing:

- Expensive as one has to spend both time and money to perform white box testing.
- Every possibility that few lines of code is missed accidentally.
- Indepth knowledge about the programming language is necessary to perform white box testing.

What is Functional Testing?

Functional testing is a quality assurance (QA) process[1] and a type of black-box testing that bases its test cases on the specifications of the software component under test. Functions are tested by feeding them input and examining the output, and internal program structure is rarely considered (unlike white-box testing).

Functional Testing is a testing technique that is used to test the features/functionality of the system or Software, should cover all the scenarios including failure paths and boundary cases.

It is basically defined as a type of testing which verifies that each function of the software application works in conformance with the requirement and specification. This testing is not concerned about the source code of the application. Each functionality of the software application is tested by providing appropriate test input, expecting the output and comparing the actual output with the expected output. This testing focuses on checking of user interface, APIs, database, security, client or server application and functionality of the Application Under Test.

The other major Functional Testing techniques include:

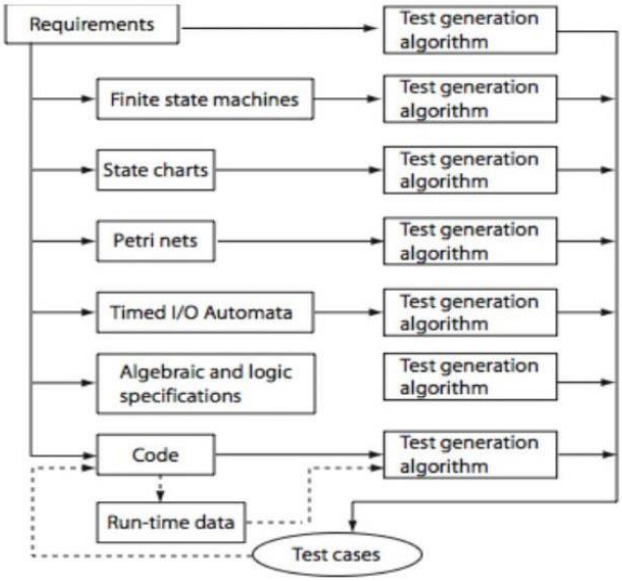
- Unit Testing
- Integration Testing
- Smoke Testing
- User Acceptance Testing
- Localization Testing
- Interface Testing
- Usability Testing
- System Testing
- Regression Testing
- Globalization Testing

Test-generation Strategies

Test generation Any form of test generation uses a source document. In the most informal of test methods, the source document resides in the mind of the tester who generates tests based on a knowledge of the requirements. In several commercial environments, the process is a bit more formal. The tests are generated using a mix of formal and informal methods either directly from the requirements document serving as the source. In more advanced test processes, requirements serve as a source for the development of formal models.

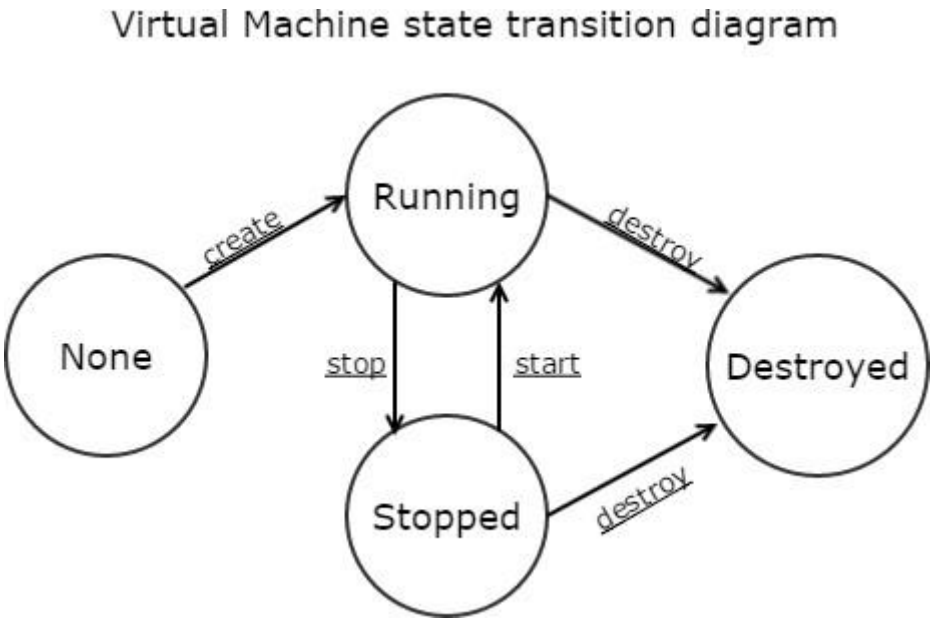
Test generation strategies Model based: require that a subset of the requirements be modeled using a formal notation (usually graphical). Models: Finite State Machines, Timed automata, Petri net, etc. Specification based: require that a subset of the requirements be modeled using a formal mathematical notation. Examples: B, Z, and Larch. Code based: generate tests directly from the code.

Test generation strategies (Summary)



A '**Finite State Machine** (**FSM**)' is a system that will be in different discrete **states** (like ‒ready‖, ‒not ready‖, ‒open‖, ‒closed‖,…) depending on the inputs or stimuli. The discrete **states** that the system ends up with, depends on the rules of the transition of the system.

For example, VM states from user perspective are like:



Statechart diagram is one of the five UML diagrams used to model the dynamic nature of a system. They define different **states** of an object during its lifetime and these **states** are changed by events. **Statechart** diagrams are useful to model the reactive systems.

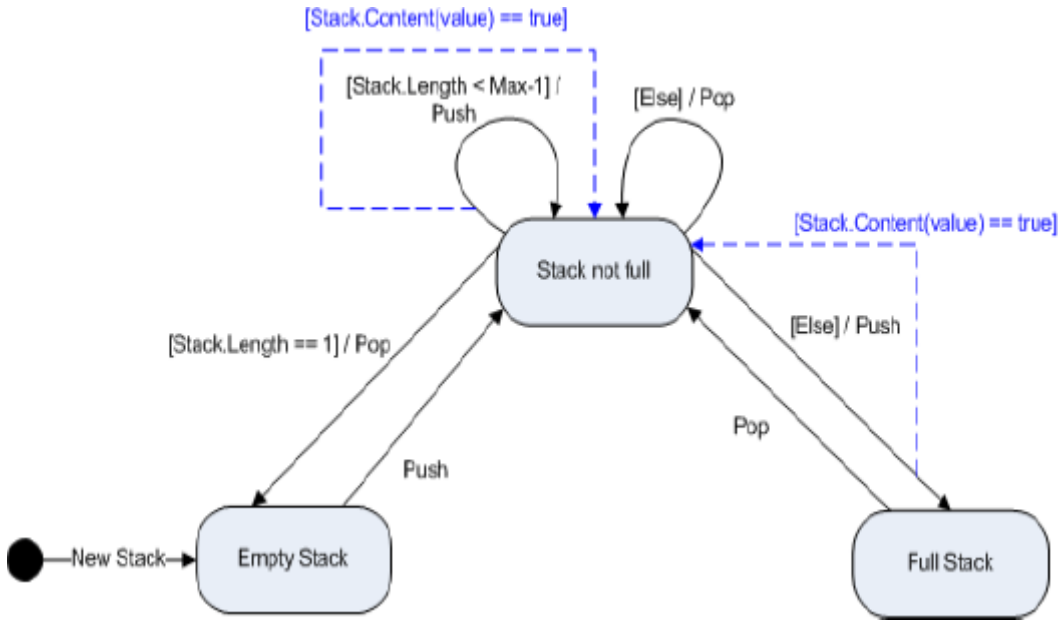
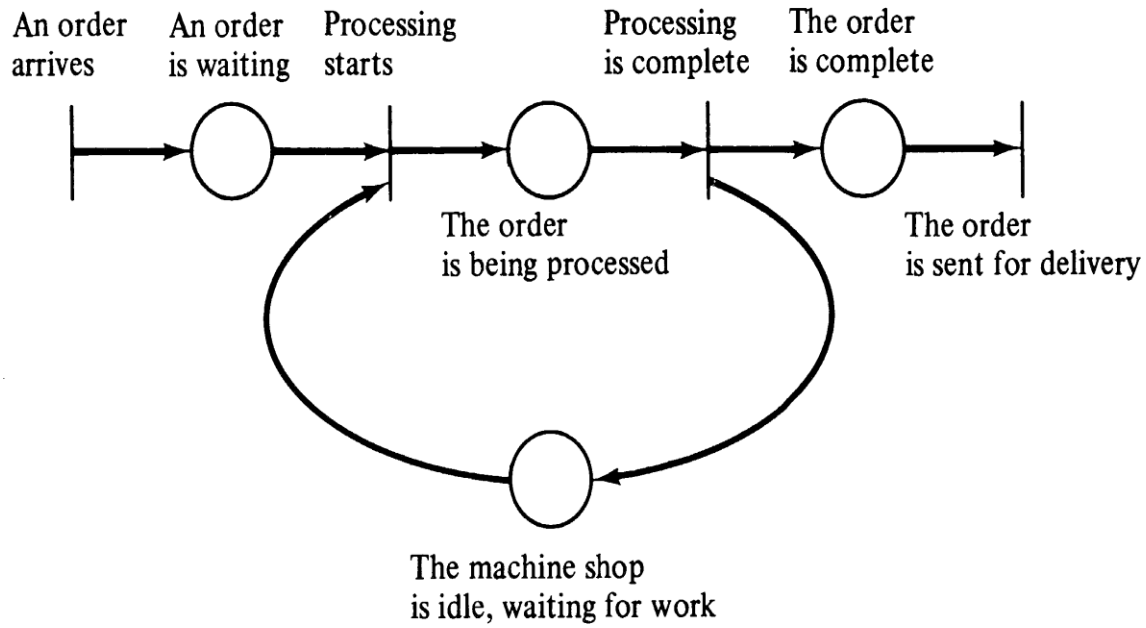


Figure 2**:** Statechart diagram with an around advice on method Push.

Petri nets were designed for and are used mainly for *modeling*. Many systems, especially those with independent components, can be modeled by a Petri net. The systems may be of many different kinds: computer hardware, computer software, physical systems, social systems, and so on. Petri nets are used to model the occurrence of various events and activities in a system. In particular, Petri nets may model the flow of information or other resources within a system.

The Petri net of Figure 3.1 is a Petri net model of the machine shop example given above. We have labeled each transition and place with the corresponding event or condition.



Timed I/O automata

In theoretical computer science, automata theory is the study of abstract machines (or more appropriately, abstract 'mathematical' machines or systems) and the computational problems that can be solved using these machines. These abstract machines are called automata. This automaton consists of
• states (represented in the figure by circles),
• and transitions (represented by arrows). As the automaton sees a symbol of input, it makes a transition (or jump) to another state, according to its transition function (which takes the current state and the recent symbol as its inputs). Uses of Automata: compiler design and parsing.

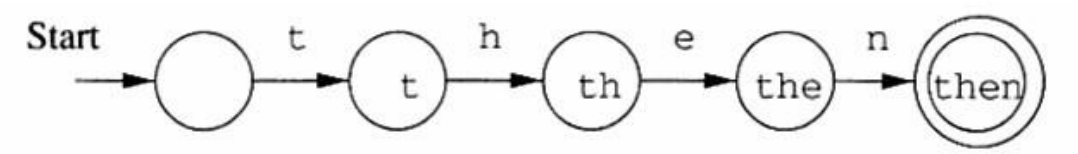


Figure 1.2: A finite automaton modeling recognition of then

Test Metrics

Software test metrics is to monitor and control process and product. It helps to drive the project towards our planned goals without deviation.

Metrics answer different questions. It‘s important to decide what questions you want answers to.

Software test metrics are classified into two types

1. Process metrics
2. Product metrics

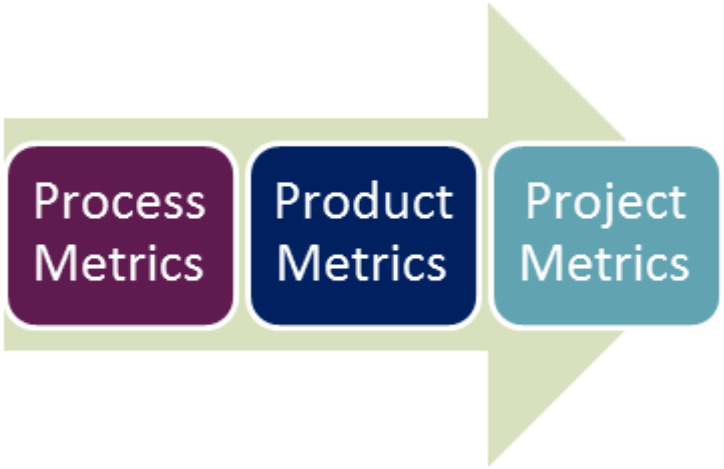
The ideal example to understand metrics would be a weekly mileage of a car compared to its ideal mileage

Why Test Metrics are Important?

"We cannot improve what we cannot measure" and Test Metrics helps us to do exactly the same.

- Take decision for next phase of activities
- Evidence of the claim or prediction
- Understand the type of improvement required
- Take decision or process or technology change

Types of Test Metrics



- **Process Metrics:** It can be used to improve the process efficiency of the SDLC (Software Development Life Cycle)
- **Product Metrics:** It deals with the quality of the software product
- **Project Metrics:** It can be used to measure the efficiency of a project team or any testing tools being used by the team members

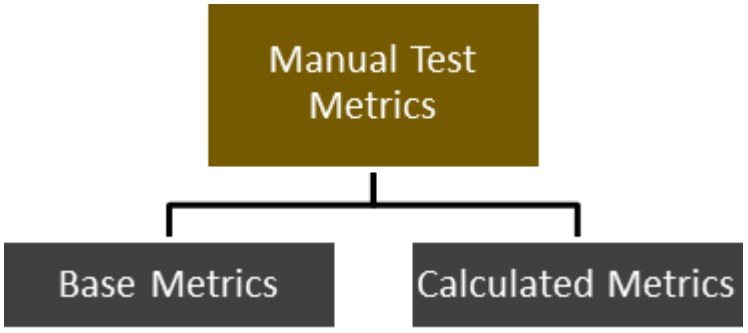
Identification of correct testing metrics is very important. Few things need to be considered before identifying the test metrics

- Fix the target audience for the metric preparation
- Define the goal for metrics
- Introduce all the relevant metrics based on project needs
- Analyze the cost benefits aspect of each metrics and the project lifestyle phase in which it results in the maximum output

Manual Test Metrics

In Software Engineering, Manual test metrics are classified into two classes

- **Base Metrics**
- **Calculated Metrics**



Base metrics is the raw data collected by Test Analyst during the test case development and execution (**# of test cases executed, # of test cases**).

While calculated metrics are derived from the data collected in base metrics. Calculated metrics is usually followed by the test manager for test reporting purpose (**% Complete, % Test Coverage**).

Depending on the project or business model some of the important metrics are

Test Metrics Life Cycle

Different stages of Metrics life cycle	Steps during each stage
• Analysis	• Identification of the Metrics • Define the identified QA Metrics
• Communicate	• Explain the need for metric to stakeholder and testing team • Educate the testing team about the data points to need to be captured for processing the metric
• Evaluation	• Capture and verify the data • Calculating the metrics value using the data captured
• Report	• Develop the report with an effective conclusion • Distribute the report to the stakeholder and respective representative • Take feedback from stakeholder

Error and fault taxonomies

Error and Fault Taxonomies Process versus Product

process refers to how we do something, and

product is the end result of a process SQA is more concerned with reducing errors endemic in the development process, while testing is more concerned with discovering faults in a product.

Faults can be classified in several ways:
• the development phase where the corresponding error occurred,
• the consequences of corresponding failures,

• difficulty to resolve,
• risk of no resolution, and so on.

Fault Taxonomies

Input / Output Faults

Type	Instances
Input	Correct input not accepted
	Incorrect input accepted
	Description wrong or missing
	Parameters wrong or missing
Output	Wrong format
	Wrong result
	Correct result at wrong time (too early, too late)
	Incomplete or missing result
	Spurious result
	Spelling/grammar
	Cosmetic

Fault Taxonomies

Logic Faults

Missing case(s)
Duplicate case(s)
Extreme condition neglected
Misinterpretation
Missing condition
Extraneous condition(s)
Test of wrong variable
Incorrect loop iteration
Wrong operator (e.g., < instead of ≤)

Fault Taxonomies

Computation Faults

Incorrect algorithm
Missing computation
Incorrect operand
Incorrect operation
Parenthesis error
Insufficient precision (round-off, truncation)
Wrong built-in function

Fault Taxonomies

Interface Faults

Incorrect interrupt handling
I/O timing
Call to wrong procedure
Call to nonexistent procedure
Parameter mismatch (type, number)
Incompatible types
Superfluous inclusion

Fault Taxonomies

Data Faults

Incorrect initialization	Incorrect data dimension
Incorrect storage/access	Incorrect subscript
Wrong flag/index value	Incorrect type
Incorrect packing/unpacking	Incorrect data scope
Wrong variable used	Sensor data out of limits
Wrong data reference	Off by one
Scaling or units error	Inconsistent data

Levels of testing

What are the levels of testing?

A level of software testing is a process where every unit or component of a software/system is tested. The main goal of system testing is to evaluate the system's compliance with the specified needs.

There are many different testing levels which help to check behavior and performance for software testing. These testing levels are designed to recognize missing areas and reconciliation between the development lifecycle states. In SDLC models there are characterized phases such as requirement gathering, analysis, design, coding or execution, testing, and deployment.

All these phases go through the process of software testing levels. There are mainly four testing levels are:

1. Unit Testing
2. Integration Testing
3. System Testing
4. Acceptance Testing
5. Each of these testing levels has a specific purpose. These testing level provide value to the software development lifecycle.

1) **Unit testing:**

A Unit is a smallest testable portion of system or application which can be compiled, liked, loaded, and executed. This kind of testing helps to test each module separately.

The aim is to test each part of the software by separating it. It checks that component are fulfilling functionalities or not. This kind of testing is performed by developers.

2) **Integration testing:**

Integration means combining. For Example, In this testing phase, different software modules are combined and tested as a group to make sure that integrated system is ready for system testing.

Integrating testing checks the data flow from one module to other modules. This kind of testing is performed by testers.

3) **System testing:**

System testing is performed on a complete, integrated system. It allows checking system's compliance as per the requirements. It tests the overall interaction of components. It involves load, performance, reliability and security testing.

System testing most often the final test to verify that the system meets the specification. It evaluates both functional and non-functional need for the testing.

4) **Acceptance testing:**

Acceptance testing is a test conducted to find if the requirements of a specification or contract are met as per its delivery. Acceptance testing is basically done by the user or customer. However, other stockholders can be involved in this process.

Conclusion:

- A level of software testing is a process where every unit or component of a software/system is tested.
- The primary goal of system testing is to evaluate the system's compliance with the specified needs.
- In Software Engineering, four main levels of testing are Unit Testing, Integration Testing, System Testing and Acceptance Testing.

A diagrammatic variation of the waterfall model, known as the V-Model in ISTQB parlance, is given in Figure 1.8; this variation emphasizes the correspondence between testing and design levels.

A practical relationship exists between levels of testing versus specification-based and code based testing. Most practitioners agree that code-based testing is most appropriate at the unit level, whereas specification-based testing is most appropriate at the system level.

This is generally true; however, it is also a likely consequence of the base information produced during the requirements specification, preliminary design, and detailed design phases.

The constructs defined for code-based testing make the most sense at the unit level, and similar constructs are only now becoming available for the integration and system levels of testing.

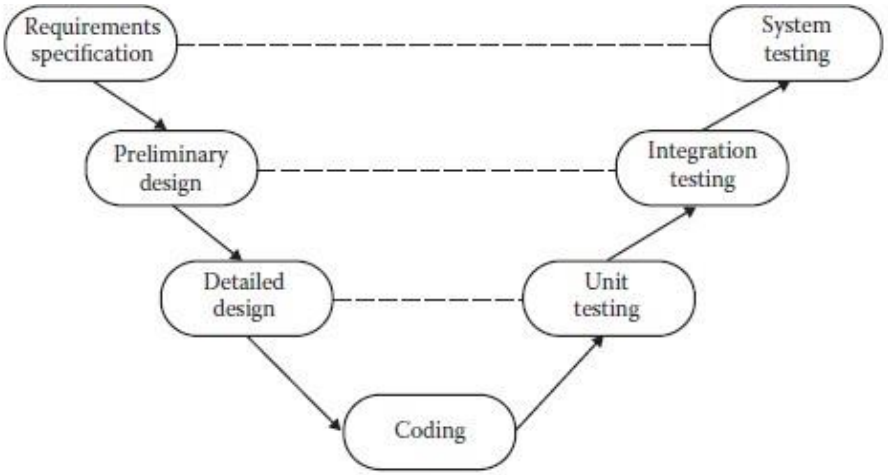


Figure 1.8 Levels of abstraction and testing in waterfall model.

Testing and Verification

What is Verification?

The verifying process includes checking documents, design, code, and program.

What is Validation?

Validation is a dynamic mechanism of Software testing and validates the actual product.

Verification vs Validation: Key Difference

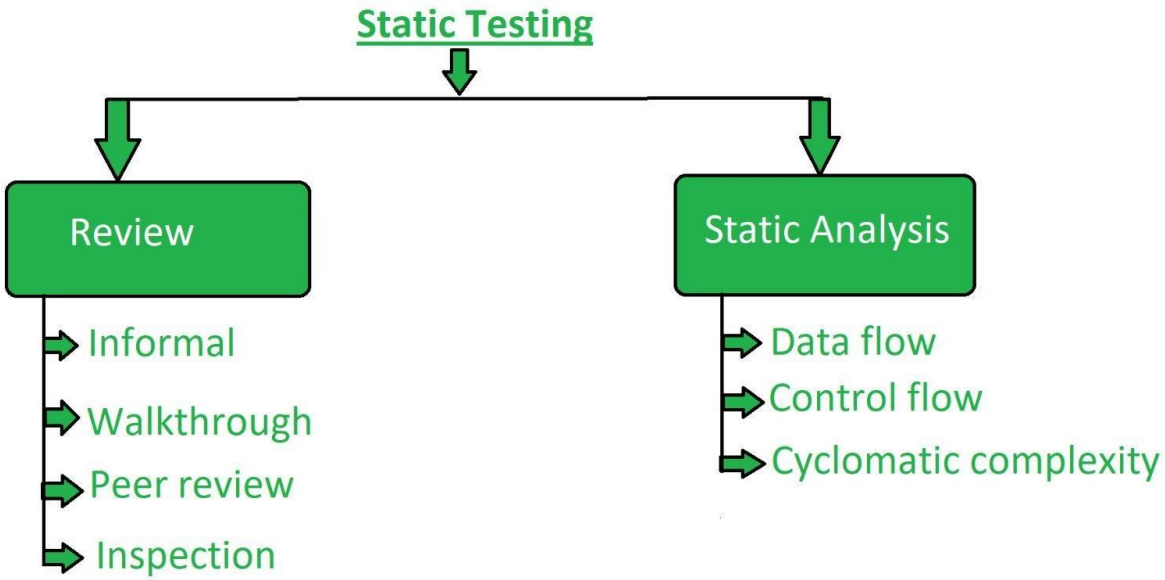
Verification	Validation
• The verifying process includes checking documents, design, code, and program	• It is a dynamic mechanism of testing and validating the actual product
• It does ***not*** involve executing the code	• It always involves executing the code
• Verification uses methods like reviews, walkthroughs, inspections, and desk- checking etc.	• It uses methods like Black Box Testing, White Box Testing, and non-functional testing

• Whether the software conforms to specification is checked	• It checks whether the software meets the requirements and expectations of a customer
• It finds bugs early in the development cycle	• It can find bugs that the verification process can not catch
• Target is application and software architecture, specification, complete design, high level, and database design etc.	• Target is an actual product
• QA team does verification and make sure that the software is as per the requirement in the SRS document.	• With the involvement of testing team validation is executed on software code.
• It comes before validation	• It comes after verification

Static Testing

Static Testing is a type of a Software Testing method which is performed to check the defects in software without actually executing the code of the software application. Whereas in Dynamic Testing checks the code is executed to detect the defects.
Static testing is performed in early stage of development to avoid errors as it is easier to find sources of failures and it can be fixed easily. The errors that can‘t not be found using Dynamic Testing, can be easily found by Static Testing.

Static Testing Techniques:
There are mainly two type techniques used in Static Testing:



1. **Review:**

In static testing review is a process or technique that is performed to find the potential defects in the design of the software.

It is process to detect and remove errors and defects in thedifferent supporting documents like software requirements specifications.

People examine the documents and sorted out errors, redundancies and ambiguities.

Review is of four types:

- **Informal:**
  In informal review the creator of the documents put the contents in front of audience and everyone gives their opinion and thus defects are identified in the early stage.
- **Walkthrough:**
  It is basically performed by experienced person or expert to check the defects so that there might not be problem further in the development or testing phase.
- **Peer review:**
  Peer review means checking documents of one-another to detect and fix the defects. It is basically done in a team of colleagues.
- **Inspection:**
  Inspection is basically the verification of document the higher authority like the verification of software requirement specifications (SRS).

2. **Static Analysis:**

Static Analysis includes the evaluation of the code quality that is written by developers. Different tools are used to do the analysis of the code and comparison of the same with the standard.

It also helps in following identification of following defects:

(a) Unused variables
(b) Dead code
(c) Infinite loops
(d) Variable with undefined value
(e) Wrong syntax

Static Analysis is of three types:

- **Data Flow:**

  Data flow is related to the stream processing

- **Control Flow:**

  Control flow is basically how the statements or instructions are executed.

- **Cyclomatic Complexity:**

  Cyclomatic complexity is the measurement of the complexity of the program that is basically related to the number of independent paths in the control flow graph of the program.

Problem Statements:

A **PROBLEM STATEMENT** is a concise description of an issue to be addressed or a condition to be improved upon. It identifies the gap between the current (problem) state and desired (goal) state of a process or product. Focusing on the facts, the problem statement should be designed to address the Five Ws.

The first condition of solving a problem is to understand the problem, which can be done by way of a problem statement.

Problem statements are widely used by businesses and organizations to execute process improvement projects. A simple and well-defined problem statement will be used by the project team to understand the problem and work toward developing a solution.

It will also provide management with specific insights into the problem so that they can make appropriate project-approving decisions. As such, it is crucial for the problem statement to be clear and unambiguous.

The **Five Ws** are questions whose answers are considered basic in information gathering or problem solving.

- Who
- What
- When
- Where
- Why

Generalized pseudocode,

Pseudocode is an informal high-level description of the operating principle of a computer program or other algorithm. It uses the structural conventions of a normal programming language, but is intended for human reading rather than machine reading. Pseudocode typically omits details that are essential for machine understanding of the algorithm, such as variable declarations, system-specific code and some subroutines. The programming language is augmented with natural language description details, where convenient, or with compact mathematical notation. The purpose of using pseudocode is that it is easier for people to understand than conventional programming language code, and that it is an efficient and environment-independent description of the key principles of an algorithm.

- Pseudocode provides a *"language neutral"* way to express program source code.
- Pseudocode given here is based on visual basic.

Table 2.1 Generalized Pseudocode

Language Element	*Generalized Pseudocode Construct*
Comment	' <text>
Data structure declaration	Type <type name> <list of field descriptions> End <type name>
Data declaration	Dim <variable> As <type>
Assignment statement	<variable> = <expression>
Input	Input (<variable list>)
Output	Output (<variable list>)
Condition	<expression> <relational operator> <expression>
Compound condition	<Condition> <logical connective> <Condition>
Sequence	statements in sequential order

Simple selection	If <condition> Then
	  <then clause>
	EndIf

Selection	If <condition>
	  Then <then clause>
	  Else <else clause>
	EndIf

Multiple selection	Case <variable> Of Case 1: <predicate> <Case clause> ... Case *n*: <predicate> <Case clause> EndCase
Counter-controlled repetition	For <counter> = <start> To <end> <loop body> EndFor
Pretest repetition	While <condition> <loop body> EndWhile

Language Element	*Generalized Pseudocode Construct*
Posttest repetition	Do <loop body> Until <condition>
Procedure definition (similarly for functions and o–o methods)	<procedure name> (Input: <list of variables>; Output: <list of variables>) <body> End <procedure name>
Interunit communication	Call <procedure name> (<list of variables>; <list of variables>)
Class/Object definition	<name> (<attribute list>; <method list>, <body>) End <name>
Interunit communication	msg <destination object name>.<method name> (<list of variables>)
Object creation	Instantiate <class name>.<object name> (list of attribute values)
Object destruction	Delete <class name>.<object name>
Program	Program <program name> <unit list> End<program name>

The triangle problem

The triangle problem is the most widely used example in software testing literature.

2.2.1 Problem Statement

Simple version: The triangle program accepts three integers, a, b, and c, as input. These are taken to be sides of a triangle. The output of the program is the type of triangle determined by the three sides: Equilateral, Isosceles, Scalene, or NotATriangle. Sometimes, this problem is extended to include right triangles as a fifth type; we will use this extension in some of the exercises.

Improved version: The triangle program accepts three integers, a, b, and c, as input. These are taken to be sides of a triangle. The integers a, b, and c must satisfy the following conditions:

c1. $1 \leq a \leq 200$ c4. $a < b + c$
c2. $1 \leq b \leq 200$ c5. $b < a + c$
c3. $1 \leq c \leq 200$ c6. $c < a + b$

The output of the program is the type of triangle determined by the three sides: Equilateral,

Isosceles, Scalene, or NotATriangle. If an input value fails any of conditions c1, c2, or c3, the program notes this with an output message, for example, ―Value of b is not in the range of permitted values.‖ If values of a, b, and c satisfy conditions c4, c5, and c6, one of four mutually exclusive outputs is given:

1. If all three sides are equal, the program output is Equilateral.
2. If exactly one pair of sides is equal, the program output is Isosceles.
3. If no pair of sides is equal, the program output is Scalene.
4. If any of conditions c4, c5, and c6 is not met, the program output is NotATriangle.

2.2.2 Discussion

Perhaps one of the reasons for the longevity of this example is that it contains clear but complex logic. It also typifies some of the incomplete definitions that impair communication among customers, developers, and testers. The first specification presumes the developers know some details about triangles, particularly the triangle inequality: the sum of any pair of sides must be strictly greater than the third side.

2.2.3 Traditional Implementation

The traditional implementation of this grandfather of all examples has a rather FORTRAN-like style. The flowchart for this implementation appears in Figure 2.1. Figure 2.2 is a flowchart for the improved version. The flowchart box numbers correspond to comment numbers in the (FORTRANlike) pseudocode program given next.

The variable ―match‖ is used to record equality among pairs of the sides. A classic intricacy of the FORTRAN style is connected with the variable ―match‖: notice that all three tests for the triangle inequality do not occur. If two sides are equal, say a and c, it is only necessary to compare a + c with b. (Because b must be greater than zero, a + b must be greater than c because c equals a.) This observation clearly reduces the number of comparisons that must be made. The efficiency of this version is obtained at the expense of clarity (and ease of testing).

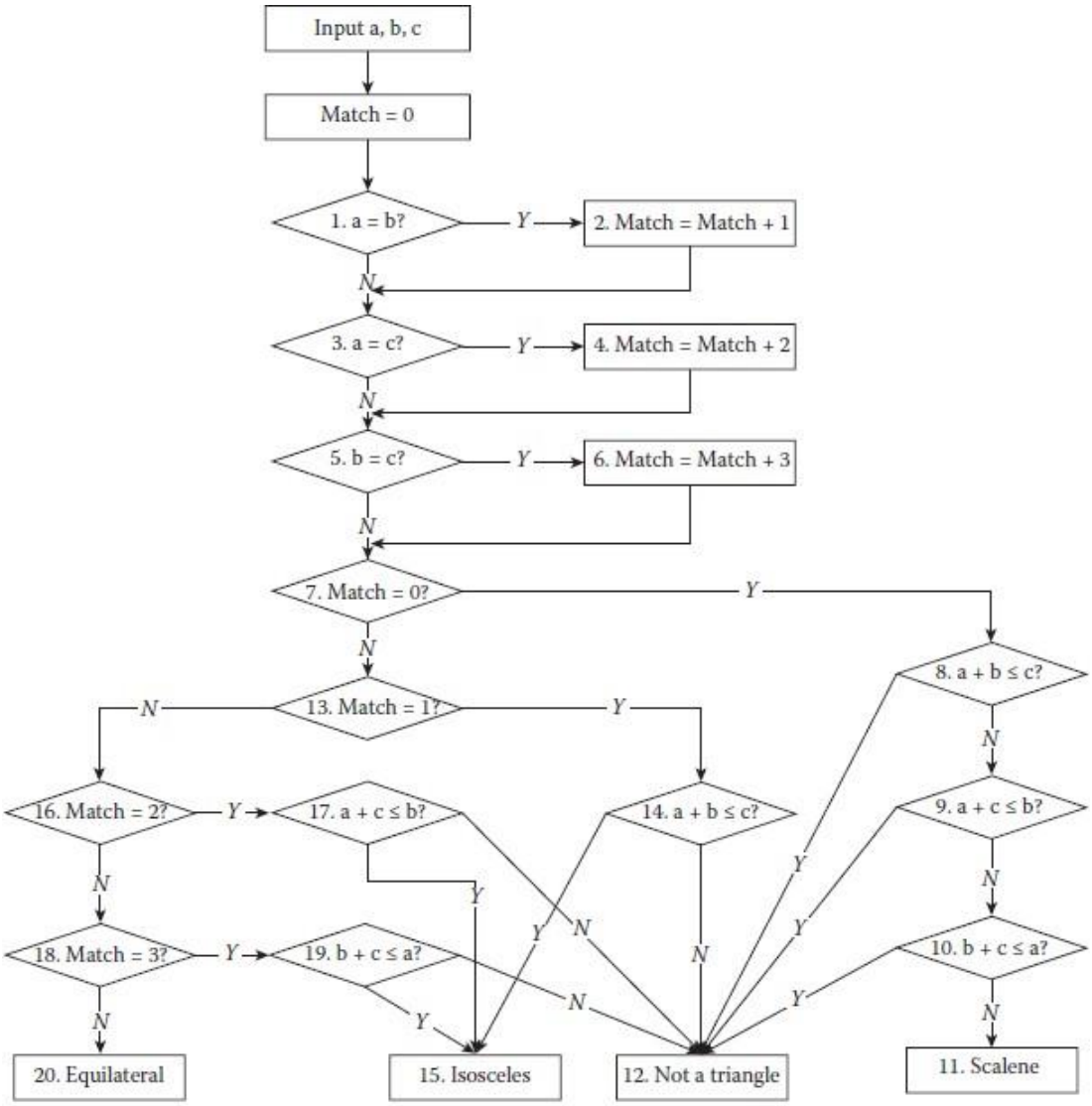


Figure 2.1 Flowchart for traditional triangle program implementation.

The pseudocode for this is given next. **[Simple version]**

Program triangle1

Dim a, b, c, match As INTEGER

Output(—Enter 3 integers which are sides of a triangle‖)

Input(a, b, c)

Output(—Side A is‖,a)
Output(—Side B is‖,b)
Output(—Side C is‖,c)

match = 0
If a = b _(1)

```
Then match = match + 1                    _(2)
EndIf

If a = c                                  _(3)
Then match = match + 2                    _(4)
EndIf

If b = c                                  _(5)
Then match = match + 3                    _(6)
EndIf

If match = 0                              _(7)
Then If (a + b)≤ c                        _(8)
Then Output(―NotATriangle‖)                          _(12.1)
Else If (b + c) ≤ a                       _(9)
Then Output(―NotATriangle‖)                           _(12.2)
Else If (a + c) ≤ b                       _(10)
Then Output(―NotATriangle‖)                          _(12.3)
Else Output (–Scalene‖)                   _(11)

EndIf
EndIf
EndIf

Else If match = 1                         _(13)
Then If (a + c) ≤ b                       _(14)
Then Output(―NotATriangle‖)                          _(12.4)
Else Output (–Isosceles‖)                            _(15.1)
EndIf

Else If match=2                           _(16)
Then If (a + c) ≤ b
Then Output(―NotATriangle‖)                          (12.5)
Else Output (–Isosceles‖)                            _(15.2)
EndIf

Else If match = 3                         _(18)
Then If (b + c) ≤ a                       _(19)
Then Output(―NotATriangle‖)                          _(12.6)
Else Output (–Isosceles‖)                            _(15.3)
EndIf

Else Output (–Equilateral‖)               _(20)
EndIf
EndIf
EndIf
EndIf

End Triangle1
```

2.2.4 Structured Implementations

The pseudocode for **[Improved Version]**

```
Step 1: Get Input
Output(―Enter 3 integers which are sides of a triangle‖)
Input(a,b,c)
Output(―Side A is‖,a)
Output(―Side B is‖,b)
Output(―Side C is‖,c)

Step 2: Is A Triangle?‘
If (a < b + c) AND (b < a + c) AND (c < a + b)
Then IsATriangle = True
Else IsATriangle = False
EndIf

Step 3: Determine Triangle Type
If IsATriangle
Then If (a = b) AND (b = c)
Then Output (―Equilateral‖)
Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
Then Output (–Scalene‖)
Else Output (–Isosceles‖)
EndIf
EndIf
Else Output(―Not a Triangle‖)
EndIf
End triangle2
```

The pseudocode for **[Final Version]**

Third version

```
Program triangle3

Dim a, b, c As Integer
Dim c1, c2, c3, IsATriangle As Boolean
Step 1: Get Input
Do
Output(―Enter 3 integers which are sides of a triangle‖)
Input(a, b, c)
c1 = (1 ≤ a) AND (a ≤ 300)
c2 = (1 ≤ b) AND (b ≤ 300)
c3 = (1 ≤ c) AND (c ≤ 300)

If NOT(c1)
Then Output(―Value of a is not in the range of permitted values‖)
EndIf
If NOT(c2)
Then Output(―Value of b is not in the range of permitted values‖)
EndIf
If NOT(c3)
```

```
ThenOutput(―Value of c is not in the range of permitted values‖)
EndIf
Until c1 AND c2 AND c3
Output(―Side A is‖,a)
Output(―Side B is‖,b)
Output(―Side C is‖,c)
_Step 2: Is A Triangle?
If (a < b + c) AND (b < a + c) AND (c < a + b)
Then IsATriangle = True
Else IsATriangle = False
```

2.3 T he NextDate Function

The complexity in the triangle program is due to the relationships between inputs and correct outputs. We will use the NextDate function to illustrate a different kind of complexity—logical relationships among the input variables.

2.3.1 Problem Statement

NextDate is a function of three variables: month, date, and year. It returns the date of the day after the input date. The month, date, and year variables have integer values subject to these conditions

(the year range ending in 2012 is arbitrary, and is from the first edition):
c1. $1 \leq \text{month} \leq 12$
c2. $1 \leq \text{day} \leq 31$
c3. $1812 \leq \text{year} \leq 2012$

As we did with the triangle program, we can make our problem statement more specific. This entails defining responses for invalid values of the input values for the day, month, and year.

We can also define responses for invalid combinations of inputs, such as June 31 of any year.

If any of conditions c1, c2, or c3 fails, NextDate produces an output indicating the corresponding variable has an out-of-range value—for example, ―Value of month not in the range 1...12.‖ Because numerous invalid day–month–year combinations exist, NextDate collapses these into one message: ―Invalid Input Date.‖

2.3.2 Discussion

Two sources of complexity exist in the NextDate function: the complexity of the input domain discussed previously, and the rule that determines when a year is a leap year. A year is 365.2422 days long; therefore, leap years are used for the ―extra day‖ problem. If we declared a leap year every fourth year, a slight error would occur.

The Gregorian calendar (after Pope Gregory) resolves this by adjusting leap years on century years. Thus, a year is a leap year if it is divisible by 4, unless it is a century year. Century years are leap years only if they are multiples of 400 (Inglis, 1961); thus, 1992, 1996, and 2000 are leap years, while the year 1900 is not a leap year. The NextDate function also illustrates a sidelight of software testing.

2.3.3 Implementations

Program NextDate1 _Simple version

```
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
Output (―Enter today‘s date in the form MM DD YYYY‖)

Input (month, day, year)
Case month Of
Case 1: month Is 1,3,5,7,8, Or 10: _31 day months (except Dec.)
If day < 31
Then tomorrowDay = day + 1
Else
tomorrowDay = 1
tomorrowMonth = month + 1
EndIf


Case 2: month Is 4,6,9, Or 11 _30 day months
If day < 30
Then tomorrowDay = day + 1
Else
tomorrowDay = 1
tomorrowMonth = month + 1
EndIf


Case 3: month Is 12: _December
If day < 31
Then tomorrowDay = day + 1
Else
tomorrowDay = 1
tomorrowMonth = 1
If year = 2012
Then Output (―2012 is over‖)
Else tomorrow.year = year + 1
EndIf


Case 4: month is 2: _February
If day < 28
Then tomorrowDay = day + 1
Else
If day = 28
Then If ((year is a leap year)
Then tomorrowDay = 29 _leap year
Else _not a leap year
tomorrowDay = 1
tomorrowMonth = 3
EndIf
```

```
Else If day = 29
Then If ((year is a leap year)
Then tomorrowDay = 1
tomorrowMonth = 3
Else _not a leap year
Output(—Cannot have Feb.‖, day)

EndIf
EndIf
EndIf
EndIf
EndCase


Output (—Tomorrow‘s date is‖, tomorrowMonth, tomorrowDay, tomorrowYear)
End NextDate
```

Program NextDate2 Improved version

```
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
Dim c1, c2, c3 As Boolean

Do
Output (—Enter today‘s date in the form MM DD YYYY‖)

Input (month, day, year)
c1 = (1 ≤ day) AND (day ≤ 31)
c2 = (1 ≤ month) AND (month ≤ 12)
c3 = (1812 ≤ year) AND (year ≤ 2012)

If NOT(c1)
Then Output(—Value of day not in the range 1..31‖)
EndIf

If NOT(c2)
Then Output(—Value of month not in the range 1..12‖)
EndIf

If NOT(c3)
Then Output(—Value of year not in the range 1812..2012‖)
EndIf

Until c1 AND c2 AND c2

Case month Of
Case 1: month Is 1,3,5,7,8, Or 10: _31 day months (except Dec.)
If day < 31
Then tomorrowDay = day + 1
```

```
Else
tomorrowDay = 1
tomorrowMonth = month + 1
EndIf


Case 2: month Is 4,6,9, Or 11 _30 day months
If day < 30
Then tomorrowDay = day + 1
Else
If day = 30
Then tomorrowDay = 1
tomorrowMonth = month + 1
Else Output(―Invalid Input Date‖)
EndIf
EndIf


Case 3: month Is 12: _December
If day < 31
Then tomorrowDay = day + 1
Else
tomorrowDay = 1
tomorrowMonth = 1
If year = 2012
Then Output (―Invalid Input Date‖)
Else tomorrow.year = year + 1
EndIf
EndIf


Case 4: month is 2: _February
If day < 28
Then tomorrowDay = day + 1

Else

If day = 28
Then
If (year is a leap year)
Then tomorrowDay = 29 _leap day
Else _not a leap year
tomorrowDay = 1
tomorrowMonth = 3
EndIf

Else

If day = 29
Then
```

```
If (year is a leap year)
Then tomorrowDay = 1
tomorrowMonth = 3

Else

If day > 29
Then Output(―Invalid Input Date‖)
EndIf
EndIf
EndIf
EndIf
EndIf
EndCase
Output (―Tomorrow‘s date is‖, tomorrowMonth, tomorrowDay, tomorrowYear)

End NextDate2
```

2.4 T he Commission Problem

Our third example is more typical of commercial computing. It contains a mix of computation and decision making, so it leads to interesting testing questions. Our main use of this example will be in our discussion of data flow and slice-based testing.

2.4.1 Problem Statement

A rifle salesperson in the former Arizona Territory sold rifle locks, stocks, and barrels made by a gunsmith in Missouri. Locks cost $45, stocks cost $30, and barrels cost $25. The salesperson had to sell at least one lock, one stock, and one barrel (but not necessarily one complete rifle) per month, and production limits were such that the most the salesperson could sell in a month was 70 locks, 80 stocks, and 90 barrels. After each town visit, the salesperson sent a telegram to the Missouri gunsmith with the number of locks, stocks, and barrels sold in that town.

At the end of a month, the salesperson sent a very short telegram showing –1 lock sold. The gunsmith then knew the sales for the month were complete and computed the salesperson‘s commission as follows: 10% on sales up to (and including) $1000, 15% on the next $800, and 20% on any sales in excess of $1800.

2.4.2 Discussion

This example is somewhat contrived to make the arithmetic quickly visible to the reader. It might be more realistic to consider some other additive function of several variables, such as various calculations found in filling out a US 1040 income tax form. (We will stay with rifles.)

This problem separates into three distinct pieces: the input data portion, in which we could deal with input data validation (as we did for the triangle and NextDate programs), the sales calculation, and the commission calculation portion. This time, we will omit the input data validation portion. We will replicate the telegram convention with a sentinel-controlled while loop that is typical of MIS data gathering applications.

