SOFTWARE TESTING METHODOLOGIES

COURSE FILE

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

(2015-2016)

Contents

Course coordinator **Program Coordinator** **HOD**

Syllabus:

UNIT-I:
Introduction:- Purpose of testing, Dichotomies, model for testing, consequences of bugs, taxonomy of bugs

UNIT-II:
Flow graphs and Path testing:- Basics concepts of path testing, predicates, path predicates and achievable paths, path sensitizing, path instrumentation, application of path testing.

UNIT-III:
Transaction Flow Testing:-transaction flows, transaction flow testing techniques. Dataflow testing:- Basics of dataflow testing, strategies in dataflow testing, application of dataflow testing.

UNIT-IV:
Domain Testing:-domains and paths, Nice & ugly domains, domain testing, domains and interfaces testing, domain and interface testing, domains and testability.

UNIT-V:
Paths, Path products and Regular expressions:- path products & path expression, reduction procedure, applications, regular expressions & flow anomaly detection.

UNIT-VI:
Logic Based Testing:- overview, decision tables, path expressions, kv charts, specifications.

UNIT-VII:
State, State Graphs and Transition testing:- state graphs, good & bad state graphs, state testing, Testability tips.

UNIT-VIII:
Graph Matrices and Application:-Motivational overview, matrix of graph, relations, power of a matrix, node reduction algorithm, building tools. (Student should be given an exposure to a tool like JMeter or Win-runner).

TEXT BOOKS :
1. Software Testing techniques – Baris Beizer, Dreamtech, second edition.
2. Software Testing Tools – Dr.K.V.K.K.Prasad, Dreamtech.

REFERENCES :
1. The craft of software testing – Brian Marick, Pearson Education.
2. Software Testing Techniques – SPD(Oreille)
3. Software Testing in the Real World – Edward Kit, Pearson.
4. Effective methods of Software Testing, Perry, John Wiley.
5. Art of Software Testing – Meyers, John Wiley.

Vision of the Department

To produce globally competent and socially responsible computer science engineers contributing to the advancement of engineering and technology which involves creativity and innovation by providing excellent learning environment with world class facilities.

Mission of the Department

1. To be a center of excellence in instruction, innovation in research and scholarship, and service to the stake holders, the profession, and the public.

2. To prepare graduates to enter a rapidly changing field as a competent computer science engineer.

3. To prepare graduate capable in all phases of software development, possess a firm understanding of hardware technologies, have the strong mathematical background necessary for scientific computing, and be sufficiently well versed in general theory to allow growth within the discipline as it advances.

4. To prepare graduates to assume leadership roles by possessing good communication skills, the ability to work effectively as team members, and an appreciation for their social and ethical responsibility in a global setting.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

1. To provide graduates with a good foundation in mathematics, sciences and engineering fundamentals required to solve engineering problems that will facilitate them to find employment in industry and / or to pursue postgraduate studies with an appreciation for lifelong learning.

2. To provide graduates with analytical and problem solving skills to design algorithms, other hardware / software systems, and inculcate professional ethics, inter-personal skills to work in a multi-cultural team.

3. To facilitate graduates to get familiarized with the art software / hardware tools, imbibing creativity and innovation that would enable them to develop cutting-edge technologies of multi-disciplinary nature for societal development.

PROGRAM OUTCOMES (PO)

1. An ability to apply knowledge of mathematics, science and engineering to develop and analyze computing systems.
2. an ability to analyze a problem and identify and define the computing requirements appropriate for its solution under given constraints.
3. An ability to perform experiments to analyze and interpret data for different applications.
4. An ability to design, implement and evaluate computer-based systems, processes, components or programs to meet desired needs within realistic constraints of time and space.
5. An ability to use current techniques, skills and modern engineering tools necessary to practice as a CSE professional.
6. An ability to recognize the importance of professional, ethical, legal, security and social issues and addressing these issues as a professional.
7. An ability to analyze the local and global impact of systems /processes /applications /technologies on individuals, organizations, society and environment.
8. An ability to function in multidisciplinary teams.
9. An ability to communicate effectively with a range of audiences.
10. Demonstrate knowledge and understanding of the engineering, management and economic principles and apply them to manage projects as a member and leader in a team.
11. A recognition of the need for and an ability to engage in life-long learning and continuing professional development
12. Knowledge of contemporary issues.
13. An ability to apply design and development principles in producing software systems of varying complexity using various project management tools.
14. An ability to identify, formulate and solve innovative engineering problems.

Course Objectives

The aim of this course is,

- To study the fundamental concepts of software testing which includes objectives, process, criteria, strategies, and methods.
- To discuss various software testing types and levels of testing like black and white box testing along with levels unit test, integration, regression, and system testing.

- It also helps to learn the types of bugs, testing levels with which the student can very well identify a bug and correct as when it happens.
- It provides knowledge on transaction flow testing and data flow testing techniques so that the flow of the program is tested as well.
- To learn the domain testing, path testing and logic based testing to explore the testing process easier.
- To know the concepts of state graphs, graph matrixes and transition testing along with testability tips to enhance the testing process in different way.
- To expose the advanced software testing topics, such as object-oriented software testing methods, and component-based software testing issues, challenges, and solutions.
- To gain software testing experience by applying software testing knowledge and methods to practice-oriented software testing projects.
- To gain the techniques and skills on how to use modern software testing tools to support software testing projects

Course Outcomes

- Know the basic concepts of software testing and its essentials.
- Able to identify the various bugs and correcting them after knowing the consequences of the bug.
- Use of program's control flow as a structural model is the corner stone of testing.
- Performing functional testing using control flow and transaction flow graphs.
- Know the basic techniques for deriving test cases
- Follow an effective, step-by-step process for identifying needed areas of testing, designing test conditions and building and executing test cases.
- Able to test a domain or an application and identifying the nice and ugly domains.
- Able to make a path expression and reduce them very well when needed.
- Can use the testing tools and perform the testing of any type with good perfection.
- Apply appropriate software testing tools, techniques and methods for even more effective systems during both the test planning and test execution phases of a software development project.
- Well developed knowledge in comparing the various testing strategies as well.

