

Introduction to Testing

Dr. Durga Prasad Mohapatra
Professor
Department Of CSE
NIT, Rourkela.

Defect Reduction Techniques

- Review
- Testing
- Formal verification
- Development process
- Systematic methodologies

Why Test?

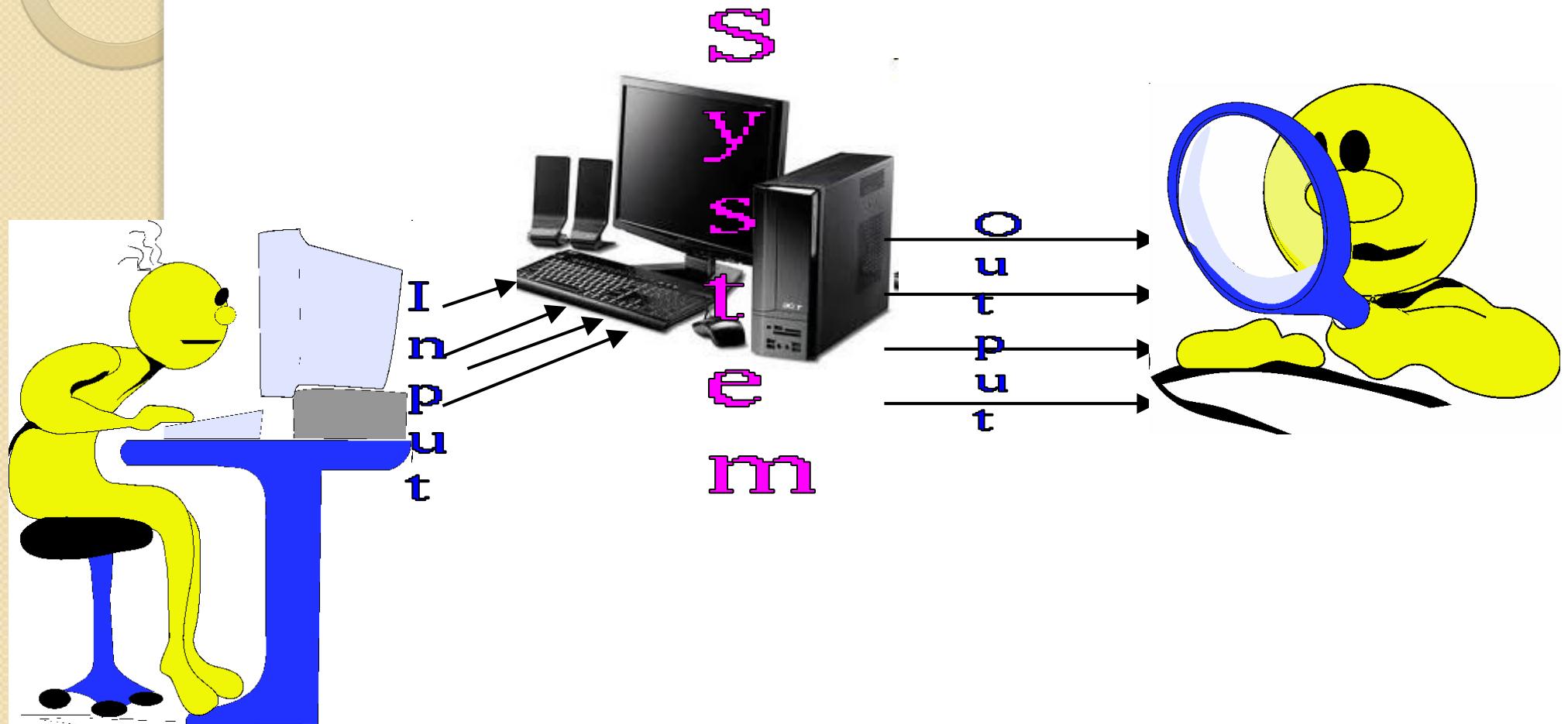


- Ariane 5 rocket self-destructed 37 seconds after launch
- Reason: A control software bug that went undetected
 - Conversion from 64-bit floating point to 16-bit signed integer value had caused an exception
 - The floating point number was larger than 32767
 - Efficiency considerations had led to the disabling of the exception handler.
- Total Cost: over \$1 billion

How Do You Test a Program?

- Input test data to the program.
- Observe the output:
 - Check if the program behaved as expected.

How Do You Test a Program?



How Do You Test a Program?

- If the program does not behave as expected:
 - Note the conditions under which it failed.
 - Later debug and correct.

What's So Hard About Testing ?

- Consider `int proc1(int x, int y)`
- Assuming a 64 bit computer
 - Input space = 2^{128}
- Assuming it takes 10secs to key-in an integer pair
 - It would take about a billion years to enter all possible values!
 - Automatic testing has its own problems!

Testing Facts

- **Consumes largest effort among all phases**
 - Largest manpower among all other development roles
 - Implies more job opportunities
- **About 50% development effort**
 - But 10% of development time?
 - How?

Testing Facts

- Testing is getting more complex and sophisticated every year.
 - Larger and more complex programs
 - Newer programming paradigms