2.4.3 Implementation

```
Program Commission (INPUT,OUTPUT)

Dim locks, stocks, barrels As Integer
Dim lockPrice, stockPrice, barrelPrice As Real
Dim totalLocks,totalStocks,totalBarrels As Integer
Dim lockSales, stockSales, barrelSales As Real
Dim sales,commission : REAL

lockPrice = 45.0
stockPrice = 30.0
barrelPrice = 25.0
totalLocks = 0
totalStocks = 0
totalBarrels = 0

Input(locks)
While NOT(locks = -1) _Input device uses -1 to indicate end of data
Input(stocks, barrels)
totalLocks = totalLocks + locks
totalStocks = totalStocks + stocks
totalBarrels = totalBarrels + barrels
Input(locks)
EndWhile

Output(—Locks sold:‖, totalLocks)
Output(—Stocks sold:‖, totalStocks)
Output(—Barrels sold:‖, totalBarrels)

lockSales = lockPrice * totalLocks
stockSales = stockPrice * totalStocks
barrelSales = barrelPrice * totalBarrels
sales = lockSales + stockSales + barrelSales
Output(—Total sales:‖, sales)

If (sales > 1800.0)
Then
commission = 0.10 * 1000.0
commission = commission + 0.15 * 800.0
commission = commission + 0.20 * (sales–1800.0)
Else If (sales > 1000.0)
Then
commission = 0.10 * 1000.0
commission = commission + 0.15*(sales–1000.0)
Else commission = 0.10 * sales
EndIf
EndIf
Output(—Commission is $‖,commission)
End Commission
```

2.5 T he SATM System

To better discuss the issues of integration and system testing, we need an example with larger scope (Figure 2.3).

The ATM described here is minimal, yet it contains an interesting variety of functionality and interactions that typify the client side of client–server systems.

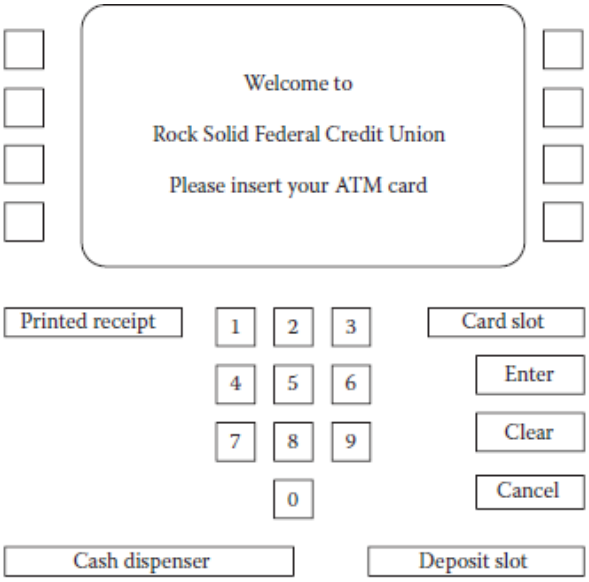


Figure 2.3 SATM terminal.

2.5.1 Problem Statement

The SATM system communicates with bank customers via the 15 screens shown in Figure 2.4.

Using a terminal with features as shown in Figure 2.3, SATM customers can select any of three transaction types: deposits, withdrawals, and balance inquiries. For simplicity, these transactions can only be done on a checking account.

When a bank customer arrives at an SATM station, screen 1 is displayed. The bank customer accesses the SATM system with a plastic card encoded with a personal account number (PAN), which is a key to an internal customer account file, containing, among other things, the customer‘s name and account information.

If the customer‘s PAN matches the information in the customer account file, the system presents screen 2 to the customer. If the customer‘s PAN is not found, screen 4 is displayed, and the card is kept.

At screen 2, the customer is prompted to enter his or her personal identification number (PIN). If the PIN is correct (i.e., matches the information in the customer account file), the system displays screen 5; otherwise, screen 3 is displayed.

The customer has three chances to get the PIN correct; after three failures, screen 4 is displayed, and the card is kept.

On entry to screen 5, the customer selects the desired transaction from the options shown on screen. If balance is requested, screen 14 is then displayed. If a deposit is requested, the status of the deposit envelope slot is determined from a field in the terminal control file. If no problem is known, the system displays screen 7 to get the transaction amount. If a problem occurs with the deposit envelope slot,

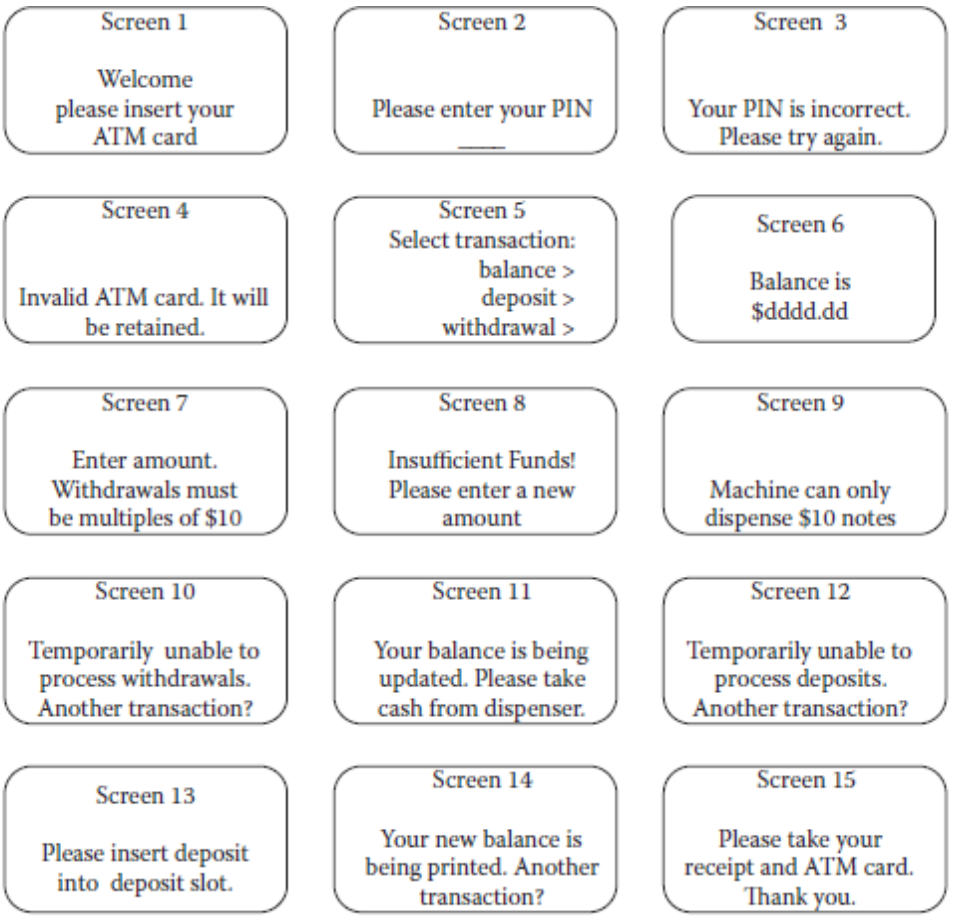


Figure 2.4 SATM screens.

The system displays screen 12. Once the deposit amount has been entered, the system displays screen 13, accepts the deposit envelope, and processes the deposit. The system then displays screen 14.

If a withdrawal is requested, the system checks the status (jammed or free) of the withdrawal chute in the terminal control file. If jammed, screen 10 is displayed; otherwise, screen 7 is displayed so the customer can enter the withdrawal amount. Once the withdrawal amount is entered, the system checks the terminal status file to see if it has enough currency to dispense.

If it does not, screen 9 is displayed; otherwise, the withdrawal is processed. The system checks the customer balance (as described in the balance request transaction); if the funds in the account are insufficient, screen 8 is displayed. If the account balance is sufficient, screen 11 is displayed and the money is dispensed. The balance is printed on the transaction receipt as it is for a balance request transaction.

After the cash has been removed, the system displays screen 14.

When the –No‖ button is pressed in screens 10, 12, or 14, the system presents screen 15 and returns the customer‘s ATM card. Once the card is removed from the card slot, screen 1 is displayed.

When the –Yes‖ button is pressed in screens 10, 12, or 14, the system presents screen 5 so the customer can select additional transactions.

2.6 T he Currency Converter

The currency conversion program is another event-driven program that emphasizes code associated with a GUI. A sample GUI is shown in Figure 2.5.

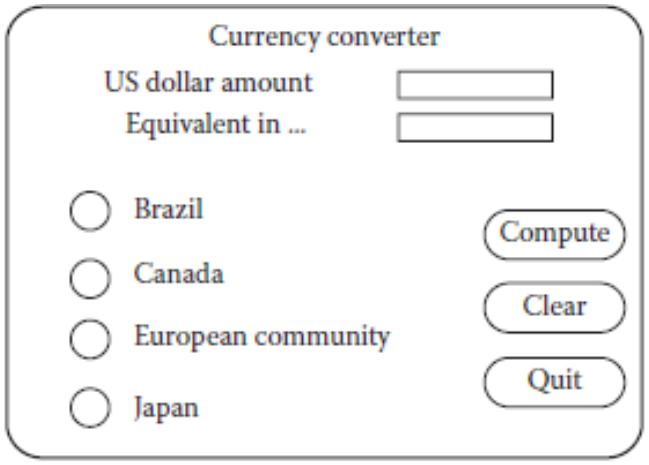


Figure 2.5 Currency converter graphical user interface.

Problem Statement

The currency converter application converts U.S. dollars to any of four currencies: Brazilian reals, Canadian dollars, European Community euros, and Japanese yen. The user can revise inputs and perform repeated currency conversion.

System Functions

In the first step, sometimes called project inception, the customer/user describes the application in very general terms. This might take the form of –user stories,‖ which are precursors to use cases.

From these, three types of system functions are identified: evident, hidden, and frill. Evident functions are the obvious ones. Hidden functions might not be discovered immediately, and frills are the –bells and whistles‖ that so often occur. Table 15.3 lists the system functions for the currency converter application.

Table 15.3 System Functions for Currency Converter Application

Reference No.	*Function*	*Category*
R1	Start application	Evident
R2	End application	Evident
R3	Input US dollar amount	Evident
R4	Select country	Evident
R5	Perform conversion calculation	Evident
R6	Clear user inputs and program outputs	Evident
R7	Maintain exclusive–or relationship among countries	Hidden
R8	Display country flag images	Frill

Presentation Layer

Pictures are still worth a thousand words. The third step in Larman‘s approach is to sketch the user interface; our version is in Figure 2.5. This much information can support a customer walkthrough to demonstrate that the system functions identified can be supported by the interface.

High-Level Use Cases

The use case development begins with a very high-level view. Notice, as the succeeding levels of use cases are elaborated, much of the early information is retained.

It is convenient to have a short, structured naming convention for the various levels of use cases. Here, for example, HLUC refers to high-level use case (where would we be without acronyms?).

Very few details are provided in a high-level use case; they are insufficient for test case identification.

The main point of high-level use cases is that they capture a narrative description of something that happens in the system to be built.

HLUC 1	Start application.
Description	The user starts the currency conversion application in Windows®.

HLUC 2	End application.
Description	The user ends the currency conversion application in Windows.

HLUC 3	Convert dollars.
Description	The user inputs a US dollar amount and selects a country; the application computes and displays the equivalent in the currency of the selected country.

HLUC 4	Revise inputs.
Description	The user resets inputs to begin a new transaction.

HLUC 5	Repeated conversions, same dollar amount.
Description	The user inputs a US dollar amount and selects a country; the application computes and displays the equivalent in the currency of the selected country.

HLUC 6	Revise inputs.
Description	A US dollar amount has been entered OR a country has been selected.

HLUC 7	Abnormal case: no country selected.
Description	User enters a dollar amount and clicks on the Compute button without selecting a country.

HLUC 8	Abnormal case: no dollar amount entered.
Description	User selects a country and clicks on the Compute button without entering a dollar amount.

HLUC 9	Abnormal case: no dollar amount entered and no country selected.
Description	User clicks on the Compute button without entering a dollar amount and without selecting a country.

Essential Use Cases

Essential use cases add ‒actor‖ and ‒system‖ events to a high-level use case. Actors in UML are sources of system-level inputs (i.e., port input events). Actors can be people, devices, adjacent systems, or abstractions such as time. Since the only actor is the User, that part of an essential use case is omitted. The numbering of actor actions (port input events) and system responses (port output events) shows their approximate sequences in time.

EUC-1	Start application
Description	The user starts the currency conversion application in Windows.
Event Sequence	
Input Events	Output Events
1. The user starts the application, either with a Run ... command or by double clicking the application icon.	
	2. The currency conversion application GUI appears on the monitor and is ready for user input.

EUC-3	Convert dollars
Description	The user inputs a US dollar amount and selects a country; the application computes and displays the equivalent in the currency of the selected country.
Event Sequence	
Input Events	Output Events
1. The user enters a dollar amount.	
	2. The dollar amount is displayed on the GUI.
3. The user selects a country.	
	4. The name of the country's currency is displayed.
	5. The flag of the country is displayed.
6. The user requests a conversion calculation.	
	7. The equivalent currency amount is displayed.

EUC-7	Abnormal case: no country selected
Description	The user enters a dollar amount and clicks on the cmdCompute button without selecting a country.
Event Sequence	
Input Events	Output Events
1. The user enters a dollar amount.	2. The dollar amount is displayed on the GUI.
3. User clicks the Compute button.	4. A message box appears with the caption "must select a country."
5. The user closes the message box.	6. The message box is no longer visible.
	7. The flag of the country is no longer visible.

C PROGRAM FOR CONVERTING US DOLLARS TO FRANC, POUNDS, YEN, EUROS AND CANADIAN DOLLAR

```
#include<stdio.h>
#include<stdlib.h>
    #define Swiss_Franc_rate 0.6072;                /*Swiss Franc rate*/
    #define British_Pounds_rate 1.4320;             /*British Pound rate*/
    #define Japanese_Yen_rate 0.0081;               /*Japanese Yen rate*/
    #define Canadian_Dollar_rate 0.6556;            /*Canadian Dollar rate*/
    #define Euros_rate 0.8923;


int main(void){

/*Declare floaters*/

float Swiss_Franc;                          /*Swiss Franc*/
float British_Pounds;                       /*British pounds*/
float Japanese_Yen;                         /*Japanese Yen*/
float Canadian_Dollar;                      /*Canadian Dollar*/
float Euros;                                /*European Union Euro*/
float USD;                                   /*US Dollar*/
int   choice;

                                            /*Title*/

printf("       Currency Conversion Program\n");

printf("--------------------------\n\n");


                                            /*Menu*/

    printf("1)   Swiss Franc              \n");
```

```
    printf("2)   British Pound          \n");
    printf("3)   Japanese Yen           \n");
    printf("4)   Canadian Dollar        \n");
    printf("5)   Euro                 \n");
    printf("6)   Exit the Program       \n");


printf("\n");
printf("\n");

                                        /*Input from User*/

printf("Please enter your choice (1-6): ");
scanf("%d",&choice);

while((choice<1) || (choice>6)){
printf("Invalid entry, please Enter 1-6: ");
scanf("%i",&choice);
}

if(choice==1){
printf("Please the amount:  ");
scanf("%f",&Swiss_Franc);

                                        /*Conversion Calculation 1*/
Swiss_Franc = USD / Swiss_Franc_rate;

}

if(choice==2){
printf("Please enter the amount:  ");
scanf("%f",&British_Pounds);

    /*Conversion Calculation 2*/
British_Pounds = USD / British_Pounds_rate;
}


if(choice==3){
printf("Please enter the amount:  ");
scanf("%f",&Japanese_Yen);

    /*Conversion Calculation 3*/
Japanese_Yen = USD / Japanese_Yen_rate
}


if(choice==4){
printf("Please enter the amount:  ");
scanf("%f",&Canadian_Dollar);
```

```
    /*Conversion Calculation 4*/
Canadian_Dollar = USD / Canadian_Dollar_rate;
}

if(choice==5){
printf("Please enter the amount:  ");
scanf("%f",&Euros);

    /*Conversion Calculation 5*/
Euros = USD / Euros_rate;
}

if(choice==6){
printf("Exit the program:  ");

while (getchar() != '\n')
     continue;
   goto top;
 }
 printf("Goodbye!\n");
 return 0;

}
```

Saturn Windshield Wiper Controller

The windshield wiper on some Saturn automobiles is controlled by a lever with a dial. The lever has four positions: OFF, INT (for intermittent occurring at irregular intervals; not continuous or steady), LOW, and HIGH;

The dial has three positions, numbered simply 1, 2, and 3.

The dial positions indicate three intermittent speeds.

The dial position is relevant only when the lever is at the INT position.

The decision table below shows the windshield wiper speeds (in wipes per minute) for the lever and dial positions.

c1. Lever	OFF	INT	INT	INT	LOW	HIGH
c2. Dial	n/a	1	2	3	n/a	n/a
a1. Wiper	0	4	6	12	30	60

Garage Door Controller

A system to open a garage door is composed of several components: a drive motor, a drive chain, the garage door wheel tracks, a lamp, and an electronic controller. This much of the system is powered by commercial 110 V electricity. Several devices communicate with the garage door controller—a wireless keypad (usually in an automobile), a digit keypad on the outside of the garage door, and a wall-mounted button.

In addition, there are two safety features, a laser beam near the floor and an obstacle sensor. These latter two devices operate only when the garage door is closing. If the light beam is interrupted (possibly by a pet), the door immediately stops, and then reverses direction until the door is fully open. If the door encounters an obstacle while it is closing (say a child‘s tricycle left in the path of the door), the door stops and reverses direction until it is fully open.

There is a third way to stop a door in motion, either when it is closing or opening. A signal from any of the three devices (wireless keypad, digit keypad, or wall-mounted control button).

The response to any of these signals is different—the door stops in place. A subsequent signal from any of the devices starts the door in the same direction as when it was stopped. Finally, there are sensors that detect when the door has moved to one of the extreme positions, either fully open or fully closed. When the door is in motion, the lamp is lit, and remains lit for approximately 30 seconds after the door reaches one of the extreme positions.

The three signaling devices and the safety features are optional additions to the basic garage door opener. This example will be used in Chapter 17 in the discussion of systems of systems. For now, a SysML context diagram of the garage door opener is given in Figure 2.6.

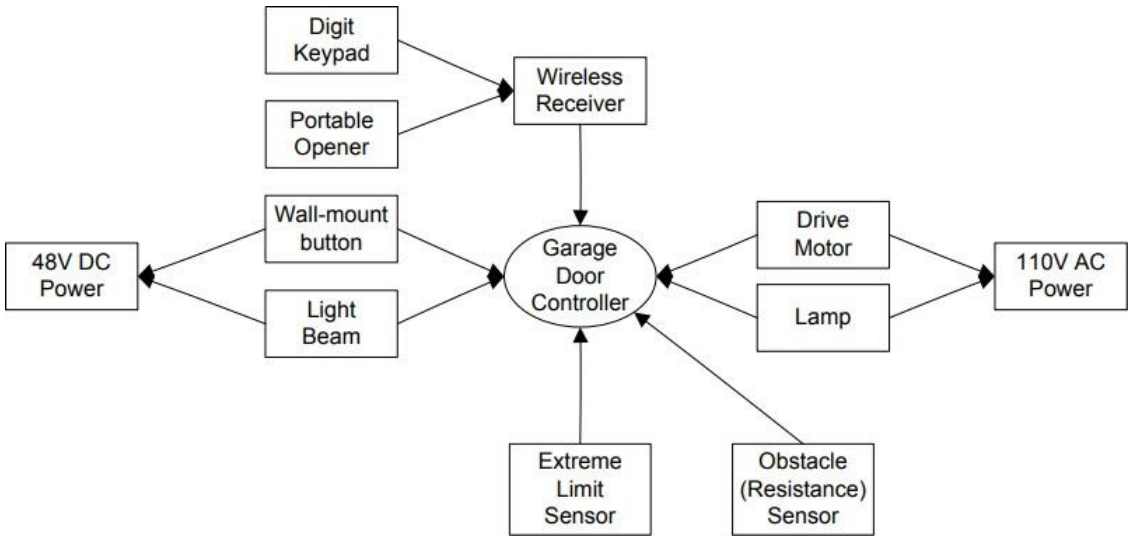


Figure 2.6 **SysML diagram of garage door controller.**

MODULE 2

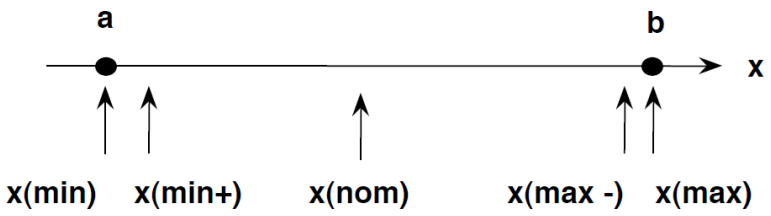
Functional Testing: Boundary value analysis, Robustness testing, Worst-case testing, Robust Worst testing for triangle problem, Nextdate problem and commission problem, Equivalence classes, Equivalence test cases for the triangle problem, NextDate function, and the commission problem, Guidelines and observations, Decision tables, Test cases for the triangle problem, NextDate function, and the commission problem, Guidelines and observations. Fault Based Testing: Overview, Assumptions in fault based testing, Mutation analysis, Fault-based adequacy criteria, Variations on mutation analysis.

Boundary value analysis

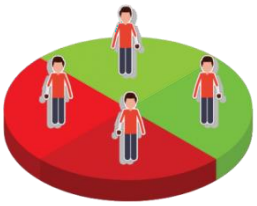
Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values.

- So these extreme ends like Start- End, Lower- Upper, Maximum-Minimum, Just Inside-Just Outside values are called boundary values and the testing is called "boundary testing".
- The basic idea in boundary value testing is to select input variable values at their:

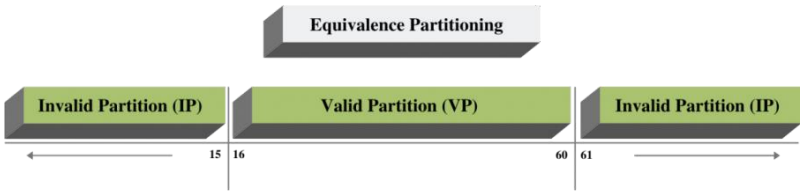
1. Minimum
2. Just above the minimum
3. A nominal value
4. Just below the maximum
5. Maximum



The first step of Boundary value analysis is to create Equivalence Partitioning, which would look like below.



Equivalence Partitioning



Now Concentrate on the Valid Partition, which ranges from 16-60. We have a 3 step approach to identify boundaries:

- Identify Exact Boundary Value of this partition Class – which is 16 and 60.
- Get the Boundary value which is one less than the exact Boundary – which is 15 and 59.
- Get the Boundary Value which is one more than the precise Boundary – which is 17 and 61.

If we combine them all, we will get below combinations for Boundary Value for the Age Criteria.

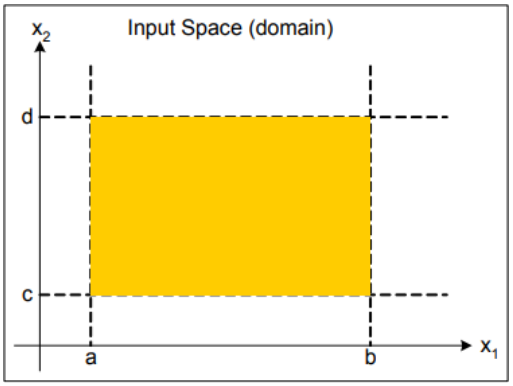
Valid Boundary Conditions : Age = 16, 17, 59, 60
Invalid Boundary Conditions : Age = 15, 61

It‟s straightforward to see that valid boundary conditions fall under Valid partition class, and invalid boundary conditions fall under Invalid partition class.

The Focus of BVA Boundary Value Analysis focuses on the input variables of the function. For the purposes of this report I will define two variables (I will only define two so that further examples can be kept concise) X1 and X2. Where X1 lies between A and B and X2 lies between C and D.

$A \leq X1 \leq B$
$C \leq X2 \leq D$

The values of A, B, C and D are the extremities of the input domain. These are best demonstrated by figure 4.1.

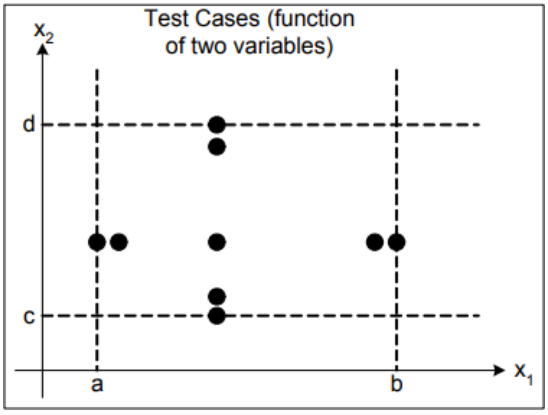


The Yellow shaded area of the graph shows the acceptable/legitimate input domain of the given function. As the name suggests Boundary Value Analysis focuses on the boundary of the input space to recognize test cases. The idea and motivation behind BVA is that errors tend to occur near the extremities of the input variables. The defects found on the boundaries of these input variables can obviously be the result of countless possibilities. Figure 4.1 4 But there are many common faults that result in errors more collated towards the boundaries of input variables. For example if the programmer forgot to count from zero or they just miscalculated. Errors in the code concerning loop counters being off by one or the use of a < operator instead of ≤. These are all very common mistakes and accompanied with other common errors we find an increasing need to perform Boundary Value Analysis.

5.0 Applying Boundary Value Analysis In the general application of Boundary Value Analysis can be done in a uniform manner. The basic form of implementation is to maintain all but one of the variables at their nominal (normal or average) values and allowing the remaining variable to take on its extreme values. The values used to test the extremities are:

- Min ..... Minimal
- Min+ ..... Just above Minimal
- Nom ..... Average
- Max- ..... Just below Maximum
- Max ..... Maximum

In continuing our example this results in the following test cases shown in figures 5.1 and 5.2:



{<x1nom, x2min>, <x1nom, x2min+ >,<x1nom, x2nom>,<x1nom, x2max- >, <x1nom, x2max>, <x1min, x2nom >, <x1min+, x2nom >, <x1nom, x2nom >, <x1max-, x2nom >, <x1max, x2nom > }

You maybe wondering why it is we are only concerned with one of the values taking on their extreme values at any one particular time. The reason for this is that generally Boundary Value Analysis uses the Critical Fault Assumption. There are advantages and shortcomings of this method.

5.1 Some Important examples To be able to demonstrate or explain the need for certain methods and their relative merits I will introduce two testing examples proposed by P.C. Jorgensen [1]. These examples will provide more extensive ranges to show where certain testing techniques are required and provide a better overview of the methods usability.

• The NextDate problem

The NextDate problem is a function of three variables: day, month and year. Upon the input of a certain date it returns the date of the day after that of the input. The input variables have the obvious conditions:

$1 \leq \text{Day} \leq 31.$
$1 \leq \text{month} \leq 12.$
$1812 \leq \text{Year} \leq 2012.$

(Here the year has been restricted so that test cases are not too large). There are more complicated issues to consider due to the dependencies between variables. For example there is never a 31st of April no matter what year we are in. The nature of these dependencies is the reason this example is so useful to us. All errors in the NextDate problem are denoted by "Invalid Input Date."

• The Triangle problem In fact the first introduction of the Triangle problem is in 1973, Gruenburger. There have been many more references to this problem since making this one of the most popular example to be used in conjunction with testing literature.

The triangle problem accepts three integers (a, b and c) as its input, each of which are taken to be sides of a triangle. The values of these inputs are used to determine the type of the triangle (Equilateral, Isosceles, Scalene or not a triangle).

For the inputs to be declared as being a triangle they must satisfy the six conditions:

C1. $1 \leq a \leq 200$.
C2. $1 \leq b \leq 200$.
C3. $1 \leq c \leq 200$.

C4. $a < b + c$.
C5. $b < a + c$.
C6. $c < a + b$.

Otherwise this is declared not to be a triangle. The type of the triangle, provided the conditions are met, is determined as follows:

1. If all three sides are equal, the output is Equilateral.
2. If exactly one pair of sides is equal, the output is Isosceles.
3. If no pair of sides is equal, the output is Scalene.

5.2 Critical Fault Assumption

The Critical Fault Assumption also known as the single fault assumption in reliability theory. The assumption relies on the statistic that failures are only rarely the product of two or more simultaneous faults. Upon using this assumption we can reduce the required calculations dramatically.

The amount of test cases for our example as you can recall was 9. Upon inspection we find that the function f that computes the number of test cases for a given number of variables n can be shown as:

$f = 4n + 1$

As there are four extreme values this accounts for the 4n. The addition of the constant one constitutes for the instance where all variables assume their nominal value.

5.3 Generalizing BVA

There are two approaches to generalizing Boundary Value Analysis. We can do this by the number of variables or by the ranges these variables use. To generalize by the number of variables is relatively simple. This is the approach taken as shown by the general Boundary Value Analysis technique using the critical fault assumption.

Generalizing by ranges depends on the type of the variables. For example in the NextDate example proposed by P.C. Jorgensen [1], we have variable for the year, month and day. Languages similar to the likes of FORTRAN would normally encode the month‟s variable so that January corresponded to 1 and February corresponded to 2 etc. Also it would be possible in some languages to declare an enumerated type {Jan, Feb, Mar,……, Dec}. Either way this type of declaration is relatively simple because the ranges have set values.

When we do not have explicit bounds on these variable ranges then we have to create our own. These are known as artificial bounds and can be illustrated via the use of the Triangle problem. The point raised by P.C. Jorgensen was that we can easily impose a lower bound on the length of an edge for the tri-angle as an edge with a negative length would be “silly”. The problem occurs when trying to decide upon an upper bound for the length of each length. We could use a certain set integer, we could allow the program to use the highest possible integer (normally denoted as something to the effect of MaxInt). The arbitrary nature of this problem can lead to messy results or non concise test cases.

5.4 Limitations of BVA

Boundary Value Analysis works well when the Program Under Test (PUT) is a “function of several independent variables that represent bounded physical quantities” [1]. When these conditions are met BVA works well but when they are not we can find deficiencies in the results.

For example the NextDate problem, where Boundary Value Analysis would place an even testing regime equally over the range, tester‟s intuition and common sense shows that we require more emphasis towards the end of February or on leap years.

The reason for this poor performance is that BVA cannot compensate or take into consideration the nature of a function or the dependencies between its variables. This lack of intuition or understanding for the variable nature means that BVA can be seen as quite rudimentary.

Robustness testing

6.0 Robustness Testing

Robustness testing can be seen as and extension of Boundary Value Analysis. The idea behind Robustness testing is to test for clean and dirty test cases. By clean I mean input variables that lie in the legitimate input range. By dirty I mean using input variables that fall just outside this input domain.

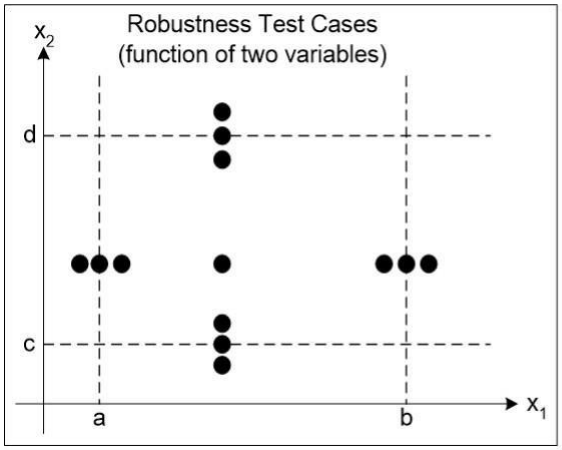
In addition to the aforementioned 5 testing values (min, min+, nom, max-, max) we use two more values for each variable (min-, max+), which are designed to fall just outside of the input range.

If we adapt our function f to apply to Robustness testing we find the following equation:

$f = 6n + 1$

I have equated this solution by the same reasoning that lead to the standard BVA equation. Each variable now has to assume 6 different values each whilst the other values are assuming their nominal value (hence the 6n), and there is again one instance whereby all variables assume their nominal value (hence the addition of the constant 1). These result can be seen in figures 6.1 and 6.2.

Robustness testing ensues a sway in interest, where the previous interest lied in the input to the program, the main focus of attention associated with Robustness testing comes in the expected outputs when and input variable has exceeded the given input domain. For example the NextDate problem when we an entry like the 31st June we would expect an error message to the effect of "that date does not exist; please try again". Robustness testing has the desirable property that it forces attention on exception handling. Although Robustness testing can be somewhat awkward in strongly typed languages it can show up altercations. In Pascal if a value is defined to reside in a certain range then and values that falls outside that range result in the run time errors that would terminate any normal execution. For this reason exception handling mandates Robustness testing.



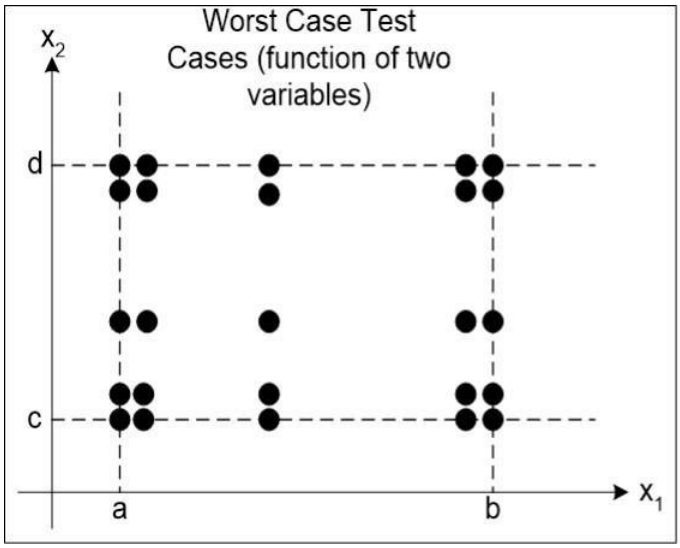
$\langle x_{1nom}, x_{2min} \rangle$
$\langle x_{1nom}, x_{2min+} \rangle$
$\langle x_{1nom}, x_{2nom} \rangle$
$\langle x_{1nom}, x_{2max-} \rangle$
$\langle x_{1nom}, x_{2max} \rangle$
$\langle x_{1min}, x_{2nom} \rangle$
$\langle x_{1min+}, x_{2nom} \rangle$
$\langle x_{1nom}, x_{2nom} \rangle$
$\langle x_{1max-}, x_{2nom} \rangle$
$\langle x_{1max}, x_{2nom} \rangle$
$\langle x_{1min-}, x_{2nom} \rangle$
$\langle x_{1nom}, x_{2max+} \rangle$
$\langle x_{1max+}, x_{2nom} \rangle$
$\langle x_{1nom}, x_{2min-} \rangle$

Worst-case testing

Boundary Value analysis uses the critical fault assumption and therefore only tests for a single variable at a time assuming its extreme values. By disregarding this assumption we are able to test the outcome if more than one variable were to assume its extreme value. In an electronic circuit this is called Worst Case Analysis. In Worst-Case testing we use this idea to create test cases.

To generate test cases we take the original 5-tuple set (min, min+, nom, max-, max) and perform the Cartesian product of these values. The end product is a much larger set of results than we have seen before.

We can see from the results in figures 7.1 and 7.2 that worst case testing is a more comprehensive testing technique. This can be shown by the fact that standard Boundary Value Analysis test cases are a proper subset of Worst-Case test cases.



$\langle x_{1min}, x_{2min} \rangle$
$\langle x_{1min}, x_{2min+} \rangle$
$\langle x_{1min}, x_{2nom} \rangle$
$\langle x_{1min}, x_{2max-} \rangle$
$\langle x_{1min}, x_{2max} \rangle$
$\langle x_{1min+}, x_{2min} \rangle$
$\langle x_{1min+}, x_{2min+} \rangle$
$\langle x_{1min+}, x_{2nom} \rangle$
$\langle x_{1min+}, x_{2max-} \rangle$
$\langle x_{1min+}, x_{2max} \rangle$
$\langle x_{1nom}, x_{2min} \rangle$
$\langle x_{1nom}, x_{2min+} \rangle$
$\langle x_{1nom}, x_{2nom} \rangle$
$\langle x_{1nom}, x_{2max-} \rangle$
$\langle x_{1nom}, x_{2max} \rangle$

$\langle x_{1max-}, x_{2min} \rangle$
$\langle x_{1max-}, x_{2min+} \rangle$
$\langle x_{1max-}, x_{2nom} \rangle$
$\langle x_{1max-}, x_{2max-} \rangle$
$\langle x_{1max-}, x_{2max} \rangle$
$\langle x_{1max}, x_{2min} \rangle$
$\langle x_{1max}, x_{2min+} \rangle$
$\langle x_{1max}, x_{2nom} \rangle$
$\langle x_{1max}, x_{2max-} \rangle$
$\langle x_{1max}, x_{2max} \rangle$

These test cases although more comprehensive in their coverage, constitute much more endeavour. To compare we can see that Boundary Value Analysis results in 4n + 1 test case where Worst-Case testing results in 5^n test cases. As each variable has to assume each of its variables for each permutation (the Cartesian product) we have 5 to the n test cases.

For this reason Worst-Case testing is generally used for situations that require a higher degree of testing (where failure of the program would be very costly)with less regard for the time and effort required as for many situations this can be too expensive to justify.

Robust Worst testing for triangle problem,

If the function under test were to be of the greatest importance we could use a method named Robust Worst-Case testing which as the name suggests draws it attributes from Robust and Worst-Case testing.

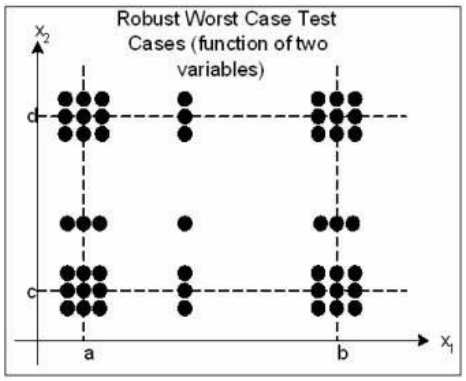
Test cases are constructed by taking the Cartesian product of the 7-tuple set defined in the Robustness testing chapter. Obviously this results in the largest set of test results we have seen so far and requires the most effort to produce.

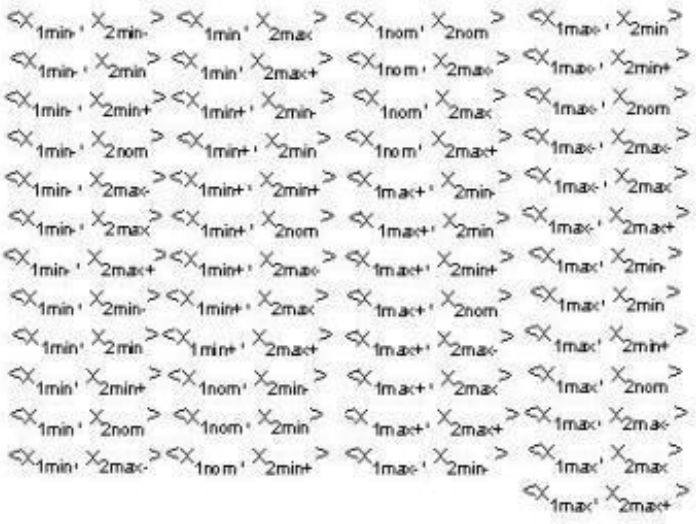
We can see that the function f (to calculate the number of test cases required) can be adapted to calculate the amount of Robust Worst-Case test cases. As there are now 7 values each variable can assume we find the function f to be:

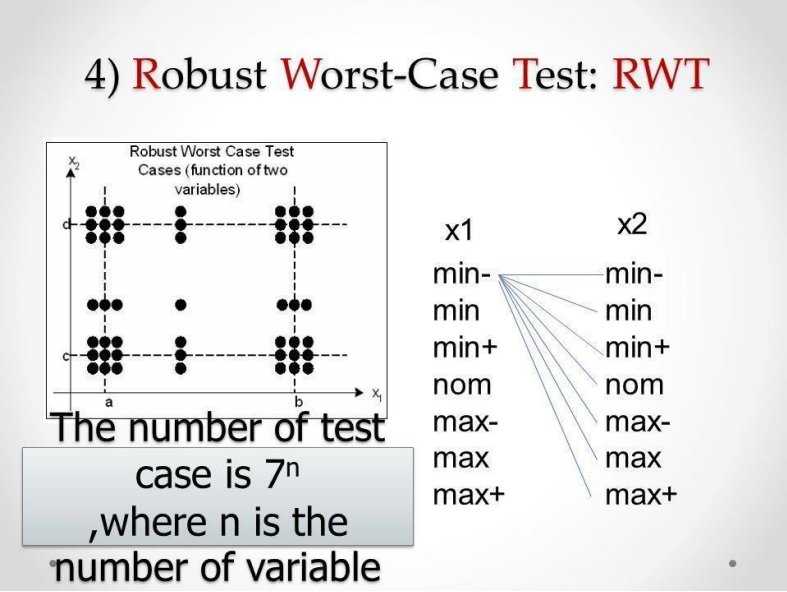
$f = 7^n$

This function has also been reached in the paper A Testing and analysis tool for Certain 3-Variable functions [2].

The results for the continuing example can be seen in figures 7.3 and 7.4.







For each example I will show test cases for the standard Boundary Value Analysis and the Worst-case testing techniques. These will show how the test cases are performed and how comprehensive the results are. There will not be test cases for Robustness testing or robust Worst-case testing as the cases covered should explain how the process works. Too many test cases would prove to be monotonous when trying to explain a concept, however when presenting a real project when the figures are more "necessary" all test cases should be detailed and explained to their full extent.

Robust worst case testing for triangle problem

Standard Boundary Value Analysis test cases:

min = 1
min+ = 2
nom = 100
max- = 199
max = 200

Boundary Value Analysis Test Cases

Case	a	b	c	Expected Output
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a Triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	199	100	Isosceles
9	100	200	100	Not a Triangle
10	1	100	100	Isosceles
11	2	100	100	Isosceles
12	199	100	100	Isosceles
13	200	100	100	Not a Triangle

Worst-Case Analysis test cases:

Worst Case Test Cases (60 of 125)

Case	a	b	c	Expected Output	Case	a	b	c	Expected Output
1	1	1	1	Equilateral	31	2	2	1	Isosceles
2	1	1	2	Not a Triangle	32	2	2	2	Equilateral
3	1	1	100	Not a Triangle	33	2	2	100	Not a Triangle
4	1	1	199	Not a Triangle	34	2	2	199	Not a Triangle
5	1	1	200	Not a Triangle	35	2	2	200	Not a Triangle
6	1	2	1	Not a Triangle	36	2	100	1	Not a Triangle
7	1	2	2	Isosceles	37	2	100	2	Not a Triangle
8	1	2	100	Not a Triangle	38	2	100	100	Isosceles
9	1	2	199	Not a Triangle	39	2	100	199	Not a Triangle
10	1	2	200	Not a Triangle	40	2	100	200	Not a Triangle
11	1	100	1	Not a Triangle	41	2	199	1	Not a Triangle
12	1	100	2	Not a Triangle	42	2	199	2	Not a Triangle
13	1	100	100	Isosceles	43	2	199	100	Not a Triangle
14	1	100	199	Not a Triangle	44	2	199	199	Isosceles
15	1	100	200	Not a Triangle	45	2	199	200	Scalene
16	1	199	1	Not a Triangle	46	2	200	1	Not a Triangle
17	1	199	2	Not a Triangle	47	2	200	2	Not a Triangle
18	1	199	100	Not a Triangle	48	2	200	100	Not a Triangle
19	1	199	199	Isosceles	49	2	200	199	Scalene
20	1	199	200	Not a Triangle	50	2	200	200	Isosceles
21	1	200	1	Not a Triangle	51	100	1	1	Not a Triangle
22	1	200	2	Not a Triangle	52	100	1	2	Not a Triangle
23	1	200	100	Not a Triangle	53	100	1	100	Isosceles
24	1	200	199	Not a Triangle	54	100	1	199	Not a Triangle
25	1	200	200	Isosceles	55	100	1	200	Not a Triangle
26	2	1	1	Not a Triangle	56	100	2	1	Not a Triangle
27	2	1	2	Isosceles	57	100	2	2	Not a Triangle
28	2	1	100	Not a Triangle	58	100	2	100	Isosceles
29	2	1	199	Not a Triangle	59	100	2	199	Not a Triangle
30	2	1	200	Not a Triangle	60	100	2	200	Not a Triangle

Again this is only up to 60 of 125 test cases.