Mapping of Course to PEOs and POs

S.No.	Course Outcome	POs
1	Know the basic concepts of software testing and its essentials.	1,2,3
2	Able to identify the various bugs and correcting them after knowing the consequences of the bug	2,3,4,12
3	Use of program's control flow as a structural model is the corner stone of testing.	1,3,5
4	Performing functional testing using control flow and transaction flow graphs.	1,8,5
5	Know the basic techniques for deriving test cases	2,5,8,13
6	Follow an effective, step-by-step process for identifying needed areas of testing, designing test conditions and building and executing test cases.	
7	Able to test a domain or an application and identifying the nice and ugly domains	1,3,14
8	Able to make a path expression and reduce them very well when needed. .	3,7
11	Apply appropriate software testing tools, techniques and methods for even more effective systems during both the test planning and test execution phases of a software development project.	3,5,10,13

12	Well developed knowledge in comparing the various testing strategies as well.	7,13,14

Course	PEOS	POs
STM	PEO1,PEO2,PEO3	PO1PO2,PO3,PO4,PO5,PO6,PO7,PO11,PO12,PO13

Mapping of Course outcomes with Programme outcomes:

*When the course outcome weightage is < 40%, it will be given as moderately correlated (1).
*When the course outcome weightage is >40%, it will be given as strongly correlated (2).

Pos	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
STM															
CO1: Know the basic concepts of software testing and its essentials.	**2**	**1**	**2**			**2**		**1**			**2**				
CO 2: Able to identify the various bugs and correcting them after knowing the consequences of the bug	**2**		**2**	**2**		**1**			**1**				2		**Professional core**
CO 3: Use of program's control flow as a structural model is the corner stone of testing.	**2**	**2**			**1**		**2**		**1**			**1**		1	
CO 4: Performing	**2**		**1**	**1**		**2**			**2**		**2**		1		

functional testing using control flow and transaction flow graphs.															
CO 5: Know the basic techniques for deriving test cases	**1**			**1**			**2**		**2**		**2**	**2**			
CO 6: Follow an effective, step-by-step process for identifying needed areas of testing, designing test conditions and building and executing test cases.	**2**	**1**	**1**	**2**		**2**	**1**				**1**				
CO 7: Able to test a domain or an application and identifying the nice and ugly domains	**1**			**1**			**1**			**2**		**1**		2	
CO 8: Able to make a path expression and reduce them very well when needed.	**2**		**2**		**1**		**1**			**2**		**1**			

CO 9: Apply appropriate software testing tools, techniques and methods for even more effective systems during both the test planning and test execution phases of a software development project.	**2**	**1**			**1**	**1**	**2**		**2**			**2**	2	1	
CO 10: Well developed know ledge in compari ng the various testing st rategies as well.	**2**	**2**	**2**		**1**		**1**		**2**		**1**	**1**	2		

Learning Outcomes

Upon successful completion of this course, the student will be able to:

- Have an ability to apply software testing knowledge and engineering methods.
- To apply the fundamental knowledge of testing real time scenarios
- To test a simple application of their choice and to understand those learnt techniques in software development life cycle.
- Have an ability to design and conduct a software test process for a software testing project.
- Have an ability to identify the needs of software test automation, and define and develop a test tool to support test automation**.**
- Have an ability understand and identify various software testing problems, and solve these problems by designing and selecting software test models, criteria, strategies, and methods.
- Have an ability to use various communication methods and skills to communicate with their teammates to conduct their practice-oriented software testing projects.
- Have basic understanding and knowledge of **contemporary** issues in software testing, such as component-based software testing problems.
- Have an ability to use software testing methods and modern software testing tools for their testing projects.

Prerequisites

- Proper knowledge on software engineering and their concepts
- Enough knowledge on object oriented modeling and techniques
- Knowing the different types and levels of software testing process
- Good programming skills and debugging skills.

Instructional learning outcomes

S.No	Unit	Contents	Outcomes
1	I	Introduction:- Purpose of testing, Dichotomies, model for testing, consequences of bugs, taxonomy of bugs	At the end of the chapter • Convinces the student that this course is indeed important • Gives a broad overview on purpose and goals for testing • Gives a better idea on bugs. • Optimistic notions about bugs • It gives good scenario on tests and testing levels on different software models
2	II	Flow graphs and Path testing:- Basics concepts of path testing, predicates, path predicates and achievable paths, path sensitizing, path instrumentation, application of path testing.	At the end of the chapter • Student will be able to create control flow graphs from programs • Students will able to test the path with loops. • It finds a set of solutions to the path predicate expression.

3	**III**	Transaction Flow Testing:-transaction flows, transaction flow testing techniques. Dataflow testing:- Basics of dataflow testing, strategies in dataflow testing, application of dataflow testing.	At the end of the chapter • Student has a clear understanding of Specifying requirements of big, online and complicated transaction flow. • Students develop Confidence in the program data flow. data dominated design. source code for data declarations.
4	**IV**	Domain Testing:-domains and paths, Nice & ugly domains, domain testing, domains and interfaces testing, domain and interface testing, domains and testability.	At the end of the chapter • Ability to perform Domain testing by viewing programs as input data classifiers. • Performing functional and structural testing. • Classification of different domains for testing program
5	**V**	Paths, Path products and Regular expressions:- path products & path expression, reduction procedure, applications, regular expressions & flow anomaly detection.	At the end of the chapter • Understanding purpose and application of flow graphs, performing syntax and state testing • Representing path and regular expressions • Identifying structured and unstructured flow graphs

6	**VI**	Logic Based Testing:- overview, decision tables, path expressions, kv charts, specifications.	At the end of the chapter • Modeling logic based testing with decision tables • Constructing decision tables, and reducing Boolean expressions by karnaugh maps • Specifying path expressions and Kv charts for consistency
7	**VII**	State, State Graphs and Transition testing:- state graphs, good & bad state graphs, state testing, Testability tips.	At the end of the chapter • Graphical representation of state graphs • Knowing the properties of state graphs, advantages and its disadvantages • Software implementation of sate graphs • Differentiating between good and bad state graphs

8	**VIII**	Graph Matrices and Application:- Motivational overview, matrix of graph, relations, power of a matrix, node reduction algorithm, building tools. (Student should be given an exposure to a tool like JMeter or Win-runner).	At the end of the chapter • Introducing graph matrices, understanding testing theory • Applications of graph matrices • Implementation of node-reduction algorithms • Brief idea on software testing tools like JMeter or Win Runner

11.Class Time Tables IV A,B,C sec

Department of Computer Science & Engineering

ODD SEMESTER

Year/Sem/Sec: IV-B.Tech I-Semester A-Section **Room No: LH-30** **A.Y : 2015 - 16** **WEF:22-06-2015(V1)**

Class Teacher: T.SOWMYA

Time	09.30-10.20	10.20-11.10	11.10-12.00	12.00-12.50	12.50-1.30	1.30-2.20	2.20-3.10	3.10-4.00
Period	1	2	3	4	LUNCH	5	6	7
Monday	DWDM	DP	STM	LP		DWDM	CG	CC
Tuesday	LP	LP LAB				CG	DP	CC
Wednesday	CG	DWDM	CC	LP		DP	STM	LP
Thursday	STM	STM LAB				CC	CG	STM
Friday	DP	DWDM	CRT			CRT		LIBRARY
Saturday	CC	STM	CG	DP		DWDM	LP	SPORTS