Overview of Testing Activities

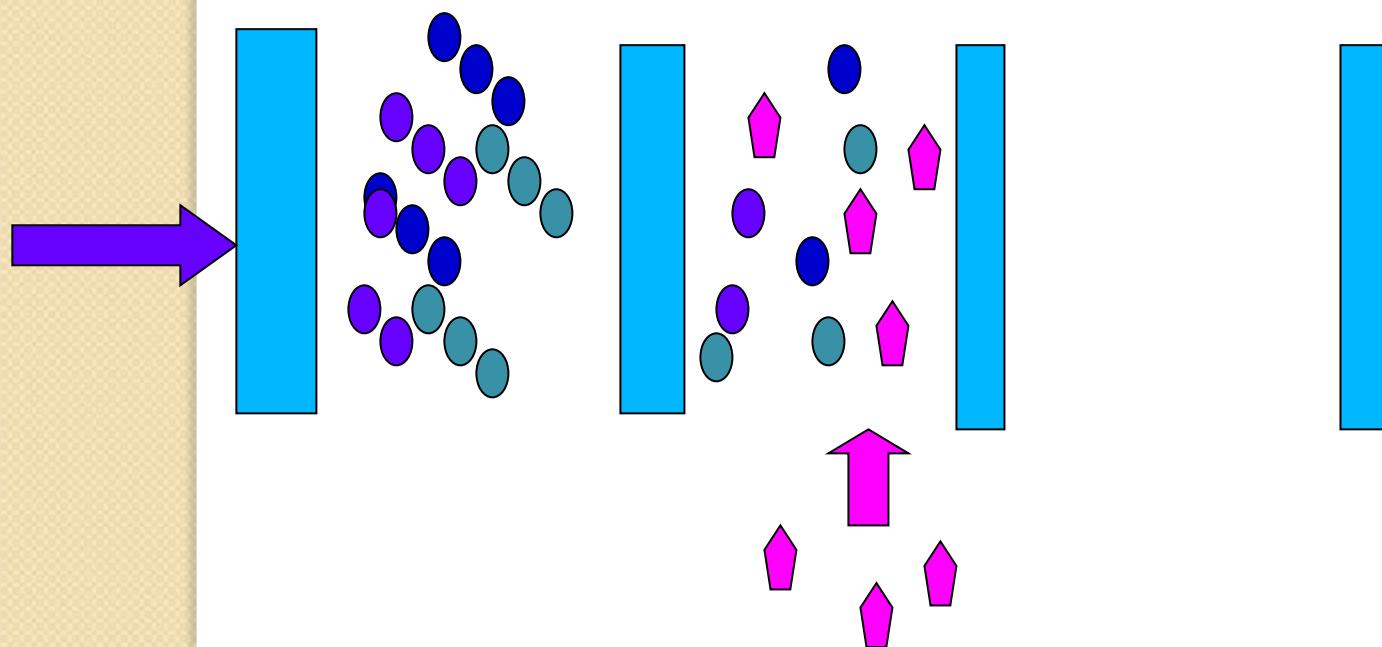
- Test Suite Design
- Run test cases and observe results to detect failures.
- Debug to locate errors
- Correct errors.

Error, Faults, and Failures

- A failure is a manifestation of an error (also defect or bug).
 - Mere presence of an error may not lead to a failure.

Pesticide Effect

- Errors that escape a fault detection technique:
 - Can not be detected by further applications of that technique.



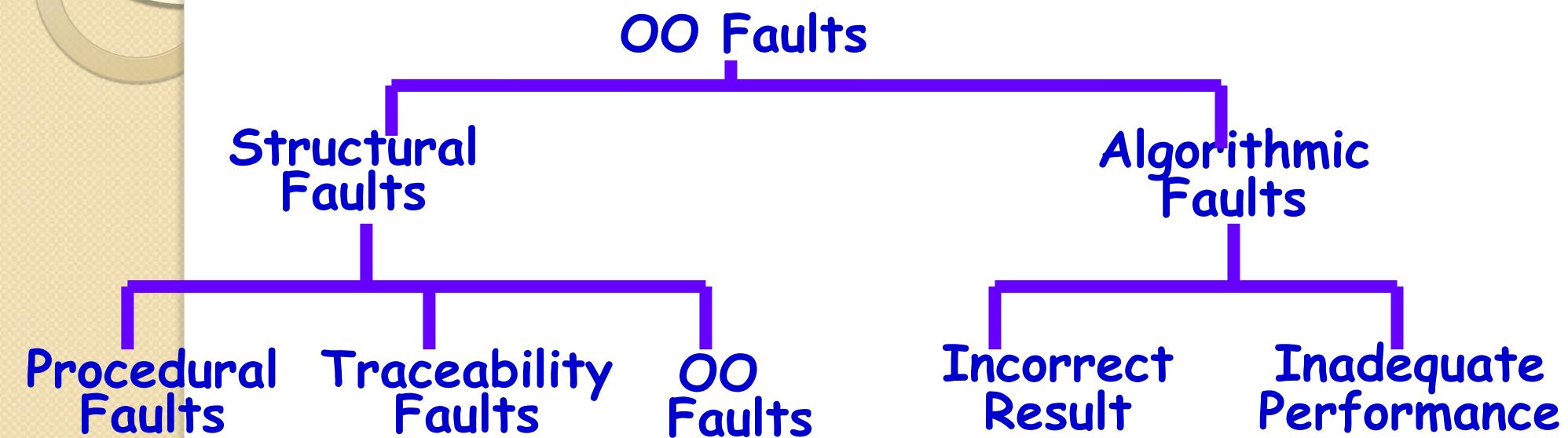
Pesticide Effect

- Assume we use 4 fault detection techniques and 1000 bugs:
 - Each detects only 70% bugs
 - How many bugs would remain
 - $1000 * (0.3)^4 = 81$ bugs

Fault Model

- Types of faults possible in a program.
- Some types can be ruled out
 - Concurrency related-problems in a sequential program

Fault Model of an OO Program



Hardware Fault-Model

- Simple:
 - Stuck-at 0
 - Stuck-at 1
 - Open circuit
 - Short circuit
- Simple ways to test the presence of each
- Hardware testing is fault-based testing



Software Testing

- Each test case typically tries to establish correct working of some functionality
 - Executes (covers) some program elements
 - For restricted types of faults, fault-based testing exists.

Test Cases and Test Suites

- Test a software using a set of carefully designed test cases:
 - The set of all test cases is called the test suite

Test Cases and Test Suites

- A **test case** is a triplet [I,S,O]
 - I is the data to be input to the system,
 - S is the state of the system at which the data will be input,
 - O is the expected output of the system.

Aim of Testing

- The aim of testing is to identify all defects in a software product.
- However, in practice even after thorough testing:
 - one cannot guarantee that the software is error-free.

Aim of Testing

- The input data domain of most software products is very large:
 - it is not practical to test the software exhaustively with each input data value.

Aim of Testing

- Testing does however expose many errors:
 - testing provides a practical way of reducing defects in a system
 - increases the users' confidence in a developed system.

Aim of Testing

- Testing is an important development phase:
 - requires the maximum effort among all development phases.
- In a typical development organization:
 - maximum number of software engineers can be found to be engaged in testing activities.

Aim of Testing

- Many engineers have the wrong impression:
 - testing is a secondary activity
 - it is intellectually not as stimulating as the other development activities, etc.