Robust worst case testing Nextdate problem

month	day	year
min = 1	min = 1	min = 1812
min+ = 2	min+ = 2	min+ = 1813
nom = 6	nom = 15	nom = 1912
max- = 11	max- = 30	max- = 2011
max = 12	max = 31	max = 2012

Boundary Value Analysis Test Cases

Case	month	day	year	Expected Output
1	6	15	1812	June 16, 1812
2	6	15	1813	June 16, 1813
3	6	15	1912	June 16, 1912
4	6	15	2011	June 16, 2011
5	6	15	2012	June 16, 2012
6	6	1	1912	June 2, 1912
7	6	2	1912	June 3, 1912
8	6	30	1912	July 1, 1912
9	6	31	1912	error
10	1	15	1912	January 16, 1912
11	2	15	1912	February 16, 1912
12	11	15	1912	November 16, 1912
13	12	15	1912	December 16, 1912

Worst case analysis test case

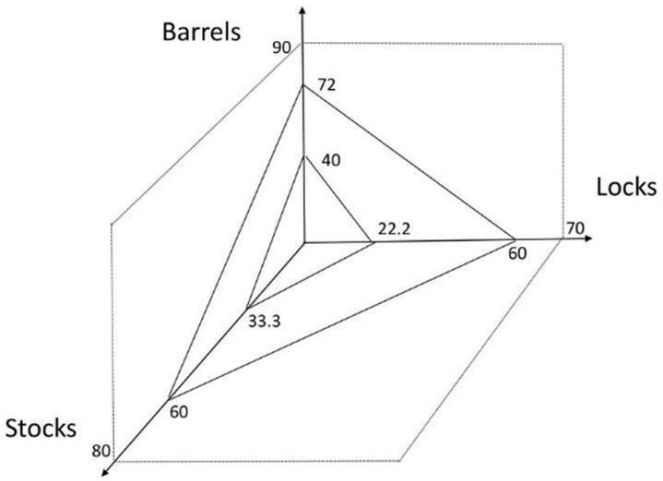
Worst Case Test Cases (60 of 125)

Case	month	day	year	Expected Output	Case	month	day	year	Expected Output
1	1	1	1812	January 2, 1812	31	2	2	1812	February 3, 1812
2	1	1	1813	January 2, 1813	32	2	2	1813	February 3, 1813
3	1	1	1912	January 2, 1912	33	2	2	1912	February 3, 1912
4	1	1	2011	January 2, 2011	34	2	2	2011	February 3, 2011
5	1	1	2012	January 2, 2012	35	2	2	2012	February 3, 2012
6	1	2	1812	January 3, 1812	36	2	15	1812	February 16, 1812
7	1	2	1813	January 3, 1813	37	2	15	1813	February 16, 1813
8	1	2	1912	January 3, 1912	38	2	15	1912	February 16, 1912
9	1	2	2011	January 3, 2011	39	2	15	2011	February 16, 2011
10	1	2	2012	January 3, 2012	40	2	15	2012	February 16, 2012
11	1	15	1812	January 16, 1812	41	2	30	1812	error
12	1	15	1813	January 16, 1813	42	2	30	1813	error
13	1	15	1912	January 16, 1912	43	2	30	1912	error
14	1	15	2011	January 16, 2011	44	2	30	2011	error
15	1	15	2012	January 16, 2012	45	2	30	2012	error
16	1	30	1812	January 31, 1812	46	2	31	1812	error
17	1	30	1813	January 31, 1813	47	2	31	1813	error
18	1	30	1912	January 31, 1912	48	2	31	1912	error
19	1	30	2011	January 31, 2011	49	2	31	2011	error
20	1	30	2012	January 31, 2012	50	2	31	2012	error
21	1	31	1812	February 1, 1812	51	6	1	1812	June 2, 1812
22	1	31	1813	February 1, 1813	52	6	1	1813	June 2, 1813
23	1	31	1912	February 1, 1912	53	6	1	1912	June 2, 1912
24	1	31	2011	February 1, 2011	54	6	1	2011	June 2, 2011
25	1	31	2012	February 1, 2012	55	6	1	2012	June 2, 2012
26	2	1	1812	February 2, 1812	56	6	2	1812	June 3, 1812
27	2	1	1813	February 2, 1813	57	6	2	1813	June 3, 1813
28	2	1	1912	February 2, 1912	58	6	2	1912	June 3, 1912
29	2	1	2011	February 2, 2011	59	6	2	2011	June 3, 2011
30	2	1	2012	February 2, 2012	60	6	2	2012	June 3, 2012

As we can see there are only 60 of 125 test cases in this example, this shows the vast amount of test cases produced.

Robust worst case testing commission problem

- Rifle salespersons in the Arizona Territory sold rifle locks, stocks, and barrels made by a gunsmith in Missouri
- Lock = $45.00, stock = $30.00, barrel = $25.00
- Each salesperson had to sell at least one complete rifle per month ($100)
- The most one salesperson could sell in a month was 70 locks, 80 stocks, and 90 barrels
- Each salesperson sent a telegram to the Missouri company with the total order for each town (s)he visits
- 1≤towns visited≤10, per month
- Commission: 10% on sales up to $1000, 15% on the next $800, and 20% on any sales in excess of $1800



Output Boundary Value Test Cases

Case #	Locks	Stocks	Barrels	Sales	Comm.	Comments
1	1	1	1	100	10	min
2	10	10	9	975	97.5	border-
3	10	9	10	970	97	border-
4	9	10	10	955	95.5	border-
5	**10**	**10**	**10**	**1000**	**100**	**border**
6	10	10	11	1025	103.75	border+
7	10	11	10	1030	104.5	border+
8	11	10	10	1045	106.75	border+

Equivalence classes

Equivalence Class Testing, which is also known as Equivalence Class Partitioning (ECP) and Equivalence Partitioning, is an important software testing technique used by the team of testers for grouping and partitioning of the test input data, which is then used for the purpose of testing the software product into a number of different classes.

These different classes resemble the specified requirements and common behavior or attribute(s) of the aggregated inputs. Thereafter, the test cases are designed and created based on each class attribute(s) and one element or input is used from each class for the test execution to validate the software functioning, and simultaneously validates the similar working of the software product for all the other inputs present in their respective classes.

For an int variable in some program, it might be possible to test the project when every program value is input for the variable. This is true because, on any specific machine, only a finite number of values can be assigned to an int variable. However, the number of values is large, and the testing would be very time consuming and not likely worthwhile.

The number of possible values is much larger for variables of type float or String.

Thus, for almost every program, it is impossible to test all possible input values.

To get around the impossibility of testing for every possible input value, the possible input values for a variable are normally divided into categories, usually called blocks or equivalence classes.

The objective is to put values into the same equivalence class if the project should have similar (equivalent) behavior for each value of the equivalence class.

Now, rather than testing the project for all possible input values, the project is tested for an input value from each equivalent class.

The rationale for defining an equivalence class is as follows: If one test case for a particular equivalence class exposes an error, all other test cases in that equivalence class will likely expose the same error.

Using standard notation from discrete mathematics, the objective is to partition the input values for each variable, where a partition is defined as follows:

Definition 16.1: A partition of a set A is the division of the set into subsets
Ai, i = 1, 2, . . . ,m,

called blocks or equivalence classes, such that each element of A is in exactly one of the equivalence classes.

Often the behavior of a program is a function of the relative values of several variables.

In this case, it is necessary for the partition to reflect the values of all the variables involved. As an example, consider the following informal specification of a program:

Given the three sides of a triangle as integers x, y, and z, it is desired to have a program to determine the type of the triangle: equilateral, isosceles, or scalene.

The behavior (i.e., output) of the program depends on the values of the three integers. However, as previously remarked, it is infeasible to try all possible combinations of the possible integer values.

Traditional equivalence class testing simply partitions the input values into valid and nonvalid

values, with one equivalence class for valid values and another for each type of invalid values.

Note that this implies an individual test case to cover each invalid equivalence class. The rationale for doing this is that if invalid inputs can contain multiple errors, the detection of one error may result in other error checks not being made.

For the triangle example, there are several types of invalid values. The constraints can be divided into the following categories:

C 1. The values of x, y, and z are greater than zero.
C 2. The length of the longest side is less than the sum of the lengths of the other two sides.

To guarantee that each invalid situation is checked independently, an invalid equivalence class should be set up for each of the variables having a nonpositive value:

1. $\{(x, y, z) \mid x \leq 0, y, z > 0\}$

2. $\{(x, y, z) \mid y \leq 0, x, z > 0\}$

3. $\{(x, y, z) \mid z \leq 0, x, y > 0\}$

However, each of the variables can be the one that has the largest value (i.e., corresponds to the longest side). Thus, three more invalid equivalence classes are needed:

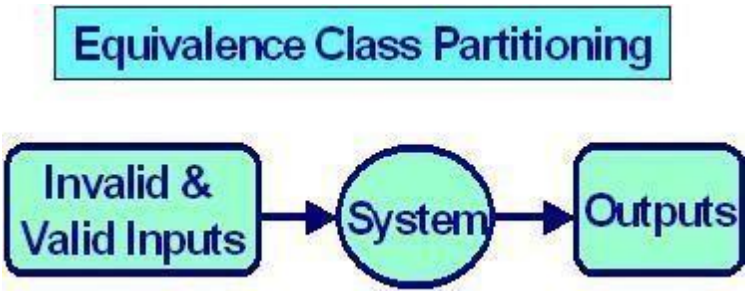
4. $\{(x, y, z) \mid x \geq y, x \geq z, x \geq y + z\}$

5. $\{(x, y, z) \mid y \geq x, y \geq z, y \geq x + z\}$

6. $\{(x, y, z) \mid z \geq x, z \geq y, z \geq x + y\}$

In the current example, possible test cases for each equivalence class are the following:
1. (−1, 2, 3), (0, 2, 3)
2. (2, −1, 3), (2, 0, 3)
3. (2, 3, −1), (2, 3, 0)
4. (5, 2, 3), (5, 1, 2)
5. (2, 5, 3), (1, 5, 2)
6. (2, 3, 5), (1, 2, 5)

The above are not handled by BVA technique as we can see massive redundancy in the tables of test cases. In this technique, the input and the output domain is divided into a finite number of equivalence classes.



Then, we select one representative of each class and test our program against it. It is assumed by the tester that if one representative from a class is able to detect error then why should he consider other cases. Furthermore, if this single representative test case did not detect any error then we assume that no other test case of this class can detect error. In this method we consider both valid and invalid input domains. The system is still treated as a black-box meaning that we are not bothered about its internal logic.

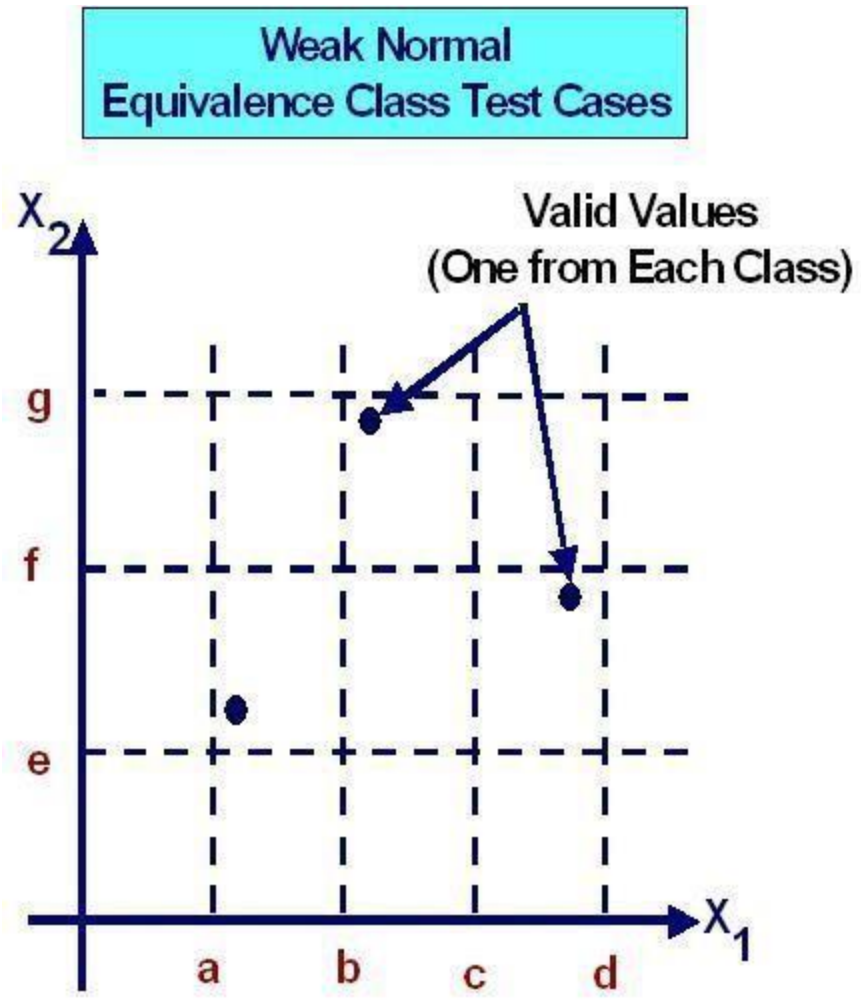
The idea of equivalence class testing is to identify test cases by using one element from each equivalence class. If the equivalence classes are chosen wisely, the potential redundancy among test cases can be reduced.

Types of equivalence class testing:

Following four types of equivalence class testing are presented here

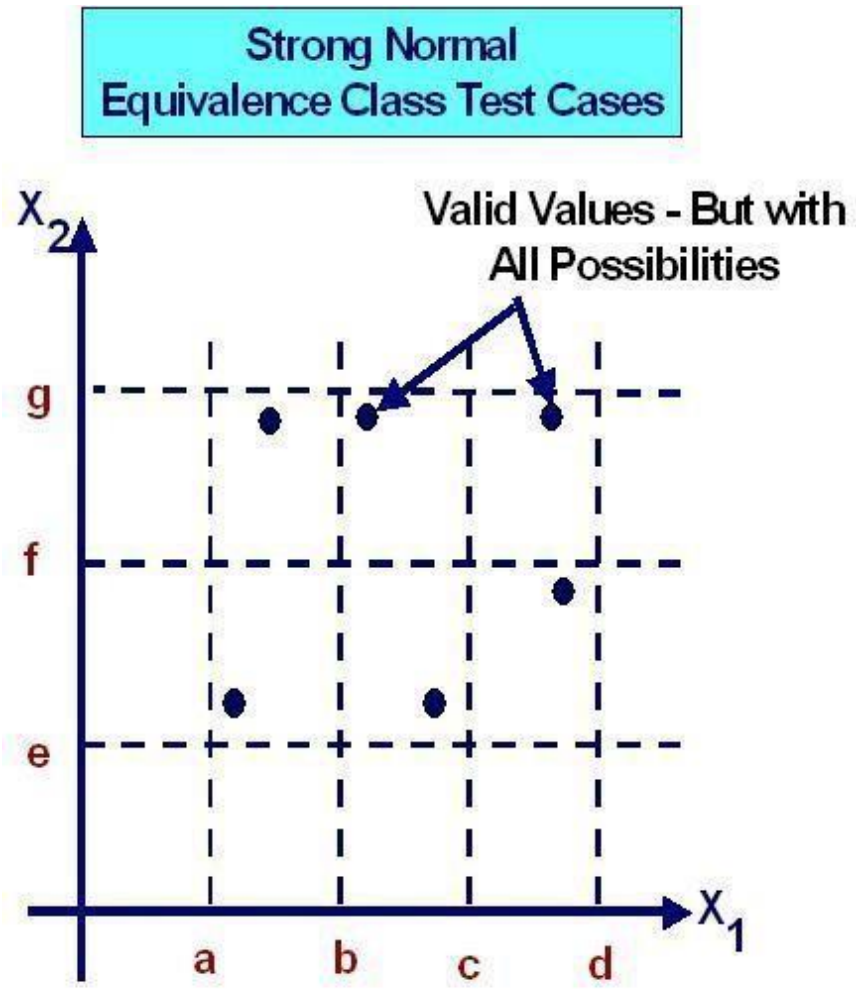
1) Weak Normal Equivalence Class Testing.
2) Strong Normal Equivalence Class Testing.
3) Weak Robust Equivalence Class Testing.
4) Strong Robust Equivalence Class Testing.

1) Weak Normal Equivalence Class Testing:
The word „weak" means „single fault assumption". This type of testing is accomplished by using one variable from each equivalence class in a test case. We would, thus, end up with the weak equivalence class test cases as shown in the following figure.



Each dot in above graph indicates a test data. From each class we have one dot meaning that there is one representative element of each test case. In fact, we will have, always, the same number of weak equivalence class test cases as the classes in the partition.

2) Strong Normal Equivalence Class Testing:
This type of testing is based on the multiple fault assumption theory. So, now we need test cases from each element of the Cartesian product of the equivalence classes, as shown in the following figure.

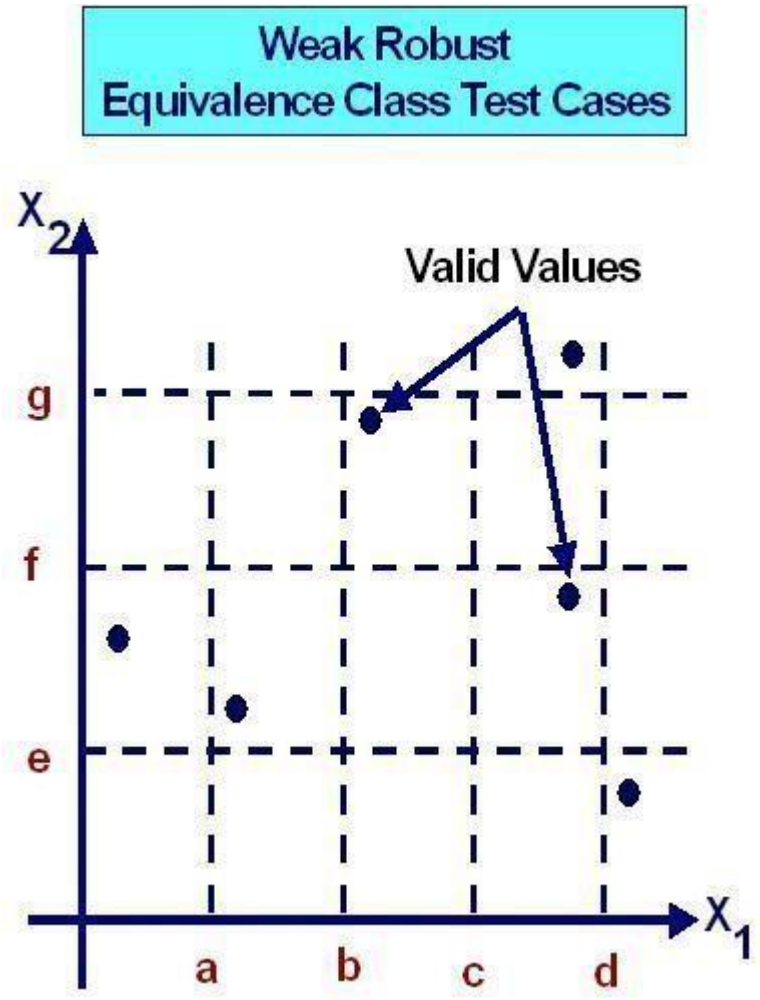


Just like we have truth tables in digital logic, we have similarities between these truth tables and our pattern of test cases. The Cartesian product guarantees that we have a notion of "completeness" in following two ways

a) We cover all equivalence classes.
b) We have one of each possible combination of inputs.

3) Weak Robust Equivalence Class Testing:

The name for this form of testing is counter intuitive and oxymoronic. The word" weak" means single fault assumption theory and the word „Robust" refers to invalid values. The test cases resulting from this strategy are shown in the following figure.



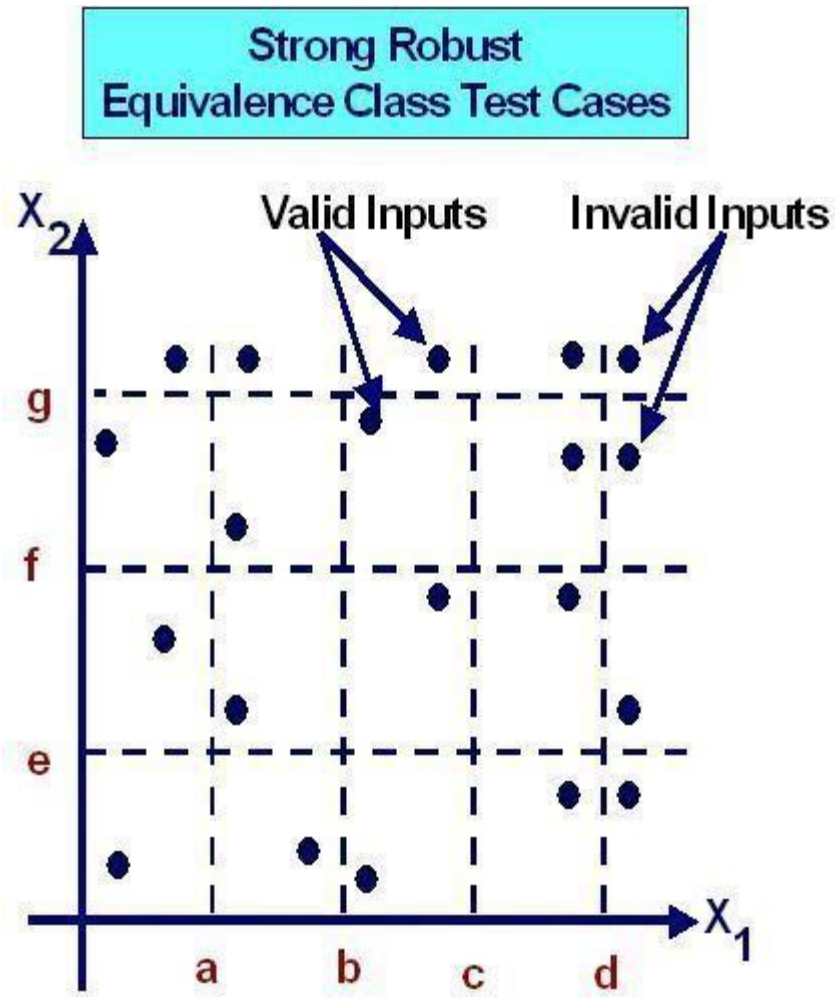
Following two problems occur with robust equivalence testing.

a) Very often the specification does not define what the expected output for an invalid test case should be. Thus, testers spend a lot of time defining expected outputs for these cases.

b) Strongly typed languages like Pascal, Ada, eliminate the need for the consideration of invalid inputs. Traditional equivalence testing is a product of the time when languages such as FORTRAN, C and COBOL were dominant. Thus this type of error was quite common.

4) Strong Robust Equivalence Class Testing:

This form of equivalence class testing is neither counter intuitive nor oxymoronic, but is just redundant. As explained earlier also, „robust‟ means consideration of invalid values and the „strong‟ means multiple fault assumption. We obtain the test cases from each element of the Cartesian product of all the equivalence classes as shown in the following figure.



We find here that we have 8 robust (invalid) test cases and 12 strong or valid inputs. Each one is represented with a dot. So, totally we have 20 test cases (represented as 20 dots) using this technique.

Guidelines for Equivalence Class Testing:
The following guidelines are helpful for equivalence class testing

1) The weak forms of equivalence class testing (normal or robust) are not as comprehensive as the corresponding strong forms.
2) If the implementation language is strongly typed and invalid values cause run-time errors then there is no point in using the robust form.
3) If error conditions are a high priority, the robust forms are appropriate.
4) Equivalence class testing is approximate when input data is defined in terms of intervals and sets of discrete values. This is certainly the case when system malfunctions can occur for out-of-limit variable values.
5) Equivalence class testing is strengthened by a hybrid approach with boundary value testing (BVA).

6) Equivalence class testing is used when the program function is complex. In such cases, the complexity of the function can help identify useful equivalence classes.
7) Strong equivalence class testing makes a presumption that the variables are independent and the corresponding multiplication of test cases raises issues of redundancy. If any dependencies occur, they will often generate "error" test cases.
8) Several tries may be needed before the "right" equivalence relation is established.
9) The difference between the strong and weak forms of equivalence class testing is helpful in the distinction between progression and regression testing.

Equivalence test cases for the triangle Problem

Four possible outputs –
NotA-Triangle, Scalene, Isosceles and Equilateral.

R1 = { : the triangle with sides a.b and c is equilateral}
R2 = { : the triangle with sides a,b and c is isosceles}
R3 = { : the triangle with sides a,b and c is isosceles}
R4 = { : sides a,b and c do not form a triangle}

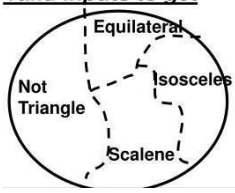
Test Case	a	b	c	Expected Output
W N1	5	5	5	Equilateral
W N2	2	2	3	Isosceles
W N3	3	4	5	Scalene
W N4	4	1	2	Not a Triangle

Consider: Weak Normal Equivalence Test Cases for Triangle Problem

"valid" inputs:
1<= a <= 200
1<= b <= 200
1<= c <= 200
and
for triangle:
a < b + c
b < a + c
c < b + a

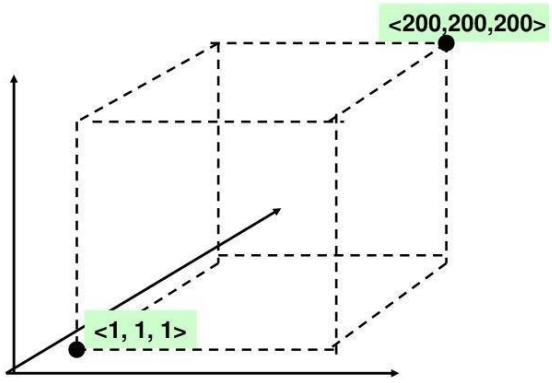
	inputs		
output	a	b	c
Not triangle	35	10	4
Equilateral	35	35	35
Isosceles	24	24	7
Scalene	35	18	24

Valid Inputs to get



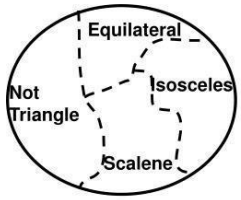
Strong Normal Equivalence Test Cases for Triangle Problem • Since there is no further sub-intervals inside the valid inputs for the 3 sides a, b, and c, are Strong Normal Equivalence is the same as the Weak Normal Equivalence

Weak Robust Equivalence Test Cases for Triangle Problem



Now, on top of the earlier 4 normal test cases, include the "invalid" inputs

Valid Inputs



Include 6 invalid test case *in addition to Weak Normal* above:

<201, 45, 50 >
<45, 204, 78 >
<50, 78, 208 >

below:

< -5, 76, 89 >
< 56, -20, 89 >
< 56, 89, 0 >

Strong Robust Equivalence Test Cases for Triangle Problem

- Similar to Weak robust, but all combinations of "invalid" inputs must be *included to the Strong Normal.*
- Look at the "cube" figure and consider the corners (two diagonal ones)

a) Consider one of the corners <200,200,200> : there should be $(2^3 - 1) = 7$ cases of "invalids"

< 201, 201, 201 >
< 201, 201, 50 >
< 201, 50 , 201 >
< 201, 50 , 50 >
< 50 , 201, 50 >
< 50 , 201, 201 >
< 50, 50 , 201 >

b) There will be 7 more "invalids" when we consider the other corner , <1,1,1 >:

< 0, 0, 0 >
< 0, 0, 5 >
< 0, 10, 0 >
< 0, 8, 10>
<7, 0, 9 >
<8, 0, 0 >
<8, 9, 0 >

NextDate function

Next Date Function Problem „
Valid Equivalence Classes

M1 = { month : 1 ≤ month ≤ 12 }
D1 = { day: 1 ≤ day ≤ 31 }
Y1 = { year: 1812 ≤ year ≤ 2012 } „

Invalid Equivalence Classes
M2 = { month : month < 1 }
M3 = { month : month > 12 }
D2 = { day: day < 1 }
D3 = { day: day > 31 }
Y2 = { year: year < 1812 }
Y3 = { year: year > 2012 }

- Valid classes = Independent variables
- One weak and strong normal ECT.

Day	Month	Year	Expected Output
15	6	1912	16/6/1912

- Weak Robust Test Cases

Day	Month	Year	Expected Output
15	6	1912	16/6/1912
-1	6	1912	day not in range
32	6	1912	day not in range
15	-1	1912	month not in range
15	13	1912	month not in range
15	6	1811	year not in range
15	6	2013	year not in range

Strong robust ECT

Test Case	Month	Day	Year	Expected Output
SR1	-1	15	1912	Value of month not in the range 1..12
SR2	6	-1	1912	Value of day not in the range 1..31
SR3	6	15	1811	Value of year not in the range 1812..2012
SR4	-1	-1	1912	Value of month not in the range 1..12
				Value of day not in the range 1..31
SR5	6	-1	1811	Value of day not in the range 1..31
				Value of year not in the range 1812..2012
SR6	-1	15	1811	Value of month not in the range 1..12
				Value of year not in the range 1812..2012
SR7	-1	-1	1811	Value of month not in the range 1..12
				Value of day not in the range 1..31
				Value of year not in the range 1812..2012

Previous test cases were poor. „

Focus on Equivalence Relation. „
What must be done to an input date? „
We produce a new set of Equivalence Classes.

M1 = { month: month has 30 days } „
M2 = { month: month has 31 days } „
M3 = { month: month is February } „
D1 = { day: $1 \leq day \leq 28$ } „
D2 = { day: day = 29 } „
D3 = { day: day = 30 } „
D4 = { day: day = 31 } „
Y1 = { year: year = 2000 } „
Y2 = { year: year is a leap year } „
Y3 = { year: year is a common year }

So, now let us again identify the various equivalence class test cases:

1) Weak Normal Equivalence Class: As done earlier as well, the inputs are mechanically selected from the approximate middle of the corresponding class.

Test Case ID	Month (mm)	Day (dd)	Year (yyyy)	Expected Output
WN1	6	14	2000	6/15/2000
WN2	7	29	1996	7/30/1996
WN3	2	30	2002	2/31/2002 (Impossible)
WN4	6	31	2000	7/1/2000 (Impossible)

The random / mechanical selection of input values makes no consideration of our domain knowledge and thus we have two impossible dates. This will always be a problem with „automatic" test case generation because all of our domain knowledge is not captured in the choice of equivalence classes.

2) Strong Normal Equivalence Class: The strong normal equivalence class test cases for the revised classes are:

Test Case ID	Month (mm)	Day (dd)	Year (yyyy)	Expected	Output
SN1	6	14	2000	6/15/2000	
SN2	6	14	1996	6/15/1996	
SN3	6	14	2002	6/15/2002	
SN4	6	29	2000	6/30/2000	
SN5	6	29	1996	6/30/1996	
SN6	6	29	2002	6/30/2002	
SN7	6	30	2000	6/31/2000	(Impossible)
SN8	6	30	1996	6/31/1996	(Impossible)
SN9	6	30	2002	6/31/2002	(Impossible)
SN10	6	31	2000	7/1/2000	(Invalid Input)
SN11	6	31	1996	7/1/1996	(Invalid Input)
SN12	6	31	2002	7/1/2002	(Invalid Input)
SN13	7	14	2000	7/15/2000	
SN14	7	14	1996	7/15/1996	
SN15	7	14	2002	7/15/2002	
SN16	7	29	2000	7/30/2000	
SN17	7	29	1990	7/30/1996	
SN18	7	29	2002	7/30/2002	
SN19	7	30	2000	7/31/2000	
SN20	7	30	1996	7/31/1996	
SN21	7	30	2002	7/31/2002	
SN22	7	31	2000	8/1/1996	
SN23	7	31	1996	8/1/2000	
SN24	7	31	2002	8/1/2002	
SN25	2	14	2000	7/15/2000	
SN26	2	14	1996	2/15/1996	
SN27	2	14	2002	2/15/2002	
SN28	2	29	2000	3/1/2000	(Invalid Input)
SN29	2	29	1996	3/1/1996	
SN30	2	29	2002	3/1/2002	(Impossible Date)
SN31	2	30	2000	3/1/2000	(Impossible Date)
SN32	2	30	1996	3/1/1996	(Impossible Date)
SN33	2	30	2002	3/1/2002	(Impossible Date)
SN34	6	31	2000	7/1/2000	(Impossible Date)
SN35	6	31	1996	7/1/1996	(Impossible Date)
SN36	6	31	2002	3/1/2002	(Impossible Date)

So, three month classes, four day classes and three year classes results in 3 * 4 * 3 = 36 strong normal equivalence class test cases. Furthermore, adding two invalid classes for each variable will result in 150 strong robust equivalence class test cases.

It is quite difficult to describe all such 150 classes here.
There are 150 strong-robust test cases (5 x 6 x 5)

commission problem

Class for Commission Problem

Test data : price Rs for lock - 45.0 , stock - 30.0 and barrel - 25.0

sales = total lock * lock price + total stock * stock price + total barrel * barrel price

commission : 10% up to sales Rs 1000 , 15 % of the next Rs 800 and 20 % on any sales in excess of 1800

Pre-condition : lock = -1 to exit and 1< =lock < = 70 , 1<=stock <=80 and 1<=barrel<=90

Brief Description : The salesperson had to sell at least one complete rifle per month.

Checking boundary value for locks, stocks and barrels and commission

Valid Classes

L1 ={LOCKS :1 <=LOCKS<=70}
L2 ={Locks=-1}(occurs if locks=-1 is used to control input iteration)
L3 ={stocks : 1<=stocks<=80}
L4= {barrels :1<=barrels<=90}

Invalid Classes
L3 ={locks: locks=0 OR locks<-1}
L4 ={locks: locks> 70}
S2 ={stocks : stocks<1}
S3 ={stocks : stocks >80}
B2 ={barrels : barrels <1}
B3 =barrels : barrels >90}

Commission Problem Output Equivalence Class Testing
(Weak & Strong Normal Equivalence Class)

Commission Problem Output Equivalence Class Testing

(Weak & Strong Normal Equivalence Class)

Case Id	Description	Input Data			Expected Output		Actual output		Status	Comment
		Total Locks	Total Stocks	Total Barrels	Sales	Commission	Sales	Commission		
1	Enter the value within the range for lock, stocks and barrels	35	40	45	3900	640				

Weak Robustness Equivalence Class

Case Id	Description	Input Data			Expected Output	Actual output	Status	Comment
		Locks	Stocks	Barrels				
WR1	Enter the value locks = -1	-1	40	45	Terminates the input loop and proceed to calculate sales and commission (if Sales > 0)			
WR2	Enter the value less than -1 or equal to **zero** for locks and other valid inputs	0	40	45	Value of Locks not in the range 1..70			
WR3	Enter the value greater than 70 for locks and other valid inputs	71	40	45	Value of Locks not in the range 1..70			
WR4	Enter the value less than or equal than 0 for stocks and other valid inputs	35	0	45	Value of stocks not in the range 1..80			
WR5	Enter the value greater than 80 for stocks and other valid inputs	35	81	45	Value of stocks not in the range 1..80			
WR6	Enter the value less than or equal 0 for barrels and other valid inputs	35	40	0	Value of Barrels not in the range 1..90			
WR7	Enter the value greater than 90 for barrels and other valid inputs	35	40	91	Value of Barrels not in the range 1..90			

Strong Robustness Equivalence Class

Case Id	Description	Input Data			Expected Output	Actual output	Status	Comment
		Locks	Stocks	Barrels				
SR1	Enter the value less than -1 for locks and other valid inputs	-2	40	45	Value of Locks not in the range 1..70			
SR2	Enter the value less than or equal than 0 for stocks and other valid inputs	35	-1	45	Value of stocks not in the range 1..80			
SR3	Enter the value less than or equal 0 for barrels and other valid inputs	35	40	-2	Value of Barrels not in the range 1..90			
SR4	Enter the locks and stocks less than or equal to 0 and other valid inputs	-2	-1	45	Value of Locks not in the range 1..70			
					Value of stocks not in the range 1..80			
SR5	Enter the locks and barrel less than or equal to 0 and other valid inputs	-2	40	-1	Value of Locks not in the range 1..70			
					Value of Barrels not in the range 1..90			
SR6	Enter the stocks and barrel less than or equal to 0 and other valid inputs	35	-1	-1	Value of stocks not in the range 1..80			
					Value of Barrels not in the range 1..90			
SR7	Enter the stocks and barrel less than or equal to 0 and other valid inputs	-2	-2	-2	Value of Locks not in the range 1..70			
					Value of stocks not in the range 1..80			
					Value of Barrels not in the range 1..90			

Some addition equivalence Boundary checking

Case Id	Description	Input Data			Expected Output		Actual output			
		Total Locks	Total Stocks	Total Barrels	Sales	Commission	Sales	Commission	Status	Comment
OR1	Enter the value for lock, stocks and barrels where 0 < Sales < 1000	5	5	5	500	50				
OR2	Enter the value for lock, stocks and barrels where 1000 < Sales < 1800	15	15	15	1500	175				
OR3	Enter the value for lock, stocks and barrels where Sales < 1800	25	25	25	2500	360				

Guidelines and observations

Guidelines and Observations
Now that we have gone through three examples, we conclude with some observations about, and guidelines for equivalence class testing.
1. The traditional form of equivalence class testing is generally not as thorough as weak equivalence class testing, which in turn, is not as thorough as the strong form of equivalence class testing.
2. The only time it makes sense to use the traditional approach is when the implementation language is not strongly typed.
3. If error conditions are a high priority, we could extend strong equivalence class testing to include invalid classes.
4. Equivalence class testing is appropriate when input data is defined in terms of ranges and sets of discrete values. This is certainly the case when system malfunctions can occur for out-of-limit variable values.
5. Equivalence class testing is strengthened by a hybrid approach with boundary value testing. (Wecan "reuse" the effort made in defining the equivalence classes.)
6. Equivalence class testing is indicated when the program function is complex. In such cases, the complexity of the function can help identify useful equivalence classes, as in the NextDate function.
7. Strong equivalence class testing makes a presumption that the variables are independent when the Cartesian Product is taken. If there are any dependencies, these will often generate "error" test cases, as they did in the NextDate function. (The decision table technique in Chapter 7 resolves this problem.)
8. Several tries may be needed before "the right" equivalence relation is discovered, as we saw in the NextDate example. In other cases, there is an "obvious" or "natural" equivalence relation. When in doubt, the best bet is to try to second guess aspects of any reasonable implementation.

Decision tables

Decision Table Test case design technique is one of the testing techniques. You could find other testing techniques such as Equivalence Partitioning, Boundary Value Analysis

In Decision table technique, we deal with combinations of inputs. To identify the test cases with decision table, we consider conditions and actions. We take conditions as inputs and actions as outputs.

Stub	Rule 1	Rule 2	Rules 3,4	Rule 5	Rule 6	Rules 7,8
c1	T	T	T	F	F	F
c2	T	T	F	T	T	F
c3	T	F	-	T	F	-
a1	X	X		X		
a2	X				X	
a3		X		X		
a4			X			X

condition stubs	condition entries
action stubs	action entries

Examples on Decision Table Test Case Design Technique:

Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognized	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

Take an example of transferring money online to an account which is already added and approved.

Here the conditions to transfer money are ACCOUNT ALREADY APPROVED, OTP (One Time Password) MATCHED, SUFFICIENT MONEY IN THE ACCOUNT.

And the actions performed are TRANSFER MONEY, SHOW A MESSAGE AS INSUFFICIENT AMOUNT, BLOCK THE TRANSACTION INCASE OF SUSPICIOUS TRANSACTION.

Here we decide under what condition the action be performed Now let‟s see the tabular column below.

DECISION TABLE

ID	CONDITIONS/ACTIONS	TEST CASE 1	TEST CASE 2	TEST CASE 3	TEST CASE 4	TEST CASE 5
Condition 1	Account Already Approved	T	T	T	T	F
Condition 2	OTP (One Time Password) Matched	T	T	F	F	X
Condition 3	Sufficient Money in the Account	T	F	T	F	X
Action 1	Transfer Money	Execute				
Action 2	Show a Message as 'Insufficient Amount'		Execute			
Action 3	Block The Transaction Incase of Suspicious Transaction			Execute	Execute	X

In the first column I took all the conditions and actions related to the requirement. All the other columns represent Test Cases.

T = True, F = False, X = Not possible

From the case 3 and case 4, we could identify that if condition 2 failed then system will execute Action 3. So we could take either of case 3 or case 4

So finally concluding with the below tabular column.

ID	CONDITIONS/ACTIONS	TEST CASE 1	TEST CASE 2	TEST CASE 3	TEST CASE 4
Condition 1	Account Already Approved	T	T	T	F
Condition 2	OTP (One Time Password) Matched	T	T	F	X
Condition 3	Sufficient Money in the Account	T	F	T	X
Action 1	Transfer Money	Execute			
Action 2	Show a Message as 'Insufficient Amount'		Execute		
Action 3	Block The Transaction Incase of Suspicious Transaction			Execute	X

Decision Table Interpretation
Conditions are interpreted as
Input
Equivalence classes of inputs

Actions are interpreted as
Output
Major functional processing portions

With a complete decision table
We have a complete set of test cases

The ability to recognize complete decision table puts us into a challenge of identifying redundant and inconsistent rules

A redundant decision table

Rule 4 and 9

Conditions	1–4	5	6	7	8	9
c1	T	F	F	F	F	T
c2	–	T	T	F	F	F
c3	–	T	F	T	F	F
a1	X	X	X	–	–	X
a2	–	X	X	X	–	–
a3	X	–	X	X	X	X

- **Here rules 4 and rule 9 are identical; redundant**

A inconsistent decision table rule 4 and rule 9

Redundancy OK; but Inconsistency?

- **Now consider rules 4 and 9**

Table 7.10 An Inconsistent Decision Table

Conditions	1–4	5	6	7	8	9
c1	T	F	F	F	F	T
c2	–	T	T	F	F	F
c3	–	T	F	T	F	F
a1	X	X	X	–	–	–
a2	–	X	X	X	–	X
a3	X	–	X	X	X	–

Test cases for the triangle problem

Triangle Decision Table

	1	2	3	4	5	6	7	8	9
C1: <a, b,c > forms a triangle?	F	T	T	T	T	T	T	T	T
C3: a = b?	–	T	T	T	T	F	F	F	F
C4: a = c?	–	T	T	F	F	T	T	F	F
C5: b = c?	–	T	F	T	F	T	F	T	F
A1: Not a Triangle	X								
A2: Scalene									X
A3: Isosceles					X		X	X	
A4: Equilateral		X							
A5: Impossible			**X**	**X**		**X**			

Action added by a tester showing impossible rules

Triangle Decision Table – refined

	1	2	3	4	5	6	7	8	9	10	11
C1-1: a < b+c?	F	T	T	T	T	T	T	T	T	T	T
C1-2: b < a+c?	–	F	T	T	T	T	T	T	T	T	T
C1-3: c < a+b?	–	–	F	T	T	T	T	T	T	T	T
C2: a = b?	–	–	–	T	T	T	T	F	F	F	F
C3: a = c?	–	–	–	T	T	F	F	T	T	F	F
C4: b = c?	–	–	–	T	F	T	F	T	F	T	F
A1: Not a Triangle	X	X	X								
A2: Scalene											X
A3: Isosceles							X		X	X	
A4: Equilateral				X							
A5: Impossible					X	X		X			

Similar to equivalence classes we can refine the conditions

Triangle Test Cases

Case ID	a	b	c	Expected Output
1	4	1	2	Not a Triangle
2	1	4	2	Not a Triangle
3	1	2	4	Not a Triangle
4	5	5	5	Equilateral
5	???	???	???	Impossible
6	???	???	???	Impossible
7	2	2	3	Isosceles
8	???	???	???	Impossible
9	2	3	2	Isosceles
10	3	2	2	Isosceles
11	3	4	5	Scalene

Don't Care Entries and Rule Counts
Limited entry tables with N conditions have 2^N rules.

Don't care entries reduce the number of explicit rules by implying the existence of non-explicitly stated rules.
How many rules does a table contain including all the implied rules due to don't care entries?

Don't Care Entries and Rule Counts – 2
Each don't care entry in a rule doubles the count for the rule
For each rule determine the corresponding rule count
Total the rule counts

1	C1-1: a < b+c?	F	T	T	T	T	T	T	T	T	T
2	C1-2: b < a+c?	–	F	T	T	T	T	T	T	T	T
3	C1-3: c < a+b?	–	–	F	T	T	T	T	T	T	T
4	C2: a = b?	–	–	–	T	T	T	T	F	F	F
5	C3: a = c?	–	–	–	T	T	F	F	T	T	F
6	C4: b = c?	–	–	–	T	F	T	F	T	F	T
	Rule count	32	16	8	1	1	1	1	1	1	1

+/ = 64
= 2^6

Test Cases for the Triangle Problem

- 11 test cases
  - 3 impossible
  - 3 not triangle
  - 1 equilateral
  - 1 scalene
  - 3 isosceles
- ? means invalid

Table 7.11 Test Cases from Table 7.3

Case ID	*a*	*b*	*c*	*Expected Output*
DT1	4	1	2	Not a Triangle
DT2	1	4	2	Not a Triangle
DT3	1	2	4	Not a Triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

NextDate function

The NextDate problem illustrates the problem of dependencies in the input domain Decision tables can highlight such dependencies

Impossible dates can be clearly marked as a separate action

C1: month in M1?	T	–	–
C2: month in M2?	–	T	–
C3: month in M3?	–	–	T
A1: Impossible			
A2: Next Date			

Because a month is in an equivalence class
we cannot have T for more than one entry.
The do not care entries are really F.

NextDate Equivalence Classes – for 1st try

M1 = {month : 1 .. 12 | days(month) = 30 }
M2 = {month : 1 .. 12 | days(month) = 31 }
M3 = {month : {2} }
D1 = {day : 1 .. 28}
D2 = {day : {29} }
D3 = {day : {30} }
D4 = {day : {31} }
Y1 = {year : 1812 .. 2012 | leap_year (year) }
Y2 = {year : 1812 .. 2012 | common_year (year) }

First try decision table yields 256 rules

C1: month in M1?	T	T	T	T	T	T	T	T				
C2: month in M2?									T	T	T	T
C3: month in M3?												
C4: day in D1?	T	T							T	T		
C5: day in D2?			T	T							T	T
C6: day in D3?					T	T						
C7: day in D4?							T	T				
C8: year in Y1?	T		T		T		T		T		T	
C9: year in Y2?		T		T		T		T		T		T
A1: Impossible							X	X				
A2: Next Date	X	X	X	X	X	X			X	X	X	X

NextDate Equivalence Classes – for 2nd try

M1 = {month : 1 .. 12 | days(month) = 30 }
M2 = {month : 1 .. 12 | days(month) = 31 }
M3 = {month : {2} }
D1 = {day : 1 .. 28}
D2 = {day : {29} }
D3 = {day : {30} }
D4 = {day : {31} }
Y1 = {year : {2000} }
Y2 = {year : 1812 .. 2012 | leap_year (year) A year ≠ 2000 }
Y3 = {year : 1812 .. 2012 | common_year (year) }

Second try decision table yields 36 rules
3 months * 4 days * 3 year = 36 rules

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
C1: month in	M1	M1	M1	M1	M2	M2	M2	M2	M3	M3	M3	M3	M3	M3	M3	M3
C2: day in	D1	D2	D3	D4	D1	D2	D3	D4	D1	D1	D1	D2	D2	D2	D3	D3
C3: year in	–	–	–	–	–	–	–	–	Y1	Y2	Y3	Y1	Y2	Y3	–	–
A1: Impossible				X								X		X	X	X
A2: Increment day	X	X			X	X	X			X						
A3: Reset day			X					X	X		X		X			
A4: Increment month			X					???	X		X		X			
A5: reset month								???								
A6: Increment year								???								