S.No	Subject(T/P)	Faculty Name
1	LINUX PROGRAMMING	M.VAMSI KRISHNA
2	SOFTWARE TESTING METHODOLOGIES	A.LALITHA

3	DATA WAREHOUSING DATA MINING	T.SOUMYA	
4	COMPUTER GRAPHICS	P.SWATHI	
5	CLOUD COMPUTING	Y.PHANI KISHORE	
6	DESIGN PATTENS	CH.V.ANUPNMA	
7	SOFTWARE TESTING METHODOLOGIES LAB	A.LALITHA/CH.V.ANUPAMA	
8	DATA WAREHOUSING DATA MINING LAB	M.VAMSI KIRSHNA/T.SOUMYA/MADHURI	
9	*-Tutorial Hour/Discussion Hour		

TT. Coord:_____ HOD:_____ Dean Academics:-_____

Principal:_____

Geethanjali College of Engineering & Technology

Department of Computer Science & Engineering

ODD SEMESTER

Year/Sem/Sec: IV-B.Tech I-Semester B-Section **Room No: LH-31** **A.Y : 2015 - 16** **WEF:22-06-2015(V1)**

Class Teacher: N.RADHIKA AMARESHWARI

Time	09.30-10.20	10.20-11.10	11.10-12.00	12.00-12.50	12.50-1.30	1.30-2.20	2.20-3.10	3.10-4.00
Period	1	2	3	4		5	6	7
Monday	LP	DWDM LAB				DP	STM	CG
Tuesday	STM	DWDM	CC	LP		STM LAB		
Wednesday	DWDM	CG	CC	CG		STM	LP	DP
Thursday	DP	DWDM	LP	DWDM		STM	CG	CC
Friday	CC	DP	CRT			CRT		LIBRARY
Saturday	CG	STM	CC	LP		DP	DWDM	SPORTS

S.No	Subject(T/P)	FACULTY NAME
1	LINUX PROGRAMMING	V.SHIVA NARAYANA REDDY

2	SOFTWARE TESTING METHODOLOGIES			N.RADHIKA				
3	DATA WAREHOUSING DATA MINING			Y.SWATHI TEJA				
4	COMPUTER GRAPHICS			B.SUBBA RAO				
5	CLOUD COMPUTING			G.PRASOONA				
6	DESIGN PATTENS			K.VIJAY BHASKAR				
7	SOFTWARE TESTING METHODOLOGIES LAB			N.RADHIKA/B.SUBBA RAO/G.PRASOONA				
8	DATA WAREHOUSING DATA MINING LAB			V S N REDDY/Y.SWATHI TEJA/B.SUBBA RAO				
9	*-Tutorial Hour/Discussion Hour							

TT. Coord:_____ **HOD:_____** **Dean Academics:-_____**

Principal:_____

Geethanjali College of Engineering & Technology

Department of Computer Science & Engineering

ODD SEMESTER

Year/Sem/Sec: IV-B.Tech I-Semester C-Section **Room No: LH-32** **A.Y : 2015 - 16** **WEF:22-06-2015(v1)**

Class Teacher: A.LALITHA

Time	09.30-10.20	10.20-11.10	11.10-12.00	12.00-12.50	12.50-1.30	1.30-2.20	2.20-3.10	3.10-4.00
Period	1	2	3	4	LUNCH	5	6	7
Monday	CG	LP	DWDM	DP		CC	STM	LP
Tuesday	DP	STM	CG	DP		CC	LP	CG
Wednesday	STM	STM LAB				CG	DWDM	DP
Thursday	CC	DWDM	CG	DWDM		DMDW LAB		
Friday	LP	STM	CRT			CRT		LIBRARY

Saturday	DWDM	LP	CC	STM		CG	DP	SPORTS

S.No	Subject(T/P)			FACULTY NAME				
1	LINUX PROGRAMMING			M.VAMSI KRISHNA				
2	SOFTWARE TESTING METHODOLOGIES			A.LALITHA				
3	DATA WAREHOUSING DATA MINING			T.SOUMYA				
4	COMPUTER GRAPHICS			P.SWATHI				
5	CLOUD COMPUTING			Y.PHANI KISHORE				
6	DESIGN PATTENS			CH.V. ANUPAMA				
7	SOFTWARE TESTING METHODOLOGIES LAB			A.LALITHA/CH.V.ANUPAMA/B. SUBBA RAO				
8	DATA WAREHOUSING DATA MINING LAB			M.VAMSI KRISHNA/T.SOUMYA/MADHURI/Y.SWATHI TEJA				
9	*-Tutorial Hour/Discussion Hour							

TT. Coord:_____ **HOD:_____** **Dean Academics:-_____**

Principal:_____

12.Individual Time Table

Faculty Name:A.LALITHA **Sub/Lab: STM / CT& ST LAB (A & C) IV CSE I Sem** **A.Y:2015-16 I semester**

Time	09.30-10.20	10.20-11.10	11.10-12.00	12.00-12.50	12.50-1.30	1.30-2.20	2.20-3.10	3.10-4.00
Period	**1**	**2**	**3**	**4**		**5**	**6**	**7**
Monday			**A**		**LUNCH**		C	
Tuesday		**C**						
Wednesday	**C**	**CT& ST LAB C**					**A**	
Thursday	**A**	**CT& ST LAB A**						**A**
Friday		**C**						
Saturday		**A**		**C**				

Faculty Name:N RADHIKA	Sub/Lab: STM / CT&ST LAB (A & B) IV CSE I Sem						A.Y:2015-16 I semester	
Time	09.30-10.20	10.20-11.10	11.10-12.00	12.00-12.50	12.50-1.30	1.30-2.20	2.20-3.10	3.10-4.00
Period	**1**	**2**	**3**	**4**		**5**	**6**	**7**
Monday					LUNCH		**B**	
Tuesday	**B**					**CT &ST LAB B**		
Wednesday		**CT &ST LAB C**				**B**		
Thursday		**CT &ST LAB A**				**B**		
Friday		**B**						
Saturday								