Aim of Testing

- Testing a software product is in fact:
 - as much challenging as initial development activities such as specification, design, and coding.
- Also, testing involves a lot of creative thinking.

Levels of Testing

- Software products are tested at three levels:
 - Unit testing
 - Integration testing
 - System testing

Unit testing

- During unit testing, modules are tested in isolation:
 - If all modules were to be tested together:
 - it may not be easy to determine which module has the error.

Unit testing

- Unit testing reduces debugging effort several folds.
 - Programmers carry out unit testing immediately after they complete the coding of a module.



Integration testing

- After different modules of a system have been coded and unit tested:
 - modules are integrated in steps according to an integration plan
 - partially integrated system is tested at each integration step.

System Testing

- System testing involves:
 - validating a fully developed system against its requirements.

Verification versus Validation

- Verification is the process of determining:
 - Whether output of one phase of development conforms to its previous phase.
- Validation is the process of determining:
 - Whether a fully developed system conforms to its SRS document.

Verification versus Validation

- Verification is concerned with phase containment of errors,
 - Whereas the aim of validation is that the final product be error free.

Design of Test Cases

- Exhaustive testing of any non-trivial system is impractical:
 - Input data domain is extremely large.
- Design an **optimal test suite**:
 - Of reasonable size and
 - Uncovers as many errors as possible.

Design of Test Cases

- If test cases are selected randomly:
 - Many test cases would not contribute to the significance of the test suite,
 - Would not detect errors not already being detected by other test cases in the suite.
- Number of test cases in a randomly selected test suite:
 - Not an indication of effectiveness of testing.

Design of Test Cases

- Testing a system using a large number of randomly selected test cases:
 - Does not mean that many errors in the system will be uncovered.
- Consider following example:
 - Find the maximum of two integers x and y .

Design of Test Cases

- The code has a simple programming error:

```
If (x>y) max = x;  
else max = x;
```
- Test suite $\{(x=3,y=2);(x=2,y=3)\}$ can detect the error,
- A larger test suite $\{(x=3,y=2);(x=4,y=3);(x=5,y=1)\}$ does not detect the error.

Design of Test Cases

- Systematic approaches are required to design an **optimal test suite**:
 - Each test case in the suite should detect different errors.

Design of Test Cases

- There are essentially three main approaches to design test cases:
 - **Black-box approach**
 - **White-box (or glass-box) approach**
 - **Grey-box (or model based) approach**

Black-Box Testing

- Test cases are designed using only **functional specification** of the software:
 - Without any knowledge of the internal structure of the software.
- For this reason, black-box testing is also known as **functional testing**.

Black-box Testing Techniques

- There are many approaches to design black box test cases:
 - Equivalence class partitioning
 - Boundary value analysis
 - State table based testing
 - Decision table based testing
 - Cause-effect graph based testing
 - Orthogonal array testing
 - Positive-negative testing

White-box Testing

- Designing white-box test cases:
 - Requires knowledge about the internal structure of software.
 - **White-box testing is also called structural testing.**

White-Box Testing Techniques

- There exist several popular white-box testing methodologies:
 - Statement coverage
 - Branch coverage
 - Path coverage
 - Condition coverage
 - MC/DC coverage
 - Mutation testing
 - Data flow-based testing

Coverage-Based Testing Versus Fault-Based Testing

- Idea behind coverage-based testing:
 - Design test cases so that certain program elements are executed (or covered).
 - Example: statement coverage, path coverage, etc.
- Idea behind fault-based testing:
 - Design test cases that focus on discovering certain types of faults.
 - Example: Mutation testing.

Why Both BB and WB Testing?

Black-box

- Impossible to write a test case for every possible set of inputs and outputs
- Some code parts may not be reachable
- Does not tell if extra functionality has been implemented.

White-box

- Does not address the question of whether or not a program matches the specification
- Does not tell you if all of the functionality has been implemented
- Does not discover missing program logic

Grey Box / Model Based Testing

- In grey box testing, test cases are designed from design documents / models, such as UML diagrams.
- Grey-box testing is also called model based testing.
- Mainly used for testing of O-O systems.



Summary

- Discussed importance of testing and the basic concepts of testing.
- Presented the levels of testing.
 - Unit testing
 - Integration testing
 - System testing
- Discussed the fundamentals of black box testing, white box testing and grey box testing.

References

1. R. Mall, Fundamentals of Software Engineering, (Chapter - 10), Fifth Edition, PHI Learning Pvt. Ltd., 2018.



Thank You

Agile Models

What is Agile Software Development?

- Agile: Easily moved, light, nimble, active software processes
- How agility achieved?
 - Fitting the process to the project
 - Avoidance of things that waste time

Agile Model

- To overcome the shortcomings of the waterfall model of development.
 - Proposed in mid-1990s
- The agile model was primarily designed:
 - To help projects to adapt to change requests
- In the agile model:
 - The requirements are decomposed into many small incremental parts that can be developed over one to four weeks each.

Ideology: Agile Manifesto

- Individuals and interactions *over*
 - process and tools <http://www.agilemanifesto.org>
- Working Software *over*
 - comprehensive documentation
- Customer collaboration *over*
 - contract negotiation
- Responding to change *over*
 - following a plan

Agile Methodologies

- XP
- Scrum
- Unified process
- Crystal
- DSDM
- Lean

Agile Model: Principal Techniques

- **User stories:**
 - Simpler than use cases.
- **Metaphors:**
 - Based on user stories, developers propose a common vision of what is required.
- **Spike:**
 - Simple program to explore potential solutions.
- **Refactor:**
 - Restructure code without affecting behavior, improve efficiency, structure, etc.

- At a time, only one increment is planned, developed, deployed at the customer site.
 - No long-term plans are made.
 - An iteration may not add significant functionality,
- Agile Model: Nitty Gritty**
- But still a new release is invariably made at the end of each iteration
 - Delivered to the customer for regular use.

Methodology

- Face-to-face communication favoured over written documents.
- To facilitate face-to-face communication,
 - Development team to share a single office space.
 - Team size is deliberately kept small (5-9 people)
 - This makes the agile model most suited to the development of small projects.