December problem in rule 8
And February 28 problem in rule 9,11 and 12

So we go for Try 3

M1 = {month : 1 .. 12 | days(month) = 30 }
M2 = {month : 1 .. 12 | days(month) = 31 ∧ month ≠ 12 }
M3 = {month : {12} }
M4 = {month : {2} }
D1 = {day : 1 .. 27}
D2 = {day : {28} }
D3 = {day : {29} }
D4 = {day : {30} }
D5 = {day : {31} }
Y1 = {year : 1812 .. 2012 | leap_year (year) }
Y2 = {year : 1812 .. 2012 | common_year (year) }

Handle end of month and year better

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
C1: month in	M1	M1	M1	M1	M1	M2	M2	M2	M2	M2	M3	M3	M3	M3	M3	M4	M4	M4	M4	M4	M4	M4
C2: day in	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5	D1	D2	D2	D3	D3	D4	D5
C3: year in	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	Y1	Y2	Y1	Y2	–	–
A1: Impossible					X															X	X	X
A2: Increment day	X	X	X			X	X	X	X		X	X	X	X		X	X					
A3: Reset day				X						X					X			X	X			
A4: Increment month				X						X								X	X			
A5: reset month															X							
A6: Increment year															X							

Table 7.16 Decision Table Test Cases for NextDate

Case ID	Month	Day	Year	Expected Output
1–3	April	15	2001	April 16, 2001
4	April	30	2001	May 1, 2001
5	April	31	2001	Invalid Input Date
6–9	January	15	2001	January 16, 2001
10	January	31	2001	February 1, 2001
11–14	December	15	2001	December 16, 2001
15	December	31	2001	January 1, 2002
16	February	15	2001	February 16, 2001
17	February	28	2004	February 29, 2004
18	February	28	2001	March 1, 2001
19	February	29	2004	March 1, 2004
20	February	29	2001	Invalid Input Date
21, 22	February	30	2001	Invalid Input Date

Commission problem

Test Case Name : Boundary Value for Commission Problem
Experiment Number : 5
Test data : price Rs for lock - 45.0 , stock - 30.0 and barrel - 25.0
sales = total lock * lock price + total stock * stock price + total barrel * barrel price
commission : 10% up to sales Rs 1000 , 15 % of the next Rs 800 and 20 % on any sales in excess of 1800
Pre-condition : lock = -1 to exit and 1< =lock < = 70 , 1<=stock <=80 and 1<=barrel<=90
Brief Description : The salesperson had to sell at least one complete rifle per month.

CHECKING BOUNDARY VALUE FOR LOCKS, STOCKS AND BARRELS AND COMMISSION
Commission Problem Output Boundary Value Analysis Cases

Case Id	Description	Input Data			Expected Output		Actual output		Status	
		Total Locks	Total Stocks	Total Barrels	Sales	Comm-ission	Sales	Comm -ission		Comment
1	Enter the min value for locks, stocks and barrels	1	1	1	100	10				output minimum
2	Enter the min value for 2 items and min +1 for any one item	1	1	2	125	12.5				output minimum +
3		1	2	1	130	13				output minimum +
4		2	1	1	145	14.5				output minimum +
5	Enter the value sales approximately mid value between 100 to 1000	5	5	5	500	50				Midpoint
6	Enter the values to calculate the commission for sales nearly less than 1000	10	10	9	975	97.5				Border point -
7		10	9	10	970	97				Border point -
8		9	10	10	955	95.5				Border point -
9	Enter the values sales exactly equal to 1000	10	10	10	1000	100				Border point
10	Enter the values to calculate the commission for sales nearly greater than 1000	10	10	11	1025	103.75				Border point +
11		10	11	10	1030	104.5				Border point +
12		11	10	10	1045	106.75				Border point +

13	Enter the value sales approximately mid value between 1000 to 1800	14	14	14	1400	160				Midpoint
14	Enter the values to calculate the commission for sales nearly less than 1800	18	18	17	1775	216.25				Border point -
15		18	17	18	1770	215.5				Border point -
16		17	18	18	1755	213.25				Border point -
17	Enter the values sales exactly equal to 1800	18	18	18	1800	220				Border point
18	Enter the values to calculate the commission for sales nearly greater than 1800	18	18	19	1825	225				Border point +
19		18	19	18	1830	226				Border point +
20		19	18	18	1845	229				Border point +
21	Enter the values normal value for lock, stock and barrel	48	48	48	4800	820				Midpoint
22	Enter the max value for 2 items and max - 1 for any one item	70	80	89	7775	1415				Output maximum -
23		70	79	90	7770	1414				Output maximum -
24		69	80	90	7755	1411				Output maximum -
25	Enter the max value for locks, stocks and barrels	70	80	90	7800	1420				Output maximum

Output Special Value Test Cases

Case Id	Description	Input Data			Expected Output		Actual output		Status	Comment
		Total Locks	Total Stocks	Total Barrels	Sales	Commission	Sales	Commission		
1	Enter the random values such that to calculate commission for sales nearly less than 1000	11	10	8	995	99.5				Border point -
2	Enter the random values such that to calculate commission for sales nearly greater than 1000	10	11	9	1005	100.75				Border point +
3	Enter the random values such that to calculate commission for sales nearly less than 1800	18	17	19	1795	219.25				Border point -
4	Enter the random values such that to calculate commission for sales nearly greater than 1800	18	19	17	1805	221				Border point +

Test Case Name :Equivalence Class for Commission Problem
Experiment Number : 6
Test data : price Rs for lock - 45.0 , stock - 30.0 and barrel - 25.0
sales = total lock * lock price + total stock * stock price + total barrel * barrel price
commission : 10% up to sales Rs 1000 , 15 % of the next Rs 800 and 20 % on any sales in excess of 1800
Pre-condition : lock = -1 to exit and 1< =lock < = 70 , 1<=stock <=80 and 1<=barrel<=90
Brief Description : The salesperson had to sell at least one complete rifle per month.

Checking boundary value for locks, stocks and barrels and commission
Valid Classes
L1 ={LOCKS :1 <=LOCKS<=70}
L2 ={Locks=-1}(occurs if locks=-1 is used to control input iteration)
L3 ={stocks : 1<=stocks<=80}
L4= {barrels :1<=barrels<=90}
Invalid Classes
L3 ={locks: locks=0 OR locks<-1}
L4 ={locks: locks> 70}
S2 ={stocks : stocks<1}
S3 ={stocks : stocks >80}
B2 ={barrels : barrels <1}

B3 =barrels : barrels >90}

Commission Problem Output Equivalence Class Testing

(Weak & Strong Normal Equivalence Class)

Case Id	Description	Input Data			Expected Output		Actual output		Status	Comment
		Total Locks	Total Stocks	Total Barrels	Sales	Commission	Sales	Commission		
1	Enter the value within the range for lock, stocks and barrels	35	40	45	3900	640				

Weak Robustness Equivalence Class

Case Id	Description	Input Data			Expected Output	Actual output	Status	Comment
		Locks	Stocks	Barrels				
WR1	Enter the value locks = -1	-1	40	45	Terminates the input loop and proceed to calculate sales and commission (if Sales > 0)			
WR2	Enter the value less than -1 or equal to **zero** for locks and other valid inputs	0	40	45	Value of Locks not in the range 1..70			
WR3	Enter the value greater than 70 for locks and other valid inputs	71	40	45	Value of Locks not in the range 1..70			
WR4	Enter the value less than or equal than 0 for stocks and other valid inputs	35	0	45	Value of stocks not in the range 1..80			
WR5	Enter the value greater than 80 for stocks and other valid inputs	35	81	45	Value of stocks not in the range 1..80			
WR6	Enter the value less than or equal 0 for barrels and other valid inputs	35	40	0	Value of Barrels not in the range 1..90			
WR7	Enter the value greater than 90 for barrels and other valid inputs	35	40	91	Value of Barrels not in the range 1..90			

Strong Robustness Equivalence Class

Case Id	Description	Input Data			Expected Output	Actual output	Status	Comment
		Locks	Stocks	Barrels				
SR1	Enter the value less than -1 for locks and other valid inputs	-2	40	45	Value of Locks not in the range 1..70			
SR2	Enter the value less than or equal than 0 for stocks and other valid inputs	35	-1	45	Value of stocks not in the range 1..80			
SR3	Enter the value less than or equal 0 for barrels and other valid inputs	35	40	-2	Value of Barrels not in the range 1..90			
SR4	Enter the locks and stocks less than or equal to 0 and other valid inputs	-2	-1	45	Value of Locks not in the range 1..70 Value of stocks not in the range 1..80			
SR5	Enter the locks and barrel less than or equal to 0 and other valid inputs	-2	40	-1	Value of Locks not in the range 1..70 Value of Barrels not in the range 1..90			
SR6	Enter the stocks and barrel less than or equal to 0 and other valid inputs	35	-1	-1	Value of stocks not in the range 1..80 Value of Barrels not in the range 1..90			
SR7	Enter the stocks and barrel less than or equal to 0 and other valid inputs	-2	-2	-2	Value of Locks not in the range 1..70 Value of stocks not in the range 1..80 Value of Barrels not in the range 1..90			

Some addition equivalence Boundary checking

Case Id	Description	Input Data			Expected Output		Actual output		Status	Comment
		Total Locks	Total Stocks	Total Barrels	Sales	Commission	Sales	Commission		
OR1	Enter the value for lock, stocks and barrels where 0 < Sales < 1000	5	5	5	500	50				
OR2	Enter the value for lock, stocks and barrels where 1000 < Sales < 1800	15	15	15	1500	175				
OR3	Enter the value for lock, stocks and barrels where Sales < 1800	25	25	25	2500	360				

Test Case Name :Decision Table for Commission Problem

Experiment Number : 7

Test data : price Rs for lock - 45.0 , stock - 30.0 and barrel - 25.0

sales = total lock * lock price + total stock * stock price + total barrel * barrel price

commission : 10% up to sales Rs 1000 , 15 % of the next Rs 800 and 20 % on any sales

in excess of 1800

Pre-condition : lock = -1 to exit and 1< =lock < = 70 , 1<=stock <=80 and 1<=barrel<=90
Brief Description : The salesperson had to sell at least one complete rifle per month.

Input data decision Table

RULES		R1	R2	R 3	R4	R5	R 6	R7	R8	R10
Conditions	C1: Locks = -1	T	F	F	F	F	F	F	F	F
	C2 : 1 ≤ Locks ≤ 70	-	T	T	F	T	F	F	F	T
	C3 : 1 ≤ Stocks ≤ 80	-	T	F	T	F	T	F	F	T
	C4 : 1 ≤ Barrels ≤ 90	-	F	T	T	F	F	T	F	T
Actions	a1 : Terminate the input loop	X								
	a2 : Invalid locks input				X		X	X	X	
	a3 : Invalid stocks input			X		X		X	X	
	a4 : Invalid barrels input		X			X	X		X	
	a5 : Calculate total locks, stocks and barrels		X	X	X	X	X	X		X
	a5 : Calculate Sales	X								
	a6: proceed to commission decision table	X								

Commission calculation Decision Table (Precondition : lock = -1)

RULES		R1	R2	R3	R4
Condition	C1 : tlocks>0 && tstocks>0 && tbarrels>0	T	T	T	F
	C1 : Sales > 0 AND Sales ≤ 1000	T	F	F	
	C2 : Sales > 1001 AND sales ≤ 1800		T	F	
	C3 : sales ≥1801			T	
Actions	A1 : Cannot calculate the commission				X
	A2 : comm= 10%*sales	X			
	A3 : comm = 10%*1000 + (sales-1000)*15%		X		
	A4 : comm = 10%*1000 + 15% * 800 + (sales-1800)*20%			X	

Guidelines and observations.

As with the other testing techniques, decision table based testing works well for some applications (like NextDate) and is not worth the trouble for others (like Commission Problem). Not surprisingly, the situations in which it works well are those where there is a lot of decision making (like the Triangle Problem), and those in which there are important logical relationships among input variables (like the NextDate function).

1. The decision table technique is indicated for applications characterized by any of the following:
prominent If-Then-Else logic logical relationships among input variables calculations involving subsets of the input variables cause and effect relationships between inputs and outputs high cyclomatic (McCabe) complexity

2. Decision tables don‟t scale up very well (a limited entry table with n conditions has 2n rules). There are several ways to deal with this: use extended entry decision tables, algebraically simplify tables, “factor” large tables into smaller ones, and look for repeating patterns of condition entries. For more on these techniques

3. As with other techniques, iteration helps. The first set of conditions and actions you identify may
be unsatisfactory. Use it as a stepping stone, and gradually improve on it until you are satisfied with a decision table.

Fault Based Testing:

A model of potential program faults is a valuable source of information for evaluating and designing test suites. Some fault knowledge is commonly used in functional and structural testing, for example when identifying singleton and error values for parameter characteristics in category partition testing, or when populating catalogs with erroneous values, but a fault model can also be used more directly. Fault-based testing uses a fault model directly to hypothesize potential faults in a program under test, and to create or evaluate test suites based on its efficacy in detecting those hypothetical faults.

Overview,

Engineers study failures to understand how to prevent similar failures in the future. For example, failure of the Tacoma Narrows Bridge led to new understanding of oscillation in high wind, and the introduction of analyses to predict and prevent such destructive oscillation in subsequent bridge design. The causes of an airline crash are likewise extensively studied, and when traced to a structural failure they frequently result in a directive to apply diagnostic tests to all aircraft considered potentially vulnerable to similar failures.

Experience with common software faults sometimes leads to improvements in design methods and programming languages. For example, the main purpose of automatic memory management in Java is not to spare the programmer the trouble of releasing unused memory, but to prevent the programmer from making the kind of memory management errors (dangling pointers, redundant deallocations, and memory leaks) that frequently occur in C and C++ programs. Automatic array bounds checking cannot prevent a programmer from using an index expression outside array bounds, but can make it much less likely that the fault escapes detection in testing, as well as limiting the damage incurred if it does lead to operational failure (eliminating, in particular, the buffer overflow attack as a means of subverting privileged programs). Type checking reliably detects many other faults during program translation. Of course, not all programmer errors fall into classes that can be prevented or statically detected using better programming languages. Some faults must be detected through testing, and there too we can use knowledge about common faults to be more effective. The basic concept of fault-based testing is to select test cases that would distinguish the program under test from alternative programs that contain hypothetical faults. This is usually approached by modifying the program under test to actually produce the hypothetical faulty programs. Fault seeding can be used to evaluate the thoroughness of a test suite (that is, as an element of a test adequacy criterion), or for selecting test cases to augment a test suite, or to estimate the number of faults in a program.

Assumptions in fault based testing,

16.2 Assumptions in Fault-Based Testing The effectiveness of fault-based testing depends on the quality of the fault model, and on some basic assumptions about the relation of the seeded faults

to faults that might actually be present. In practice the seeded faults are small syntactic changes,

like replacing one variable reference by another in an expression, or changing a comparison from < to <=. We may hypothesize that these are representative of faults actually present in the program.

COMPETENT PROGRAMMER HYPOTHESIS

Put another way, if the program under test has an actual fault, we may hypothesize that it differs from another, corrected program by only a small textual change. If so, then we need merely distinguish the program from all such small variants (by selecting test cases for which either the original or the variant program fails) to ensure detection of all such faults. This is known as the competent programmer hypothesis, an assumption that the program under test is "close to" (in the sense of textual difference) a correct program.

COUPLING EFFECT HYPOTHESIS

Some program faults are indeed simple typographical errors, and others that involve deeper errors of logic may nonetheless be manifest in simple textual differences. Sometimes, though, an error of logic will result in much more complex differences in program text. This may not invalidate fault-based testing with a simpler fault model, provided test cases sufficient for detecting the simpler faults are sufficient also for detecting the more complex fault. This is known as the coupling effect.

The coupling effect hypothesis may seem odd, but can be justified by appeal to a more plausible hypothesis about interaction of faults. A complex change is equivalent

Fault Based Testing: Terminology

Original program: The program unit (e.g., C function or Java class) to be tested.

Program location: A region in the source code. The precise definition is defined relative to the syntax of a particular programming language. Typical locations are statements, arithmetic and boolean expressions, and procedure calls.

Alternate expression: Source code text that can be legally substituted for the text at a program location. A substitution is legal if the resulting program is syntactically correct (i.e., it compiles without errors).

Alternate program: A program obtained from the original program by substituting an alternate expression for the text at some program location.

Distinct behavior of an alternate program *R* for a test *t*: The behavior of an alternate program *R* is distinct from the behavior of the original program *P* for a test *t*, if *R* and *P* produce a different result for *t*, or if the output of *R* is not defined for *t*.

Distinguished set of alternate programs for a test suite *T*: A set of alternate programs are distinct if each alternate program in the set can be distinguished from the original program by at least one test in *T*.

to several smaller changes in program text. If the effect of one of these small changes is not masked by the effect of others, then a test case that differentiates a variant based on a single change may also serve to detect the more complex error. Fault-based testing can guarantee fault detection only if the competent programmer hypothesis and the coupling effect hypothesis hold. But guarantees are more than we expect from other approaches to designing or evaluating test suites, including the structural and functional test adequacy criteria discussed in earlier chapters. Fault-based testing techniques can be useful even if we decline to take the leap of faith required to fully accept their underlying assumptions. What is essential is to recognize the dependence of these techniques, and any inferences about software quality based on fault-based testing, on the quality of the fault model. This also implies that developing better fault models, based on hard data about real faults rather than guesses, is a good investment of effort.

Mutation analysis,

Mutation analysis is the most common form of software fault-based testing. A fault model is used to produce hypothetical faulty programs by creating variants of the program under test. Variants are created by "seeding" faults, that is, by making a small change to the program under test following a pattern in the fault model. The patterns operator for changing program text are

called mutation operators, and each variant program is called a mutant.

Mutation Analysis: Terminology

Original program under test: The program or procedure (function) to be tested.

Mutant: A program that differs from the original program for one syntactic element, e.g., a statement, a condition, a variable, a label, etc.

Distinguished mutant: A mutant that can be distinguished for the original program by executing at least one test case.

Equivalent mutant: A mutant that cannot be distinguished from the original program.

Mutation operator: A rule for producing a mutant program by syntactically modifying the original program.

Mutants should be plausible as faulty programs. Mutant programs that are rejected by a compiler, or which fail almost all tests, are not good models of the faults we seek to uncover with systematic testing. We say a mutant is valid if it is syntactically correct. We say a mutant is useful if, in addition to being valid, its behavior differs from the behavior of the original program for no more than a small subset of program test cases.

A mutant obtained from the program of Figure 16.1 by substituting while for switch in the statement at line 13 would not be valid, since it would result in a compile-time error. A mutant obtained by substituting 1000 for 0 in the statement at line 4 would be valid, but not useful, since the mutant would be distinguished from the program under
test by all inputs and thus would not give any useful information on the effectiveness of a test suite. Defining mutation operators that produce valid and useful mutations is a non-trivial task.

Since mutants must be valid, mutation operators are syntactic patterns defined relative to particular programming languages. Figure 16.2 shows some mutation operators for the C language. Constraints are associated with mutation operators to guide selection of test cases likely to distinguish mutants from the original program. For example, the mutation operator svr (scalar variable replacement) can be applied only to variables of compatible type (to be valid), and a test case that distinguishes the mutant from the original program must execute the modified statement in a state in which the original variable and its substitute have different values.

Many of the mutants of Figure 16.2 can be applied equally well to other procedural languages, but in general a mutation operator that produces valid and useful mutants for a given language may not apply to a different language or may produce invalid or useless mutants for another language. For example, a mutation operator that removes the “friend” keyword from the declaration of a C++ class would not be applicable to Java, which does not include friend classes.

```
1
2 /** Convert each line from standard input */
3 void transduce() {
4 #define BUFLEN 1000
5 char buf[BUFLEN]; /* Accumulate line into this buffer */
6 int pos = 0; /* Index for next character in buffer */
7
8 char inChar; /* Next character from input */
9
10 int atCR = 0; /* 0=”within line”, 1=”optional DOS LF” */
11
12 while ((inChar = getchar()) != EOF ) {
13 switch (inChar) {
14 case LF:
15 if (atCR) { /* Optional DOS LF */
16 atCR = 0;
17 } else { /* Encountered CR within line */
18 emit(buf, pos);
19 pos = 0;
20 }
21 break;
22 case CR:
23 emit(buf, pos);
24 pos = 0;
25 atCR = 1;
26 break;
27 default:
28 if (pos >= BUFLEN-2) fail("Buffer overflow");
29 buf[pos++] = inChar;
30 } /* switch */
31 }
32 if (pos > 0) {
33 emit(buf, pos);
34 }
35 }
```

Figure 16.1: Program transduce converts line endings among Unix, DOS, and Macintosh conventions. The main procedure, which selects the output line end convention, and the output procedure emit are not shown.

id	operator	description	constraint
Operand Modifications			
crp	constant for constant replacement	replace constant $C1$ with constant $C2$	$C1 \neq C2$
scr	scalar for constant replacement	replace constant C with scalar variable X	$C \neq X$
acr	array for constant replacement	replace constant C with array reference $A[I]$	$C \neq A[I]$
scr	struct for constant replacement	replace constant C with struct field S	$C \neq S$
svr	scalar variable replacement	replace scalar variable X with a scalar variable Y	$X \neq Y$
csr	constant for scalar variable replacement	replace scalar variable X with a constant C	$X \neq C$
asr	array for scalar variable replacement	replace scalar variable X with an array reference $A[I]$	$X \neq A[I]$
ssr	struct for scalar replacement	replace scalar variable X with struct field S	$X \neq S$
vie	scalar variable initialization elimination	remove initialization of a scalar variable	
car	constant for array replacement	replace array reference $A[I]$ with constant C	$A[I] \neq C$
sar	scalar for array replacement	replace array reference $A[I]$ with scalar variable X	$A[I] \neq X$
cnr	comparable array replacement	replace array reference with a comparable array reference	
sar	struct for array reference replacement	replace array reference $A[I]$ with a struct field S	$A[I] \neq S$
Expression Modifications			
abs	absolute value insertion	replace e by abs(e)	$e < 0$
aor	arithmetic operator replacement	replace arithmetic operator ψ with arithmetic operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
lcr	logical connector replacement	replace logical connector ψ with logical connector ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
ror	relational operator replacement	replace relational operator ψ with relational operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
uoi	unary operator insertion	insert unary operator	
cpr	constant for predicate replacement	replace predicate with a constant value	
Statement Modifications			
sdl	statement deletion	delete a statement	
sca	switch case replacement	replace the label of one case with another	
ses	end block shift	move } one statement earlier and later	

Figure 16.2: A sample set of mutation operators for the C language, with associated constraints to select test cases that distinguish generated mutants from the original program.

Fault-based adequacy criteria,

Given a program and a test suite T, mutation analysis consists of the following steps:

Select mutation operators: If we are interested in specific classes of faults, we may select a set of mutation operators relevant to those faults.

Generate mutants: Mutants are generated mechanically by applying mutation operators to the original program.

Distinguish mutants: Execute the original program and each generated mutant with the test cases in T. A mutant is killed when it can be distinguished from the original program.

Figure 16.3 shows a sample of mutants for program Transduce, obtained by applying the mutant operators in Figure 16.2. Test suite T S

T S = {1U,1D,2U,2D,2M,End,Long}

kills Mj , which can be distinguished from the original program by test cases 1D, 2U, 2D, and 2M.

Mutants Mi, Mk, and Ml are not distinguished from the original program by any test in T S. We say that mutants not killed by a test suite are live.

A mutant can remain live for two reasons:
• The mutant can be distinguished from the original program, but the test suite T does not contain a test case that distinguishes them, i.e., the test suite is not adequate with respect to the mutant.

• The mutant cannot be distinguished from the original program by any test case,
i.e., the mutant is equivalent to the original program.

Given a set of mutants SM and a test suite T, the fraction of non-equivalent mutants killed by T measures the adequacy of T with respect to SM. Unfortunately, the problem of identifying equivalent mutants is undecidable in general, and we could err either by claiming that a mutant is equivalent to the program under test when it is not, or by counting some equivalent mutants among the remaining live mutants.

The adequacy of the test suite T S evaluated with respect to the four mutants of

Figure 16.3 is 25%. However, we can easily observe that mutant Mi is equivalent to the original program, i.e., no input would distinguish it. Conversely, mutants Mk and Ml seems to be non-equivalent to the original program, i.e., there should be at least one test case that distinguishes each of them from the original program.

Thus the adequacy of T S, measured after eliminating the equivalent mutant Mi , is 33%.

Mutant Ml is killed by test case Mixed, which represents the unusual case of an input file containing both DOS- and Unix-terminated lines. We would expect that Mixed would kill also Mk , but this does not actually happen: both Mk and the original program produce the same

result for Mixed. This happens because both the mutant and the original program fail in the same way.1 The use of a simple oracle for checking

ID	Operator	line	Original/ Mutant	1U	1D	2U	2D	2M	End	Long	Mixed
M_i	ror	28	`(pos >= BUFLEN-2)` `(pos == BUFLEN-2)`	-	-	-	-	-	-	-	-
M_j	ror	32	`(pos > 0)` `(pos >= 0`	-	x	x	x	x	-	-	-
M_k	sdl	16	`atCR = 0` *nothing*	-	-	-	-	-	-	-	-
M_l	ssr	16	`atCR = 0` `pos = 0`	-	-	-	-	-	-	-	x

Test case	Description
1U	One line, Unix line-end
1D	One line, DOS line-end
2U	Two lines, Unix line-end
2D	Two lines, DOS line-end

Test case	Description
2M	Two lines, Mac line-end
End	Last line not terminated with line-end sequence
Long	Very long line (greater than buffer length)
Mixed	Mix of DOS and Unix line ends in the same file

Figure 16.3: A sample set of mutants for program Transduce generated with mutation operators from Figure 16.2.

Variations on mutation analysis.

The mutation analysis process described above, which kills mutants based on the outputs produced by execution of test cases, is known as strong mutation. It can generate a number of mutants quadratic in the size of the program. Each mutant must be compiled and executed with each test case until it is killed. The time and space required for compiling all mutants and for executing all test cases for each mutant may be impractical.

The computational effort required for mutation analysis can be reduced by reducing the number of mutants generated and the number of test cases to be executed. Weak mutation analysis reduces the number of tests to be executed by killing mutants when they produce a different intermediate state, rather than waiting for a difference in the final result or observable program behavior.

With weak mutation, a single program can be seeded with many faults. A “metamutant” program is divided into segments containing original and mutated source code, with a mechanism to select which segments to execute. Two copies of the meta-mutant are executed in tandem, one with only original program code selected, and the other with a set of live mutants selected. Execution is paused after each segment to compare the program state of the two versions. If the state is equivalent, execution resumes with the next segment of original and mutated code. If the state differs, the mutant is marked as dead, and execution of original and mutated code is restarted with a new selection

Mutation Analysis vs Structural Testing

For typical sets of syntactic mutants, a mutation-adequate test suite will also be adequate with respect to simple structural criteria such as statement or branch coverage. Mutation adequacy can simulate and subsume a structural coverage criterion if the set of mutants can be killed only by satisfying the corresponding test coverage obligations.

Statement coverage can be simulated by applying the mutation operator sdl (statement deletion) to each statement of a program. To kill a mutant whose only difference from the program under test is the absence of statement *S* requires executing the mutant and the program under test with a test case that executes *S* in the original program. Thus to kill all mutants generated by applying the operator sdl to statements of the program under test, we need a test suite that causes the execution of each statement in the original program.

Branch coverage can be simulated by applying the operator cpr (constant for predicate replacement) to all predicates of the program under test with constants *True* and *False*. To kill a mutant that differs from the program under test for a predicate *P* set to the constant value *False*, we need to execute the mutant and the program under test with a test case that causes the execution of the *True* branch of *P*. To kill a mutant that differs from the program under test for a predicate *P* set to the constant value *True*, we need to execute the mutant and the program under test with a test case that causes the execution of the *False* branch of *P*.

A test suite that satisfies a structural test adequacy criterion may or may not kill all the corresponding mutants. For example, a test suite that satisfies the statement coverage adequacy criterion might not kill an sdl mutant if the value computed at the statement does not effect the behavior of the program on some possible executions.

of live mutants.

Weak mutation testing does not decrease the number of program mutants that must be considered, but it decreases the number of test executions and compilations. This performance benefit has a cost in accuracy: weak mutation analysis may “kill” a mutant even if the changed intermediate state would not have an effect on the final output or observable behavior of the program.

Like structural test adequacy criteria, mutation analysis can be used either to judge the thoroughness of a test suite or to guide selection of additional test cases. If one is designing test cases to kill particular mutants, then it may be important to have a complete set of mutants generated by a set of mutation operators. If, on the other hand, the goal is a statistical estimate of the extent to which a test suite distinguishes programs with seeded faults from the original program, then only a much smaller statistical sample of mutants is required. Aside from its

limitation to assessment rather than creation of test suites, the main limitation of statistical

mutation analysis is that partial coverage is meaningful only to the extent that the generated mutants are a valid statistical model of occurrence frequencies of actual faults. To avoid reliance on this implausible assumption, the target coverage should be 100% of the sample; statistical sampling may keep the sample small enough to permit careful examination of equivalent mutants.

Fault seeding can be used statistically in another way: To estimate the number of faults remaining in a program. Usually we know only the number of faults that have been detected, and not the number that remains. However, again to the extent that the fault model is a valid statistical model of actual fault occurrence, we can estimate that the ratio of actual faults found to those still remaining should be similar to the ratio of seeded faults found to those still remaining.

Once again, the necessary assumptions are troubling, and one would be unwise to place too much confidence in an estimate of remaining faults. None the less, a prediction with known weaknesses is better than a seat-of-the-pants guess, and a set of estimates derived in different ways is probably the best that one can hope for.

MODULE 3

Structural Testing: Overview, Statement testing, Branch testing, Conditiontesting , Path testing: DD paths, Test coverage metrics, Basis path testing,guidelines and observations, Data –Flow testing: Definition-Use testing, Slicebasedtesting, Guidelines and observations**. Test Execution:** Overview of testexecution, from test case specification to test cases, Scaffolding, Generic versusspecific scaffolding, Test oracles, Self-checks as oracles, Capture and replay

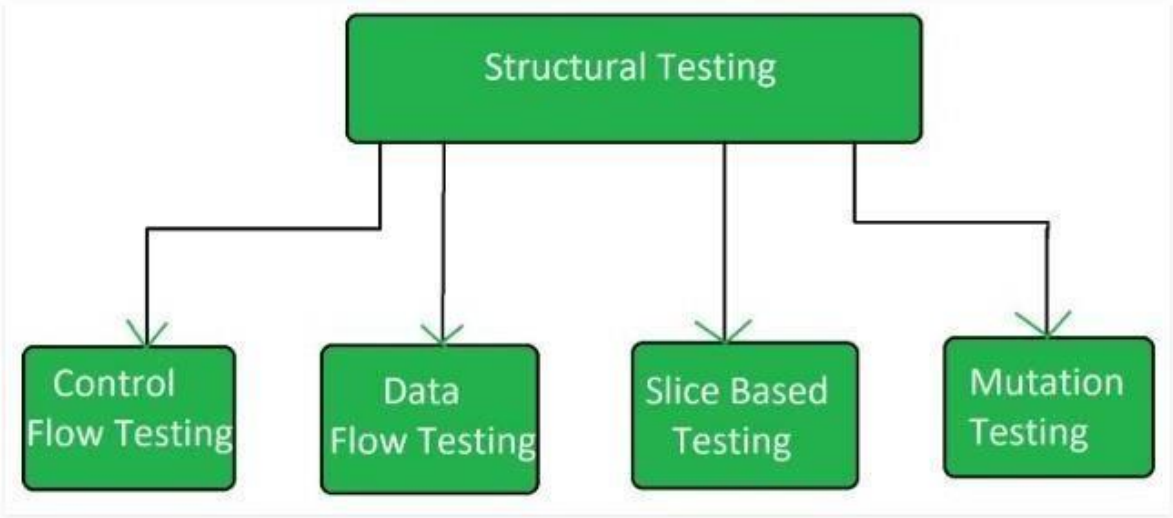
Structural Testing: Overview

Structural testing is a type of software testing which uses the internal design of the software for testing or in other words the software testing which is performed by the team which knows the development phase of the software, is known as structural testing.

Structural testing is basically related to the internal design and implementation of the software i.e. it involves the development team members in the testing team. It basically tests different aspects of the software according to its types. Structural testing is just the opposite of behavioral testing.

Types of Structural Testing:

There are 4 types of Structural Testing:



Control Flow Testing:

Control flow testing is a type of structural testing that uses the programs's control flow as amodel. The entire code, design and structure of the software have to be known for this type of testing. Often this type of testing is used by the developers to test their own code and implementation. This method is used to test the logic of the code so that required result can be obtained.

Data Flow Testing:

It uses the control flow graph to explore the unreasonable things that can happen to data. The detection of data flow anomalies are based on the associations between values and variables. Without being initialized usage of variables. Initialized variables are not used once.

Slice Based Testing:

It was originally proposed by Weiser and Gallagher for the software maintenance. It is useful for software debugging, software maintenance, program understanding and quantification of functional cohesion. It divides the program into different slices and tests that slice which can majorly affect the entire software.

Mutation Testing:

Mutation Testing is a type of Software Testing that is performed to design new software tests and also evaluate the quality of already existing software tests. Mutation testing is related to modification a program in small ways. It focuses to help the tester develop effective tests or locate weaknesses in the test data used for the program.

Advantages of Structural Testing:

- It provides thorough testing of the software.
- It helps in finding out defects at an early stage.
- It helps in elimination of dead code.
- It is not time consuming as it is mostly automated.

Disadvantages of Structural Testing:

- It requires knowledge of the code to perform test.
- It requires training in the tool used for testing.
- Sometimes it is expensive.

What is Code coverage?

Code coverage is a measure which describes the degree of which the source code of the program has been tested. It is one form of white box testing which finds the areas of the program not exercised by a set of test cases. It also creates some test cases to increase coverage and determining a quantitative measure of code coverage.

In most cases, code coverage system gathers information about the running program. It also combines that with source code information to generate a report about the test suite's code coverage.

- Statement Coverage
- Decision Coverage
- Branch Coverage
- Condition Coverage

Statement testing,

Statement Coverage

What is Statement Coverage?

Statement coverage is a white box test design technique which involves execution of all the executable statements in the source code at least once. It is used to calculate and measure the number of statements in the source code which can be executed given the requirements.

Statement coverage is used to derive scenario based upon the structure of the code under test.

$$\text{Statement Coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} \times 100$$

Scenario to calculate Statement Coverage for given source code. Here we are taking two different scenarios to check the percentage of statement coverage for each scenario.

Source Code:

```
Prints (int a, int b) {                    ----------  Printsum is a function
int result = a+ b;
   If (result> 0)
```

```
        Print ("Positive", result)
Else
        Print ("Negative", result)
}----------------------------End of the source code
```

Scenario 1:

If A = 3, B = 9

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

The statements marked in yellow color are those which are executed as per the scenario

Number of executed statements = 5, Total number of statements = 7

Statement Coverage: 5/7 = 71%

Likewise we will see scenario 2,

Scenario 2:

If A = -3, B = -9

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

The statements marked in yellow color are those which are executed as per the scenario.

Number of executed statements = 6

Total number of statements = 7

$$\text{Statement Coverage} = \frac{Number\ of\ executed\ statments}{Total\ number\ of\ statments}$$

Statement Coverage: 6/7 = 85%

What is covered by Statement Coverage?

1. Unused Statements
2. Dead Code
3. Unused Branches
4. Missing Statements

Branch testing,

Branch Coverage

In the branch coverage, every outcome from a code module is tested. For example, if the outcomes are binary, you need to test both True and False outcomes.

It helps you to ensure that every possible branch from each decision condition is executed at least a single time.

By using Branch coverage method, you can also measure the fraction of independent code segments. It also helps you to find out which is sections of code don't have any branches.

The formula to calculate Branch Coverage:

$$Branch\ Coverage = \frac{Number\ of\ Executed\ Branches}{Total\ Number\ of\ Branches}$$

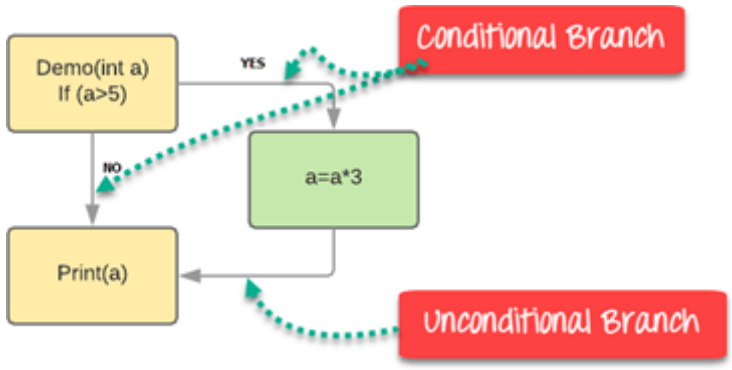
Example of Branch Coverage

To learn branch coverage, let's consider the same example used earlier

Consider the following code

```
Demo(int a) {
```

```
If (a> 5)
        a=a*3
Print (a)
}
```



Branch Coverage will consider unconditional branch as well

Test Case	Value of A	Output	Decision Coverage	Branch Coverage
1	2	2	50%	**33%**
2	6	18	50%	**67%**

Branch coverage Testing offers the following advantages:

- Allows you to validate-all the branches in the code
- Helps you to ensure that no branched lead to any abnormality of the program's operation
- Branch coverage method removes issues which happen because of statement coverage testing
- Allows you to find those areas which are not tested by other testing methods
- It allows you to find a quantitative measure of code coverage
- Branch coverage ignores branches inside the Boolean expressions

Condition testing ,

Condition Coverage

Conditional coverage or expression coverage will reveal how the variables or sub expressions in the conditional statement are evaluated. In this coverage expressions with logical operands are only considered.

For example, if an expression has Boolean operations like AND, OR, XOR, which indicated total possibilities.

Conditional coverage offers better sensitivity to the control flow than decision coverage. Condition coverage does not give a guarantee about full decision coverage

The formula to calculate Condition Coverage:

$$Condition\ Coverage = \frac{Number\ of\ Executed\ Operands}{Total\ Number\ of\ Operands}$$

Example:

```
1   IF (x < y) AND (a>b) THEN
```

For the above expression, we have 4 possible combinations

- TT
- FF
- TF
- FT

Consider the following input

X=3 Y=4	(x<y)	TRUE	Condition Coverage is ¼ = 25%
A=3 B=4	(a>b)	FALSE	

Path testing: DD paths,

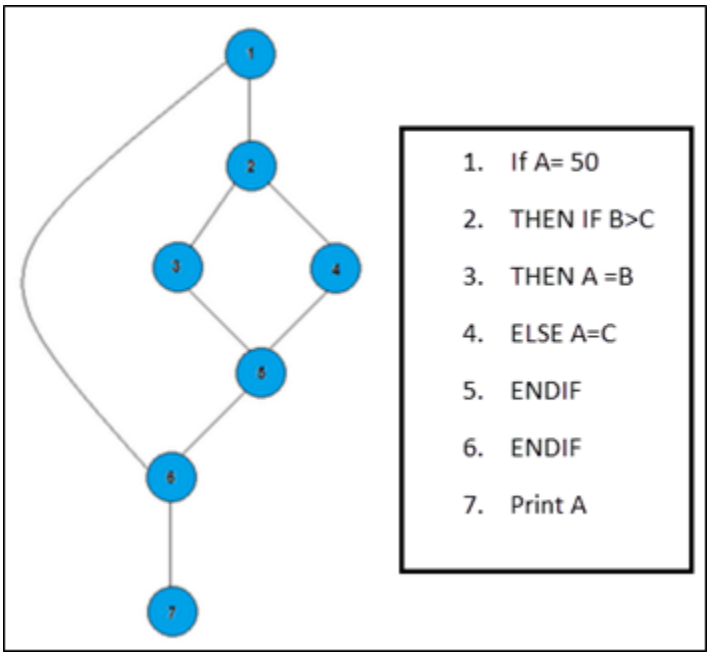
What is Path Testing?

Path testing is a structural testing method that involves using the source code of a program in order to find every possible executable path. It helps to determine all faults lying within a pieceof code. This method is designed to execute all or selected path through a computer program.

What is Basis Path Testing?

The basis path testing is same, but it is based on a White Box Testing method, that defines test cases based on the flows or logical path that can be taken through the program. In software engineering, Basis path testing involves execution of all possible blocks in a program and achieves maximum path coverage with the least number of test cases. It is a hybrid of branch testing and path testing methods.

The objective behind basis path in software testing is that it defines the number of independent paths, thus the number of test cases needed can be defined explicitly (maximizes the coverage of each test case).



In the above example, we can see there are few conditional statements that is executed depending on what condition it suffice. Here there are 3 paths or condition that need to be tested to get the output,

- **Path 1**: 1,2,3,5,6, 7

- **Path 2**: 1,2,4,5,6, 7
- **Path 3**: 1, 6, 7

Steps for Basis Path testing

The basic steps involved in basis path testing include

- Draw a control graph (to determine different program paths)
- Calculate Cyclomatic complexity (metrics to determine the number of independent paths)
- Find a basis set of paths
- Generate test cases to exercise each path

Advantages of Basic Path Testing

- It helps to reduce the redundant tests
- It focuses attention on program logic
- It helps facilitates analytical versus arbitrary case design
- Test cases which exercise basis set will execute every statement in a program at least once

Conclusion:

Basis path testing helps to determine all faults lying within a piece of code.

Program Graphs

Program graphs are a graphical representation of a program's source code. Thenodes of the program graph represent the statement fragments of the code, andthe edges represent the program's flow of control.

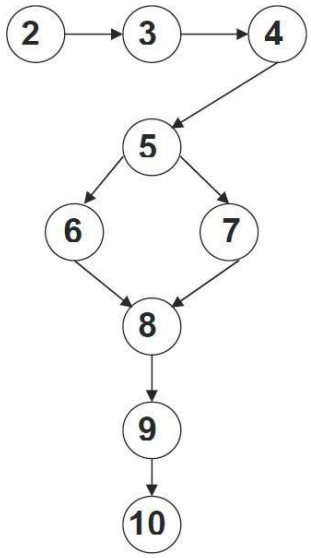
Figure 1.1 shows pseudocode for a simple program that simply subtractstwo integers and outputs the result to the terminal. The number subtracteddepends on which is the larger of the two; this stops a negative number frombeing output.

1. Program 'Simple Subtraction'
2. Input (x, y)
3. Output (x)
4. Output (y)
5. If x > y then DO
6. x – y = z
7. Else y – x = z
8. EndIf
9. Output (z)
10. Output "End Program"

Figure 1.1 Pseudocode for the simple subtraction program.

The construction of a program graph for this simple code is a basic task. Eachline number is used to enumerate the relevant nodes of the graph. It is notnecessary to include basic declarations and module titles in the program graph,and so line 1 of the pseudocode in Figure 1.1 will be ignored.

For a path to beexecutable it must start at line 2 of the pseudocode, and end at line 10. In thecorresponding program graph of this code in Figure 1.2, this is demonstrated bythe fact that every legal path must begin at the source node and end at the sinknode.



Due to the simplicity of our code example, it is a trivial task to find all of thepossible executable paths within the program graph shown in Figure 1.2. Startingat the source node and ending at the sink node, there exist two possible paths.

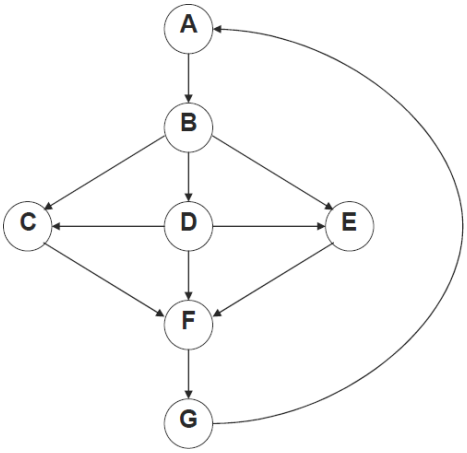
The first path would be the result of the If-Then clause being taken, and thesecond would be the result of the Else clause being taken.

A program graph provides us with some interesting details about thestructure of a piece of code. In the example graph of Figure 1.2, we can see thatnodes 2 through to 4 and nodes 9 to 10 are sequences.

This means that thesenodes represent simple statements such as variable declarations, expressions orbasic input/output commands. Nodes 5 through to 8 are a representation of an ifthen-else construct, while nodes 2 and 10 are the source and sink nodes of theprogram respectively.

By examining a program graph, a tester can garner an important piece ofinformation; is the program structured or unstructured? It is at this point that animportant distinction must be made between structure and simplicity.

Aprogram may contain thousands of lines of code and remain structured, whereasa piece of code only ten lines long may contain a loop that results in a loss ofstructure, and thus spores a potentially large number of execution paths. This isshown by the simple program graph in Figure1.3



Although containing fewer nodes than the program graph in Figure 1.2,this program graph would be much more complex to test, solely because it lacksstructure. This reason behind this lack of structure is due to the program graphcontaining a loop construct in which there exists internal branching. As a result,if the loop from node G to node A had 18 repetitions, it would see the number ofdistinct possible execution paths rise to 4.77 trillion [Jorgensen, 2002].