13. Lecture Schedule

S. No	Unit No	Total no. of Periods	Topics to be covered	Regular / Additional	Teaching aids used LCD/OHP/BB	Date
1	**1**	1	Importance of the subject	regular	BB	
2		1	Course objectives and outcomes	regular	BB	
3		1	Purpose of testing	regular	BB	
4		1	Goals of testing	regular	BB	
5		2	Dichotomies	regular	BB	
6		1	Model of testing	regular	BB	
7		1	Role of models	regular	BB	
8		1	Consequences of bugs	regular	BB	
9		1	Bugs effects	regular	BB	
10		2	Taxonomy of bugs	regular	BB	
11		1	A Project case study	regular	BB	
12		1	Summary & assignment questions	regular		
12	**2**	1	Basic concepts of path testing	regular	BB	
13		1	Predicates	regular	BB	
14		2	Path predicates and achievable paths	regular	PPT	
15		1	Path sensitizing	regular	BB	
16		1	Path instrumentation	regular	BB	
17		1	Application of path testing	regular	BB	
18		1	Summary & assignment questions	regular	BB	
19	**3**	1	Transaction flow , testing	regular	PPT	

20		2	Transaction flow testing techniques	regular	PPT	
21		1	Dataflow testing: Basics of dataflow testing	regular	BB	
22		1	Strategies in dataflow testing	regular	BB	
23		1	Applications of dataflow testing	additional	BB	
24	**4**	2	Domains and paths	regular	BB	
25		2	Nice & ugly domains	regular	BB	
26		2	Domain testing	regular	BB	
27		3	Domains and interfaces testing	regular	BB	
28		3	Domain and interface testing	regular	BB	
29		2	Domains and testability	regular	BB	
30		1	Revision of domain testing	regular	BB	
31	**5**	1	Path products & path expressions	regular	BB/OHP	
32		2	Reduction procedure	regular	BB	
33		1	applications	regular	BB	
34		1	Regular expressions	regular	BB/OHP	
35		2	Flow anomaly detection	regular	BB	
36		1	Flow anomaly detection	regular	BB	
37		1	Flow anomaly detection	regular	BB	
38		1	Revision of path expressions	regular	BB	
39	**6**	1	overview	regular	BB	
40		1	Decision tables	regular	BB/OHP	
41		1	Path expressions	regular	BB	
42		1	KV charts	regular	BB/OHP/PPT	
43		1	KV charts	regular	BB/OHP/PPT	
44		2	specifications	regular	BB	
45		1	Revision of decisions tables	regular	BB	
46	**7**	1	State graphs	regular	BB/OHP	
47		1	State graphs	regular	BB/OHP	
48		1	Good & Bad state graphs	regular	BB/OHP	
49		2	Good & Bad state graphs	regular	BB/OHP	
50		1	State testing	regular	BB	
51		1	Testability tips	regular	BB	
52		1	Revision of state graphs	regular	BB	
53	**8**	2	Motivational overview	regular	BB	
54		1	Matrix of graph ,relations	regular	BB	

			, power of a matrix			
55		1	Node reduction algorithm	regular	BB	
56		1	Building tools	regular	BB	
57		1	Usage of JMeter and Win runner tools for functional/Regression testing	regular	PPT	
58		1	Creation of test script for unattended testing	regular	PPT	
59		1	Synchronization of test case	regular	BB	
60		1	Common modeling technique	regular	BB	
61		1	Rapid testing, performance testing of a data base application	regular	PPT	

Total no of classes: 61

14. Detailed Notes

UNIT-I

The Purpose of Testing

Test design and testing takes longer than program design and coding. In testing each and every module is tested. Testing is a systematic approach that the given one is correct or not. Testing find out the bugs in a given software.

Productivity and Quality in Software

Once in production, each and every stage is subjected to quality control and testing from component source inspection to final testing before shipping. If flaws are discovered at any stage, that part will be discarded or cycled back for rework and correction. The assembly line's productivity is measured by the sum of the cost of the materials, the rework, discarded components and the cost of quality assurance and testing.

Note :

By testing we get the quality of a software.

If we give the guarantee for the quality of a product then it is called Quality Assurance.

Goals for Testing

Testing and test design as a parts of quality assurance, should also focus on bug prevention. To the extent that testing and test design do not prevent bugs, they should be able to discover symptoms caused by bugs. Bug prevention is testing first goal. Bug prevention is better than the detection and correction. There is no retesting and no time wastage. The act of testing, the act of designing tests is one of the bug preventions. Before coding test can be performed. "Test, then code". Testing discover and eliminates bugs before coding.

Bug prevention – Primary goal

Bug discovery – Secondary goal

Phases in a Tester's Mental Life

Why testing ?

Phase 0 : There's no difference between testing and debugging. Here there is no effective testing, no quality assurance and no quality.

Phase 1 : The purpose of testing is to show that the software works. Testing increases, software works decreases. There is a difference between testing and debugging. If testing fails the software doesn't work.

Phase 2 : The purpose of testing is to show that the software doesn't works. The test reveals a bug, the programmer corrects it, the test designer designs and executes another test intended to demonstrate another bug. It is never ending sequence.

Phase 3 : The purpose of testing is not to prove anything, but to reduce the perceived risk of not working to an acceptable value. Here testing implements the quality control. To the extent that testing catches bugs and to the extent that those bugs are fixed, testing does improve the product. If a test is passed, then the product's quality does not change, but our perception of that quality does.

Note : testing pass or fail reduces our perception of risk about a software product.

Phase 4 : Here what testing can do and not to do. The testability is that goal for two reasons : 1.Reduce the labor of testing.

2.Testable code has fewer bugs that code that's hard to test.

Test design : Design means documenting or modeling. In test design phase the given system is tested that bugs are present or not. If test design is not formally designed no one is sure whether there was a bug or not. So, test design is a important one to get the system without any bugs.

Testing isn't everything

We must first review, inspect, read, do walkthroughs and then test. The major methods in decreasing order of effectiveness as follows :

Inspection methods : It includes walkthroughs, desk checking, formal inspection and code reading. These methods appear to be as effective as testing, but the bugs caught do not completely overload.

Design style : It includes testability, openness and clarity to prevent bugs.

Static Analysis Methods : It includes of strong typing and type checking. It eliminates an entire category of bugs.

Languages : The source language can help reduce certain kinds of bugs. Programmers find new kinds of bugs in new languages, so the bug rate seems to be independent of the languages used.

Design methodology and Development Environment : Design methodology can prevent many kinds of bugs. Development process used and the environment in which what methodology is embedded.

The pesticide paradox and the complexity Barrier

1.Pesticide Paradox :. Every method you use to prevent or find bugs leaves a residue of subtler bugs again which those methods are ineffectual

2.Complexity Barrier : Software complexity grows to the limits of our ability to manage hat complexity.

Some Dichotomies

Testing versus Debugging : The phrase " Test and Debug " is treated as a single word.

The purpose of testing is to show that a program has bugs.