Effectiveness of Communication Modes

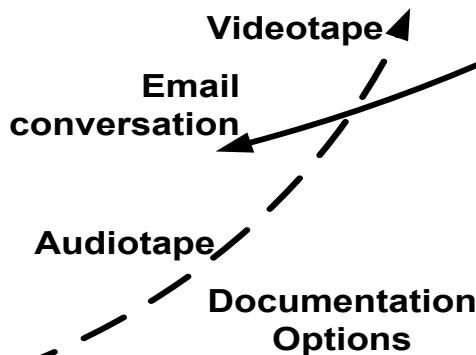
Communication Effectiveness

Cold

Hot

Richness of Communication Channel

9



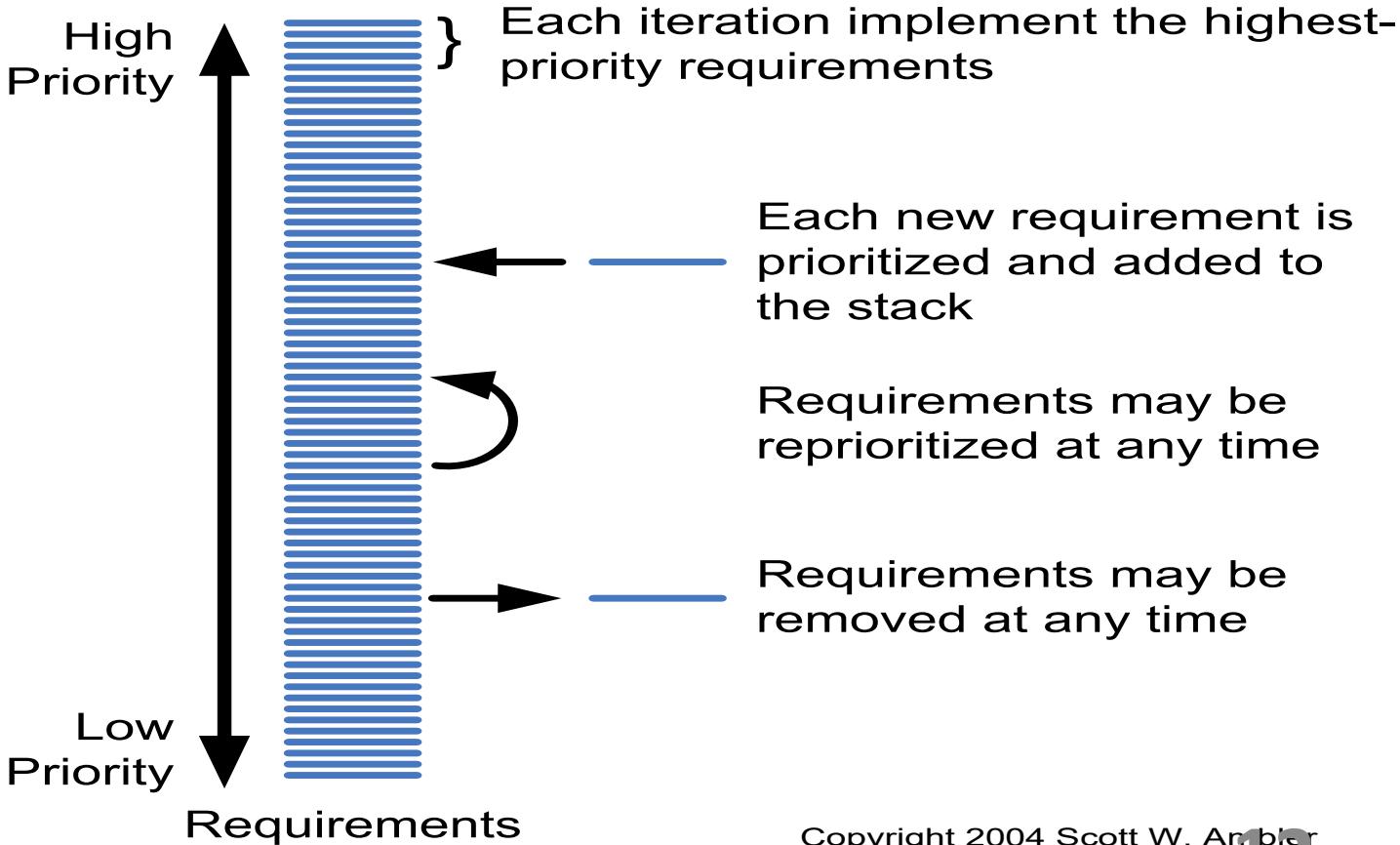
Agile Model: Principles

- The primary measure of progress:
 - Incremental release of working software
- Important principles behind agile model:
 - Frequent delivery of versions --- once every few weeks.
 - Requirements change requests are easily accommodated.
 - Close cooperation between customers and developers.
 - Face-to-face communication among team members.

Agile Documentation

- Travel light:
 - You need far less documentation than you think.
- Agile documents:
 - Are concise
 - Describe information that is less likely to change
 - Describe “good things to know”
 - Are sufficiently accurate, consistent, and detailed
- Valid reasons to document:
 - Project stakeholders require it
 - To define a contract model
 - To support communication with an external group
 - To think something through

Agile Software Requirements Management



Adoption Detractors

- Sketchy definitions, make it possible to have
 - Inconsistent and diverse definitions
- High quality people skills required
- Short iterations inhibit long-term perspective
- Higher risks due to feature creep:
 - Harder to manage feature creep and customer expectations

Agile Model Shortcomings

- Derives agility through developing tacit knowledge within the team, rather than any formal document:
 - Can be misinterpreted...
 - External review difficult to get...
 - When project is complete, and team disperses, maintenance becomes difficult...

- Steps of **Agile Model versus Waterfall Model**

Waterfall model are a planned sequence:

- Requirements-capture, analysis, design, coding, and testing .

- Progress is measured in terms of delivered artefacts:

- Requirement specifications, design documents, test plans, code reviews, etc.

- In contrast agile model sequences:

- Delivery of working versions of a product in several increments.

Agile Model versus Iterative Waterfall Model

- As regards to similarity:
 - We can say that Agile teams use the waterfall model on a small scale.

Agile versus RAD Model

- Agile model does not recommend developing prototypes:
 - Systematic development of each incremental feature is emphasized.
- In contrast:
 - RAD is based on designing quick-and-dirty prototypes, which are then refined into production quality code.