Thisdemonstrates how an unstructured program can lead to difficulties in evenfinding every possible path, while testing each path would be an infeasible task.From this we can conclude that when writing a program, a software engineershould attempt to keep it structured in order to make the testing process assimple as possible.

When studying the work of Thomas McCabe later in thispaper, we will be looking at how he has analysed program graphs and devised amethodology to retain a program's structure, thus keeping test cases to aminimum.

DD-Paths

The reason that program graphs play such an important role in structural testingis due to the fact that they form the basis of a number of testing methods,including one based on a construct known as decision-to-decision paths (morecommonly referred to as DD-Paths).

The idea is to use DD-Paths to create acondensation graph of a piece of software's program graph, in which a numberof constructs are collapsed into single nodes known as DD-Paths.

DD-Paths are chains of nodes in a directed graph that adhere to certaindefinitions. Each chain can be broken down into a different type of DD-Path, theresult of which ends up as being a graphof DD-Paths. The length of a chaincorresponds to the number of edges that the chain contains.

The definitions ofeach different type of DD-Path that a chain can be reduced to are given asfollows:

Type 1: A single node with an in-degree = 0.
Type 2: A single node with an out-degree = 0.
Type 3: A single node with in-degree >= 2 or out-degree >= 2.
Type 4: A single node with in-degree = 1 and out-degree = 1.
Type 5: The chain is of a maximal length >= 1.

All programs must have an entry and an exit and so every program graphmust have a source and sink node.

Type 1 and Type 2 are needed to provide uswith the capability of defining these key nodes as initial and final DD-Paths.

Type 3 deals with slightly more complex structured constructs that often appearin a program graph such as If-Then-Else statements and Case statements. Thisdefinition is particularly important, as it allows for branching to be dealt with inthe testing process, a concept that will be examined more closely when we cometo analyse test coverage metrics.

Type 4 allows for basic nodes such asexpressions and declarations to be defined as DD-Paths. As it is these types ofnodes that make up the main part of a program.

Type 5 is used to take chains ofthese nodes and condense them into a single node. It is important that we findthe final node within a chain in order to have the smallest number of nodes aspossible to test; it is for this reason that the definition of a Type 5 DD-Path mustexamine the maximal length of the chain.

In order to successfully demonstrate how the above definitions can beused to create a DD-Path graph, we will apply them to the program graph inFigure 1.2. The result of this application is a DD-Path graph of the simplesubtraction problem, as shown in Figure 1.4.

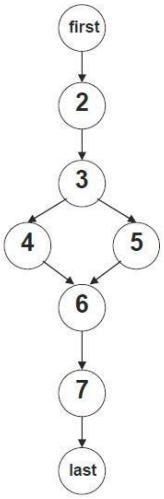


Figure 1.4 A DD-Path graph of the simple subtraction program

We can immediately identify some differences between the programgraph in Figure 1.1 and its DD-Path graph. The source and sink nodes of thegraph have been replaced by the words ‘first’ and ‘last’ in order to identify thenodes that conform to Type 1 and Type 2 DD-Paths. Perhaps more interestingly,there exists one less node.

This is due to the fact that nodes 3 and 4 in the originalprogram graph were a chain of maximal length >=1, and so they have beencondensed into a single node in the DD-Path graph

There also exist similarities between the two graphs. Node 7 remainsunchanged while the If- Then-Else construct is still visible. Nodes 3 and 6 obeythe Type 3 definition, while nodes 4 and 6are simply chains of length 1 and soare defined as Type 4 DD-Paths.

Having defined the concept of DD-Paths we can now see that theconstruction of a DD-Path graph presents testers with all possible linear codesequences. Test cases can be set up to execute each of these sequences, meaningall paths within the DD-Path graph of the program can be tested. As a result,DD-Paths can be used as a test coverage metric; software engineers know that ifthey can test every DD-Path then all faults within the DD-Path graph of aprogram are likely to be located.

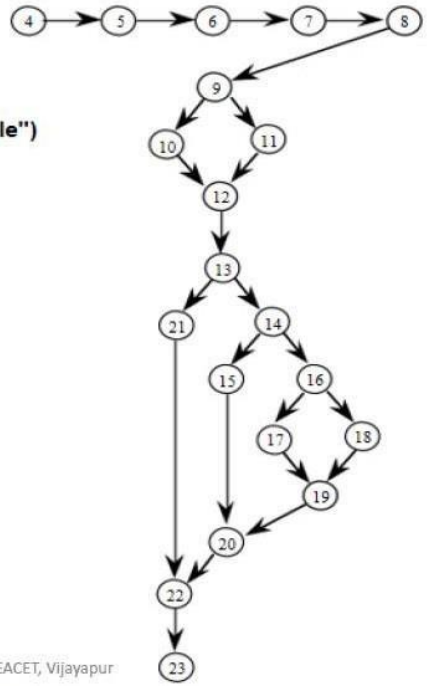
TRIANGLE PROBLEM

Triangle Program Specification
• Inputs: a, b, and c are non-negative integers, taken to be sides of a
triangle
• Output: type of triangle formed by a, b, and c
– Not a triangle
– Scalene (no equal sides)
– Isosceles (exactly 2 sides equal)
– Equilateral (3 sides equal)
• To be a triangle, a, b, and c must satisfy the triangle inequalities:

– a < b + c,
– b < a + c, and
– c < a + b

Program Graph for Triangle Problem

```
1.  Program triangle
2.   Dim a,b,c As Integer
3.  Dim IsATriangle As Boolean
    'Step 1:  Get Input
4.  Output("Enter 3 integers which are sides of a triangle")
5.   Input(a,b,c)
6.   Output("Side A is ",a)
7.   Output("Side B is ",b)
8.  Output("Side C is ",c)
    'Step 2:  Is A Triangle?
9.   If (a < b + c) AND (b < a + c) AND (c < a + b)
10.     Then IsATriangle = True
11.     Else IsATriangle = False
12   EndIf
    'Step 3:  Determine Triangle Type
13.  If IsATriangle
14.     Then   If (a = b) AND (b = c)
15.               Then Output ("Equilateral")
16.               Else   If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
17.                         Then   Output ("Scalene")
18.                         Else   Output ("Isosceles")
19.                      EndIf
20.            EndIf
21.     Else   Output("Not a Triangle")
22.  EndIf
23.  End triangle2
```

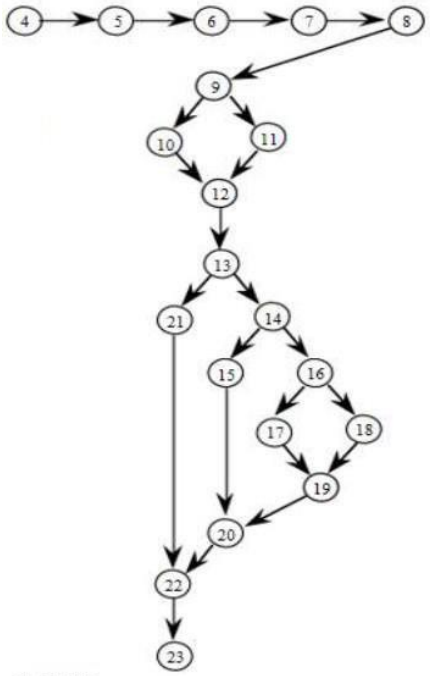


Trace for a = 5, b = 5, c = 5

Trace for a = 2, b = 5, c = 5

Trace for a = 3, b = 4, c = 5

Trace for a = 2, b = 3, c = 7



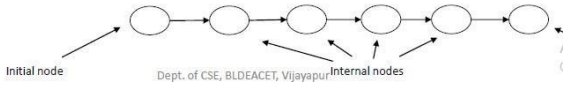
- Nodes 4 through 8 are sequences
- Nodes 9 through 12 are if-then-else construct
- Nodes 13 through 22 are nested if-then-else construct
- Node 4 – source node
- Node 23 – sink node
- No loops exist, so this is a directed acyclic graph

Importance of Program graph

- Program execution corresponds to paths from the source to the sink nodes.
- There is explicit description of the relationship between a test case and the part of the program it exercises.
- Can deal with the potentially large number of execution paths in a program.

DD-Paths

- The best known form of structural testing is based on a construct known as a *decision-to-decision* path.
- Concentrate only on decision nodes. Nodes which are in sequence are combined in single node.
- A DD-Path is a *chains* obtained from a program graph, where a chain is a path in which the initial and terminal nodes are distinct, and every interior node has indegree = 1, and outdegree = 1.
- DD-Paths are used to create *DD-Path Graphs*.
- Note that the initial node is 2-connected to every other node in the chain, and there are no instances of 1- or 3- connected nodes.
- An example of a chain is shown below:



DD-Paths

A DD-Path (decision-to-decision) is a chain in a program graph such that

Case 1: it consists of a single node with indegree = 0, (source)

Case 2: it consists of a single node with outdegree = 0, (sink)

Case 3: it consists of a single node with indegree >= 2 or outdegree >= 2,

Case 4: it consists of a single node with indegree = 1 and outdegree = 1,

Case 5: it is a maximal chain of length >= 1.

- Cases 1 and 2 establish the unique source and sink nodes of the program graph structured program as initial and final DD-Paths.
- Case 3 deals with complex nodes; it ensures that no node is contained in more than one DD-Paths.
- Case 4 is needed for short branches.
- Case 5 is the normal case, in which a DD-Paths is a single-entry, single-exit sequence of nodes(chain).

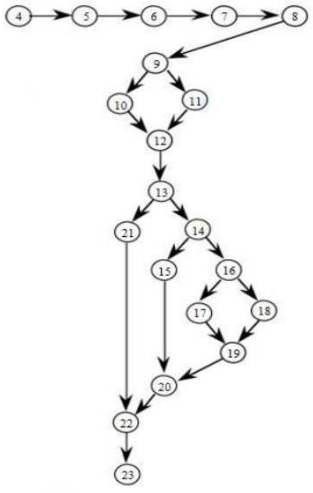
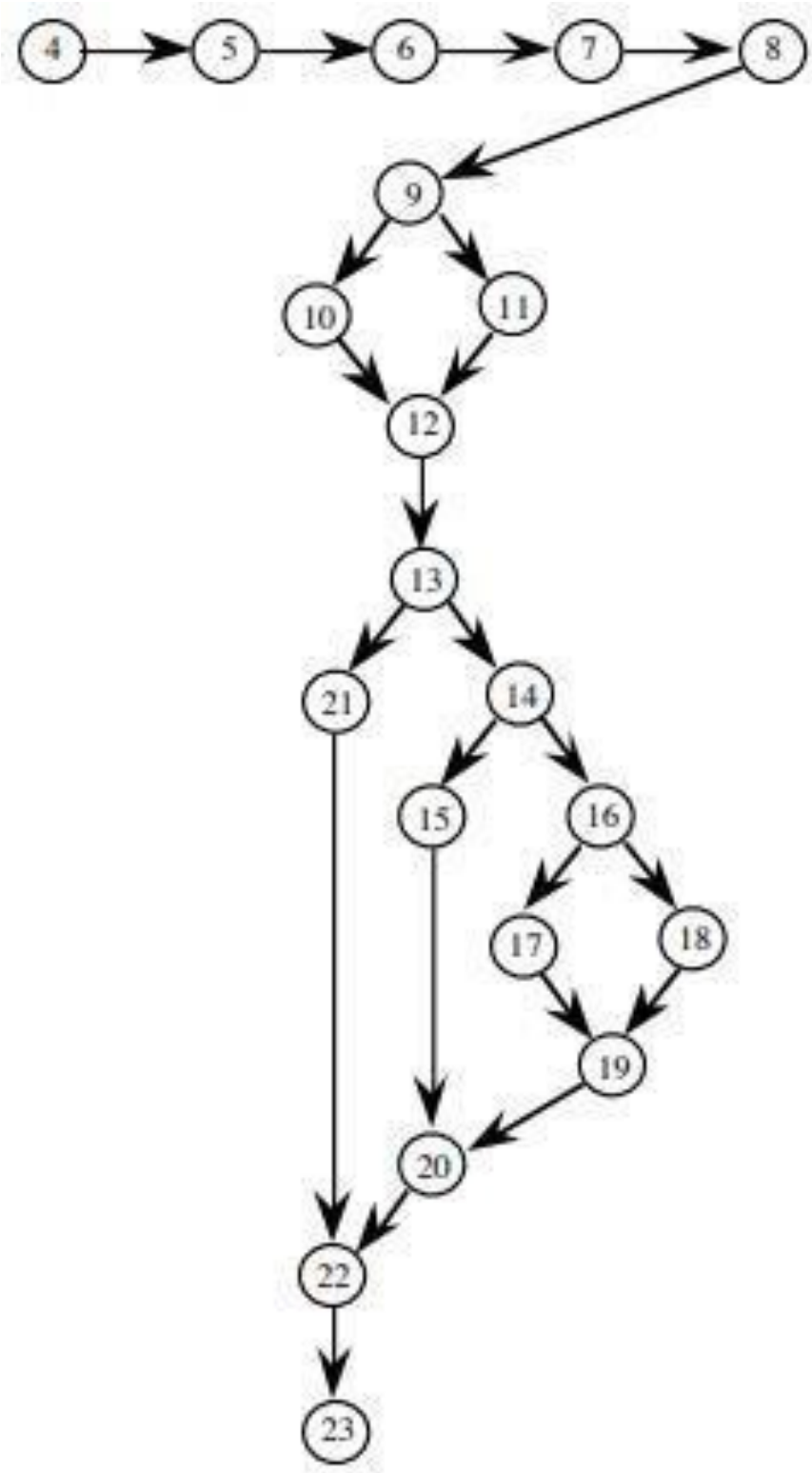
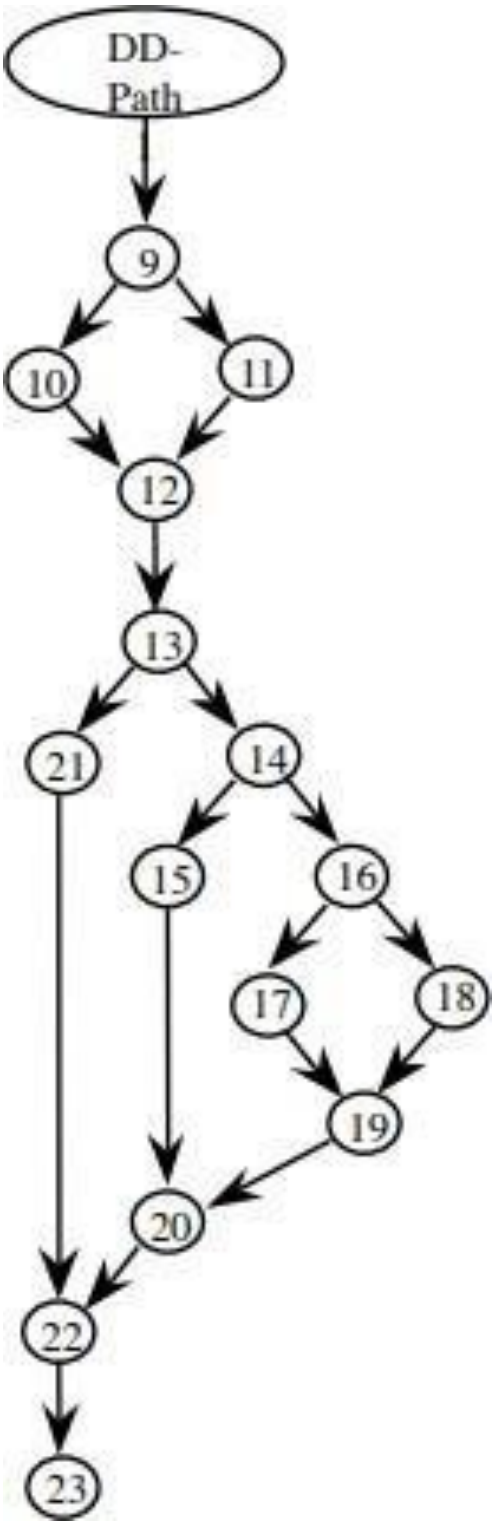


Table 9.1 Types of DD-Paths in Figure 9.1

Program Graph Nodes	*DD-Path Name*	*Case of Definition*
4	first	1
5–8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3
21	G	4
22	O	3
23	last	2


4
5
6
7
8
9
10
11
12
13
21
14
15
16
17
18
19
20
22
23


DD-
Path
9
10
11
12
13
21
14
15
16
17
18
19
20
22
23

Test coverage metrics,

Code-Based Test Coverage Metrics

• Used to evaluate a given set of test cases
• Often required by
– contract
– U.S. Department of Defense
– company-specific standards
• Elegant way to deal with the gaps and redundancies that are unavoidable with specification-based test cases.
• BUT
– coverage at some defined level may be misleading
– coverage tools are needed

Code-Based Test Coverage Metrics
(E. F. Miller, 1977 dissertation)
• C0: Every statement
• C1: Every DD-Path
• C1p: Every predicate outcome
• C2: C1 coverage + loop coverage
• Cd: C1 coverage +every pair of dependent DD-Paths
• CMCC: Multiple condition coverage
• Cik: Every program path that contains up to k repetitions of a loop (usually k = 2)
• Cstat: "Statistically significant" fraction of paths
• C∞: All possible execution paths

1. Statement and Predicate Testing
➢ The statement and predicate levels (C0 and C1
) collapse into one consideration.
➢ Statement coverage based testing aims to devise test cases that collectively exercise all statements in a program.
➢ This coverage metrics require that we find a set of test cases such that, when executed, every node of the program graph is traversed at least once.

2. DD-Path Testing (C1P)

➢When every DD-Path is traversed (C1 metric), each predicate outcome has beenexecuted; this amounts to traversing every edge in the DD-path graph. As opposed toonly every node.
➢For if-then and if-then-else statements, both the true and false branches arecovered(C1Pcoverage)
➢For CASE statement each clause is covered.

3. Dependent Pairs of DD-Paths (Cd
)
• In simple C1coverage criterion we are interested simply to traverse all edges in the DDPath graph.
• If we enhance this coverage criterion by ensuring that we also traverse dependent pairsof DD-Paths also we may have the chance of revealing more errors that are based ondata flow dependencies.
• More specifically, two DD-Paths are said to be dependent iff there is a define/referencerelationship between these DD-Paths, in which a variable is defined (receives a value)in one DD-Path and is referenced in the other.
• In Cdtesting we are interested on covering all edges of the DD-Path graph and alldependent 08-03-2018 DD-Path pairs.

For Ex: DD-Path graph of Triangle problem
➢ C and H are such pairs, as DD-Paths D and H pairs.
➢The variable IsATriangle is set to TRUE at node C and FALSE at node D.
➢ Node H is the branch taken when IsATriangle is TRUE in the condition at node B, so any path containing nodes D and H is infeasible.

4. Multiple Condition Coverege (CMCC)
• Now if we consider that the predicates P1 is a compound predicate (i.e. (A or B)) thenMultiple Condition Coverage Testing requires that each possible combination of inputsbe tested for each decision.
• Example: “if (A or B)” requires 4 test cases:
A = True, B = True
A = True, B = False
A = False, B = True
A = False, B = False
• The problem: For n conditions, 2ntest cases are needed, and this grows exponentiallywith n

For example, take the following statement:
If x == 2 || x == 6 && Boolean == true then Do

For example, take the following statement:

If x == 2 || x == 6 && Boolean == true then Do

X	X	Boolean	Validity
T	T	T	Invalid
T	T	F	Invalid
T	F	T	Valid
F	T	T	Valid
T	F	F	Valid
F	T	F	Valid
F	F	T	Valid
F	F	F	Valid

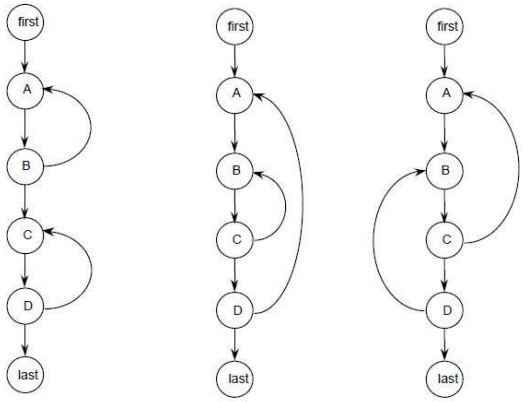
5. Loop Coverage (C2)

➢ Loops are highly fault-prone portion of source code.

➢ The simple view of loop testing coverage is that we must devise test cases thatexercise the two possible outcomes of the decision of a loop condition that is one totraverse the loop and the other to exit (or not enter) the loop.

➢An extension would be to consider a modified boundary value analysis approachwhere the loop index is given a minimum, minimum +, a nominal, a maximum -, anda maximum value or even robustness testing.

Concatenated, Nested, and Knotted Loops



Concatenated Loops are simply a sequence of disjoint loops.

➢ Concatenated loops occur when it is possible to leave one loop and immediately enter

into another.
➢If the iteration values of one loop affect those of another loop, they must be treated inthe same way as nested loops.
Nested Loops: one loop is present inside another loop.
➢ Nested loops can present difficulties to a software engineer.
➢Five tests for a single loop would be increased to 25 tests for a pair of nested loops, and125 tests for three nested loops.
➢This exponential increase of required tests means that nested loops should be avoided asa program construct.
➢However in some cases this construct may be unavoidable

When it is possible to branch into (or out from) the middle of a loop, and these branchesare internal to other loops, the result is Beizer's Knotted Loop (Horrible Loops).
➢Once a loop is tested, then the tester can collapse it into a single node to simplify thegraph for the next loop tests. In the case of nested loops we start with the inner mostloop and we proceed outwards.

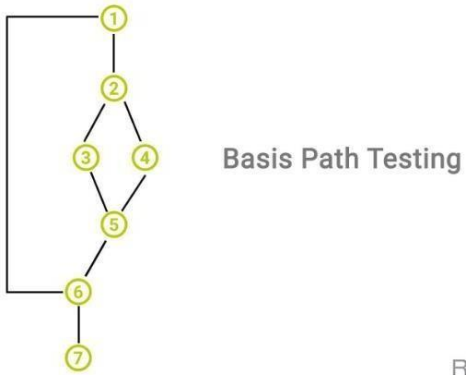
Basis path testing,

What Is Basis Path Testing?
Through utilizing a white box method, basis path testing can attain maximum path coverage using the minimum number of test cases.

Every possible block of code in a program is executed through the lowest number of test cases. It does this by identifying the number of independent paths so that the number of test cases required can be explicitly defined, thus maximizing the coverage of each test case.

Basis path testing is effective because it ensures full branch coverage without needing to cover all the possible paths. As already mentioned, this can be time-consuming and costly. Branch coverage is another testing method that aims to verify that every branch extending from every decision point is tested at least once. This way, all the branches in the code can be validated to make sure that none result in the application behaving abnormally. It so happens then that, basis path testing is considered to be a hybrid of path and branch testing methods.

Example
To illustrate how to implement the steps of basis path testing, we have included an example. Below is a flow diagram showing nodes for logical paths, statements, and conditionals changing the flow of execution.



This provides a simple example of what basis path testing looks like. There are a number of conditional statements that are executed depending on input parameters. In this case, there are 3 paths or conditions to be tested to determine the output:

Path 1: 1 2 3 5 6 7
Path 2: 1 2 4 5 6 7
Path 3: 1 6 7

Steps For Carrying Out Testing

As an overview, the steps for carrying out this testing method includes:

- Drafting a control flow graph to identify the possible program paths
- Calculating the number of independent paths through a process known as cyclomatic complexity which we discuss below
- Define the set of basis paths to be tested
- Generate test cases to evaluate the program flow for each path

Cyclomatic Complexity

Cyclomatic complexity is a software metric and another key process in implementing basis path testing. A software metric is a quantitative measurement of time, quality, size, and cost of an attribute of software.

In this case, cyclomatic complexity measures the complexity of a program by identifying all independent paths through which the processes flow.

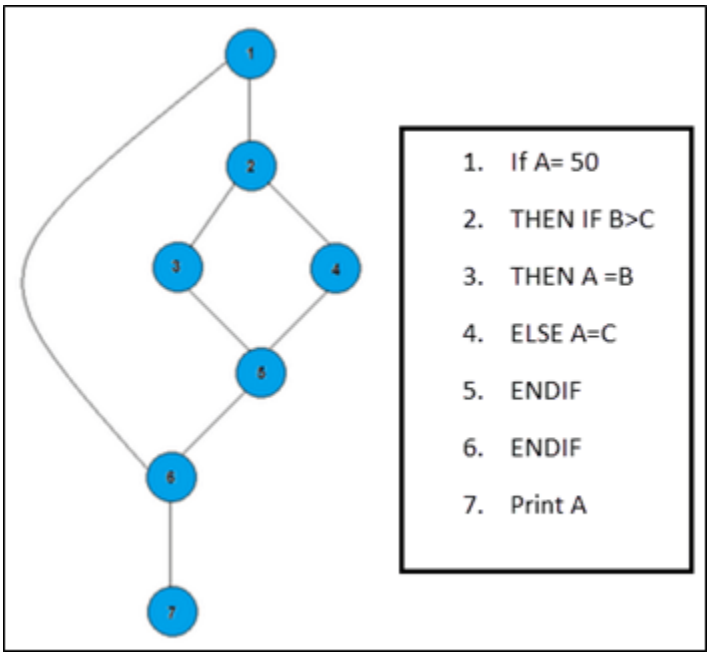
The metric is based on a control flow representation of a program and was developed in 1976 by Thomas McCabe. His model uses a flow graph that consists of nodes and edges to present a visualization of the control flow of a program. Nodes symbolize the processing tasks and edges control flow between the nodes. Nodes are the entry and exit points of processes in the program sequence while independent paths add a new process to the program flow. They have at least one edge which has not been followed in any other paths.

A mathematical representation of the cyclomatic complexity of program code can be calculated as follows:
V(G) = E – N + 2
Where
E = number of edges
N = number of nodes
V(G) = P + 1
Where
P = number of predicate nodes (nodes that contain conditions)
Once the number of paths or conditions has been calculated, the number of tests to be written is known. For example, 3 paths will mean that at least one test should be generated to cover each path.

V(G) = number of regions in graph

The properties of cyclomatic complexity are as follows:

- V(G) is the highest number of independent paths shown in the graph
- V(G) is always greater than or equal to 1
- If V(G) is equal to 1 then G will have one path
- Ideally, minimize the complexity score to 10 – the higher the score, the more complex the code



1. V(G) = e – n + 2p
e = 8
n = 7

p = 1
V(G) = 8 – 7 + 2(1) = 3

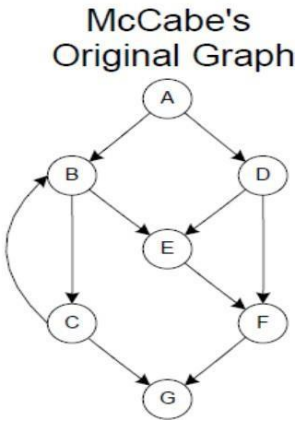
2. V(G) = predicate node + 1
predicate node = 2 (node 1 and node 2)
V(G) = 2 + 1 = 3

3. V(G) = number of regions in graph
number of regions = 3
V(G) = 3

Path/Edge Traversal

Path/Edge Traversed	1	2	3	4	5	6	7	8
Path 1: 1, 2, 3, 5, 6, 7	1		**1**		**1**	1		1
Path 2: 1, 2, 4, 5, 6, 7	1	**1**		**1**		1		1
Path 3: 1, 6, 7							**1**	1

McCabe's Example



V(G) = 10 - 7 + 2(1)
= 5

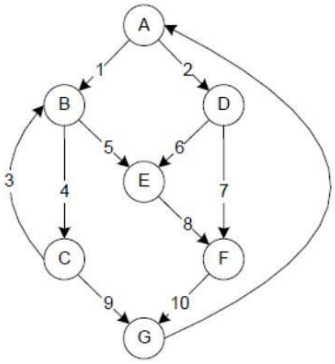
McCabe's Baseline Method

- Pick a "baseline" path that corresponds to normal execution. (The baseline should have as many decisions as possible.)
- To get succeeding basis paths, retrace the baseline until you reach a decision node. "Flip" the decision (take another alternative) and continue as much of the baseline as possible.
- Repeat this until all decisions have been flipped. When you reach V(G) basis paths, you're done.
- If there aren't enough decisions in the first baseline path, find a second baseline and repeat steps 2 and 3.

Following this algorithm, we get basis paths for McCabe's example.

Resulting basis paths

First baseline path
p1: A, B, C, G
Flip decision at C
p2: A, B, C, B, C, G
Flip decision at B
p3: A, B, E, F, G
Flip decision at A
p4: A, D, E, F, G
Flip decision at D
p5: A, D, F, G



Path/Edge Traversal

Path / Edges	*e1*	*e2*	*e3*	*e4*	*e5*	*e6*	*e7*	*e8*	*e9*	*e10*
p1: A, B, C, G	1	0	0	1	0	0	0	0	1	0
p2: A, B, C, B, C, G	1	0	**1**	2	0	0	0	0	1	0
p3: A, B, E, F, G	1	0	0	0	**1**	0	0	1	0	1
p4: A, D, E, F, G	0	1	0	0	0	**1**	0	1	0	1
p5: A, D, F, G	0	1	0	0	0	0	**1**	0	0	1
ex1: A, B, C, B, E, F, G	1	0	1	1	1	0	0	1	0	1
ex2: A, B, C, B, C, B, C, G	1	0	2	3	0	0	0	0	1	0

Find cyclomatic complexity of triangle problem
1. V(G) = e – n + 2p
e = 23
n = 20
p = 1
V(G) = 23 – 20 + 2(1) = 5
2. V(G) = predicate node +1
= 4 + 1 = 5
(Predicate nodes are 9, 13, 14, 16)
3. Nos. of regions = 5

Find cyclomatic complexity of triangle problem DD-Path graph
1. V(G) = e – n + 2p
e = 20
n = 17
p = 1
V(G) = 20 – 17 + 2(1) = 5
2. V(G) = predicate node +1
= 4 + 1 = 5
(Predicate nodes are B, F, H, J)
3. Nos. of regions = 5

Basis Paths

Original	p1: FIRST-A-B-C-E-F-H-J-K-M-N-O-LAST	Scalene
Flip p1 at B	p2: FIRST-A-B-D-E-F-H-J-K-M-N-O-LAST	Infeasible
Flip p1 at F	p3: FIRST-A-B-C-E-F-G-O-LAST	Infeasible
Flip p1 at H	p4: FIRST-A-B-C-E-F-H-I-N-O-LAST	Equilateral
Flip p1 at J	p5: FIRST-A-B-C-E-F-H-J-L-M-N-O-LAST	Isosceles

NOTE: there is no basis path for the Not A Triangle case.

Observations on McCabe's Basis Path Method
Problems with Basis Path
➢What is the significance of a path as a linear combination of basis paths?
➢What does 2p2 mean?
➢ Execute path p2 twice?
➢What does –p1 part mean?
➢ Execute path p1 backward?

➢ Undo the most recent execution on p1?
➢Don't do p1 next time?
➢In the path ex2 = 2p2 - p1 should a tester run path p2 twice, and then not do path p1 thenext time? This is theory run amok(uncontrollable).

Is there any guarantee that basis paths are feasible?
➢Is there any guarantee that basis paths will exercise interesting dependencies?

For Triangle problem, we can identify two rules:
If node C is traversed, then we must traverse node H.
If node D is traversed, then we must traverse node G.
Taken together, these rules, in conjunction with McCabe's baseline method, will yield thefollowing feasible basis path set.

p1: FIRST-A-B-C-E-F-H-J-K-M-N-O-LAST	Scalene
p6: FIRST-A-B-D-E-F-G-O-LAST	Not a Triangle
p4: FIRST-A-B-C-E-F-H-I-N-O-LAST	Equilateral
p5: FIRST-A-B-C-E-F-H-J-L-M-N-O-LAST	Isosceles

The bottom line for testers is:
➢ Programs with high cyclomatic complexity require more testing.
➢ The organizations that use the cyclomatic complexity, most set some guidelines formaximum acceptable complexity; V(G) = 10 is a common choice.
➢What happens if a unit test has a higher complexity?
➢Two possibilities: simplify the unit or plan to do more testing.
➢If the unit is well structured, its essential complexity is 1, so it can be simplified easily.
➢If the unit has an essential complexity that exceeds the guidelines, often the best choice isto eliminate the unstructured.

guidelines and observations,

Guidelines and Observations
In our study of functional testing, we observed that gaps and redundancies can both exist, and at the same time, cannot be recognized. The problem was that functional testing removes us "too far" from code. The path testing approaches to structural testing represent the case where the pendulum has swung too far the other way: moving from code to directed graph representations and program path formulations obscures important information that is present in the code, in particular the distinction between feasible and infeasible paths. In the next chapter, we look at dataflow based testing. These techniques move closer to the code, so the pendulum will swing back from the path analysis extreme.

McCabe was partly right when he observed: “It is important to understand that these are purely criteria that measure the quality of testing, and not a procedure to identify test cases” [McCabe 82]. He was referring to the DD-Path coverage metric (which is equivalent to the predicate outcome metric) and the cyclomatic complexity metric that requires at least the cyclomatic number of distinct program paths must be traversed. Basis path testing therefore gives us a lowerbound on how much testing is necessary.

Path based testing also provides us with a set of metrics that act as cross checks on functional testing. We can use these metrics to resolve the gaps and redundancies question. When we find that the same program path is traversed by several functional test cases, we suspect that this redundancy is not revealing new faults. When we fail to attain DD-Path coverage, we know that there are gaps in the functional test cases. As an example, suppose we have a program that contains extensive error handling, and we test it with boundary value test cases (rain, mi n+,nom, max-, and max). Because these are all permissible values, DD-Paths corresponding to the error handling code will not be traversed.

If we add test cases derived from robustness testing or traditional equivalence class testing, the DD-Path coverage will improve. Beyond this rather obvious use of coverage metrics, there is an opportunity for real testing craftsmanship.

The coverage metrics in Table 2 can operate in two ways: as a blanket mandated standard (e.g., all units shall be tested to attain full DD-Path coverage) or as a mechanism to selectively test portions of code more rigorously than others. We might choose multiple condition coverage for modules with complex logic, while those with extensive iteration might be tested in terms of the loop coverage techniques.

This is probably the best view of structural testing: use the properties of the source code to identify appropriate coverage metrics, and then use these as a cross check on functional testcases. When the desired coverage is not attained, follow interesting paths to identify additional (special value) test cases.

Data –Flow testing: Definition-Use testing,

Data flow testing is an unfortunate term, because most software developers immediately think about some connection with dataflow diagrams. Data flow testing refers to forms of structural testing that focus on the points at which variables receive values and the points at which these values are used (or referenced).

We will see that data flow testing serves as a “reality check” on path testing; indeed, many of the data flow testing proponents (and researchers) see this approach as a form of path testing. We will look at two mainline forms of data flow testing: one provides a set of basic definitions and a unifying structure of test coverage metrics, while the second is based on a concept called a “program slice”. Both of these formalize intuitive behaviors (and analyses) of testers, and

although they both start with a program graph, both move back in the direction of functional testing.

Most programs deliver functionality in terms of data. Variables that represent data somehow receive values, and these values are used to compute values for other variables. Since the early 1960s, programmers have analyzed source code in terms of the points (statements) at which variables receive values and points at which these values are used.

Many times, their analyses were based on concordances that list statement numbers in which variable names occur. Concordances were popular features of second generation language compilers (they are still popular with COBOL programmers). Early "data flow" analyses often centered on a set of faults that are now known as define/reference anomalies:

- a variable that is defined but never used (referenced)
- a variable that is used but never defined
- a variable that is defined twice before it is used

Define/Use Testing

Much of the formalization of define/use testing was done in the early 1980s [Rapps 85]; the definitions in this section are compatible with those in [Clarke 89], an article which summarizes most of define/use testing theory. This body of research is very compatible with the formulation we developed in chapters 4 and 9. It presumes a program graph in which nodes are statement fragments (a fragment may be an entire statement), and programs that follow the structured programming precepts.

The following definitions refer to a program P that has a program graph G(P), and a set of program variables V. The program graph G(P) is constructed as in Chapter 4, with statement fragments as nodes, and edges that represent node sequences. G(P) has a single entry node, and asingle exit node.

Definition

Node n G(P) is a ***defining node*** of the variable v V, written as DEF(v,n), iff the value of thevariable v is defined at the statement fragment corresponding to node n. Input statements, assignment statements, loop control statements, and procedure calls are all examples of statements that are defining nodes. When the code corresponding to such statements executes,the contents of the memory location(s) associated with the variables are changed.

Definition

Node n G(P) is a ***usage node*** of the variable v V, written as USE(v, n), iff the value of the variablev is used at the statement fragment corresponding to node n. Output statements, assignment statements, conditional statements, loop control statements, and procedure calls areall examples of statements that are usage nodes. When the code corresponding to such statementsexecutes, the contents of the memory location(s) associated with the variables remain unchanged.

Definition

A usage node USE(v, n) is a ***predicate use*** (denoted as P-use) iff the statement n is a predicate statement; otherwise USE(v, n) is a ***computation use*** , (denoted C-use). The nodes corresponding to predicate uses always have an outdegree ≥ 2, and nodes corresponding to computation uses always have outdegree ≤ 1.

Definition
A ***definition-use (sub)path***with respect to a variable v (denoted du-path) is a (sub)path in PATHS(P) such that, for some v V, there are define and usage nodes DEF(v, m) and USE(v, n) such that m and n are the initial and final nodes of the (sub)path.
Definition

A ***definition-clear (sub)path***with respect to a variable v (denoted dc-path) is a definition-use(sub)path in PATHS(P) with initial and final nodes DEF (v, m) and USE (v, n) such that no other node in the (sub)path is a defining node of v. Testers should notice how these definitions capture the essence of computing with stored data values. Du-paths and dc-paths describe the flow of data across source statements from points at which the values are defined to points at which the values are used. Du-paths that are not definition-clear are potential trouble spots.

Data-Flow Testing
• Data-flow testing uses the controlflowgraph to explore the unreasonablethings that can happen to data (i.e.,anomalies).
• Consideration of data-flow anomaliesleads to test path selection strategiesthat fill the gaps between complete pathtesting and branch or statement testing.

Data-Flow Testing (Cont'd)
• Data-flow testing is the name given to a familyof test strategies based on selecting pathsthrough the program's control flow in order toexplore sequences of events related to thestatus of data objects.
• E.g., Pick enough paths to assure that:
– Every data object has been initialized prior to itsuse.
– All defined objects have been used at least once

Data Object Categories
• (d) Defined, Created, Initialized
• (k) Killed, Undefined, Released
• (u) Used:
– (c) Used in a calculation
– (p) Used in a predicate

(d) Defined Objects
• An object (e.g., variable) is defined
when it:
– appears in a data declaration

– is assigned a new value
– is a file that has been opened
– is dynamically allocated

(u) Used Objects
• An object is used when it is part of acomputation or a predicate.
• A variable is used for a computation (c) whenit appears on the RHS (sometimes even theLHS in case of array indices) of anassignment statement.
• A variable is used in a predicate (p) when itappears directly in that predicate.

Example: Definition and Uses
1. read (x, y);
2. z = x + 2;
3. if (z < y)
4 w = x + 1;
else
5. y = y + 1;
6. print (x, y, w, z);

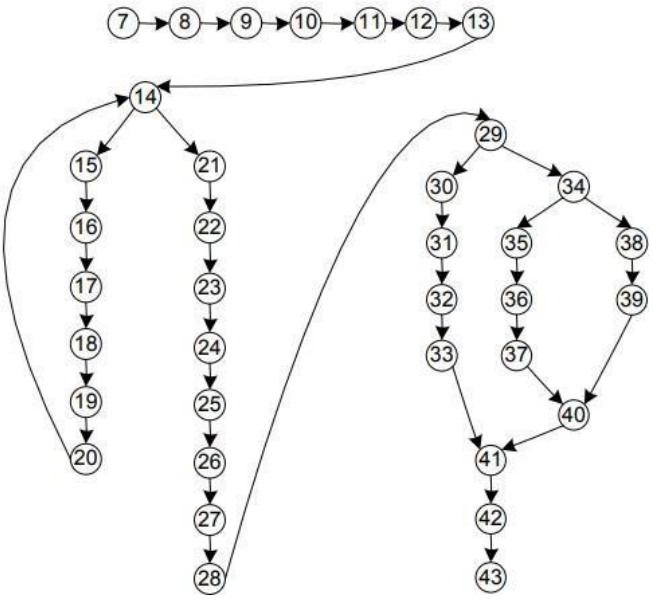
Example: Definition and Uses

	Def	C-use	P-use
1. read (x, y);	x, y		
2. z = x + 2;	z	x	
3. if (z < y)			z, y
4 w = x + 1;	w	x	
else			
5. y = y + 1;	y	y	
6. print (x, y, w, z);		x, y, w, z	

Example: first part of the Commission Program
1. Program Commission (INPUT,OUTPUT)
2. Dim locks, stocks, barrels As Integer
3. Dim lockPrice, stockPrice, barrelPrice As Real
4. Dim totalLocks, totalStocks, totalBarrels As Integer
5. Dim lockSales, stockSales, barrelSales As Real
6. Dim sales, commission As Real

```
7. lockPrice = 45.0
8. stockPrice = 30.0
9. barrelPrice = 25.0
10. totalLocks = 0
11. totalStocks = 0
12. totalBarrels = 0
13. Input(locks)
14. While NOT(locks = -1)
15. Input(stocks, barrels)
16. totalLocks = totalLocks + locks
17. totalStocks = totalStocks + stocks
18. totalBarrels = totalBarrels + barrels
19. Input(locks)
20. EndWhile
21. Output("Locks sold: ", totalLocks)
22. Output("Stocks sold: ", totalStocks)
23. Output("Barrels sold: ", totalBarrels)
23. Output("Barrels sold: ", totalBarrels)
24. lockSales = lockPrice * totalLocks
25. stockSales = stockPrice * totalStocks
26. barrelSales = barrelPrice * totalBarrels
27. sales = lockSales + stockSales + barrelSales
28. Output("Total sales: ", sales)
29. If (sales > 1800.0)
30. Then
31. commission = 0.10 * 1000.0
32. commission = commission + 0.15 * 800.0
33. commission = commission + 0.20 *(sales-1800.0)
34. Else If (sales > 1000.0)
35. Then
36. commission = 0.10 * 1000.0
37. commission = commission + 0.15 *(sales-1000.0)
38. Else
39. commission = 0.10 * sales
40. EndIf
41. EndIf
42. Output("Commission is $", commission)
43. End Commission
```

Commission Program Graph



Selected Def and Use Nodes

Variable	*Defined at Node*	*Used at Node*
lockPrice	7	24
totalLocks	10, 16	16, 21, 24
locks	13, 19	14, 16
lockSales	24	27
sales	27	28, 29, 33, 34, 37, 38
commission	31, 32, 33, 36, 37, 38	32, 33, 37, 41

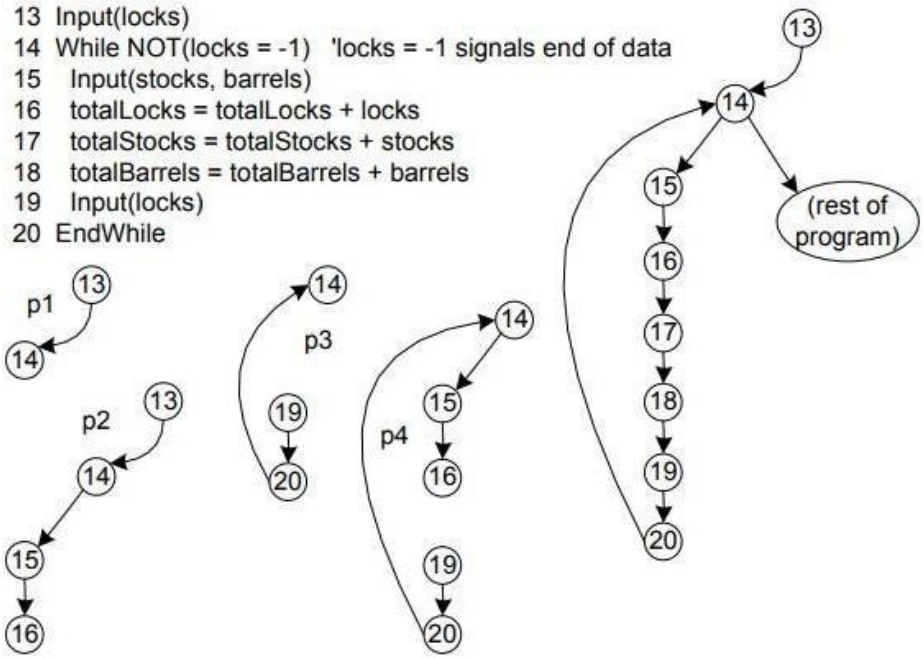
Exercise: Identify Def and Use Nodes

Variable	*Defined at Node*	*Used at Node*
stockPrice		
totalStocks		
locks	13, 19	14, 16
stockSales		
sales	27	28, 29, 33, 34, 37, 38
commission	31, 32, 33, 36, 37, 38	32, 33, 37, 41

Define/Use Paths (du-paths) for locks

- p1 = <13, 14>
- p2 = <13, 14, 15, 16>
- p3 = <19, 20, 14>
- p4 = <19, 20, 14, 15, 16>

- (all are definition clear)



Rapps-Weyuker Metrics

Associated with the concepts discussed in the previous section are a set of testcoverage metrics, also defined by Sandra Rapps and Elaine Weyuker in theearly 1980s [2]. The metrics – a set of criteria, essentially – allow the tester toselect sets of paths through the program, where "the number of paths selectedis always finite, and chosen in a systematic and intelligent manner in order tohelp us uncover errors".