The purpose of debugging is find the error or misconception that led to the program's failure and to design and implement the program changes that correct the error.

Note : Debugging usually follows testing, but they differ as to goals, methods and psychology.

Function versus Structure : Tests can be designed from a functional or structural point of view. In functional testing the program or system is treated as a blackbox.

Black box Testing : Here we don't know the inter functionality we knows only about the input and the outcome. In structural testing does look at the implementation details, as programming style, control method, source language, database design and coding details.

White box Testing : Here inter functionality is considered

Designer versus the Tester : Designing depends on a system's structural details. The more you know about the design, the likelier you are to eliminate useless tests.

Tester, test-team member or test designer contrast to the programmer and program designer. Testing includes unit testing to unit integration, component testing to component integration, system testing to system integration.

Modularity versus Efficiency : Both tests and systems can be modular. A module is a discrete, well defined small component of a system. The smaller the component, the easier is to understand but every component has interfaces with other components and all component interfaces are sources of confusion. Smaller the component less the bugs.Large components reduce external interfaces but have complicated internal logic that may be difficult or impossible to understand.Testing can and should likewise be originated in to modular components, small, independent test cases have the virtue of easy repeatability.

Small versus Large : Programming in the large means constructing programs that consist of many components written by many different persons.Programming in the small is what we do for ourselves in the privacy of our own offices or as homework exercises in an undergraduate programming course.Qualitative changes occur with size and so must testing methods and quality criteria.

The Builder Versus the Buyer :

Just as programmers and testers can merge and become one, so can builder and buyer.

1. The builder, who designs for and is accountable to

2. The buyer, who pays for the system in the hope of profits from providing services to

A MODEL FOR TESTING

The Project: a real-world context characterized by the following model project.

Application: It is a real-time system that must provide timely responses to user requests for services. It is an online system connected to remote terminals.

Staff: The programming staff consists of twenty to thirty programmers. There staff is used for system's design.

Schedule: The project will take 24 months from the start of design to formal acceptance by the customer. Acceptance will be followed by a 6-month cutover period. Computer resources for development and testing will be almost adequate.

Specification: means requirements.

Acceptance Test: The system will be accepted only after a formal acceptance test. The application is not new, so part of the formal test already exists. At first the customer will intend to design the acceptance test, but later it will become the software design team's responsibility.

Personnel: Management's attitude is positive and knowledgeable about the realities of such projects.

Standards: Programming and test standards exist and are usually followed. They understand the role of interfaces and the need for interface standards.

Objectives: The system is the first of many similar systems that will be implemented in the future. No two will be identical, but they will have 75% of the code in common. Once installed, the system is expected to operate profitably for more than 10 years.

Source: One-third of the code is new, one-third extracted from a previous, reliable, but poorly documented system, and one-third is being rehosted

History: One programmer will quit before his components are tested. Another programmer will be fired before testing begins. A facility and/or hardware delivery problem will delay testing for several weeks and force second- and third-shift work. Several important milestones will slip but the delivery date will be met.

Overview

The process starts with a program embedded in an environment, such as a computer, an operating system, or a calling program. This understanding leads us to create three models:

- a model of the environment,
- a model of the program,
- a model of the expected bugs.

From these models we create a set of tests, which are then executed. The result of each test is either expected or unexpected. If unexpected, it may lead us to revise the test, our model or concept of how the program behaves, our concept of what bugs are possible, or the program itself. Only rarely would we attempt to modify the environment.

The Environment

A program's environment is the hardware and software required to make it run. For online systems the environment may include communications lines, other systems, terminals, and operators. The environment also includes all programs that interact with—and are used to create— the program under test, such as operating system, loader, linkage editor, compiler, utility routines. If testing reveals an unexpected result, we may have to change our beliefs (our model of the environment) to find out what went wrong. But sometimes the environment could be wrong: the bug could be in the hardware or firmware after all.

Bugs

A bad specification may lead us to mistake good behavior for bugs, and vice versa. An unexpected test result may lead us to change our notion of what a bug is—that is to say, our model of bugs. If you hold any of the following beliefs, then disabuse yourself of them because as long as you believe in such things you will be unable to test effectively and unable to justify the dirty tests most programs need.

Benign Bug Hypothesis: The belief that bugs are nice, tame, and logical. Only weak bugs have a logic to them and are amenable to exposure by strictly logical means. Subtle bugs have no definable pattern— they are wild cards.

Bug Locality Hypothesis: The belief that a bug discovered within a component affects only that component's behavior; that because of structure, language syntax, and data organization, the symptoms of a bug are localized to the component's designed domain. Only weak bugs are so localized. Subtle bugs have consequences that are arbitrarily far removed from the cause in time and/or space from the component in which they exist.

Control Bug Dominance : The belief that errors in the control structure of programs dominate the bugs. While many easy bugs, especially in components, can be traced to control-flow errors, data-flow and data-structure errors are as common. Subtle bugs that violate data-structure boundaries and data/code separation can't be found by looking only at control structures.

Code/Data Separation: The belief, especially in HOL programming, that bugs respect the separation of code and data.* Furthermore, in real systems the distinction between code and data can be hard to make, and it is exactly that blurred distinction that permit such bugs to exist.

Lingua Salvator Est. : The hopeful belief that language syntax and semantics (e.g., structured coding, strong typing, complexity hiding) eliminates most bugs. True, good language features do help prevent the simpler component bugs but there's no statistical evidence to support the notion that such features help with subtle bugs in big systems.

Corrections Abide: The mistaken belief that a corrected bug remains corrected. Here's a generic counterexample. A bug is believed to have symptoms caused by the interaction of components A and B but the real problem is a bug in C, which left a residue in a data structure used by both A and B. The bug is "corrected" by changing A and B. Later, C is modified or removed and the symptoms of A and B recur. Subtle bugs are like that.

Silver Bullets: The mistaken belief that X (language, design method, representation, environment— name your own) grants immunity from bugs. Easy-to-moderate bugs may be reduced, but remember the pesticide paradox.

Sadism Suffices: The common belief, especially by independent testers, that a sadistic streak, low cunning, and intuition are sufficient to extirpate most bugs. You only catch easy bugs that way. Tough bugs need methodology and techniques, so read on.

Angelic Testers: The ludicrous belief that testers are better at test design than programmers are at code design.

Tests

Tests are formal procedures. Inputs must be prepared, outcomes predicted, tests documented, commands executed, and results observed; all these steps are subject to error. There is nothing magical about testing and test design that immunizes testers against bugs. An unexpected test result is as often cause by a test bug as it is by a real bug.* Bugs can creep into the documentation, the inputs, and the commands and becloud our observation of results. An unexpected test result, therefore, may lead us to revise the tests. Because the tests are themselves in an environment, we also have a mental model of the tests, and instead of revising the tests, we may have to revise that mental model.