Agile versus exploratory programming

- Similarity:
 - Frequent re-evaluation of plans,
 - Emphasis on face-to-face communication,
 - Relatively sparse use of documents.
- Agile teams, however, do follow defined and disciplined processes and carry out rigorous designs:
 - This is in contrast to chaotic coding in exploratory programming.

Extreme Programming (XP)

Extreme Programming Model

- Extreme programming (XP) was proposed by Kent Beck in 1999.
- The methodology got its name from the fact that:
 - Recommends taking the best practices to extreme levels.
 - If something is good, why not do it all the time.

Taking Good Practices to Extreme

- If code review is good:
 - Always review --- **pair programming**
- If testing is good:
 - Continually write and execute test cases --- **test-driven development**
- If incremental development is good:
 - Come up with new increments every few days
- If simplicity is good:
 - Create the simplest design that will support only the currently required functionality.

Taking to Extreme

- **If design is good,**
 - everybody will design daily (refactoring)
- **If architecture is important,**
 - everybody will work at defining and refining the architecture (metaphor)
- **If integration testing is important,**
 - build and integrate test several times a day (continuous integration)

- **Communication:**
 - Enhance communication among team members and with the customers.
- **Simplicity:**
 - Build something simple that will work today rather than something that takes time and yet never used
 - May not pay attention for tomorrow
- **Feedback:**
 - System staying out of users is trouble waiting to happen
- **Courage:**
 - Don't hesitate to discard code

Best Practices

- **Coding:**
 - without code it is not possible to have a working system.
 - Utmost attention needs to be placed on coding.
- **Testing:**
 - Testing is the primary means for developing a fault-free product.
- **Listening:**
 - Careful listening to the customers is essential to develop a good quality product.

Best Practices

- **Designing:**
 - Without proper design, a system implementation becomes too complex
 - The dependencies within the system become too numerous to comprehend.
- **Feedback:**
 - Feedback is important in learning customer requirements.

Extreme Programming Activities

- **XP Planning**
 - Begins with the creation of “user stories”
 - Agile team assesses each story and assigns a cost
 - Stories are grouped to form a deliverable increment
 - A commitment is made on delivery date
- **XP Design**
 - Follows the KIS principle
 - Encourage the use of CRC cards
 - For difficult design problems, suggests the creation of “spike solutions”—a design prototype
 - Encourages “refactoring”—an iterative refinement of the internal program design

Extreme Programming Activities

- **XP Coding**

- Recommends the construction of unit test cases *before* coding commences (test-driven development)
- Encourages “pair programming”

- **XP Testing**

- All unit tests are executed daily
- “Acceptance tests” are defined by the customer and executed to assess customer visible functionalities

1. **Planning** – determine scope of the next release by combining business priorities and technical estimates

Full List of XP Practices

2. **Small releases** – put a simple system into production, then release new versions in very short cycles
3. **Metaphor** – all development is guided by a simple shared story of how the whole system works
4. **Simple design** – system is to be designed as simple as possible
5. **Testing** – programmers continuously write and execute unit tests

Full List of XP Practices

7. **Refactoring** – programmers continuously restructure the system without changing its behavior to remove duplication and simplify
8. **Pair-programming** -- all production code is written with two programmers at one machine
9. **Collective ownership** – anyone can change any code anywhere in the system at any time.
10. **Continuous integration** – integrate and build the system many times a day – every time a task is completed.

Full List of XP Practices

11. **40-hour week** – work no more than 40 hours a week as a rule
12. **On-site customer** – a user is a part of the team and available full-time to answer questions
13. **Coding standards** – programmers write all code in accordance with rules emphasizing communication through the code

Emphasizes Test-Driven Development (TDD)

- Based on user story develop test cases
- Implement a quick and dirty feature every couple of days:
 - Get customer feedback
 - Alter if necessary
 - Refactor
- Take up next feature

Project Characteristics that Suggest Suitability of Extreme Programming

- Projects involving new technology or research projects.
 - In this case, the requirements change rapidly and unforeseen technical problems need to be resolved.
- Small projects:
 - These are easily developed using extreme programming.

Life Cycle Models:

Scrum

Practice Questions

- What are the stages of iterative waterfall model?
- What are the disadvantages of the iterative waterfall model?
- Why has agile model become so popular?
- What difficulties might be faced if no life cycle model is followed for a certain large project?

Suggest Suitable Life Cycle Model

- A software for an academic institution to automate its:
 - Course registration and grading
 - Fee collection
 - Staff salary
 - Purchase and store inventory
- The software would be developed by tailoring a similar software that was developed for another educational institution:
 - 70% reuse
 - 10% new code and 20% modification

Practice Questions

- Which types of risks can be better handled using the spiral model compared to the prototyping model?
- Which type of process model is suitable for the following projects:
 - A customization software
 - A payroll software for contract employees that would be add on to an existing payroll software