Paths through the program are selected, and test data – to be input intothe program – is also selected to cover these paths (the percentage of coverageaccording to the set of paths selected).

Having the set of paths contain allpossible paths of the program (known as the All-Pathscriterion, according tothe Rapps/Weyuker nomenclature) is often infeasible, as the number of loopspossible through the program – and therefore the number of potential pathsto test – can often be infinite.

Nine criteria have been defined in the literature. Three correspond to themetrics used in path testing, where the paths selected are not chosen accordingto their variables and their attributes, but rather by an analysis of the structureof the program. These metrics are known as All-Paths (which has already beenmentioned above), All-Edges and All-Nodes. All-Paths, which corresponds tothe concept of 'path coverage', is satisfied if every path of the program graphis covered in the set.

All-Edges, which corresponds to 'branch coverage', issatisfied if every edge (branch) of the program graph is covered. All-Nodes,which corresponds to 'statement coverage', is satisfied if every node is coveredby the set of paths. In addition to these metrics, six new metrics were defined:

All-DU-Paths, All-Uses, All-C-Uses/Some-P-Uses, All-P-Uses/Some-C-Uses, AllDefs and All-P-Uses. Definitions (adapted from the definitions in [2, 1]) ofthese metrics are provided below:

• The set of paths satisfies All-Defs for P if and only if, within the set ofpaths chosen, every defining node for each variable in the program hasa definition-clear path to a usage node for the same variable, within theset of paths chosen.

• The set of paths satisfies All-P-Uses for P if and only if, within the set ofpaths chosen, every defining node for each variable in the program hasa definition-clear path to every P-use node for the same variable.

• The set of paths satisfies All P-Uses/Some C-Uses for P if and only if,within the set of paths chosen, every defining node for each variable inthe program has a definition-clear path to every P-use node for the same

variable: however, if there are no reachable P-uses, the definition-clearpath leads to at least one C-use of the variable.

• The set of paths satisfies All C-Uses/Some P-Uses for P if and only if,within the set of paths chosen, every defining node for each variable inthe program has a definition-clear path to every C-use node for the samevariable: however, if there are no reachable C-uses, the definition-clear path leads to at least one P-use of the variable.

• The set of paths satisfies All-Uses for P if and only if, within the set ofpaths chosen, every defining node for each variable in the program hasa definition-clear path to every usage node for the same variable.

• The set of paths satisfies All-DU-Paths for P if and only if, the set of pathschosen contains every feasible DU-path for the program.

Different criteria are supplied so that the tester can make what is describedby Rapps andWeyuker as a "tradeoff" [2]. Although, in an ideal world, a program would be tested as thoroughly and 'completely' as possible – for example, with respect to structural testing, each and every possible combinations ofnodes, branches, conditions, etc. would be tested thoroughly with every feasible combination of test data – in reality, a number of factor impede on this.

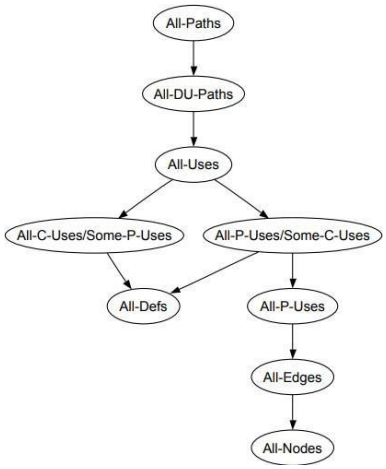
Forinstance: time constraints; financial constraints; a situation where all 'major'areas of the system under test have been deemed to have been tested satisfactorily; or even the level of criticality – is the program's stability and reliabilitya critical factor (for instance, would lives be threatened if an error occurredin the program?

Yes, if the program is controlling an aeroplane; no, if theprogram is controlling the in-flight games system for passengers!). Rapps andWeyuker have defined their "strongest" criterion to be All-DU-Paths; Jorgensenstates that "the generally accepted minimum [is] All-Edges" [1].

Rapps and Weyuker noted that there was a relationship between the different metrics: certain metrics expanded upon other metrics – that is, if a set ofpaths satisfied a certain metric, then it also satisfied all the other metrics belowit (for example, if All-Paths is satisfied, then so are All- DU-Paths and All-Uses).

A diagram, created by Rapps and Weuyker, showing the relationship betweenmetrics is shown in figure 4. This relationship was later described by Clarke etal. [3] as "subsumption".

In the diagram (figure 4), the arrows show the relationship between metrics. For example, All-Paths subsumes (or is stronger than) All-DU-Paths. However, Rapps and Weyuker describe that, during the development of the metrics,they had found that All-Defs is "not necessarily" stronger than All-Edges and



Slicebased testing,

```
1 program Example()
2 varstaffDiscount, totalPrice, finalPrice, discount, price
3 staffDiscount = 0.1
4 totalPrice = 0
5 input(price)
6 while(price != -1) do
7 totalPrice = totalPrice + price
8 input(price)
9 od
10 print("Total price: " + totalPrice)
11 if(totalPrice> 15.00) then
12 discount = (staffDiscount * totalPrice) + 0.50
13 else
14 discount = staffDiscount * totalPrice
15 fi
16 print("Discount: " + discount)
17 finalPrice = totalPrice - discount
```

Program Slices

The concept of program slicing was first proposed by Mark Weiser in the early1980s [6, 7]. According to Weiser, “slicing is a source code transformationof a program” [6], which allows a subset of a program, corresponding to aparticular behaviour, to be looked at individually.

This gives the benefit that a“programmer maintaining a large, unfamiliar program” does not have to understand “an entire system to change only a small piece” [6].

The conceptof program slicing was extended to cover software maintenance by Keith Gallagher and James Lyle in 1991 [8], extending slices to become “independentof line numbers”. Amended definitions of the program slice concept are givenin Paul Jorgensen’s book [1].

A program slice with respect to a variable at a certain point in the program,is the set of program statements from which the value of the variable at thatpoint of the program is calculated.

This definition can be amended to encompass the program graph concept: by replacing the set of program statementswith nodes of the program graph.

This allows the tester to find the list ofusage nodes from the graph, and then generate slices with them.

Program slices use the notation S(V, n), where S indicates that it is a program slice, V is the set of variables of the slice and n refers to the statementnumber (i.e. the node number with respect to the program graph) of the slice.

So, for example, with respect to the price variable given in the examplein section 2, the following are slices for each use of the variable:

• S(price, 5) = {5}
• S(price, 6) = {5, 6, 8, 9}
• S(price, 7) = {5, 6, 8, 9}
• S(price, 8) = {8}

To generate the slice S(price, 7), the following steps were taken:

• Lines 1 to 4 have no bearing on the value of the variable at line 7 (and,for that matter, for no other variable at any point), so they are not addedto the slice.

• Line 5 contains a defining node of the variable price that can affect thevalue at line 7, so 5 is added to the slice.

• Line 6 can affect the value of the variable as it can affect the flow ofcontrol of the program. Therefore, 6 is added to the slice.

• Line 7 is not added to the slice, as it cannot affect the value of the variable at line 7 in any way.

• Line 8 is added to the slice – even though it comes after line 7 in theprogram listing. This is because of the loop: after the first iteration ofthe loop, line 8 will be executed before the next execution of line 7. Theprogram graph in figure 1 shows this in a clear way.

• Line 9 signifies the end of the loop structure. This affects the flow ofcontrol (as shown in figure 1, the flow of control goes back to node 6).

This indirectly affects the value of price at line 7, as the value stored inthe variable will have almost certainly been changed at line 8. Therefore,9 is added to the slice.

• No other line of the program can be executed before line 7, and so cannotaffect the value of the variable at that point. Therefore, no other line isadded to the slice.

The program slice, as already mentioned, allows the programmer to focusspecifically on the code that is relevant to a particular variable at a certainpoint. However, the program slice concept also allows the programmer togenerate a lattice of slices: that is, a graph showing the subset relationshipbetween the different slices. For instance, looking at the previous example for the variable price, the slices S(price, 5) and S(price, 8) are subsets of S(price,7).

With respect to a program as a whole, certain variables may be related tothe values of other variables: for instance, a variable that contains a valuethat is to be returned at the end of the execution may use the values of othervariables in the program. For instance, in the main example in this document,

thefinalPrice variable uses the totalPrice variable, which itself usesthe price variable. The finalPrice variable also uses the discount variable, which uses the staffDiscount and totalPrice variables – and so on.

Therefore, the slices of the totalPrice and discount variables are a subset of theslice of the finalPrice variable at lines 17 and 18, as they both contribute to thevalue. This subset relationship‘ripples down’ to the other variables, accordingto the use-relationship described

Lattice of Slices

• Because a slice is a set of statement fragmentnumbers, we can find slices that are subsets ofother slices.
• This allows us to “work backwards” from points ina program, presumably where a fault is suspected.
• The statements leading to the value of commissionwhen it is output are an excellent example of thispattern.
• Some researchers propose that this is the waygood programmers think when they debug code.

This is shown visually in the following example:

• S(staffDiscount, 3) = {3}
• S(totalPrice, 4) = {4}
• S(totalPrice, 7) = {4, 5, 6, 7, 8}

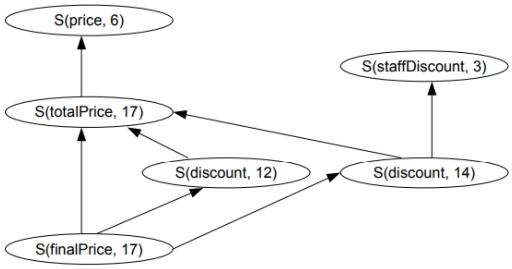


Figure 5: The Program Slice Lattice

• S(totalPrice, 11) = {4, 5, 6, 7, 8}
• S(discount, 12) = {3, 4, 5, 6, 7, 8, 11, 12}
• S(discount, 14) = {3, 4, 5, 6, 7, 8, 13, 14}
• S(finalPrice, 17) = {3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 17}

Therefore, the lattice of slices for the finalPrice variable is as shown infigure 5. This relationship, as shown in the lattice diagram, can feasibly helpduring testing, particularly if there's a fault. For instance, if there is an error inthe slice of finalPrice, then, by testing the different subset slices, you caneliminate them from the possible sources of the error (for instance, the errormay be generated from an incorrect calculation of the discount, for instance).

If there is no error in the subset slices, then the error must be found in theremaining lines of code. As it is a set of statement fragments, this means thatthe remaining lines of code are the relative complement of the slice.

In otherwords, the error is likely to be in:Fullslice − SubsetSlices
If there is an error, then there could be errors in either the subsets, thecode or both.

The relationship between slices also shows the interactions between variables in the code: if a slice for a variable x is a subset of a slice for a variabley, then the value of x must be needed by y. By generating the lattice, thetester can hopefully discover any unnecessary or undesired interactions between variables.

In the program fragment
13. Input(locks)
14. While NOT(locks = -1)
15. Input(stocks, barrels)
16. totalLocks = totalLocks + locks
17. totalStocks = totalStocks + stocks
18. totalBarrels = totalBarrels + barrels
19. Input(locks)
20.EndWhile
There are these slices on locks (notice that

statements 15, 17, and 18 do not appear):
S1: S(locks, 13) = {13}
S2: S(locks, 14) = {13, 14, 19, 20}
S3: S(locks, 16) = {13, 14, 19, 20}
S4: S(locks, 19) = {19}

Guidelines and observations.

Guidelines and Observations
Dataflow testing is clearly indicated for programs that are computationally intensive. As a corollary, in control intensive programs, if control variables are computed (P-uses), dataflow testing is also indicated. The definitions we made for define/use paths and slices give us very precise ways to describe parts of a program that we would like to test. There are academic tools that support these definitions, but they haven't migrated to the commercial marketplace. Some pieces are there; you can find programming language compilers that provide on-screen highlighting of slices, and most debugging tools let you "watch" certain variables as you step through a program execution.

Test Execution: Overview of test execution,
Whereas test design, even when supported by tools, requires insight and ingenuity in similar measure to other facets of software design, test execution must be sufficiently automated for frequentreexecution without little human involvement. This chapter describes approaches for creating the run-time support for generating and managing test data, creating scaffolding for test execution, and automatically distinguishing between correct and incorrect test case executions.

from test case specification to test cases,

Test Case Specification document described detailed summary of ***what scenarios will be tested, how they will be tested, how often they will be tested***, and so on and so forth, for a given feature. It specifies the purpose of a specific test, identifies the required inputs and expected results, provides step-by-step procedures for executing the test, and outlines the pass/fail criteria for determining acceptance.
Test Case Specification has to be done separately for each unit. Based on the approach specified in the test plan, the feature to be tested for each unit must be determined. The overall approach stated in the plan is refined into specific test techniques that should be followed and into the criteria to be used for evaluation. Based on these the test cases are specified for the testing unit.
However, a ***Test Plan is a collection of all Test Specifications*** for a given area. The Test Plan contains a high-level overview of what is tested for the given feature area.

Reason for Test Case Specification:

There are two basic reasons test cases are specified before they are used for testing:

1. *Testing has severe limitations and the effectiveness of testing depends heavily on the exact nature of the test case. Even for a given criterion the exact nature of the test cases affects the effectiveness of testing.*
2. *Constructing a good **Test Case** that will reveal errors in programs is a very creative activity and depends on the tester. It is important to ensure that the set of test cases used is of high quality. This is the primary reason for having the test case specification in the form of a document.*
   The Test Case Specification is developed in the Development Phase by the organization responsible for the formal testing of the application.

What is Test Case Specification Identifiers?

The way to uniquely identify a test case is as follows:

- ***Test Case Objectives:*** Purpose of the test
- **Test Items*:*** Items (e.g., requirement specifications, design specifications, code, etc.) required to run a particular test case. This should be provided in "Notes" or "Attachment" feature. It describes the features and conditions required for testing.
- **Input Specifications*:*** Description of what is required (step-by-step) to execute the test case (e.g., input files, values that must be entered into a field, etc.). This should be provided in "Action" field.
- **Output Specifications*:*** Description of what the system should look like after the test case is run. This should be provided in the "Expected Results" field.
- **Environmental Needs**: Description of any special environmental needs. This includes system architectures, Hardware & Software tools, records or files, interfaces, etc.

To sum up, ***Test Case Specification*** defines the exact set up and inputs for one Test Case.

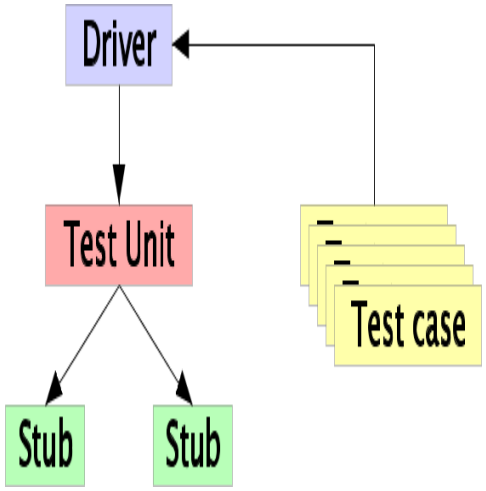
What is a Test Case?

A **TEST CASE** is a set of actions executed to verify a particular feature or functionality of your software application. A Test Case contains test steps, test data, precondition, postcondition developed for specific test scenario to verify any requirement. The test case includes specific variables or conditions, using which a testing engineer can compare expected and actual results to determine whether a software product is functioning as per the requirements of the customer.

Scaffolding,

Test Scaffolding

The *test scaffolding* denotes the auxilliary programs and classes that allow us to test a given program unit

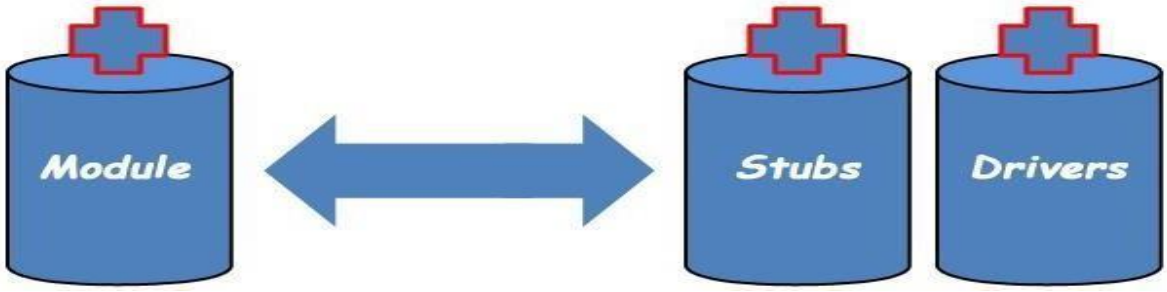


- The test units depends on a number of other units
  - These units are established as *stubs* possibly in the form of *mockups*
- The test cases are executed by means of *driver* program

It appears attractive to test the classes bottom up in order to avoid excessive use of stubs

Stubs and Drivers: Introduction

In **software testing life cycle**, there are numerous components that play a prominent part in making the process of testing accurate and hassle free. Every element related to testing strives to improve its quality and helps deliver accurate and expected results and services that are in compliance with the defined specifications. **Stubs and drivers** are two such elements used in software testing process, which act as a temporary replacement for a module. These are an integral part of software testing process as well as general software development. Therefore, to help you understand the significance of **stubs and drivers** in software testing, here is elaborated discussion on the same.



What is meant by Stubs and Drivers?

In the field of **software testing**, the term **stubs and drivers** refers to the replica of the modules, which acts as a substitute to the undeveloped or missing module. The stubs and drives are specifically developed to meet the necessary requirements of the unavailable modules and are immensely useful in getting expected results.

Stubs and drivers are two types of **test harness**, which is a collection of software and **test** that is configured together in order to test a unit of a program by stimulating variety of conditions while constantly monitoring its outputs and behaviour. Stubs and drivers are used in top-down integration and bottom-up integration testing respectively and are created mainly for the testing purpose.

Defining Stubs:

Stubs are used to test modules and are created by the team of testers during the process of **Top-Down Integration Testing**. With the assistance of these test stubs testers are capable of stimulating the behaviour of the lower level modules that are not yet integrated with the software. Moreover, it helps stimulates the activity of the missing components.

Types of Stubs:

There are basically four types of stubs used in top-down approach of integration testing, which are mentioned below:

- Displays the trace message.
- Values of parameter is displayed.
- Returns the values that are used by the modules.
- Returns the values selected by the parameters that were used by modules being tested.

Defining Drivers:

Drivers, like stubs, are used by software testers to fulfil the requirements of missing or incomplete components and modules. These are usually complex than stubs and are developed during **Bottom-Up approach of Integration Testing**. Drivers can be utilized to test the lower levels of the code, when the upper level of codes or modules are not developed. Drivers act as pseudo codes that are mainly used when the stub modules are ready, but the primary modules arenot ready.

Stubs and Drivers: Example

Consider an example of a web application, which consists of 4 modules i.e., **Module-A**, **Module-B**, **Module-C** and **Module-D**. Each of the following modules is responsible for some specific activity or functionality, as under:

Consider an example of a web application, which consists of 4 modules i.e., Module-A, Module-B, Module-C and Module-D. Each of the following modules is responsible for some specific activity or functionality, as under

Module-A ?Login page of the web application.
Module-B → Home page of the web application.
Module-C → Print Setup.
Module-D → Log out page.
modules A, B, C & D involves the interdependencies of each module over other.

It is always preferable, to perform testing, in parallel, to the development process. Thus, it implies that subsequent testing must be carried out, immediately after the development of the each module.

Module-A will be tested, as soon as, it develops. However, to carry out and validate the testing procedures in respect of module-A, there urges the need of **Module-B**, which is not yet developed. The expected functionality of the login page **(module-A)** could be validated, only if itis directed to the home page **(Module-B)**, based on the valid and correct inputs.

But, on the non-availability of the**Module-B**, it will not be possible to test **module-A**. These types of circumstances, introduces the stubs & drivers in the **process of software testing**. A dummy module, representing the basic functionality or feature of the module-B, is being developed, and thereafter, it is being integrated with the module-A, to perform testing, efficiently.

Similarly, **stubs and drivers**, are used to fulfil the requirements of other modules, such asLog out page (Module-D), needs to be directed to the login page **(Module-A)**, after successfully logging out from the application. In the event of unavailability of **Module-A**, stubs and drivers will work as a substitute for it, in order to carry out the testing of module-D.

Stubs vs Drivers
Stubs are dummy modules that always used to simulate the low level modules.
Stubs are the called programs.
Stubs are used when sub programs are under construction.
Stubs are used in top down approach
Drivers are dummy modules that always used to simulate the high level modules.
Drivers are the calling programs.
Drivers are only used when main programs are under construction.
Drivers are used in bottom up integration.

Generic versus specific scaffolding,

Generic versus Specific Scaffolding
The simplest form of scaffolding is a driver program that runs a single, specific test case. If, for example, a test case specification calls for executing method calls in a particular sequence, this is easy to accomplish by writing the code to make the method calls in that sequence.

Writing hundreds or thousands of such test-specific drivers, on the other hand, may be cumbersome and a disincentive to thorough testing. At the very least one will want to factor out some of the common driver code into reusable modules.

Sometimes it is worthwhile to write more generic test drivers that essentially interpret test case specifications. At least some level of generic scaffolding support can be used across a fairly wide class of applications.

Test oracles,

Test Oracles

It is little use to execute a test suite automatically if execution results must be manually inspected to apply a pass/fail criterion. Relying on human intervention to judge test outcomes is not merely expensive, but also unreliable.

Even the most conscientious and hard-working person cannot maintain the level of attention required to identify one failure in a hundred program executions, little more one or ten thousand.

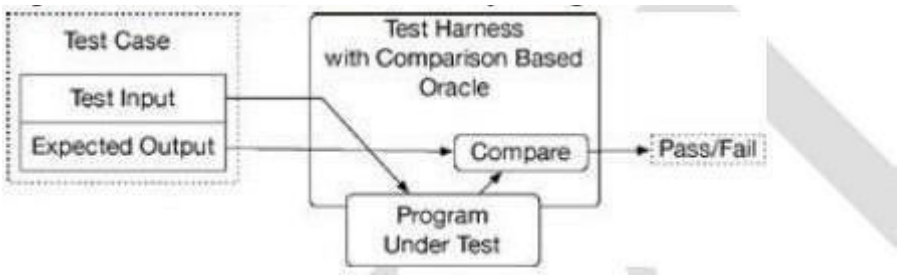
That is a job for a computer. Software that applies a pass/fail criterion to a program execution is called a *test oracle*, often shortened to *oracle*.

In addition to rapidly classifying a large number of test case executions, automated test oracles make it possible to classify behaviors that exceed human capacity in other ways, such aschecking real-time response against latency requirements or dealing with voluminous output data in a machine-readable rather than human-readable form.

Capture-replay testing, a special case of this in which the predicted output or behavior ispreserved from an earlier execution, is discussed in this chapter. A related approach is to capture the output of a trusted alternate version of the program under test.

For example, one may produce output from a trusted implementation that is for some reason unsuited for production use; it may too slow or may depend on a component that is not available in the production environment.

It is not even necessary that the alternative implementation be *more* reliable than the program under test, as long as it is sufficiently different that the failures of the real and alternate version are likely to be independent, and both are sufficiently reliable that not too much time is wasted determining which one has failed a particular test case on which they disagree.



independently compute the route to ascertain that it is in fact a valid route that starts at *A* and ends at *B.*

Oracles that check results without reference to a predicted output are often partial, in the sense that they can detect some violations of the actual specification but not others.

They check necessary but not sufficient conditions for correctness. For example, if the specification calls for finding the optimum bus route according to some metric, partial oracle a validity check is only a partial oracle because it does not check optimality.

Similarly, checking that a sort routine produces sorted output is simple and cheap, but it is only a partial oracle because the output is also required to be a permutation of the input.

A cheap partial oracle that can be used for a large number of test cases is often combined with a more expensive comparison-based oracle that can be used with a smaller set of test cases for which predicted output has been obtained.

Ideally, a single expression of a specification would serve both as a work assignment and as a source from which useful test oracles were automatically derived. Specifications are often incomplete, and their informality typically makes automatic derivation of test oracles impossible.

The idea is nonetheless a powerful one, and wherever formal or semiformal specifications (including design models) are available, it is worth- while to consider whether test oracles can be derived from them.

(Oracles) Testing of Copy/Paste in Office Suites

Summary

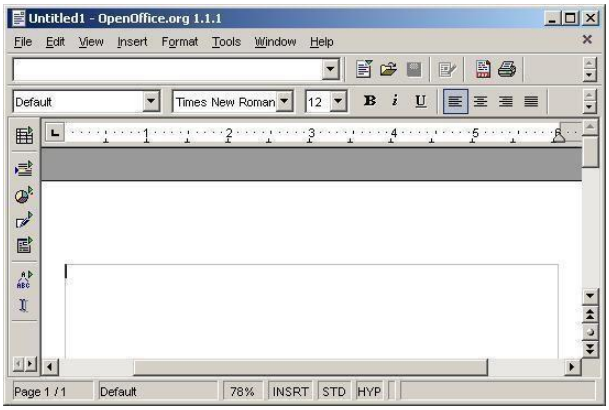
We can do a quick evaluation of the capabilities of Open Office by comparing its behaviors to Microsoft Office. In doing so, we find a critical difference in how the products handle cutting

and pasting text. In MS Office's word processor, users can create huge files by pasting large amounts of text. In OO Writer, you cannot create a file larger than 65,535 characters.

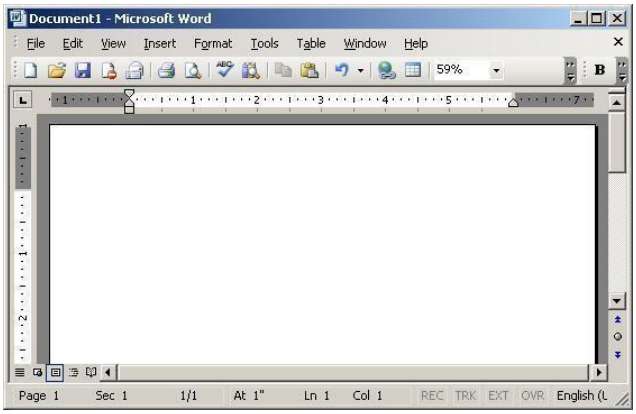
Application Description

OpenOffice.org is a free office suite that includes a word processor, a spreadsheet creator, and a presentation creator. *Writer* is the word processor component of OpenOffice.org and is used to write and edit text documents.

Microsoft Office 2003 is the most widely used office suite. *Word* is the word processor component of Office 2003 and can be used to perform the same tasks as *Writer*.



OpenOffice.org's *Writer*



Microsoft Office 2003 *Word*

Test Design

In Oracle-based testing, we compare the behavior of the program under test to the behavior of a source we consider accurate (an oracle).

One of the common early tasks when testing a program is a survey of the program's capabilities. You walk through then entire product, trying out each feature to see what the product can do, what it does well, what seems awkward, and what seems obviously unstable.

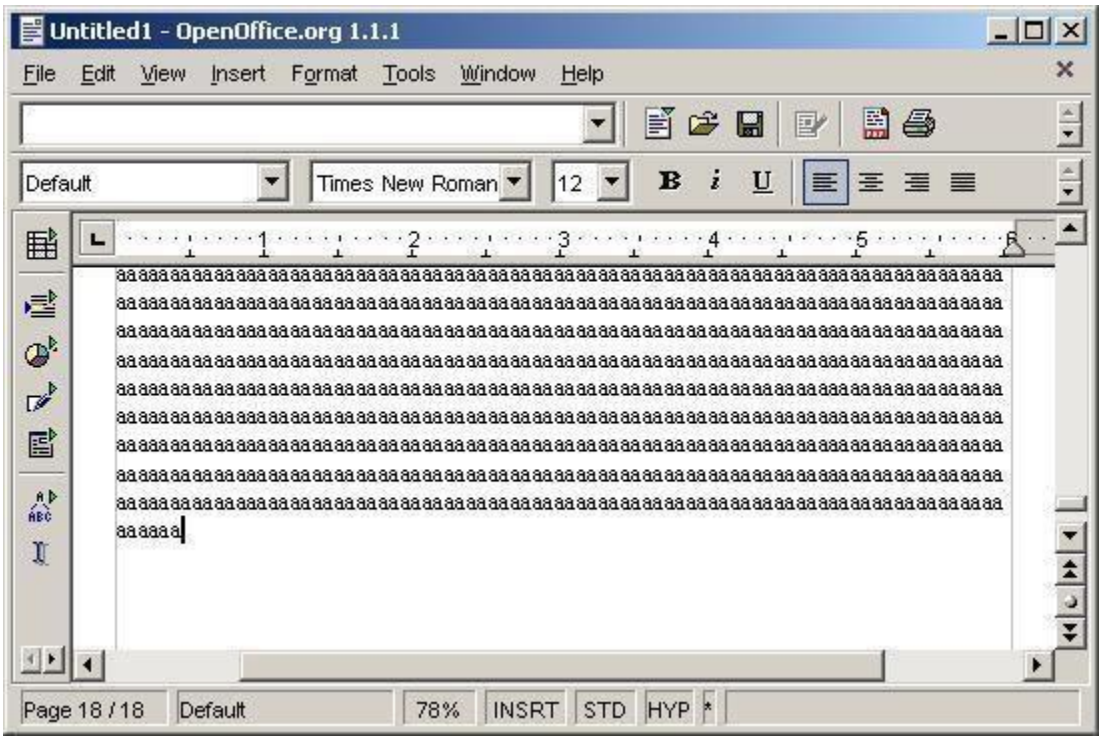
The tester doing the survey has to constantly evaluate the program: Is this behavior reasonable? Correct? In line with user expectations? A tester who is expert with this type of product will have no problem making these evaluations, but a newcomer needs a reference for guidance. An oracle is one such reference.

Open Office (OO) is an office productivity suite that was designed to compete with Microsoft Office. It makes sense to us to use Office as the reference point when surveying Open Office.

Performing the Test

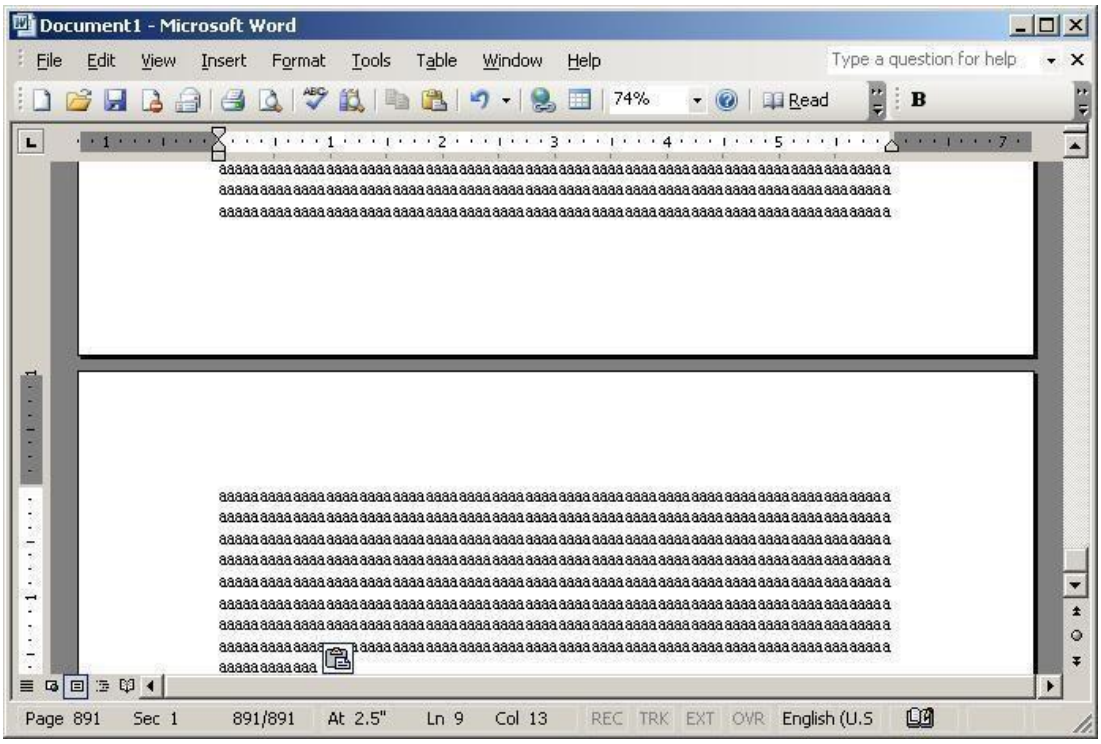
A survey involves rapid testing of many different features. We focus here on just one part of the survey, evaluation of cutting and pasting.

1. We use Open OpenOffice.org *Writer* and Microsoft *Word.*
2. In *Writer*, type a few lines of the character 'a'.
3. Highlight the characters and use *Ctrl-C* (Copy) and *Ctrl-V* (Paste) to fill the document with text.
4. Repeat step 3 to paste in as many characters as possible:



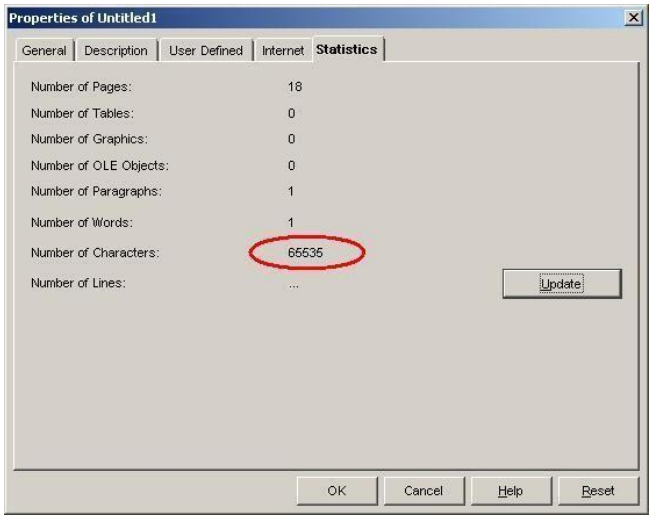
This picture shows that *Writer* has stopped accepting characters on Page 18.

5. In *Word*, type a few lines of the character 'a'.
6. Highlight the characters and use *Ctrl-C* (Copy) and *Ctrl-V* (Paste) to fill the document with text.
7. Repeat step 6 to paste in as many characters as possible:



We stopped that at page 891. *Word* will still accept characters.

Results/Relevance



***Writer*'s character count statistics**

***Word*'s character count statistics**

OpenOffice.org *Writer* stopped accepting text at 65,535 characters (about 18 pages with size 12 Times New Roman font with standard margins). At the 65,535 character limit, we are unable to add characters by pasting or by typing. We can edit the text already in the document.

Even if the character limit for Writer documents *is supposed to be* 65,535 characters, this test reveals a separate problem. When pasting text that fills the document, the overflow text was cut off without a warning. The user has thus lost data, without necessarily realizing it.

In *Word*, we saw a completely different situation. After 890 pages, it was still accepting characters. In fact, we could have kept pasting until the system ran out of memory. There is apparently no limit on the amount of characters that *Word* will accept.

How is this relevant to oracle-based testing? People often write about oracles as test automation support tools. They can be. But as we see here, even in a simple exploration, an oracle can be a useful supplement to specifications and documentation, or a surrogate for these documents if they are unavailable.

Self-checks as oracles,

A program or module specification describes *all* correct program behaviors, so an oracle based on a specification need not be paired with a particular test case.

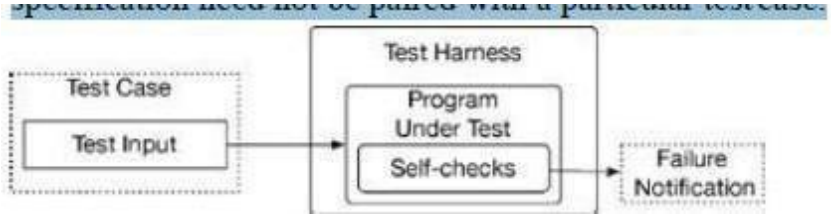


Figure 3.8: When self-checks are embedded in the program, test cases need not include predicted outputs.

Self-check assertions may be left in the production version of a system, where they provide much better diagnostic information than the uncontrolled application crash the customer may otherwise report. If this is not acceptable - for instance, if the cost of a runtime assertion check is too high - most tools for assertion processing also provide controls for activating and deactivating assertions.

It is generally considered good design practice to make assertions and self-checks be free of side-effects on program state. Side-effect free assertions are essential when assertions may be deactivated, because otherwise suppressing assertion checking can introduce program failures that appear only when one is *not* testing. Self-checks in the form of assertions embedded in program code are useful primarily for checking module and subsystem-level specifications,rather than overall program behavior.

Devising program assertions that correspond in a nuartal way to specifications (formal or informal) poses two main challenges: bridging the gap between concrete execution values and

abstractions used in specification, and dealing in a reasonable way with quantification over collections of values.

$$(|\langle k, v\rangle \in \phi(dict)|)$$
$$o = dict.get(k)$$
$$(|o = v|)$$

φ is an abstraction function that constructs the abstract model type (sets of key, value pairs) from the concrete data structure. φ is a logical association that need not be implemented whenreasoning about program correctness.

To create a test oracle, it is useful to have an actual implementation of φ. For this example, we might implement a special observer method that creates a simple textual representation of the set of (key, value) pairs. Assertions used as test oracles can then correspond directly to the specification.

Besides simplifying implementation of oracles by implementing this mapping once and using it in several assertions, structuring test oracles to mirror a correctness argument is rewarded when a later change to the program invalidates some part of that argument

In addition to an abstraction function, reasoning about the correctness of internal structures usually involves structural invariants, that is, properties of the data structure that are preserved byall operations. Structural invariants are good candidates for self checks implemented as assertions.

They pertain directly to the concrete data structure implementation, and can be implemented within the module that encapsulates that data structure. For example, if a dictionary structure is implementedas a red-black tree or an AVL tree, the balance property is an invariant of the structure that can be checked by an assertion within the module.

```
1 package org.eclipse.jdt.internal.ui.text;
2 import java.text.CharacterIterator;
3 import org.eclipse.jface.text.Assert; 4 /**
5 *A <code>CharSequence</code> based implementation of
6 * <code>CharacterIterator</code>. 7 * @since 3.0
8 */
9 public class SequenceCharacterIteratorimplements CharacterIterator{ 13 ...
14 private void invariant() {
15 Assert.isTrue(fIndex>=  fFirst);
16 Assert.isTrue(fIndex<= fLast); 17 }
49 ...
50 public SequenceCharacterIterator(CharSequence sequence, intfirst, intlast)
```

```
51 throws IllegalArgumentException {
52 if (sequence == null)
53 throw new NullPointerException();
54          if (first < 0 || first > last)
55              throw new IllegalArgumentException();
56          if (last > sequence.length())
57              throw new IllegalArgumentException();
58          fSequence= sequence;
59          fFirst= first;
60          fLast= last;
61          fIndex= first;
62          invariant();
63      }
143 ...
144     public char setIndex(int position) {
145          if (position >= getBeginIndex() && position <= getEndIndex())
146              fIndex= position;
147          else
148              throw new IllegalArgumentException();
149
150          invariant();
151          return current();
152      }
263 ...
264 }|
```

There is a natural tension between expressiveness that makes it easier to write and understand specifications, and limits on expressiveness to obtain efficient implementations.

It is not much of a stretch to say that programming languages are just formal specification languages in which expressiveness has been purposely limited to ensure that specifications can be executed with predictable and satisfactory performance.

An important way in which specifications used for human communication and reasoning about programs are more expressive and less constrained than programming languages is that they freely quantify over collections of values.

For example, a specification of database consistency might state that account identifiers are unique; that is, *for all* account records in the database, there *does not exist* another account recordwith the same identifier.

The problem of quantification over large sets of values is a variation on the basic problem of program testing, which is that we cannot exhaustively check all program behaviors.

Instead, we select a tiny fraction of possible program behaviors or inputs as representatives. The same tactic is applicable to quantification in specifications. If we cannot fully evaluate the specified property, we can at least select some elements to check (though at present we know of no program assertion packages that support sampling of quantifiers).

For example, although we cannot afford to enumerate all possible paths between two points in a large map, we may be able to compare to a sample of other paths found by the same procedure. program may use ghost variables to track entry and exit of threads from a critical section.

The postcondition of an in-place sort operation will state that the new value is sorted and a permutation of the input value. This permutation relation refers to both the "before" and "after" values of the object to be sorted.

A run-time assertion system must manage ghost variables and retained "before" values and must ensure that they have no side-effects outside assertion checking.

It may seem unreasonable for a program specification to quantify over an infinite collection, but in fact it can arise quite naturally when quantifiers are combined with negation. If we say "there is no integer greater than 1 that divides k evenly," we have combined negation with "there exists" to form a statement logically equivalent to universal ("for all") quantification over the integers. We may be clever enough to realize that it suffices to check integers between 2 and $\sqrt{k}$, but that is no longer a direct translation of the specification statement.

Capture and replay

Capture and Replay

Sometimes it is difficult to either devise a precise description of expected behavior or adequately characterize correct behavior for effective self-checks.

For example, while many properties of a program with a graphical interface may be specified in a manner suitable for comparison-based or self-check oracles, some properties are likely to require a person to interact with the program and judge its behavior.

If one cannot completely avoid human involvement in test case execution, one can at least avoid unnecessary repetition of this cost and opportunity for error. The principle is simple. The first time such a test case is executed, the oracle function is carried out by a human, and theinteraction sequence is captured.

Provided the execution was judged (by the human tester) to be correct, the captured log now forms an (input, predicted output) pair for subsequent automated retesting. The savings from automated retesting with a captured log depends on how many build- and-test cycles we can continue to use it in, before it is invalidated by some change to the program.

Distinguishing between significant and insignificant variations from predicted behavior, in order to prolong the effective lifetime of a captured log, is a major challenge for capture/replay testing. Capturing events at a more abstract level suppresses insignificant changes.

For example, if we log only the actual pixels of windows and menus, then changing even a typeface or background color can invalidate an entire suite of execution logs.
Mapping from concrete state to an abstract model of interaction sequences is sometimes possible but is generally quite limited.

A more fruitful approach is capturing input and output behavior at multiple levels of abstraction within the implementation

MODULE 4

Process Framework : Basic principles: Sensitivity, redundancy, restriction, partition, visibility, Feedback, the quality process, Planning and monitoring, Quality goals, Dependability properties ,Analysis Testing, Improving the process, Organizational factors.

Planning and Monitoring the Process: Quality and process, Test and analysis strategies and plans, Risk planning, monitoring the process, Improving the process, the quality team

Documenting Analysis and Test: Organizing documents, Test strategydocument, Analysis and test plan, Test design specifications documents, Test and analysis reports.

Process Framework : Basic principles

MENTION THE BASIS PRINCIPLES UNDERLYING A & T TECHNIQUES. ?
Analysis and testing (A&T) has been common practice since the earliest software projects.

Six principles that characterize various approaches and techniques for analysis and testing are sensitivity, redundancy, restriction, partition, visibility, and feedback.

General engineering principles:

Partition: divide and conquer
Visibility: making information accessible
Feedback: tuning the development process

Specific A&T principles:

Sensitivity: better to fail every time than sometimes
Redundancy: making intentions explicit
Restriction: making the problem easier

Sensitivity

Human developers make errors, producing faults in software. Faults may lead to failures, but faulty software may not fail on every execution.

The sensitivity principle states that it is better to fail every time than sometimes.

If a fault is detected in unit testing, the cost of repairing is relatively small.

If a fault survives at the unit level, but triggers a failure detected in integration testing, the cost of correction is much greater.

If the first failure is detected in system or acceptance testing, the cost is very high indeed, and the most costly faults are those detected by customers in the field.

A fault that triggers a failure on every execution is unlikely to survive past unit testing.

For example, a fault that results in a failure only for some unusual configurations of customer equipment may be difficult and expensive to detect. A fault that results in a failure randomly but very rarely.

The small C program that has three faulty calls to string copy procedures is shown below,

```
#include <assert.h>
char before[ ] = "=Before=";
char middle[ ] = "Middle";
char after[ ] = "=After=";

void show() {
printf("%s\n%s\n%s\n", before, middle,
after);
}

void stringCopy(char *target, const char
*source, int howBig) {
assert(strlen(source) < howBig);
strcpy(target, source);
}
```

```
int main(int argc, char *argv)
{
show();
strcpy(middle, "Muddled");  /* Fault, but may
not fail */
show();
strncpy(middle, "Muddled", sizeof(middle));  /*
Fault, may not fail */
show();
stringCopy(middle, "Muddled",sizeof(middle));
/* Guaranteed to fail */
show();
}
```

Standard C functions strcpy and strncpy may or may not fail when the source string is too long. The procedure stringCopy is sensitive: It is guaranteed to fail in an observable way if the source string is too long.

The call to strcpy, strncpy, and stringCopy all pass a source string "Muddled," which is too long to fit in the array middle. For strcpy, the fault may or may not cause an observable failure depending on the arrangement of memory.

While strncpy avoids overwriting other memory, it truncates the input without warning, and sometimes without properly null-terminating the output.

The function stringCopy, uses an assertion to ensure that, if the target string is too long, the program always fails in an observable manner.

The sensitivity principle made these faults easier to detect by making them cause failure more often by applying in three main ways:

At the design level, changing the way in which the program fails;

At the analysis and testing level, choosing a technique more reliable with respect to the property of interest;

At the environment level, choosing a technique that reduces the impact of external factors on the results.

Examples of application of the sensitivity principle:

Replacing strcpy and strncpy with stringCopy in the above program.

Run-time array bounds checking in many programming languages.