Testing and Levels

We do three distinct kinds of testing on a typical software system: unit/ component testing, integration testing, and system testing. The objectives of each class is different and therefore, we can expect the mix of test methods used to differ. They are:

Unit, Unit Testing: A unit is the smallest testable piece of software, by which I mean that it can be compiled or assembled, linked, loaded, and put under the control of a test harness or driver. A unit is usually the work of one programmer and it consists of several hundred or fewer, lines of source code. Unit testing is the testing we do to show that the unit does not satisfy its functional specification and/or that its implemented structure does not match the intended design structure. When our tests reveal such faults, we say that there is a unit bug.

Component, Component Testing : A **component** is an **integrated aggregate** of one or more units. A unit is a component; a component with subroutines it calls is a component, etc. By this (recursive)definition, a component can be anything from a unit to an entire system.
Component testing is the testing we do to show that the component does not satisfy its functional specification and/or that its implemented structure does not match the intended design structure. When our tests reveal such problems, we say that there is a **component bug**.

Integration, Integration Testing : **Integration** is a *process* by which components are aggregated to create larger components. **Integration testing** is testing done to show that even though the components were individually satisfactory, as demonstrated by successful passage of component tests, the combination of components are incorrect or inconsistent. For example, components A and B have both passed their component tests. Integration

testing is aimed as showing inconsistencies between A and B. Examples of such inconsistencies are improper call or return sequences, inconsistent data validation criteria, and inconsistent handling of data objects. Integration testing should not be confused with testing integrated
objects, which is just higher level component testing. Integration testing is specifically aimed at
exposing the problems that arise from the combination of components. The sequence, then, consists of component testing for components A and B, integration testing for the combination of A and B, and finally, component testing for the "new" component (A,B).*****

System, System Testing : A **system** is a big component. **System testing** is aimed at revealing bugs that cannot be attributed to components as such, to the inconsistencies between components, or to the planned interactions of components and other objects. System testing concerns issues and behaviors that can only be exposed by testing the entire integrated system or a major part of it. System testing includes testing for performance, security, accountability, configuration sensitivity, start-up, and recovery.

The Role of Models

Testing is a process in which we create mental models of the environment, the program, human nature,and the tests themselves. Each model is used either until we accept the behavior as correct or until themodel is no longer sufficient for the purpose.

PLAYING POOL AND CONSULTING ORACLES

Playing Pool **:** Testing is like playing pool. There's real pool and there's kiddie pool and real testing and kiddie testing. In kiddie testing the tester says, after the fact, that the observed outcome of the test was the expected outcome. In real testing *the outcome is predicted and documented before the test is run***.**

Oracles **:** An oracle is any program, process, or body of data that specifies the expected outcome of a set of tests as applied to a tested object. There are as many different kinds of oracles as there are testing concerns. The most common oracle is an input/outcome oracle—an oracle that specifies the expected outcome for a specified input.

Sources of Oracles

If every test designer had to analyze and predict the expected behavior for every test case for every component, then test design would be very expensive. The hardest part of test design is predicting the expected outcome, but we often have oracles that reduce the work. Here are some sources of oracles:

Kiddie Testing *:* Run the test and see what comes out. The observed outcome of the test was the
expected outcome.

Regression Test Suites **:** Most of the tests you need will have been run on a previous version. Most of those tests should have the same outcome for the new version. Outcome prediction is therefore needed only for changed parts of components.

Purchased Suites and Oracles **:** Highly standardized software that (should) differ only as to
implementation often has commercially available test suites and oracles. The most common examples are compilers for standard languages, communications protocols, and mathematical routines. As more software becomes standardized, more oracles will emerge as products and services.

Existing Program *:* A working, trusted program is an excellent oracle. The typical use is when the program is being rehosted to a new language, operating system, environment, configuration, or to some combination of these, with the intention that the behavior should not change as a result of the rehosting.

COMPLETE TESTING POSSIBLE?

If the objective of testing were to *prove* that a program is free of bugs, then testing not only would be practically impossible, but also would be theoretically impossible. Three different approaches can be used to demonstrate that a program is correct: tests based on structure, tests based on function, and formal proofs of correctness.

Functional Testing *:* Every program operates on a finite number of inputs.These inputs are of binary input stream. A complete functional test would consist of subjecting the program to all possible input streams. For each input the routine either accepts the stream and produces a correct outcome, accepts the stream and produces an incorrect outcome, or rejects the stream and tells us that it did so

Structural Testing *:* One should design enough tests to ensure that every path through the routine is exercised at least once. Right off that's impossible, because some loops might never terminate. A small routine can have millions or billions of paths, so total **path testing** is usually impractical, although it can be done for some routines.

Correctness Proofs *:* Formal proofs of correctness rely on a combination of functional and structural concepts. Requirements are stated in a formal language (e.g., mathematics), and each program statement is examined and used in a step of an inductive proof that the routine will produce the correct outcome for all possible input sequences.

THE TAXONOMY OF BUGS
CONSEQUENCES OF BUGS

Importance of Bugs :

Frequency : There are different bug frequency statistics.

Correction Cost : Cost of correcting the bug. That cost is the sum of two factors: (1) the cost of discovery and (2) the cost of correction. These costs go up dramatically the later in the development cycle the bug is discovered. Correction cost also depends on system size. The larger the system the more it costs to correct the same bug.

Installation Cost : Installation cost depends on the number of installations: small for a single-user program, but how about a PC operating system bug? Installation cost can dominate all other costs—fixing one simple bug and distributing the fix could exceed the entire system's development cost.

Consequences : This is measured by the mean size of the awards made by juries to the victims of your bug. A reasonable metric for bug importance is:

importance($) = frequency*(correction_cost + installation_cost + consequential_cost)

How Bugs Affect Us -- Consequences
Bug consequences range from mild to catastrophic. Consequences should be measured in human rather than machine terms because it is ultimately for humans that we write programs.