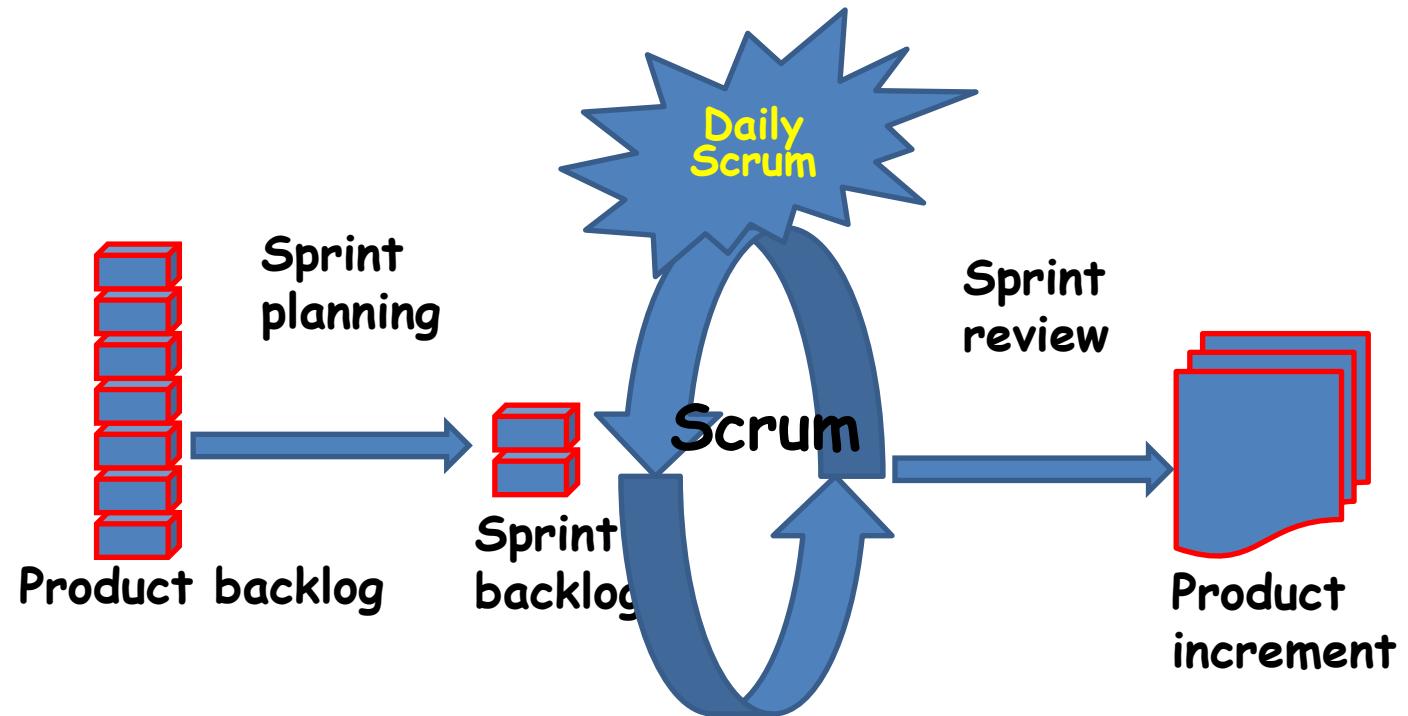
Practice Questions

- Which lifecycle model would you select for the following project which has been awarded to us by a mobile phone vendor:
 - A new mobile operating system by upgrading the existing operating system
 - Needs to work well efficiently with 4G systems
 - Power usage minimization
 - Directly upload backup data on a cloud infrastructure maintained by the mobile phone vendor

Scrum

Scrum: Characteristics

- Self-organizing teams
- Product progresses in a series of month-long **sprints**
- Requirements are captured as items in a list of **product backlog**
- One of the agile processes



Sprint

- Scrum projects progress in a series of “sprints”
 - Analogous to XP iterations or time boxes
 - Target duration is one month
- **Software increment is designed, coded, and tested during the sprint**
- **No changes entertained during a sprint**

Scrum Framework

- **Roles :** Product Owner, Scrum Master, Team
- **Ceremonies :** Sprint Planning, Sprint Review, Sprint Retrospective, and Daily Scrum Meeting
- **Artifacts :** Product Backlog, Sprint Backlog, and Burndown Chart

Key Roles and Responsibilities in a Scrum Team

- **Product Owner**
 - Represents customers' views and interests.
- **Development Team**
 - Team of five-nine people with cross-functional skill sets.
- **Scrum Master (aka Project Manager)**
 - Facilitates scrum process and resolves impediments at the team and organization level by acting as a buffer between the team and outside interference.

Product Owner

- Defines the features of the product
- Decides on release date and content
- Prioritizes features according to usefulness
- Adjusts features and priority every iteration, as needed
- Accepts or reject work results.

The Scrum Master

- Represents management in the project
- Removes impediments
- Ensures that the team is fully functional and productive
- Enables close cooperation across all roles and functions
- Shields the team from external interferences

Scrum Team

- Typically 5-10 people
- Cross-functional
 - QA, Programmers, UI Designers, etc.
- Teams are self-organizing
- Membership can change only between sprints

Sprint

- Fundamental process flow of Scrum
- It is usually a month-long iteration:
 - during this time an incremental product functionality completed
- NO outside influence allowed to interfere with the Scrum team during the Sprint
- Each day begins with the Daily Scrum Meeting

Ceremonies

- Sprint Planning Meeting
- Daily Scrum
- Sprint Review Meeting

Sprint Planning

- Goal is to produce Sprint Backlog
- Product owner works with the Team to negotiate what Backlog Items
- Scrum Master ensures Team agrees to realistic goals

Daily Scrum

- Daily
- 15-minutes
- Stand-up meeting
- Not for problem solving
- Three questions:
 1. What did you do yesterday
 2. What will you do today?
 3. What obstacles are in your way?

Daily Scrum

- Is NOT a problem solving session
- Is NOT a way to collect information about WHO is behind the schedule
- Is a meeting in which team members review what is done and make informal commitments to each other and to the Scrum Master
- Is a good way for a Scrum Master to track the progress of the Team

- Team presents what it accomplished during the sprint
- Typically takes the form of a demo of new features
- Informal
 - 2-hour prep time rule
- Participants
 - Customers
 - Management
 - Product Owner
 - Other team members

Sprint Review Meeting

Product Backlog

- A list of all desired work on the project
 - Usually a combination of
 - story-based work (“let user search and replace”)
 - task-based work (“improve exception handling”)
- List is prioritized by the Product Owner
 - Typically a Product Manager, Marketing, Internal Customer, etc.