A variety of tools and replacements for the standard memory management library are available to enhance sensitivity to memory allocation and reference faults.

The fail-fast property of Java iterators provides an immediate and observable failure when the illegal modification occurs.

A run time deadlock analysis works better if it is machine independent, i.e., if the program deadlocks when analyzed on one machine, it deadlocks on every machine

A test selection criterion works better if every selected test provides the same result, i.e., if the program fails with one of the selected tests, it fails with all of them (reliable criteria)

Redundancy

Redundancy is the opposite of independence. In software test and analysis, we wish to detect faults that could lead to differences between intended behavior and actual behavior, so the redundancy is in the form of making intentions explicit.

Redundancy can be introduced to declare intent and automatically check for consistency. Static type checking is a classic application of this principle: The type declaration is a statement of intent that is at least partly redundant with the use of a variable in the source code.

The type declaration constrains other parts of the code, so a consistency check can be applied.

Redundancy check is not limited to program source code, one can also intentionally introduce redundancy in other software artifacts (design) where, software design tools typically provide ways to check consistency between different design views or artifacts.

Redundancy is exploited instead with run-time checks which is another application of redundancy in programming.

Restriction

When there are no acceptably cheap and effective ways to check a property, checking can be done on more restrictive property or limit the check to a smaller, more restrictive class of programs.

Consider the problem of ensuring that each variable is initialized before it is used, on every execution. It is not possible for a compiler or analysis tool to precisely determine whether it holds.

The program shown below illustrates : Can the variable k ever be uninitialized the first time i is added to it? If someCondition(0) always returns true, then k will be initialized to zero on the first time through the loop, before k is incremented, so perhaps there is no potential for a run-time error - but method someCondition could be arbitrarily complex and might even depend on some condition in the environment.

Java's solution to this problem is to enforce a stricter, simpler condition: A program is not permitted to have any syntactic control paths on which an uninitialized reference could occur, regardless of whether those paths could actually be executed. The program has such a path, so the Java compiler rejects it.

The choice of programming language(s) for a project may entail a number of source code restrictions that impact test and analysis.

Additional restrictions may be imposed in the form of programming standards such as the use of type casts or pointer arithmetic in C and Other forms of restriction can apply to architectural and detailed design.

Restrictions can be imposed as the property of serializability on the schedule of transactions that happens serially in some order. This is done by using a particular locking scheme on the program at design time.

Stateless component interfaces are an example of restriction applied at the architectural level. An interface is stateless if the service does not remember anything about previous requests. One such stateless interface is the Hypertext Transport Protocol (HTTP) 1.0 of the World-Wide-Web which made Web servers much simpler and easier to test.

```
1 /** A trivial method with a potentially uninitialized variable.
2 * Maybe someCondition(0) is always true, and therefore k is
3 * always initialized before use ... but it's impossible, in
4 * general, to know for sure. Java rejects the method.
5 */
6 static void questionable() {
7 int k;
8 for (int i=0; i < 10; ++i) {
9 if (someCondition(i)) {
10 k=0;
11 } else {
12 k+=i;
13 }
14 }
15 System.out.println(k);
16 }
17 }
```

Partition

Partition, often also known as "divide and conquer," is a general engineering principle. Dividing a complex problem into sub problems to be attacked and solved independently is probably the most common human problem-solving strategy.

In Analysis and testing the partition principle is widely used and exploited.

Partitioning can be applied both at process and technique levels.

At the process level, we divide complex activities into sets of simple activities that can be attacked independently. For example, testing is usually divided into unit, integration, subsystem, and system testing. In this way, we can focus on different sources of faults at different steps, and at each step, we can take advantage of the results of the former steps.

Many static analysis techniques divide the overall analysis into two subtasks,

Simplify the system to make the proof of the desired properties feasible

And then prove the property with respect to the simplified model.

Identify a finite number of classes of test cases either from specifications (functional testing) or from program structure (structural testing) to execute.

Visibility

Visibility means the ability to measure progress or status against goals.

In software engineering, the visibility principle is in the form of process visibility, and project schedule visibility.

Quality process visibility also applies to measuring achieved (or predicted) quality against quality goals.

Visibility is closely related to observability, the ability to extract useful information from a software artifact.

A variety of simple techniques can be used to improve observability as in the Internet protocols like HTTP and SMTP (Simple Mail Transport Protocol, used by Internet mail servers) are based on the exchange of simple textual commands. The choice of simple, human-readable text rather than a more compact binary encoding has a small cost in performance and a large payoff in observability.

A variant of observability through direct use of simple text encodings is providing readers and writers to convert between other data structures and simple, human readable and editable text.

Feedback

Feedback is another classic engineering principle that applies to analysis and testing.

Feedback applies both to the process itself (process improvement) and to individual techniques.

Systematic inspection derive its success from feedback.

Participants in inspection are guided by checklists, and checklists are revised and refined based on experience.

New checklist items may be derived from root cause analysis, analyzing previously observed failures to identify the initial errors that lead to them.

SUMMARY
The discipline of test and analysis is characterized by 6 main principles:
Sensitivity: better to fail every time than sometimes
Redundancy: making intentions explicit
Restriction: making the problem easier
Partition: divide and conquer
Visibility: making information accessible
Feedback: tuning the development process
They can be used to understand advantages and limits of different approaches and compare different techniques.

The quality process

Quality process is a set of activities and responsibilities that focused primarily on ensuring adequate dependability of the software product and concerned with project schedule or with product usability.

Like other parts of an overall software process, the quality process provides a framework for selecting and arranging activities aimed at a particular goal, while also considering interactions and trade-offs with other important goals.

The quality process should be structured for, completeness: appropriate activities are planned to detect each important class of faults.

Timeliness : faults are detected at a point of high leverage which means they are detected as early as possible.

Cost-effectiveness: it is the constraints of completeness and timeliness. Cost must be considered over the whole development cycle and product life, so the dominant factor is usually the cost of repeating an activity through many change cycles.

Activities of quality process are considered as being in the domain of quality assurance or quality improvement.

Carryout quality activities at the earliest opportunity because a defect introduced in coding is far cheaper to repair during unit test than later during integration or system test, and most expensive if it is detected by a user of the fielded system.

3.2 PLANNING AND MONITORING:
Process visibility is a key factor in software quality processes.

Process visibility in software quality process emphasizes on progress against quality goals.
If one cannot gain confidence in the quality of the software system long before it reaches final testing, the quality process has not achieved adequate visibility.

A well-designed quality process balances several activities across the whole development process, selecting and arranging them to be as cost-effective as possible, and to improve early visibility.

Planning and monitoring

Planning improves early visibility which motivates the use of “proxy” measures which means the use of quantifiable attributes that are not identical to the properties wished to measure but have the advantage of being measured earlier in development. Ex: the number of faults in design or code is not a true measure of reliability, but we may count faults discovered in design inspections as an early indicator of potential quality problems

Quality goals can be achieved only through careful planning of activities that are matched to the identified objectives.

Planning is integral to the quality process.

The overall analysis and test strategy identifies company- or project-wide standards that must be satisfied, procedures required, e.g., for obtaining quality certificates techniques and tools that must be used documents that must be produced.

A complete analysis and test plan is a comprehensive description of the quality process that includes several items:

objectives and scope of A&T activities

documents and other items that must be available:
items to be tested
features to be tested and not to be tested
analysis and test activities
staff involved in A&T
constraints
pass and fail criteria
schedule
deliverables
hardware and software requirements
risks and contingencies

Quality goals

Quality process visibility includes distinction among dependability qualities.

Product qualities are the goals of software quality engineering , and process qualities are mean to achieve those goals.

Software product qualities can be divided in to those that are directly visible to a client and those that primarily affect the software development organization.

Reliability is directly visible to the client. Maintainability affects development organization , and indirectly affects client.

Properties that are directly visible to users of a software product, such as dependability, latency, usability and throughput are called external properties.

Properties that are not directly visible to end users , such as maintainability, reusability and traceability are called internal properties, even when their impact on the software development and evolution processes may indirectly affect users.

The external properties of software can be divided into dependability and usefulness.

Quality can be considered as fulfillment of required and desired properties as distinguished from specified properties.

Critical tasks in software quality analysis is to make desired properties explicit.

Dependability properties

BRIEFLY DISCUSS THE DEPENDABILITY PROPERTIES IN PROCESS FRAMEWORK. ?

Correctness:
A program or system is correct if it is consistent with its specification. A specification divides all possible system behaviors into two classes, successes (or correct executions) and failures.

All of the possible behaviors of a correct system are successes.

A program cannot be mostly correct or somewhat . It is absolutely correct on all possible behaviors, or else it is not correct.

It is very easy to achieve correctness, with respect to some (very bad) specification.

Achieving correctness with respect to a useful specification, on the other hand, is seldom practical for nontrivial systems.

Reliability:

It is a statistical approximation to correctness which means 100% reliable = correctness.

It is the likelihood of correct function for some ``unit'' of behavior.

It is relative to a specification and usage profile. The same program can be more or less reliable depending on how it is used.

Availability:
Particular measures of reliability can be used for different units of execution and different ways of counting success and failure.

Availability is an appropriate measure when a failure has some duration in time.

The availability of the router is the time in which the system is "up" (providing normal service) as a fraction of total time. Between the initial failure of a network router and its restoration we say the router is "down" or "unavailable." Thus, a network router that averages 1 hour of down time in each 24-hour period would have an availability of 23/24, or 95.8%.

Mean time between failures (MTBF) is yet another measure of reliability, also using time as the unit of execution.

The hypothetical network switch that typically fails once in a 24-hour period and takes about an hour to recover has a mean time between failures of 23 hours.

Note that availability does not distinguish between two failures of 30 minutes each and one failure lasting an hour, while MTBF does.

The definitions of correctness and reliability have (at least) two major weaknesses.

First, since the success or failure of an execution is relative to a specification, they are only as strong as the specification.

Second, they make no distinction between a failure that is a minor annoyance and a failure that results in catastrophe.

These are simplifying assumptions that we accept for the sake of precision.

Safety and hazards:

Software safety is an extension of the well-established field of system safety into software.

Safety is concerned with preventing certain undesirable behaviors, called hazards.

Software safety is typically a concern in "critical" systems such as avionics and medical systems, but the basic principles apply to any system in which undesirable behaviors can be distinguished from failure.

For example, the developers of a word processor might consider safety with respect to the hazard of file corruption separately from reliability with respect to the complete functional requirements for the word processor.

Safety is meaningless without a specification of hazards to be prevented, and in practice the first step of safety analysis is always finding and classifying hazards.

Typically, hazards are associated with some system in which the software is embedded (e.g., the medical device), rather than the software alone.

Safety is that it is concerned only with these hazards, and not with other aspects of correct functioning.

The dead-man switch of the mower , does not contribute in any way to cutting grass; its sole purpose is to prevent the operator from reaching into the mower blades while the engine runs by acting as the interlock device.

Safety is best considered as a quality distinct from correctness and reliability for two reasons. First, by focusing on a few hazards and ignoring other functionality, a separate safety specification can be much simpler than a complete system specification, and therefore easier to verify.

Second, even if the safety specification were redundant with regard to the full system specification, it is important because (by definition) we regard avoidance of hazards as more crucial than satisfying other parts of the system specification.

Robustness:

Software that fails under some conditions, which violate the premises of its design, may still be "correct" in the strict sense, yet the manner in which the software fails is important.

It is acceptable that the word processor fails to write the new file that does not fit on disk, but unacceptable to also corrupt the previous version of the file in the attempt.

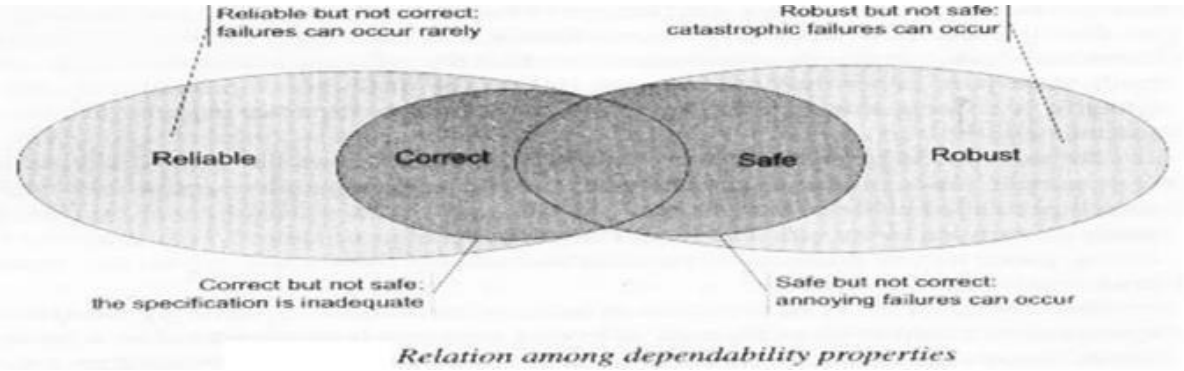
It is acceptable for the database system to cease to function when the power is cut, but unacceptable for it to leave the database in a corrupt state.

It is usually preferable for the Web system to turn away some arriving users rather than becoming too slow for all, or crashing.

Software that gracefully degrades or fails "softly" outside its normal operating parameters is robust.

Software safety is a kind of robustness, that concerns not only avoidance of hazards (e.g., data corruption) but also partial functionality under unusual situations.

Robustness, like safety, begins with explicit consideration of unusual and undesirable situations, and should include augmenting software specifications with appropriate responses to undesirable events.



Relation among dependability properties

Analysis Testing

ILLUSTRATE THE PURPOSE OF SOFTWARE ANALYSIS?

Analysis techniques that do not involve actual execution of program source code play a prominent role in overall software quality processes.

Manual inspection techniques and automated analyses can be applied at any development stage.

Inspection:-

Applied to any document including requirements documents, architectural and design documents, test plans, test cases and program source code.

Inspection also benefits by spreading good practices and shared standards of quality.

Inspection used primarily where other techniques are inapplicable and where other techniques do not provide sufficient coverage

Inspection on the other hand takes a considerable amount of time. Moreover re-inspecting a changed component can be as expensive as the initial inspection.

Automated static analyses:-
It is more limited in applicability, but used when available because substituting machine cycles for human effort is cost-effective. Due to the substantial effort for structuring a model for analysis, the cost advantage is diminished.

But their application has the ability to check for particular classes of faults for which checking with other technique are very difficult or expensive.

Sometimes the best aspects of manual inspection and automated static analysis can be obtained by carefully decomposing properties to be checked.

For example, consider property of special term in the application domain appear in a glossary of terms.

This property is not directly agreeable to an automated static analysis, since current tools cannot distinguish meaningful domain terms from other terms that have their ordinary meanings.

The property can be checked with manual inspection, but the process is tedious, expensive, and error-prone.

Hence a hybrid approach can be applied if each domain term is marked in the text. Manually checking that domain terms are marked is much faster and therefore less expensive.

3.6 Testing:

ILLUSTRATE THE PURPOSE OF SOFTWARE TEST?

Despite the attractiveness of automated static analyses, manual inspections, dynamic testing remains a dominant technique.

Dynamic testing is divided into several distinct activities that may occur at different points in a project.

Tests are executed when the corresponding code is available, but testing activities start earlier, as soon as the artifacts required for designing test case specifications are available.

Thus, acceptance and system test suites should be generated before integration and unit test suites.

By early test design tests are specified independently from code.

Moreover, test cases may highlight inconsistencies and incompleteness in the corresponding software specifications.

Early design of test cases also allows for early repair of software specifications, preventing specification faults from propagating to later stages in development.

Finally, programmers may use test cases to illustrate and clarify the software specifications, especially for errors and unexpected conditions.

Just as the "earlier is better" rule dictates using inspection to reveal flaws in requirements and design before they are propagated to program code, the same rule dictates module testing to uncover as many program faults as possible before they are incorporated in larger subsystems of the product.

Improving the process

Improving the Process:
Confronted by similar problems, developers tend to make the same kinds of errors over and over, and consequently the same kinds of software faults are often encountered project after project.

The quality process and the software development process can be improved by gathering, analyzing, and acting on data regarding faults and failures.

The goal of quality process improvement is to find cost-effective countermeasures for classes of faults that are expensive because they occur frequently, or failures they cause are expensive, or expensive to repair.

Countermeasures may be prevention or detection or quality assurance activities or aspects of software development aspects.

The first part of a process improvement is gathering sufficiently complete and accurate raw data about faults and failures.

A main obstacle is that data gathered in one project goes mainly to benefit other projects in the future and may seem to have little direct benefit for the current project.

It is therefore helpful to integrate data collection with normal development activities, such as version and configuration control, project management, and bug tracking.

Raw data on faults and failures must be aggregated into categories and prioritized. Faults may be categorized with similar causes and possible remedies.

The analysis step consists of tracing several instances of an observed fault or failure back to the human error from which it resulted, or even further to the factors that led to that human error.

The analysis also involves the reasons the fault was not detected and eliminated earlier. This process is known as "root cause analysis".

For the buffer overflow errors in network applications, the countermeasure could involve differences in programming methods or improvements to quality assurance activities or sometimes changes in management practices.

Organizational factors.

WHY ORGANIZATIONAL FACTORS ARE NEEDED IN PROCESS FRAMEWORK. ?

The quality process includes a wide variety of activities that require specific skills and attitudes and may be performed by quality specialists or by software developers.

Planning the quality process involves not only resource management but also identification and allocation of responsibilities.

A poor allocation of responsibilities can lead to major problems in which pursuit of individual goals conflicts with overall project success.

For example, splitting responsibilities of development and quality-control between a development and a quality team, and rewarding may produce undesired results.

The development team, not rewarded to produce high-quality software, may attempt to maximize productivity to the detriment of quality.

Combining development and quality control responsibilities in one undifferentiated team, while avoiding the perverse incentive of divided responsibilities, can also have unintended effects: As deadlines near, resources may be shifted from quality assurance to coding, at the expense of product quality.

Conflicting considerations support both the separation of roles and the mobility of people and roles.

At Chipmunk, responsibility for delivery of the new Web presence is distributed among a development team and a quality assurance team. The quality assurance team is divided into,

The analysis and testing group- Responsible for the dependability of the system

The usability testing group- Responsible for usability.

Responsibility for security issues is assigned to the infrastructure development group, which relies partly on external consultants for final tests based on external attack attempts.

At Chipmunk, specifications, design, and code are inspected by mixed teams,

scaffolding and oracles are designed by analysts and developers

integration, system, acceptance, and regression tests are assigned to the test and analysis team.

unit tests are generated and executed by the developers

coverage is checked by the testing team before starting integration and system testing

A specialist has been hired for analyzing faults and improving the process. The process improvement specialist works incrementally while developing the system and proposes improvements at each release.

Planning and Monitoring the Process:

Learning objectives Learning objectives
• Understand the purposes of planning and monitoring
• Distinguish strategies from plans, and understand their relation
• Understand the role of risks in planning
• Understand the potential role of tools in monitoring a quality process
• Understand team organization as an integral part of planning

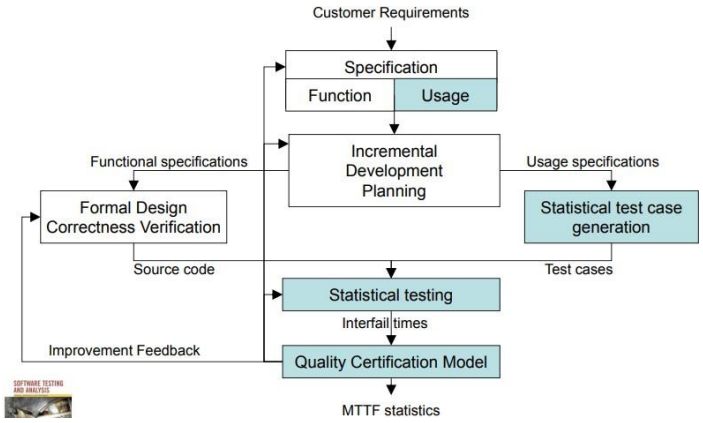
Planning:
– Scheduling activities (what steps? in what order?)
– Allocating resources (who will do it?)
– Devising unambiguous milestones for monitoring

• Monitoring: Judging progress against the plan
– How are we doing?
• A good plan must have visibility :
– Ability to monitor each step, and to make objective judgments of progress
– Counter wishful thinking and denial

Quality and process,

Quality process: Set of activities and responsibilities
– focused primarily on ensuring adequate dependability
– concerned with
project schedule or with product usability
• A framework for
– selecting and arranging activities
– considering interactions and trade-offs
• Follows the overall software process in which it is embedded
– Example: waterfall software process —> "V model": unit testing starts with implementation and finishes before integration
– Example: XP and agile methods —> emphasis on unit testing and rapid iteration for acceptance testing by customers

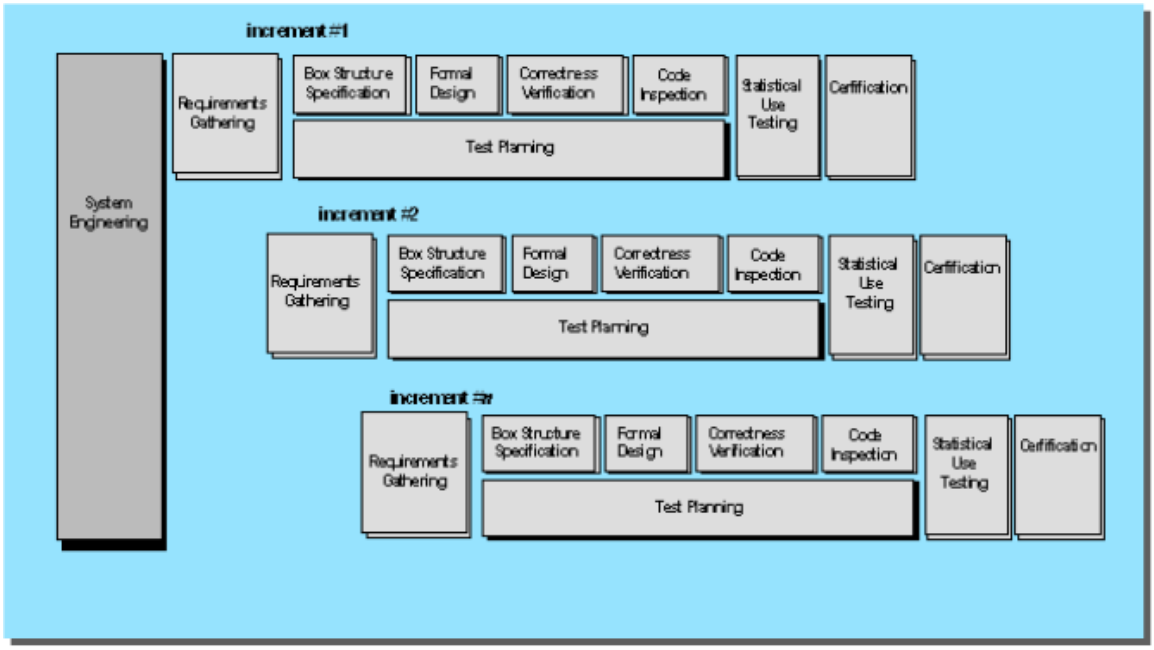
CLEANROON PROCESS MODEL



The philosophy of the "cleanroom" in hardware fabrication technologies:

It is cost-effective and time-effective to establish a fabrication approach that precludes the introduction of product defects.

Rather than fabricating a product and then working to remove defects, the cleanroom approach demands the discipline required to eliminate defects in specification and design and then fabricate in a "clean" manner.



Increment Planning —adopts the incremental strategy

Requirements Gathering —defines a description of customer level requirements (for each increment)

Box Structure Specification —describes the functional specification

Formal Design —specifications (called “black boxes”) are iteratively refined (with an increment) to become analogous to architectural and procedural designs (called “state boxes” and “clear boxes,” respectively).

Correctness Verification —verification begins with the highest level box structure (specification) and moves toward design detail and code using a set of “correctness questions.” If these do not demonstrate that the specification is correct, more formal (mathematical) methods for verification are used.

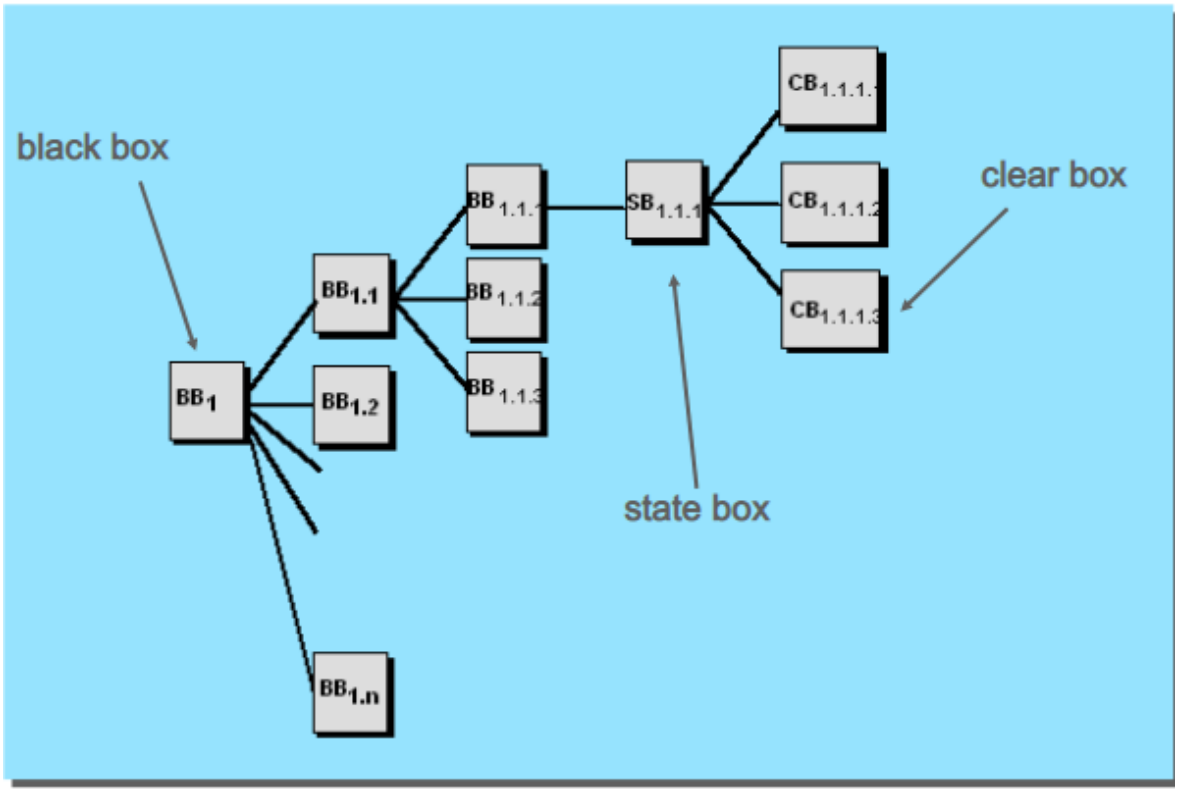
Code Generation, Inspection and Verification —the box structure specifications, represented in a specialized language, are transmitted into the appropriate programming language.

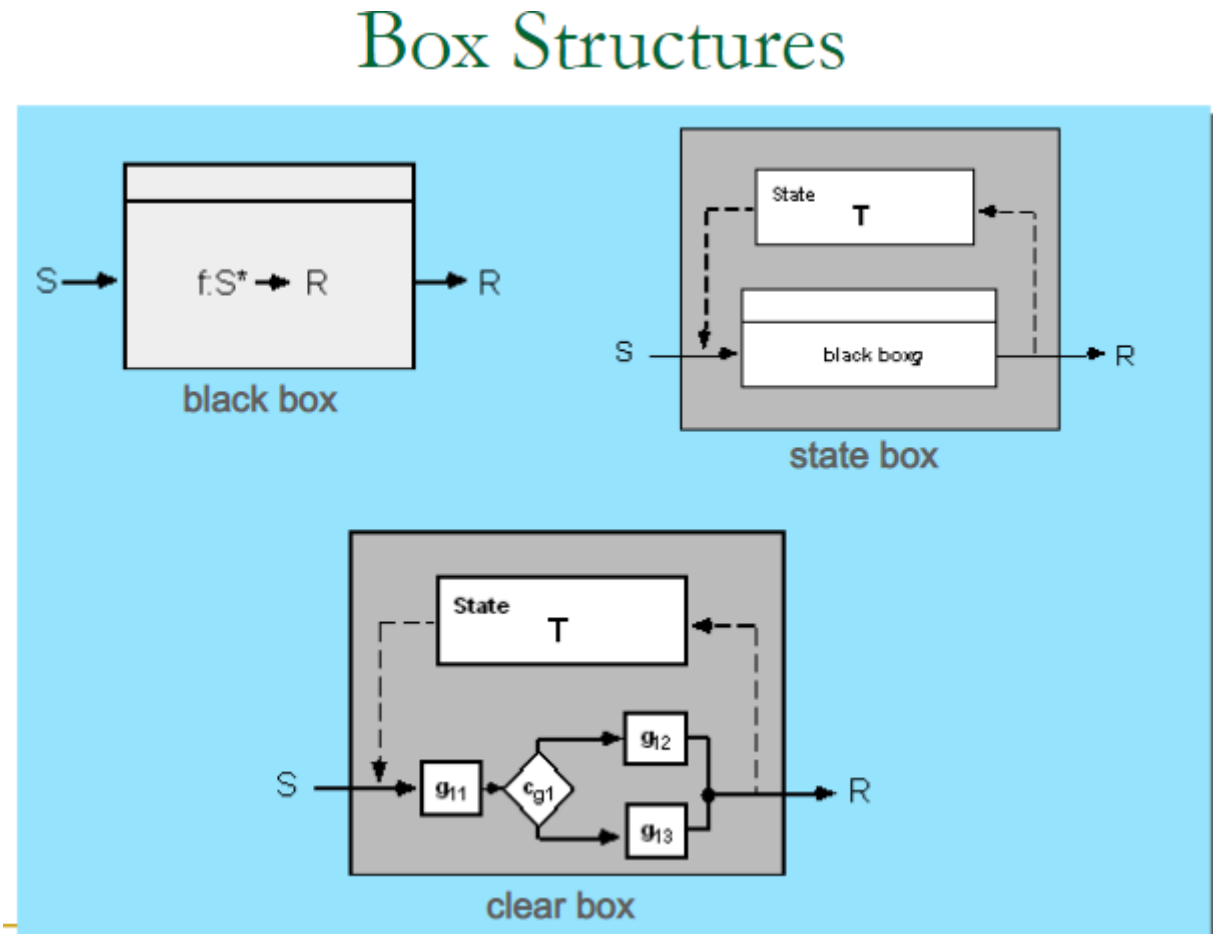
Statistical Test Planning —a suite of test cases that exercise of “probability distribution [A **probability distribution** is a list of all of the possible outcomes of a random variable along with their corresponding **probability** values.]” of usage are planned and designed

Statistical Usage Testing —execute a series of tests derived from a statistical sample (the probability distribution noted above) of all possible program executions by all users from a targeted population

Certification —once verification, inspection and usage testing have been completed (and all errors are corrected) the increment is certified as ready for integration.

Box Structure Specification





A) Black-Box Specification

A black-box specification describes an abstraction, stimuli, and response using the notation shown in . The function f is applied to a sequence, S*, of inputs (stimuli), S, and transforms them into an output (response), R. For simple software components, f may be a mathematical function, but in general, f is described using natural language (or a formal specification language).

B) State-Box Specification

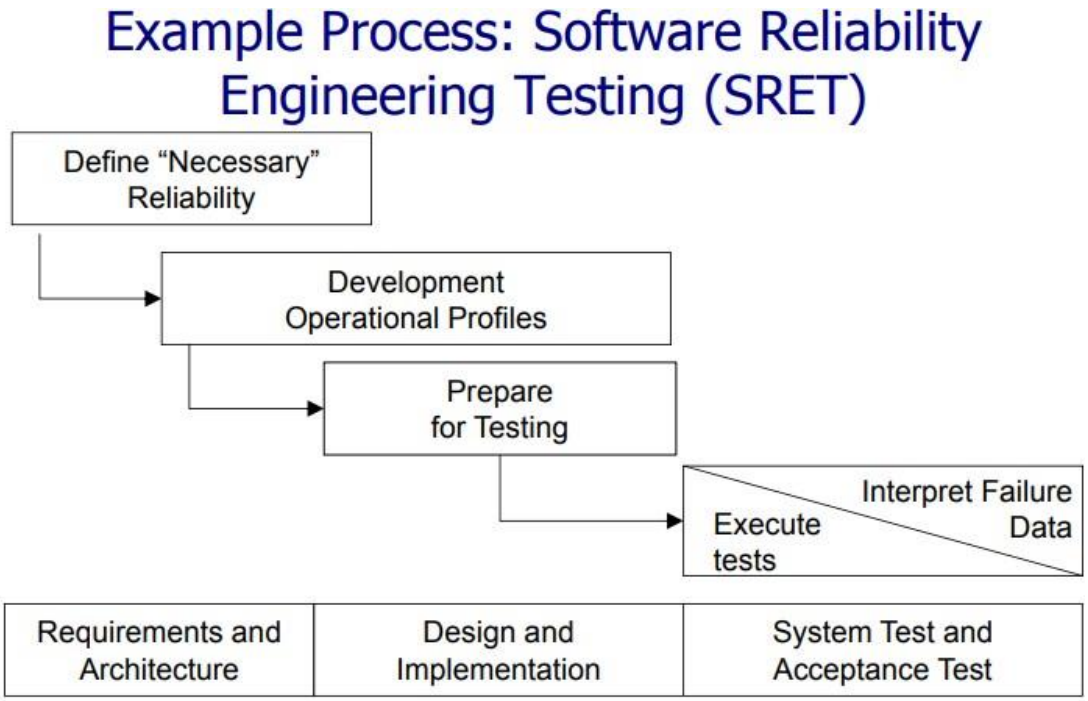
The state box is “a simple generalization of a state machine” [MIL88]. As processing occurs, a system responds to events (stimuli) by making a transition from the current state to some new state.

As the transition is made, an action may occur. The state box uses a data abstraction to determine the transition to the next state and the action (response) that will occur as a consequence of the transition. the state box incorporates a black box.

C) Clear-Box Specification

The clear-box specification is closely aligned with procedural design and structured programming. In essence, the sub function g within the state box is replaced by the structured

programming constructs that implement g. These, in turn, can be refined into lower-level clear boxes as stepwise refinement proceeds. It is important to note that the procedural specification described in the clear-box hierarchy can be proved to be correct.



What is Software Reliability Engineering (SRE)? The quantitative study of the operational behavior of software-based systems with respect to user requirements concerning reliability. SRE has been adopted either as standard or as best practice by more than 50 organizations in their software projects including AT&T, Lucent, IBM, NASA and Microsoft, plus many others worldwide. This presentation will provide an introduction to software reliability engineering

Why is SRE Important? There are several key reasons a reliability engineering program should be implemented: So that it can be determined how satisfactorily products are functioning. Avoid over-designing – products could cost more than necessary and lower profit. If more features are added to meet customer demand then reliability should be monitored to ensure that defects are not designed in, which could impact reliability. If a customer's product is not designed well, with reliability and quality in mind, then they may well turn to a COMPETITOR! Having a software reliability engineering process can make organizations more competitive as customers will always expect reliable software that is better and cheaper

Why is SRE Beneficial? For Engineers: Managing customer demands: Enables software to be produced that is more reliable; built faster and cheaper. Makes engineers more successful in meeting customer demands. In turn this avoids conflicts – risk, pressure, schedule, functionality, cost etc. For the organization: Improves competitiveness. Reduces development costs. Provides customers with quantitative reliability metrics. Places less emphasis on tools and a greater emphasis on "designing in reliability." Products can be developed that are

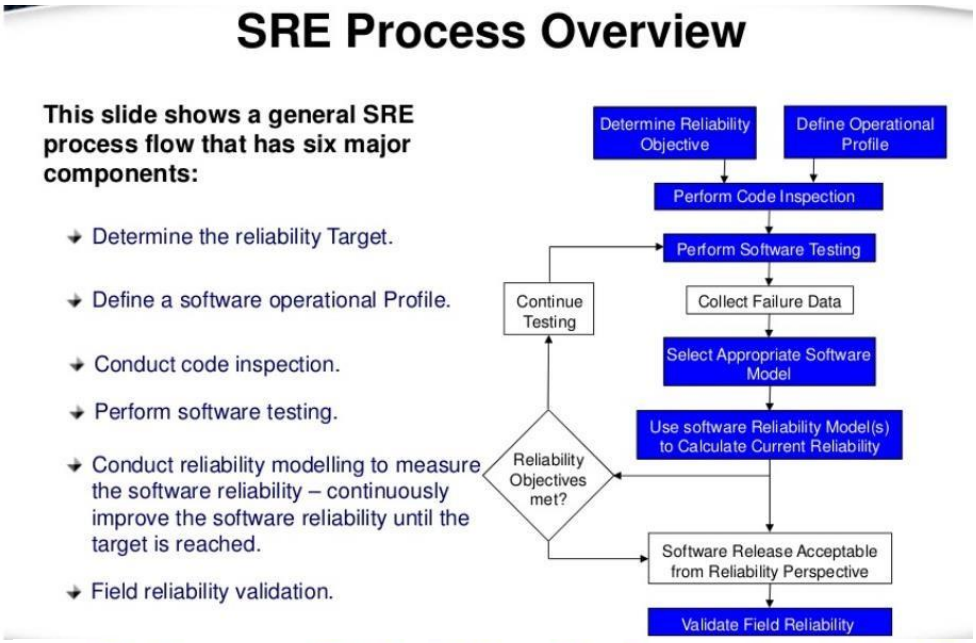
delivered to the customer at the right time, at an acceptable cost, and with satisfactory reliability.

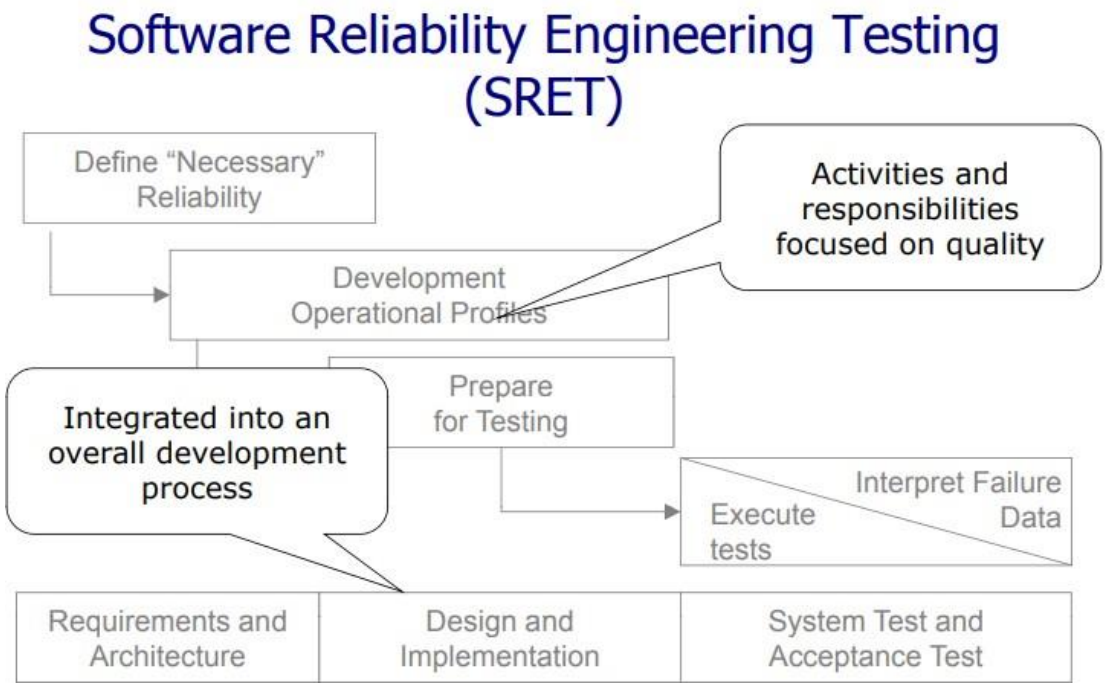
Common SRE Challenges Data is collected during test phases, so if problems are discovered it is too late for fundamental design changes to be made. Failure data collected during in-house testing may be limited, and may not represent failures that would be uncovered in the product's actual operational environment. Reliability metrics obtained from restricted testing data may result in reliability metrics being inaccurate. There are many possible models that can be used to predict the reliability of the software, which can be very confusing. Even if the correct model is selected there may be no way of validating it due to having insufficient field data.

Fault Lifecycle Techniques Prevent faults from being inserted. Avoids faults being designed into the software when it is being constructed. Remove faults that have been inserted. Detect and eliminate faults that have been inserted through inspection and test. Design the software so that it is fault tolerant. Provide redundant services so that the software continues to work even though faults have occurred or are occurring. Forecast faults and/or failures. Evaluate the code and estimate how many faults are present and the occurrences and consequences of software failures.

Preventing Faults From Being Inserted Initial approach for reliable software A fault that is never created does not cost anything to fix. This should be the ultimate objective of software engineering. This requires: A formal requirement specification always being available that has been thoroughly reviewed and agreed to. Formal inspection and test methods being implemented and used. Early interaction with end-users (field trials) and requirement refinement if necessary. The correct analysis tools and disciplined tool use. Formal programming principles and environments that are enforced. Systematic techniques for software reuse. Formal software engineering processes and tools, if applied successfully, can be very effective in preventing faults (but is no guarantee!) However, software reuse without proper verification can result in disappointment.

Removing Faults When faults are injected into the software, the next method that can be used is fault removal. Approaches: Software inspection. Software testing. Both have become standard industry practices. This presentation will focus closely on these.


SRE Process Overview
This slide shows a general SRE process flow that has six major components:
Determine the reliability Target.
Define a software operational Profile.
Conduct code inspection.
Perform software testing.
Conduct reliability modelling to measure the software reliability – continuously improve the software reliability until the target is reached.
Field reliability validation.
Determine Reliability Objective
Define Operational Profile
Perform Code Inspection
Perform Software Testing
Collect Failure Data
Select Appropriate Software Model
Use software Reliability Model(s) to Calculate Current Reliability
Reliability Objectives met?
Continue Testing
Software Release Acceptable from Reliability Perspective
Validate Field Reliability


Software Reliability Engineering Testing (SRET)
Define "Necessary" Reliability
Development Operational Profiles
Activities and responsibilities focused on quality
Prepare for Testing
Integrated into an overall development process
Execute tests
Interpret Failure Data
Requirements and Architecture
Design and Implementation
System Test and Acceptance Test

SRE Terms

- **Reliability objective:** The product's reliability goal from the customer's viewpoint.
- **Operational profile:** A set of system operational scenarios with their associated probability of occurrence.

  This encourages testers to select test cases according to the system's likely operational usage.
- **Reliability modeling:** This is an essential element of SRE that determines whether the product meets its reliability objective.

  One or more models can be used to calculate, from failure data collected during system testing, various estimates of a product's reliability as a function of test time. It can also provide the following information:
  - Product reliability at the end of various test phases.
  - Amount of additional test time required to reach the product's reliability objective.
  - The reliability growth that is still required (ratio of initial to target reliability).
  - Prediction of field reliability.
- **Field Reliability Validation:** Determination of whether the actual field reliability meets the customer's target.

Software Reliability Objectives

- **Reliability target(s) should be defined and used to:**
  - Manage customer expectations.
  - Determine how reliability growth can and will be tracked throughout the program.
  - Determine availability targets. Software reliability is commonly expressed as an availability metric though rather than as a probabilistic reliability metric. This is defined as:

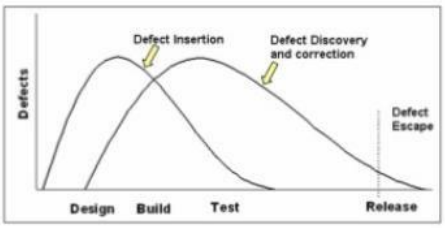
$$\text{Availability} = \frac{\text{Software uptime}}{\text{Software uptime + downtime}}$$

- **A data collection and analysis methodology also has to be defined:**
  - How inspections will be conducted.
  - How failure data will be collected.
  - How the data will be analyzed, i.e., what model will be used?
  - This helps project managers track metrics and plan resource.

Managing the Software Reliability Objective

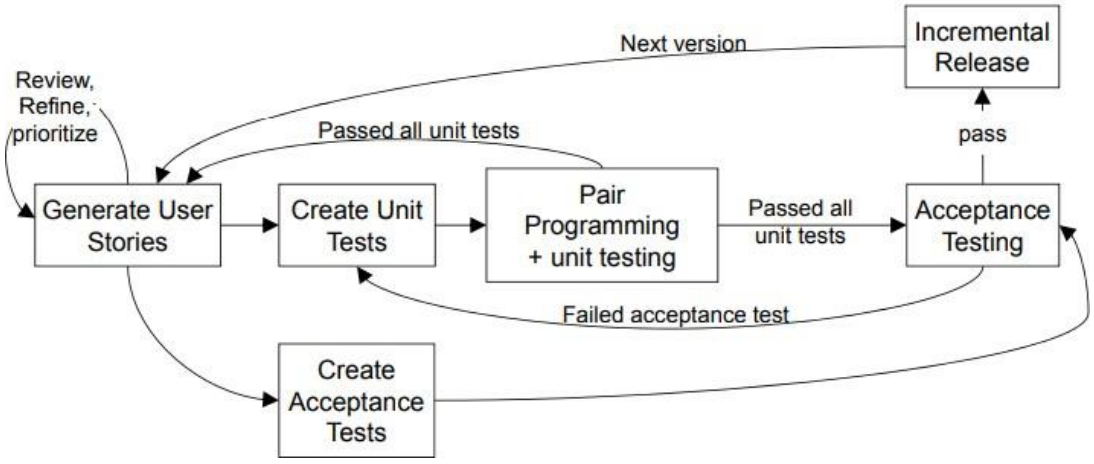
- **Defects are often inserted from the beginning of project.**

  This is usually related to the intensity of the effort, i.e. the number of engineers working on the program, the project schedule and the various design decisions that are made etc.