Consequences :

1. *Mild* : The symptoms of the bug offend us aesthetically; a misspelled output or a misaligned printout.
2. *Moderate* : Outputs are misleading or redundant. The bug impacts the system's performance.
3. *Annoying* : The system's behavior, because of the bug, is dehumanizing. Names are truncated or arbitrarily modified. Bills for $0.00 are sent. Operators must use unnatural command sequences and must trick the system into a proper response for unusual bug-related cases.
4. *Disturbing* : It refuses to handle legitimate transactions. The automatic teller machine won't give you money. My credit card is declared invalid.
5. *Serious* : It loses track of transactions: not just the transaction itself (your paycheck), but the fact that the transaction occurred. Accountability is lost.
6. *Very Serious* : Instead of losing your paycheck, the system credits it to another account or converts deposits into withdrawals. The bug causes the system to do the wrong transaction.
7. *Extreme* : The problems aren't limited to a few users or to a few transaction types. They are frequent and arbitrary instead of sporadic or for unusual cases.
8. *Intolerable* : Long-term, unrecoverable corruption of the data base occurs and the corruption is not easily discovered. Serious consideration is given to shutting the system down.
9. *Catastrophic* : The decision to shut down is taken out of our hands because the system fails.
10. *Infectious* : What can be worse than a failed system? One that corrupts other systems even though it does not fail in itself; that erodes the social or physical environment; that melts nuclear reactors or starts wars; whose influence, because of malfunction, is far greater than expected; a system that kills.

Flexible Severity Rather Than Absolutes

Many programmers, testers, and quality assurance workers have an absolutist attitude toward bugs."Everybody *knows* that a program must be *perfect* if it's to work: if there's a bug, it *must* be fixed."Metrics :

- ***Correction Cost :*** The cost of correcting a bug has almost nothing to do with symptom severity.
- Catastrophic, life-threatening bugs could be trivial to fix, whereas minor annoyances could require major rewrites to correct.
- ***Context and Application Dependency :*** The severity of a bug, for the same bug with the same
- symptoms, depends on context. For example, a roundoff error in an orbit calculation doesn't mean much in a spaceship video game but it matters to real astronauts.
- ***Creating Culture Dependency :*** What's important depends on the creators of the software and their cultural aspirations. Test tool vendors are more sensitive about bugs in their products than, say, games software vendors.
- ***User Culture Dependency :*** What's important depends on the user culture. An R&D shop might accept a bug for which there's a workaround; a banker would go to jail for that same bug; and naive users of PC software go crazy over bugs that pros ignore.
- ***The Software Development Phase :*** Severity depends on development phase. Any bug gets more severe as it gets closer to field use and more severe the longer it's been around—more severe because of the dramatic rise in correction cost with time. Also, what's a trivial or subtle bug to the designer means little to the maintenance programmer for whom all bugs are equally mysterious.

The Nightmare List and When to Stop Testing

Quantifying the Nightmare :

1. List your worst software nightmares. State them in terms of the symptoms they produce and how your user will react to those symptoms.
2. Convert the consequences of each nightmare into a cost. Usually, this is a labor cost for correcting the nightmare.
3. Order the list from the costliest to the cheapest and then discard the low-concern nightmares with which you can live.
4. Measure the kinds of bugs that are likely to create the symptoms expressed by each nightmare. Most bugs are simple goofs once you find and understand them. This can be done by bug design process.
5. For each nightmare, then, you've developed a list of possible causative bugs. Order that list bydecreasing probability. Judge the probability based on your own bug statistics, intuition, experience, etc. The same bug type will appear in different nightmares. The importance of a bug type is calculated by multiplying the expected cost of the nightmare by the probability of the bug and summing across allnightmares:

$$\text{importance of bug type } i = \sum_{\text{all nightmares}} C_j P_{(\text{bug type } i \text{ in nightmare } j)}$$

6. Rank the bug types in order of decreasing importance to you.
7. Design tests (based on your knowledge of test techniques) and design your quality assuranceinspection process by using the methods that are most effective against the most important bugs.
8. If a test is passed, then some nightmares or parts of them go away. If a test is failed, then a nightmares possible, but upon correcting the bug, it too goes away. Testing, then, gives you information you can use to revise your estimated nightmare probabilities.
9. Stop testing when the probability of all nightmares has been shown to be inconsequential as a result of hard evidence produced by testing.

General : There is no universally correct way to categorize bugs. This taxonomy is not rigid. Bugs are difficult to categorize. A given bug can be put into one or another category depending on its history and the programmer's state of mind.

Requirements, Features, and Functionality Bugs

- ***Requirements and Specifications :***
  Requirements and the specifications developed from them can be incomplete, ambiguous, or self-contradictory. They can be misunderstood or impossible to understand. The specification may assume, but not mention, other specifications and prerequisites that are known to the specifier but not to the designer. And specifications that don't have these flaws may change while the design is in progress. Features are modified, added, and deleted. The designer has to hit a moving target and occasionally misses.
  Requirements, especially as expressed in a specification (or often, as *not* expressed because there is no specification) are a major source of expensive bugs.
- ***Feature Bugs :*** Specification problems usually create corresponding feature problems. A feature can be wrong, missing, or superfluous. A missing feature or case is the easiest to detect and correct. A wrong feature could have deep design implications. Extra features were once considered desirable. "Free"features are rarely free. Removing the features might complicate the software, consume more resources, and foster more bugs.
- ***Feature Interaction :*** Features usually come in groups of related features. The features of each group and the interaction of features within each group are usually well tested. The problem is unpredictable interactions between feature groups or even between individual features.

Specification and Feature Bug Remedies

Short-Term Support : Specification languages facilitate formalization of requirements and (partial)***** inconsistency and ambiguity analysis. With formal specifications, partially to fully automatic test case generation is possible. Generally, users and developers of such products have found them to be costeffective.

Long-Term Support : Assume that we have a great specification language and that it can be used to create unambiguous, complete specifications with unambiguous, complete tests and consistent test criteria. A specification written in that language could theoretically be compiled into object code(ignoring efficiency and practicality issues). But this is just programming in HOL squared. The specification problem has been shifted to a higher level but not eliminated.

Testing Techniques

Most **functional test techniques**—that is, those techniques which are based on a behavioral description of software, such as **transaction flow testing** , **syntax testing**, **domain testing**, **logic testing**, and **state testing** are useful in testing functional bugs

Structural Bugs

Control and Sequence Bugs : Control and sequence bugs include paths left out, unreachable code, improper nesting of loops, loop-back or loop-termination criteria incorrect, missing process steps, duplicated processing, unnecessary processing, rampaging GOTO's, ill-conceived switches, spaghetti code, and worst of all, pachinko code. Control and sequence bugs at all levels are caught by testing, especially structural testing, more specifically, path testing, combined with a bottom-line functional test based on a specification. These bugs are partially prevented by language choice (e.g., languages that restrict control-flow options) and style, and most important, lots of memory.