Product Backlog

- Requirements for a system, expressed as a prioritized list of Backlog Items
 - Managed and owned by Product Owner
 - Spreadsheet (typically)

Sample Product Backlog

	Item #	Description	Est	By
Very High				
	1	Finish database versioning	16	KH
	2	Get rid of unneeded shared Java in database	8	KH
	-	Add licensing	-	-
	3	Concurrent user licensing	16	TG
	4	Demo / Eval licensing	16	TG
		Analysis Manager		
	5	File formats we support are out of date	160	TG
	6	Round-trip Analyses	250	MC
High				
	-	Enforce unique names	-	-
	7	In main application	24	KH
	8	In import	24	AM
	-	Admin Program	-	-
	9	Delete users	4	JM
	-	Analysis Manager	-	-
	10	When items are removed from an analysis, they should show up again in the pick list in lower 1/2 of the analysis tab	8	TG
	-	Query	-	-
	11	Support for wildcards when searching	16	T&A
	12	Sorting of number attributes to handle negative numbers	16	T&A
	13	Horizontal scrolling	12	T&A
	-	Population Genetics	-	-
	14	Frequency Manager	400	T&M
	15	Query Tool	400	T&M
	16	Additional Editors (which ones)	240	T&M
	17	Study Variable Manager	240	T&M
	18	Haplotypes	320	T&M
	19	Add icons for v1.1 or 2.0	-	-
	-	Pedigree Manager	-	-
	20	Validate Derived kindred	4	KH
Medium				
	-	Explorer	-	-
		Launch tab synchronization (only show queries/analyses for logged in users)	8	T&A
	21	Delete settings (?)	4	T&A

Sprint Backlog

- A subset of Product Backlog Items,
which define the work for a Sprint
 - **Created by Team members**
 - **Each Item has it's own status**
 - **Updated daily**

Sprint Backlog during the Sprint

- Changes occur:
 - Team adds new tasks whenever they need to in order to meet the Sprint Goal
 - Team can remove unnecessary tasks
 - But: Sprint Backlog can only be updated by the team
- Estimates are updated whenever there's new information

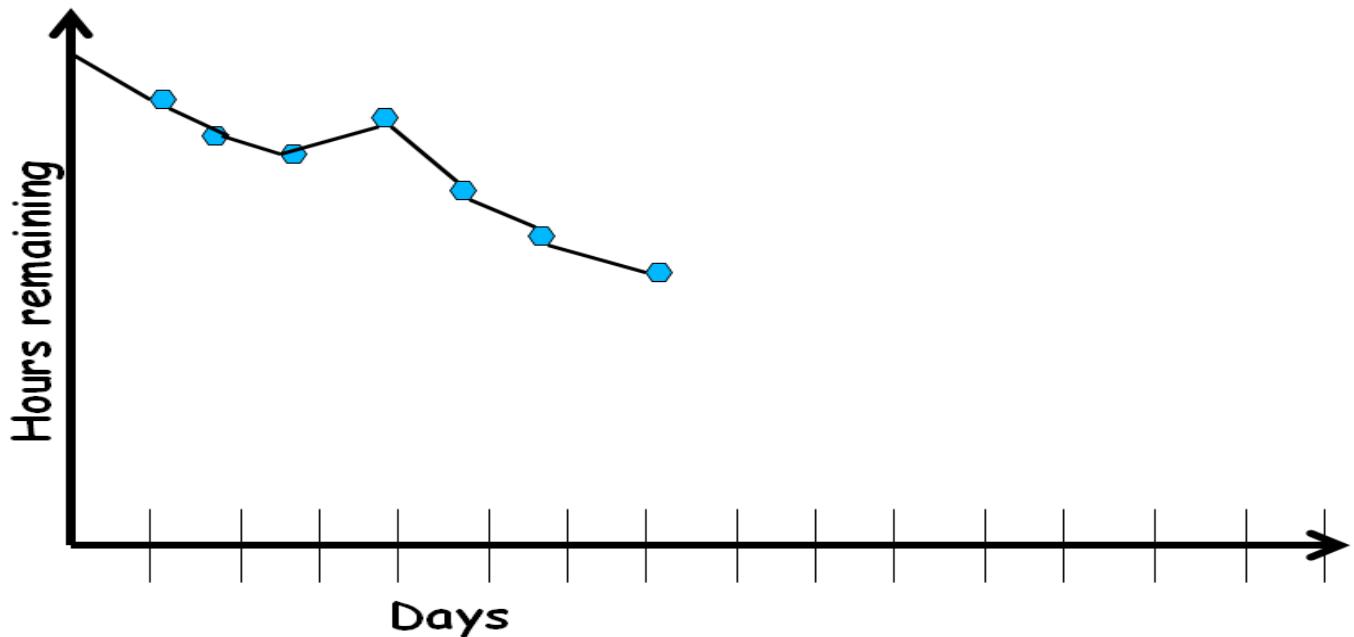
Burn down Charts

- Are used to represent “work done”.
- Are remarkably simple but effective Information disseminators
- 3 Types:
 - Sprint Burn down Chart (progress of the Sprint)
 - Release Burn down Chart (progress of release)
 - Product Burn down chart (progress of the Product)

Sprint Burn down Chart

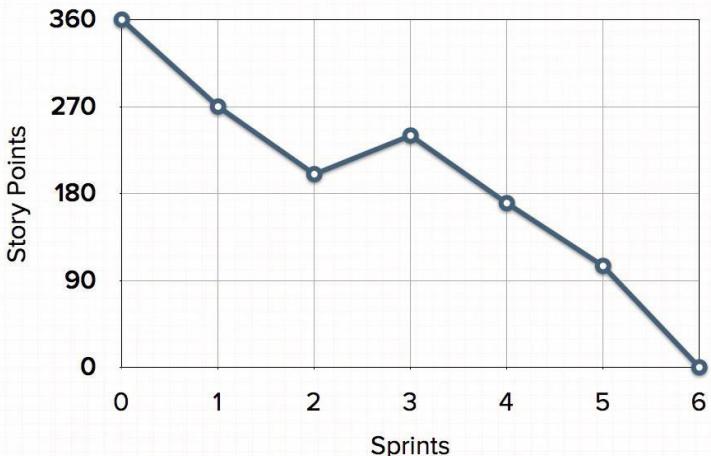
- Depicts the total Sprint Backlog hours remaining per day
- Shows the estimated amount of time to complete
- Ideally should burn down to zero to the end of the Sprint
- Actually is not a straight line