- **Defects are most often detected and addressed at a later date than the original design effort.**
  - Test efforts are relied on to discover most defects, this lag can have a negative impact on the program.
  - This can be mitigated against by using code inspection, but some testing will still be necessary. Code inspections should be conducted to IEEE 1028.
  - There is still a lag though between defect insertion and correction, which can have a negative impact on the program.
- **The eventual defect rate represents the reliability target, and as defects are discovered and addressed the software reliability is increased, or grown – this is termed 'Reliability Growth Management".**

Example Process: Extreme Programming (XP)



Back in the 1990s, the rise of the Internet necessitated a change in software development. If a company's success depended on the speed at which the company could grow and bring products to market, businesses needed to dramatically reduce the software development life cycle.

It was in this environment that Kent Beck created extreme programming (XP), an agile project management methodology that supports frequent releases in short development cycles to improve software quality and allow developers to respond to changing customer requirements.

Although you may recognize some of these practices and values from other project management methodologies, XP takes these practices to "extreme" levels, as the methodology's name suggests. In an interview with Informit, Kent explains:

"The first time I was asked to lead a team, I asked them to do a little bit of the things I thought were sensible, like testing and reviews. The second time there was a lot more on the line. I … asked the team to crank up all the knobs to 10 on the things I thought were essential and leave out everything else."

If you and your team need to quickly release and respond to customer requests, take a look at the values and rules of extreme programming—it could be a perfect fit.

Values of extreme programming methodology
XP is more than just a series of steps to manage projects—it follows a set of values that will help your team work faster and collaborate more effectively.

Simplicity
Teams accomplish what has been asked for and nothing more. XP breaks down each step of a major process into smaller, achievable goals for team members to accomplish.

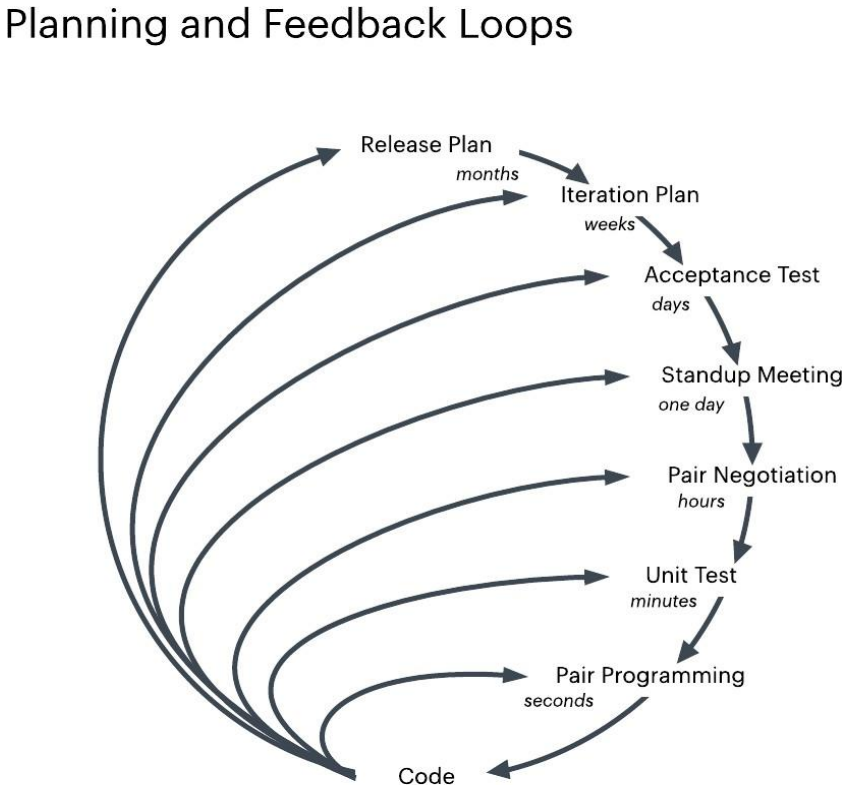
Streamlined communication
Teams work together on every part of the project, from gathering requirements to implementing code, and participate in daily standup meetings to keep all team members updated. Any concerns or problems are addressed immediately.

Consistent, constructive feedback
In XP, teams adapt their process to the project and customer needs, not the other way around. The team should demonstrate their software early and often so they can gather feedback from the customer and make the necessary changes.

Respect
Extreme programming encourages an "all for one and one for all" mentality. Each person on the team, regardless of hierarchy, is respected for their contributions. The team respects the opinions of the customers and vice versa.

Courage
Team members adapt to changes as they arise and take responsibility for their work. They tell the truth about their progress—there are no "white lies" or excuses for failure to make people feel better. There's no reason to fear because no one ever works alone.

Rules of extreme programming methodology
Don Wells published the first XP rules in 1999 to counter claims that extreme programming doesn't support activities that are necessary to software development, such as planning, managing, and designing. From planning to testing the software, follow these basic steps for each iteration.



Extreme Programming Feedback/Planning Loops (Click on image to modify online)

1. Planning

This stage is where the UX magic happens. Rather than a lengthy requirements document, the customer writes user stories, which define the functionality the customer would like to see, along with the business value and priority of each of those features. User stories don't need to be exhaustive or overly technical—they only need to provide enough detail to help the team determine how long it'll take to implement those features.

With Lucidchart, customers can create a basic flowchart and easily record and share the desired functionality.

From there, the team creates a release schedule and divides the project into iterations (one to three weeks long). Project managers might want to create a timeline or a simplified Gantt chart to share the schedule with the team.

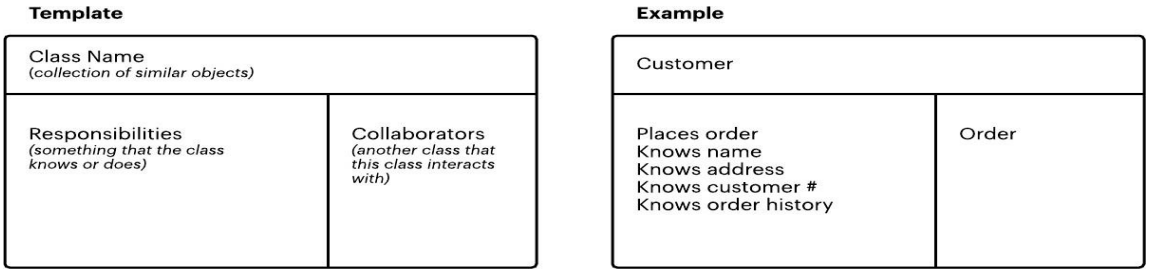
2. Managing

At this stage, the project manager will set the team up to succeed in this methodology. Everyone needs to work collaboratively and effectively communicate to avoid any slipups. This stage involves:

Creating an open workspace for your team
Setting a sustainable pace (i.e. determining the right length for iterations)
Scheduling a daily standup meeting
Measuring project velocity (the amount of work getting done on your project)
Reassigning work to avoid bottlenecks or knowledge loss
Changing the rules if XP isn't working perfectly for the team

3. Designing
This rule goes back to the value of simplicity: Start with the simplest design because it will take less time to complete than the complex solution. Don't add functionality early. Refactor often to keep your code clean and concise. Create spike solutions to explore solutions to potential problems before they put your team behind.

Kent Beck and Ward Cunningham also created class-responsibility-collaboration (CRC) cards to use as part of the XP methodology. These cards allow the entire project team to design the system and see how objects interact. If you'd like to try this brainstorming tool for yourself, get started with our Lucidchart template.



4. Coding
Then the time finally comes to implement code. XP practices collective code ownership: Everyone reviews code and any developer can add functionality, fix bugs, or refactor. For collective code ownership to work, the team should:

Choose a system metaphor (standardized naming scheme).
Practice pair programming. Team members work in pairs, at a single computer, to create code and send it into production. Only one pair integrates code at a time.
Integrate and commit code into the repository every few hours.
The customer should be available, preferably on site, during this entire process so they can answer questions and establish requirements.

5. Testing
The team performs unit tests and fixes bugs before the code can be released. They also run acceptance tests frequently.

When to use extreme programming
Still unsure whether XP will fit your team's needs, even after reading its rules and values? Extreme programming can work well for teams that:

Expect their system's functionality to change every few months.
Experience constantly changing requirements or work with customers who aren't sure what they want the system to do.
Want to mitigate project risk, especially around tight deadlines.
Include a small number of programmers (between 2 and 12 is preferable).

Are able to work closely with customers.

Are able to create automated unit and functional tests.

If collaboration and continuous development are priorities for your team, extreme programming might be worth a try. Because this highly adaptable model requires ongoing feedback from customers, anticipates errors along the way, and requires developers to work together, XP not only ensures a health product release but has also unintentionally improved productivity for development teams everywhere.

Overall Organization of a Quality Process

• Key principle of quality planning – the cost of detecting and repairing a fault increases as a function of time between committing an error and detecting the resultant faults

• therefore ... – an efficient quality plan includes matched sets of intermediate validation and verification activities that detect most faults within a short time of their introduction

• and ... – V&V steps depend on the intermediate work products and on their anticipated defects

Verification Steps for Intermediate Artifacts

• Internal consistency checks – compliance with structuring rules that define "well-formed" artifacts of that type – a point of leverage: define syntactic and semantic rules thoroughly and precisely enough that many common errors result in detectable violations

• External consistency checks – consistenc y with related artifacts – Often: conformance to a "prior" or "higher-level" specification
• Generation of correctness conjectures – Correctness conjectures: lay the groundwork for external consistency checks of other work products Often: motivate refinement of the current product

Strategies vs Plans

	Strategy	*Plan*
Scope	Organization	Project
Structure and content based on	Organization structure, experience and policy over several projects	Standard structure prescribed in strategy
Evolves	Slowly, with organization and policy changes	Quickly, adapting to project needs

Test and analysis strategies and plans,

Test and Analysis Strategy Test and Analysis Strategy
• Lessons of past experience – an organizational asset built and refined over time

• Body of explicit knowledge – more valuable than islands of individual competence – amenable to improvement – reduces vulnerability to organizational change (e.g., loss of key individuals)
• Essential for – avoiding recurring errors – maintaining consistency of the process – increasing development efficiency

Considerations in Fitting a Strategy to an Organization

• Structure and size – example
• Distinct quality groups in large organizations, overlapping of roles in smaller organizations
• Greater reliance on documents in large than small organizations
• Overall process – example
- Cleanroom requires statistical testing and forbids unit testing – fits with tight, formal specs and emphasis on reliability
- XP prescribes "test first" and pair programming – fits with fluid specifications and rapid evolution
• Application domain – example
• Safety critical domains may impose particular quality objectives and require documentation for certification

Elements of a Strategy
• Common quality requirements that apply to all or most products – unambiguous definition and measures
• Set of documents normally produced during the quality process – contents and relationships
• Activities prescribed by the overall process – standard tools and practices
• Guidelines for project staffing and assignment of roles and responsibilities

Test and Analysis Plan Test and Analysis Plan answer the following questions:
• What quality activities will be carried out?
• What are the dependencies among the quality activities and between quality and other development activities?
• What resources are needed and how will they be allocated? • How will both the process and the product be monitored?

Main Elements of a Plan
• Items and features to be verified – Scope and target of the plan
• Activities and resources – Constraints imposed by resources on activities
• Approaches to be followed – Methods and tools
• Criteria for evaluating results

Quality Goals Quality Goals
• Expressed as properties satisfied by the product – must include metrics to be monitored during the project – example: before entering acceptance testing, the product must pass comprehensive system testing with no critical or severe failures – not all details are available in the early stages of development
• Initial plan – based on incomplete information – incrementally refined

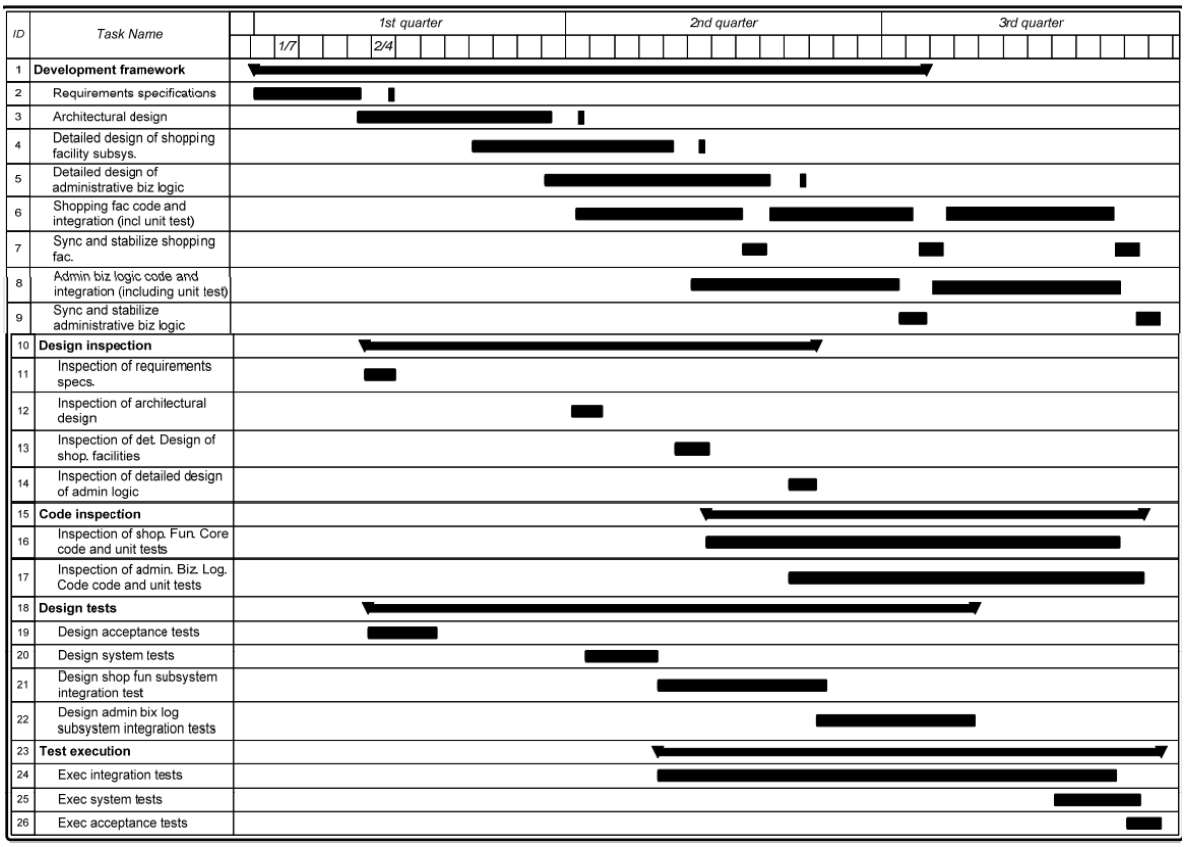
Task Schedule Task Schedule

• Initially based on – quality strategy – past experience

• Breaks large tasks into subtasks – refine as process advances

• Includes dependencies – among quality activities – between quality and development activities • Guidelines and objectives: – schedule activities for steady effort and continuous progress and evaluation without delaying development activities – schedule activities as early as possible – increase process visibility (how do we know we're on track?)

Sample Schedule



Schedule Risk Schedule Risk

• critical path = chain of activities that must be completed in sequence and that have maximum overall duration

–Schedule critical tasks and tasks that depend on critical tasks as early as possible to

• provide schedule slack

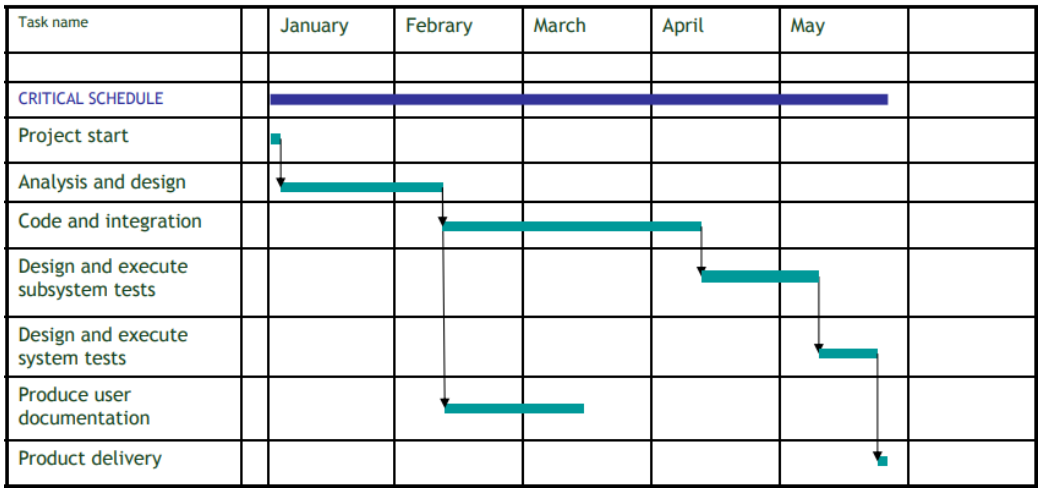
• prevent delay in starting critical tasks

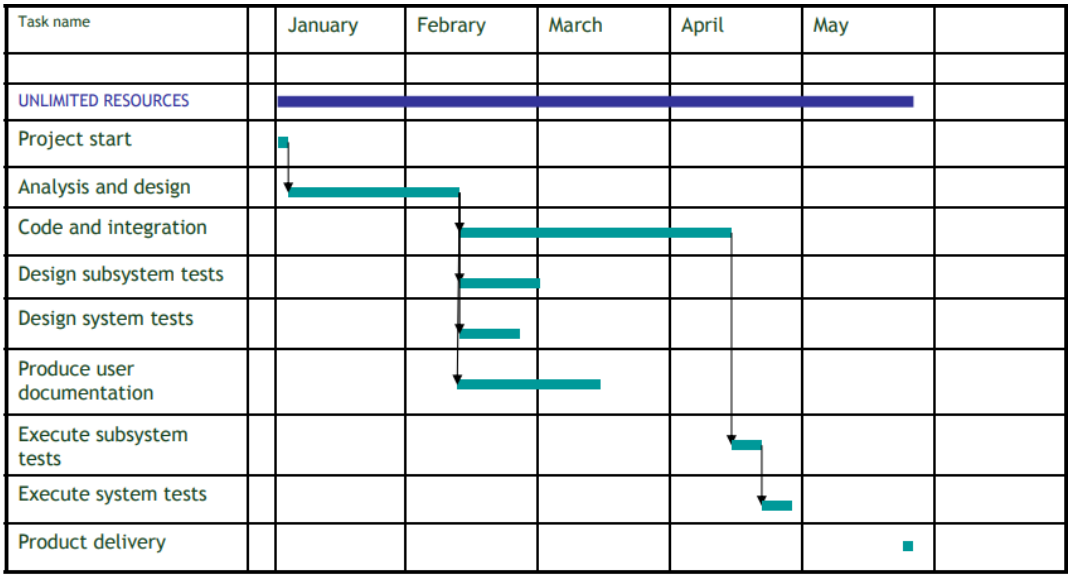
• critical dependence = task on a critical path scheduled immediately after some other task on the critical path

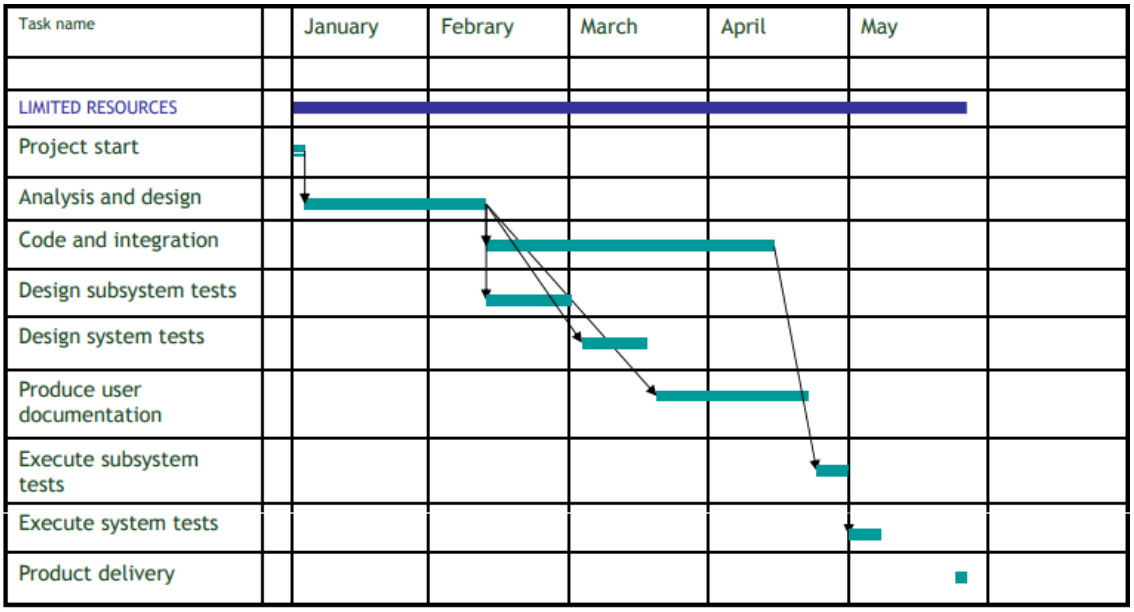
– May occur with tasks outside the quality plan (part of the project plan)

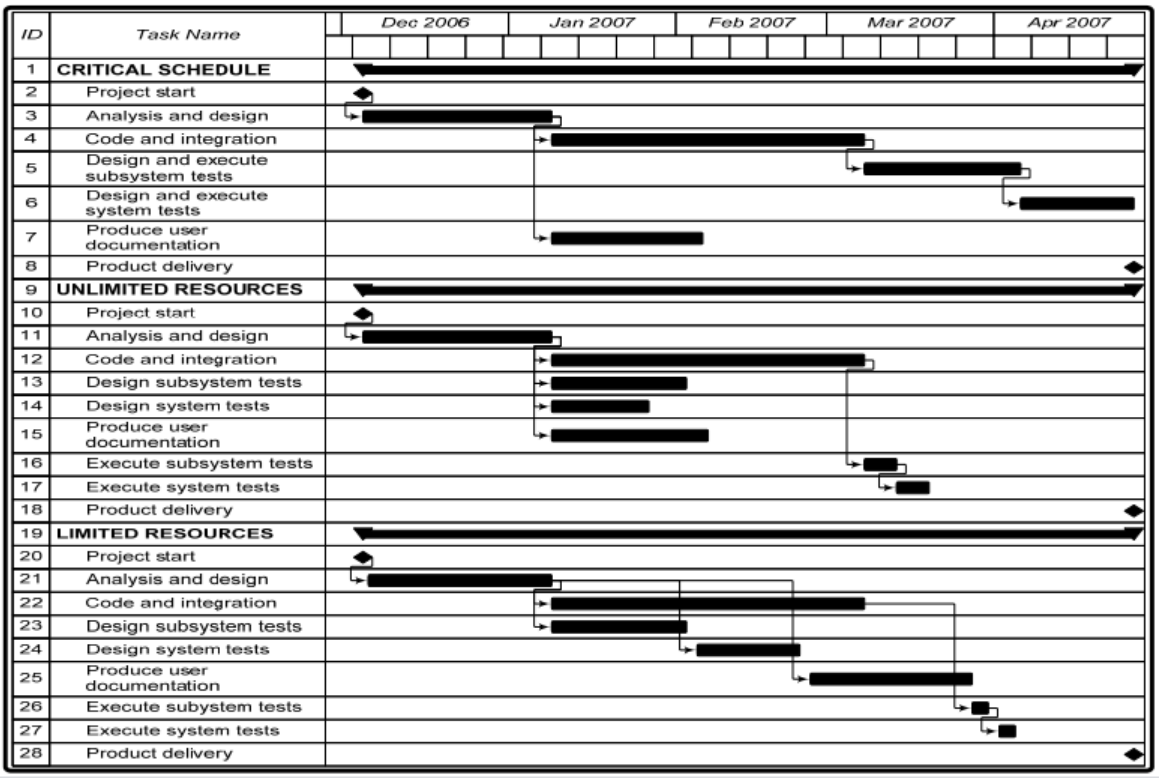
– Reduce critical dependences by decomposing tasks on critical path, factoring out subtasks that can be performed earlier

Reducing the Impact of Critical Paths









Risk planning,

Risks cannot be eliminated but they can be Risks cannot be eliminated, but they can be assessed, controlled, and monitored

• Generic management risk

– personnel

– technology
– schedule
• Quality risk
– development
– execution
– requirements

Personnel

Example Risks

- Loss of a staff member
- Staff member under-qualified for task

Control Strategies

- cross training to avoid over-dependence on individuals
- continuous education
- identification of skills gaps early in project
- competitive compensation and promotion policies and rewarding work
- including training time in project schedule

Technology

Example Risks

- High fault rate due to unfamiliar COTS component interface
- Test and analysis automation tools do not meet expectations

Control Strategies

- Anticipate and schedule extra time for testing unfamiliar interfaces.
- Invest training time for COTS components and for training with new tools
- Monitor, document, and publicize common errors and correct idioms.
- Introduce new tools in lower-risk pilot projects or prototyping exercises

Schedule

Example Risks

- Inadequate unit testing leads to unanticipated expense and delays in integration testing
- Difficulty of scheduling meetings makes inspection a bottleneck in development

Control Strategies

- Track and reward quality unit testing as evidenced by low fault densities in integration
- Set aside times in a weekly schedule in which inspections take precedence over other meetings and work
- Try distributed and asynchronous inspection techniques, with a lower frequency of face-to-face inspection meetings

Development

Example Risks

- Poor quality software delivered to testing group
- Inadequate unit test and analysis before committing to the code base

Control Strategies

- Provide early warning and feedback
- Schedule inspection of design, code and test suites
- Connect development and inspection to the reward system
- Increase training through inspection
- Require coverage or other criteria at unit test level

Test Execution

Example Risks

- Execution costs higher than planned
- Scarce resources available for testing

Control Strategies

- Minimize parts that require full system to be executed
- Inspect architecture to assess and improve testability
- Increase intermediate feedback
- Invest in scaffolding

Requirements

Example Risk

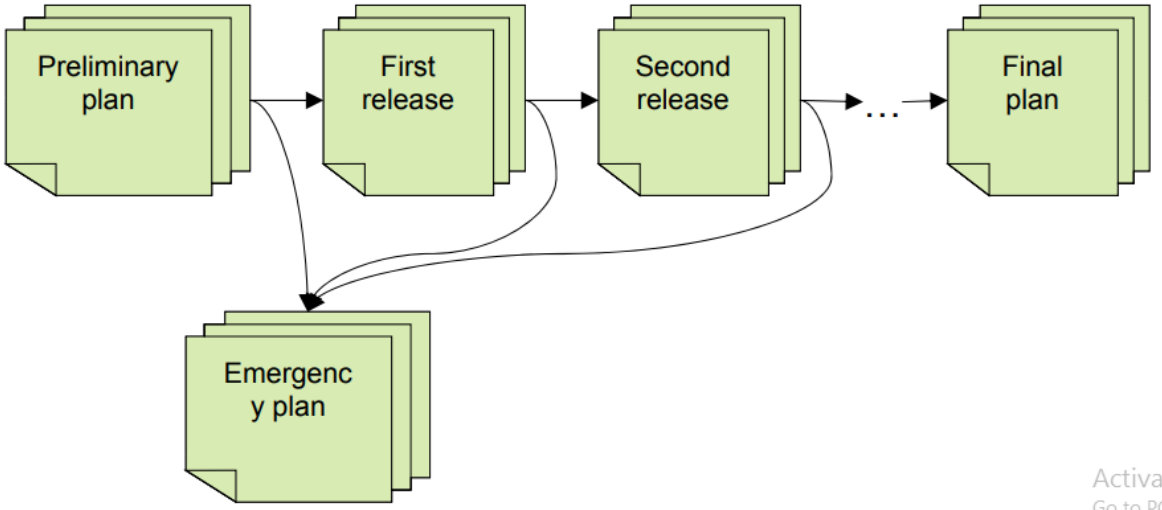
- High assurance critical requirements increase expense and uncertainty

Control Strategies

- Compare planned testing effort with former projects with similar criticality level to avoid underestimating testing effort
- Balance test and analysis
- Isolate critical parts, concerns and properties

Contingency Plan
• Part of the initial plan
– What could go wrong? How will we know, and how will we recover?
• Evolves with the plan
• Derives from risk analysis
– Essential to consider risks explicitly and in detail
• Defines actions in response to bad news
– Plan B at the ready (the sooner, the better)

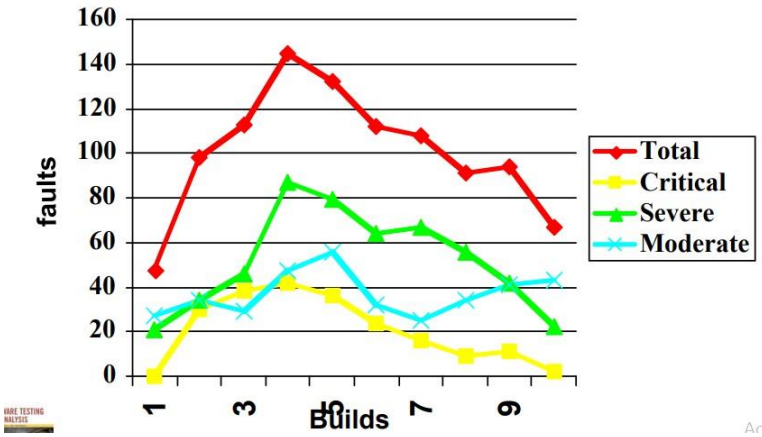
Evolution of the Plan



monitoring the process,

Identify deviations from the quality plan as
early as possible and take corrective action
• Depends on a plan that is
– realistic
– well organized
– sufficiently detailed with clear, unambiguous milestones and criteria
• A process is visible to the extent that it can be effectively monitored

Evaluate Aggregated Data by Analogy



Process Improvement
Monitoring and improvement within a project or across multiple projects:

Orthogonal Defect Classification (ODC)
&Root Cause Analysis (RCA)

Orthogonal Defect Classification (ODC)
• Accurate classification schema
– for very large projects
– to distill an unmanageable amount of detailed information
• Two main steps
– Fault classification
• when faults are detected
• when faults are fixed
– Fault analysis

ODC Fault Classification ODC Fault Classification
When faults are detected
• activity executed when the fault is revealed
• trigger that exposed the fault
• impact of the fault on the customer

When faults are fixed
• Target: entity fixed to remove the fault
• Type: type of the fault
• Source: origin of the faulty modules (in-house, library, imported, outsourced)
• Age of the faulty element (new, old, rewritten, refixed code)

ODC activities and triggers

- Review and Code Inspection
  - Design Conformance:
  - Logic/Flow
  - Backward Compatibility
  - Internal Document
  - Lateral Compatibility
  - Concurrency
  - Language Dependency
  - Side Effects
  - Rare Situation
- Structural (White Box) Test
  - Simple Path
  - Complex Path
- Functional (Black box) Test
  - Coverage
  - Variation
  - Sequencing
  - Interaction
- System Test
  - Workload/Stress
  - Recovery/Exception
  - Startup/Restart
  - Hardware Configuration
  - Software Configuration
  - Blocked Test

ODC Classification of Triggers Listed by Activity

Design Review and Code Inspection

Design Conformance A discrepancy between the reviewed artifact and a prior-stage artifact that serves as its specification.

Logic/Flow An algorithmic or logic flaw.

Backward Compatibility A difference between the current and earlier versions of an artifact that could be perceived by the customer as f**a**ilure.

Internal Document An internal inconsistency in the artifact (e.g., inconsistency between code and comments).

Lateral Compatibility An incompatibility between the artifact and some other system or module with which it should interoperate.

Concurrency A fault in interaction of concurrent processes or threads.

Language Dependency A violation of language-specific rules, standards, or best practices.

Side Effects A potential undesired interaction between the reviewed artifact and some other part of the system

Rare Situation An inappropriate response to a situation that is not anticipated in the artifact. (Error handling as specified in a prior artifact *design conformance*, not *rare situation*.)

Structural (White-Box) Test

Simple Path The fault is detected by a test case derived to cover a single program element.

Complex Path The fault is detected by a test case derived to cover a combination of program elements.

Functional (Black-Box) Test

Coverage The fault is detected by a test case derived for testing a single procedure (e.g., C function or Java method), without considering combination of values for possible parameters.

Variation The fault is detected by a test case derived to exercise a particular combination of parameters for a single procedure.

Sequencing The fault is detected by a test case derived for testing a sequence of procedure calls.

Interaction The fault is detected by a test case derived for testing procedure interactions.

System Test

Workload/Stress The fault is detected during workload or stress testing.

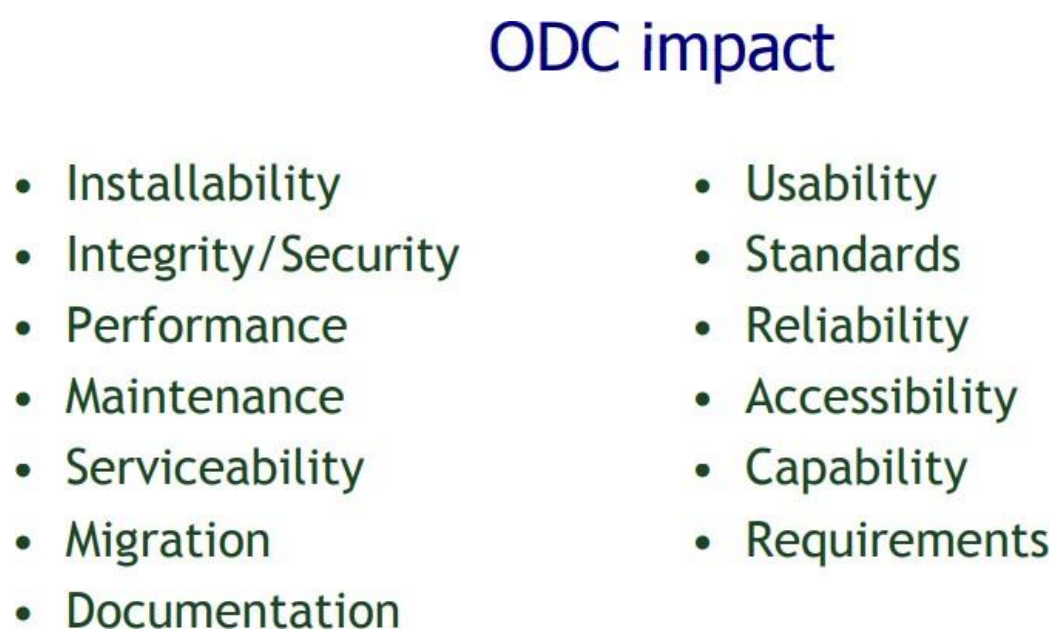
Recovery/Exception The fault is detected while testing exceptions and recovery procedures.

Startup/Restart The fault is detected while testing initialization conditions during start up or after possibly faulty shutdowns.

Hardware Configuration The fault is detected while testing specific hardware configurations

Software Configuration The fault is detected while testing specific software configurations.

Blocked Test Failure occurred in setting up the test scenario



Performance
The perceived and actual impact of the software on the time required for the customer and customer end users to complete their tasks.
Maintenance The ability to correct, adapt, or enhance the software system quickly and at minimal cost.
Serviceability Timely detection and diagnosis of failures, with minimal customer impact.
Migration Ease of upgrading to a new system release with minimal disruption to existing customer data and operations.
Documentation Degree to which provided documents (in all forms, including electronic) completely and correctly describe the structure and intended uses of the software.
Usability The degree to which the software and accompanying documents can be understood and effectively employed by the end user.
Standards The degree to which the software complies with applicable standards.
Reliability The ability of the software to perform its intended function without unplanned interruption or failure.
Accessibility The degree to which persons with disabilities can obtain the full benefit of the software system.
Capability
The degree to which the software performs its intended functions consistently with documented system requirements.
Requirements The degree to which the system, in complying with document requirements, actually meets customer expectations

ODC Classification of Defect Types for Targets *Design* and *Code*

Assignment/Initialization A variable was not assigned the correct initial value or was not assigned any initial value.
Checking Procedure parameters or variables were not properly validated before use.
Algorithm/Method A correctness or efficiency problem that can be fixed by reimplementing a single procedure or local data structure, without a design change.
Function/Class/Object
A change to the documented design is required to conform to product requirements or interface specifications.
Timing/Synchronization
The implementation omits necessary synchronization of shared resources, or violates the prescribed synchronization protocol.
Interface/Object-Oriented Messages
Module interfaces are incompatible; this can include syntactically compatible interfaces that differ in semantic interpretation of communicated data.
Relationship
Potentially problematic interactions among procedures, possibly involving different assumptions but not involving interface incompatibility.
A good RCA classification should follow the uneven distribution of faults across categories. If, for example, the current process and the programming style and environment result in many interface faults, we may adopt a finer classification for interface faults and a coarse-grain classification of other kinds of faults. We may alter the classification scheme in future projects as a result of having identified and removed the causes of many interface faults

ODC Fault Analysis (example 1/4)
• Distribution of fault types versus activities
– Different quality activities target different classes of faults
– example:
• algorithmic faults are targeted primarily by unit testing.
– a high proportion of faults detected by unit testing should belong to this class
• proportion of algorithmic faults found during unit testing
– unusually small
– larger than normal
unit tests may not have been well designed
• proportion of algorithmic faults found during unit testing unusually large
integration testing may not focused strongly enough on interface faults

ODC Fault Analysis (example 2/4)
• Distribution of triggers over time during field test
– Faults corresponding to simple usage should arise early during field test, while faults corresponding to complex usage should arise late.
– The rate of disclosure of new faults should asymptotically decrease
– Unexpected distributions of triggers over time may indicate poor system or acceptance test
• Triggers that correspond to simple usage reveal many faults late in acceptance testing
The sample may not be representative of the user population
• Continuously growing faults during acceptance test
System testing may have failed

ODC Fault Analysis (example 3/4)
Age distribution over target code
– Most faults should be located in new and rewritten code
– The proportion of faults in new and rewritten code with respect to base and re-fixed code should gradually increase
– Different patterns
⇒may indicate holes in the fault tracking and removal process
⇒may indicate inadequate test and analysis that failed in revealing faults early
– Example
• increase of faults located in base code after porting
⇒ may indicate tests for portability

Improving the process,

Improving the Process Improving the Process
• Many classes of faults that occur frequently are rooted in process and development flaws
– examples
• Shallow architectural design that does not take into account resource allocation can lead to resource allocation faults
• Lack of experience with the development environment, which leads to misunderstandings between analysts and programmers on rare and exceptional cases, can result in faults in exception handling.
• The occurrence of many such faults can be reduced by modifying the process and environment
– examples
• Resource allocation faults resulting from shallow architectural design can be reduced by introducing specific inspection tasks
• Faults attributable to inexperience with the development environment can be reduced with focused training

Improving Current and Next Processes Improving Current and Next Processes
• Identifying weak aspects of a process can be difficult
• Analysis of the fault history can help software engineers build a feedback mechanism to track relevant fault s to their root causes
– Sometimes information can be fed back directly into the current product development the current product development
– More often it helps software engineers improve the development of future products

Root cause analysis (RCA) Root cause analysis (RCA)
• Technique for identifying and eliminating process faults
– First developed in the nuclear power industry; used in many fields.
• Four main steps
– What are the faults?
– When did fault occur ? When, and when were they found?
– Why did faults occur?
– How could faults be prevented?

What are the faults?
• Identify a class of important faults
• Faults are categorized by
– severity = impact of the fault on the product
– Kind
• No fixed set of categories; Categories evolve and adapt
• Goal:
– Identify the few most important classes of faults and remove their causes
– Differs from ODC: Not trying to compare trends for different classes of faults but rather classes of faults, but rather focusing on a few important classes

Fault Severity

Level	Description	Example
Critical	The product is unusable	The fault causes the program to crash
Severe	Some product features cannot be used, and there is no workaround	The fault inhibits importing files saved with a previous version of the program, and there is no workaround
Moderate	Some product features require workarounds to use, and reduce efficiency, reliability, or convenience and usability	The fault inhibits exporting in Postscript format. Postscript can be produced using the printing facility, but with loss of usability and efficiency
Cosmetic	Minor inconvenience	The fault limits the choice of colors for customizing the graphical interface, violating the specification but causing only minor inconvenience

Pareto Distribution (80/20)
– in many populations, a few (20%) are vital and many (80%) are trivial
• Fault analysis
– 20% of the code is responsible for 80% of the fault s
• Faults tend to accumulate in a few modules
– identifying potentially faulty modules can improve the cost effectiveness of fault detection
• Some classes of faults predominate
– removing the causes of a predominant class of fault s can have a major impact on the quality of the process and of the resulting product

Why did faults occur? did faults occur?
• Core RCA step
– trace representative faults back to causes
– objective of identifying a “root” cause
• Iterative analysis

– explain the error that led to the fault
– explain the cause of that error
– explain the cause of that cause
– ...
• Rule of thumb
– "ask why six times"

Example of fault tracing Example of fault tracing
• Tracing the causes of faults requires experience Tracing the causes of faults requires experience, judgment, and knowledge of the development process
• example
– most significant class of faults = memory leaks
– cause = forgetting to release memory in exception handlers
– cause = lack of information: "Programmers can't easily
determine what needs to be cleaned up in exception handlers"
– cause = design error: cause = design error: The resource management scheme "The resource management scheme
assumes normal flow of control"
– root problem = early design problem: "Exceptional conditions were an afterthought dealt with late in design"

How could faults be prevented? How could faults be prevented?
• Many approaches depending on fault and process:
• From lightweight process changes
– example
• adding consideration of exceptional conditions to a design
inspection checklist
• To heavyweight changes:
– example
• making explicit consideration of exceptional conditions a part of all requirements analysis and design steps

The Quality Team
• The quality plan must assign roles and responsibilities to people
• Assignment of responsibility occurs at
– strategic level
• test and analysis strategy
• structure of the organization
• external requirements (e g certification agency) external requirements (e.g., certification agency)
– tactical level
• test and analysis plan

Roles and Responsibilities at Tactical Level
• balance level of effort across time
• manage personal interactions

• ensure sufficient accountability that quality tasks are not easily overlooked
• encourage objective judgment of quality
• prevent it from being subverted by schedule pressure
• foster shared commitment to quality among all team members
• develop and communicate shared knowledge and values regarding quality

Alternatives in Team Structure
• Conflicting pressures on choice of structure
– example
• autonomy to ensure objective assessment
• cooperation to meet overall project objectives
• Different structures of roles and responsibilities
– same individuals play roles of developer and tester
– most testing responsibility assigned to a distinct group
– some responsibility assigned to a distinct organization
• Distinguish
– oversight and accountability for approving a task
– responsibility for actually performing a task

Roles and responsibilities
pros and cons
• Same individuals play roles of developer and tester
– potential conflict between roles
• example
– a developer responsible for delivering a unit on schedule
– responsible for integration testing that could reveal faults that delay delivery
– requires countermeasures to control risks from conflict
• Roles assigned to different individuals
– Potential conflict between individuals
• example
– developer and a tester who do not share motivation to deliver a quality product on schedule
– requires countermeasures to control risks from conflict

Independent Testing Team Independent Testing Team
• Minimize risks of conflict between roles played by the same individual
– Example
• project manager with schedule pressures cannot
– bypass quality activities or standards
– reallocate people from testing to development
– postpone quality activities until too late in the project
• Increases risk of conflict between goals of the independent quality team and the developers
• Plan
– should include checks to ensure completion of quality activities
– Example
• developers perform module testing
• independent quality team performs integration and system testing
• quality team should check completeness of module tests

Managing Communication
• Testing and development teams must share the goal of shipping a high-quality product on schedule
– testing team
• must be perceived as relieving developers from responsibility for quality
• should not be completely oblivious to schedule pressure
• Independent quality teams require a mature development process
– Test designers must
• work on sufficiently precise specifications
• execute tests in a controllable test environment
• Versions and configurations must be well defined
• Failures and faults must be suitably tracked and monitored across versions

Testing within XP
• Full integration of quality activities with development
– Minimize communication and coordination overhead
– Developers take full responsibility for the quality of their work
– Technology and application expertise for quality tasks match expertise available for development tasks
• Plan
– check that quality activities and objective assessment are not easily tossed aside as deadlines loom
– example
• XP "test first" together with pair programming guard against some
of the inherent risks of mixing roles

Outsourcing Test and Analysis • (Wrong) motivation
– testing is less technically demanding than development and can be carried out by lower-paid and lower-skilled individuals
• Why wrong
– confuses test execution (straightforward) with analysis and test design (as demanding as design and programming)
• A better motivation
– to maximize independence
• and possibly reduce cost as (only) a secondary effect
• The plan must define
– milestones and delivery for outsourced activities
– checks on the quality of delivery in both directions

Summary
• Planning is necessary to
– order, provision, and coordinate quality activities
• coordinate quality process with overall development
• includes allocation of roles and responsibilities
– provide unambiguous milestones for judging progress
• Process visibility is key
– ability to monitor quality and schedule at each step
• intermediate verification steps: because cost grows with time between error and repair
– monitor risks explicitly, with contingency plan ready
• Monitoring feeds process improvement

– of a single project, and across projects