Logic Bugs : Bugs in logic, especially those related to misunderstanding how case statements and logic operators behave singly and in combinations, include nonexistent cases, improper layout of cases, "impossible" cases that are not impossible, a "don't-care" case that matters, improper negation of a boolean expression (for example, using "greater than" as the negation of "less than"), improper simplification and combination of cases, overlap of exclusive cases, confusing "exclusive OR" with "inclusive OR."
Logic bugs are not really different in kind from arithmetic bugs. They are likelier than arithmetic bugs because programmers, like most people, have less formal training in logic at an early age than they do in arithmetic. The best defense against this kind of bug is a systematic analysis of cases. Logic-based testing is helpful.

Processing Bugs : Processing bugs include arithmetic bugs, algebraic, mathematical function evaluation, algorithm selection, and general processing. Many problems in this area are related to incorrect conversion from one data representation to another.

Initialization Bugs : Initialization bugs are common, and experienced programmers and testers know they must look for them. Both improper and superfluous initialization occur. The latter tends to be less harmful but can affect performance. Typical bugs are as follows: forgetting to initialize working space, registers, or data areas before first use or assuming that they are initialized elsewhere; a bug in the first value of a loop-control parameter; accepting an initial value without a validation check; and initializing to the wrong format, data representation, or type.

Data-Flow Bugs and Anomalies

A **data-flow anomaly** occurs when there is a path along which we expect to do something unreasonable with data, such as using an uninitialized variable, attempting to use a variable before it exists, modifying data and then not storing or using the result, or initializing twice without an intermediate use. Although part of data-flow anomaly detection can be done by the compiler based on information known at compile time, much can be detected only by execution and therefore is a subject for testing.

Data Bugs : Data bugs include all bugs that arise from the specification of data objects, their formats,the number of such objects, and their initial values. Data bugs are at least as common as bugs in code, but they are often treated as if they did not exist at all. Underestimating the frequency of data bugs is caused by poor bug accounting.

Dynamic Versus Static : Dynamic data are transitory. Whatever their purpose, they have a relatively short lifetime, typically the processing time of one transaction. A storage object may be used to hold dynamic data of different types, with different formats, attributes, and residues. The basic problem is leftover garbage in a shared resource. This can be handled in one of three ways:
(1) cleanup after use by the user,
(2) common cleanup by the resource manager, and
(3) no cleanup. Static data are fixed in form
and content. Whatever their purpose, they appear in the source code or data base, directly or indirectly, as, for example, a number, a string of characters, or a bit pattern. Static data need not be explicit in the source code.

Coding Bugs : Coding errors of all kinds can create any of the other kinds of bugs. Syntax errors are generally not important in the scheme of things if the source language translator has adequate syntax checking. The most common kind of coding bug, and often considered the least harmful, are documentation bugs (i.e., erroneous comments). Although many documentation bugs are simple spelling errors or the result of poor writing, many are actual errors—that is, misleading or erroneous comments.

Interface, Integration, and System Bugs

External Interfaces
The external interfaces are the means used to communicate with the world. These include devices, actuators, sensors, input terminals, printers, and communication lines. Other external interface bugs include: invalid timing or sequence assumptions related to external signals; misunderstanding external input and output formats; and insufficient tolerance to bad input data.

Internal Interfaces
Internal interfaces are in principle not different from external interfaces, but there are differences in practice because the internal environment is more controlled. The external environment is fixed and the system must adapt to it but the internal environment, which consists of interfaces with other components, can be negotiated. Internal interfaces have the same problems external interfaces have, as well as a few more that are more closely related to implementation details: protocol-design bugs, input and output format bugs, inadequate protection against corrupted data, wrong subroutine call sequence, call-parameter bugs, misunderstood entry or exit parameter values. The main objective of integration testing is to test all internal interfaces.

Hardware Architecture
Software bugs related to hardware architecture originate mostly from misunderstanding how the
hardware works. Here are examples: paging mechanism ignored or misunderstood, address-generation error, I/O-device operation or instruction error, I/O-device address error, misunderstood device-status code, improper hardware simultaneity assumption, hardware race condition ignored, data format wrong for device, etc. The remedy

for hardware architecture and interface problems is two-fold: (1) good programming and testing and (2) centralization of hardware interface software in programs written by hardware interface specialists.

Operating System
Program bugs related to the operating system are a combination of hardware architecture and interface bugs, mostly caused by a misunderstanding of what it is the operating system does. and, of course, the operating system could have bugs of its own. Operating systems can lull the programmer into believing that all hardware interface issues are handled by it.

Software Architecture
Software architecture bugs are often the kind that are called "interactive." Routines can pass unit and integration testing without revealing such bugs. Many of them depend on load, and their symptoms emerge only when the system is stressed. They tend to be the most difficult kind of bug to find and exhume. Here is a sample of the causes of such bugs: assumption that there will be no interrupts, failure to block or unblock interrupts, assumption that code is reentrant or not reentrant, bypassing data interlocks, failure to close or open an interlock, etc.

Control and Sequence Bugs
System-level control and sequence bugs include: ignored timing; assuming that events occur in a
specified sequence; starting a process before its prerequisites are met (e.g., working on data before all the data have arrived from disc); waiting for an impossible combination of prerequisites; not recognizing when prerequisites have been met; specifying wrong priority, program state, or processing level; missing, wrong, redundant, or superfluous process steps.
The remedy for these bugs is in the design. Highly structured sequence control is helpful. Specialized, internal, sequence-control mechanisms, such as an internal job control language, are useful.

Integration Bugs
Integration bugs are bugs having to do with the integration of, and with the interfaces between,
presumably working and tested components. Most of these bugs result from inconsistencies or
Incompatibilities between components. All methods used to transfer data directly or indirectly between components and all methods by which components share data can host integration bugs and are therefore proper targets for integration testing. The communication methods include data structures, call sequences, registers, semaphores, communication links, protocols, and so on.

System Bugs
System bugs is a catch-all phrase covering all kinds of bugs that cannot be ascribed to components or to their simple interactions, but result from the totality of interactions between many components such as programs, data, hardware, and the operating system.

Test and Test Design Bugs

Testing
Testers have no immunity to bugs. Tests, especially system tests, require complicated scenarios and databases. They require code or the equivalent to execute, and consequently they can have bugs. Test bugs are not software bugs, it's hard to tell them apart, and much labor can be spent making the distinction.

Test Criteria
The specification is correct, it is correctly interpreted and implemented, and a seemingly proper test has been designed; but the criterion by which the software's behavior is judged is incorrect or impossible.

Remedies

Test Debugging : The first remedy for test bugs is testing and debugging the tests. The differences between test debugging and program debugging are not fundamental. Test debugging is usually easier because tests, when properly designed, are simpler than programs and do not have to make concessions to efficiency. Also, tests tend to have a localized impact relative to other tests, and therefore the complicated interactions that usually plague software designers are less frequent. We have no magic prescriptions for test debugging—no more than we have for software debugging