Sprint Burndown Chart



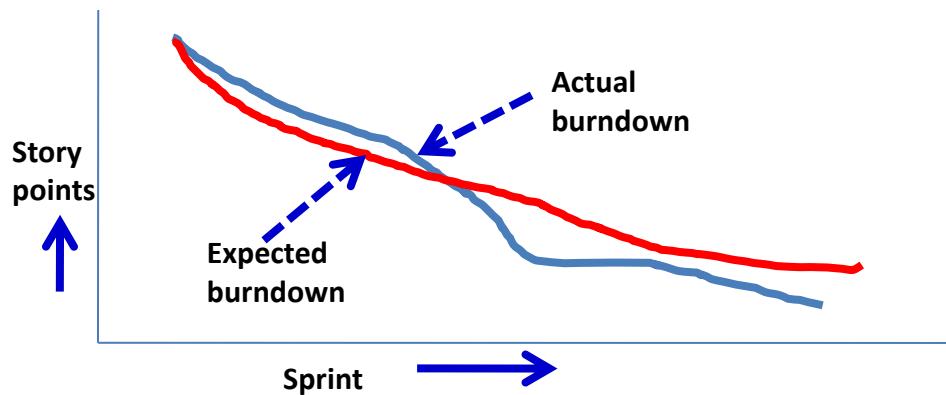
Release Burndown Chart

- Will the release be done on right time?
- How many more sprints?
- X-axis: sprints
- Y-axis: amount of story points remaining



Product Burndown Chart

- Is a “big picture” view of project’s progress (all the releases)



Scalability of Scrum

- A typical Scrum team is 6-10 people
- Jeff Sutherland - up to over 800 people
- "Scrum of Scrums" or "Meta-Scrum"

Thank You!!



Testing Agile Based Software

Prof. Durga Prasad Mohapatra

Dept. of CSE, NIT Rourkela

India



Agile software Development: Motivation

- There are many SDLC models available, such as waterfall model, prototyping model, iterative model, incremental model, spiral model and many more.
- These traditional models follow the *static* working strategy in which everything is fixed, for example cost of project, requirements or scope, schedule etc. Further, the duration is in years.
- Besides, there are several other drawbacks of each model.



Agile software Development

- In order to overcome these drawbacks, several major software vendors have been adopting a form of “intelligent” development in one or more phases of their software development processes.
- **Agile** for example, is a well-known example of a lifecycle used to build intelligent and analytical systems.
- Agile development does not promote best practices; rather, it is about adaptive planning and evolutionary development.

What is Agile Software Development?

- Agile: Easily moved, light, nimble, and active software processes.
- How agility achieved?
 - Fitting the process to the project
 - Avoiding things that waste time

Agile software Development

- ASD is based on light weight methodologies (less documentation and less planning) having dynamic nature, which is its major strength.
- This model provides an environment to accommodate frequent changes as per market standards and customer needs.
- ASD is an iterative and incremental development method and it is customer centred.

Agile software Development

- The agile process consists of multiple sprints (iterations or runs or development cycles);
 - in each sprint a specific software feature is developed, tested, refined and documented.
- However, because agile development depends on the context of the project,
 - testing is performed differently in every sprint.

Agile Model

- To overcome the shortcomings of the waterfall model of development,
 - it was proposed in mid-1990s
- The agile model was primarily designed:
 - To help projects to adapt to change requests
- In the agile model:
 - The requirements are decomposed into many small incremental parts that can be developed over one to four weeks each.

History: The Agile Manifesto

- On February 11-13, 2001, at The Lodge at Snowbird ski resort in the Wasatch mountains of Utah, seventeen people met to talk, ski, relax, and try to find common ground—and of course, to eat.
- What emerged was the Agile ‘Software Development’ Manifesto.

History: The Agile Manifesto cont...

- Representatives from Extreme Programming, SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, Pragmatic Programming, and others sympathetic to the need for an alternative to documentation driven, heavyweight software development processes convened.

History: The Agile Manifesto cont...

- Now, a bigger gathering of organizational anarchists would be hard to find, so what emerged from this meeting was symbolic—a Manifesto for Agile Software Development—signed by all participants.

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it.

Through this work we have come to value:

Individuals and interactions **over** processes and tools

Working software **over** comprehensive documentation

Customer collaboration **over** contract negotiation

Responding to change **over** following a plan

That is, while there is value in the items on the right, we value the items on the left more.



Principles behind the Agile Manifesto

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Principles behind the Agile Manifesto cont...

- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals, give them the environment and support they need, and trust them to get the job done.

Principles behind the Agile Manifesto cont...

- The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.



Principles behind the Agile Manifesto cont...

- Continuous attention to technical excellence and good design enhances agility.
- Simplicity--the art of maximizing the amount of work not done--is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly

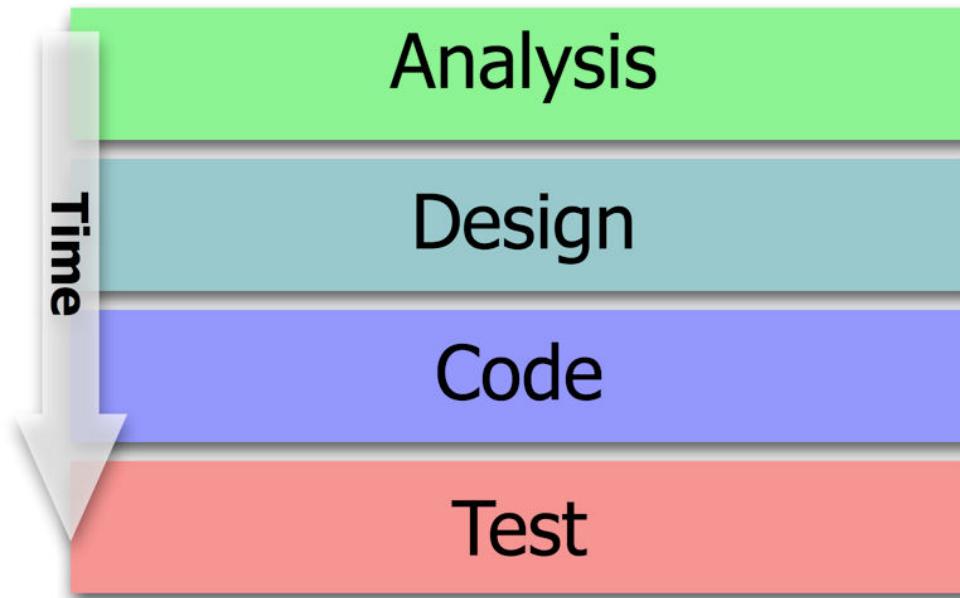
Ideology: Agile Manifesto

- Individuals and interactions over
 - process and tools <http://www.agilemanifesto.org>
- Working Software over
 - comprehensive documentation
- Customer collaboration over
 - contract negotiation
- Responding to change over
 - following a rigid plan

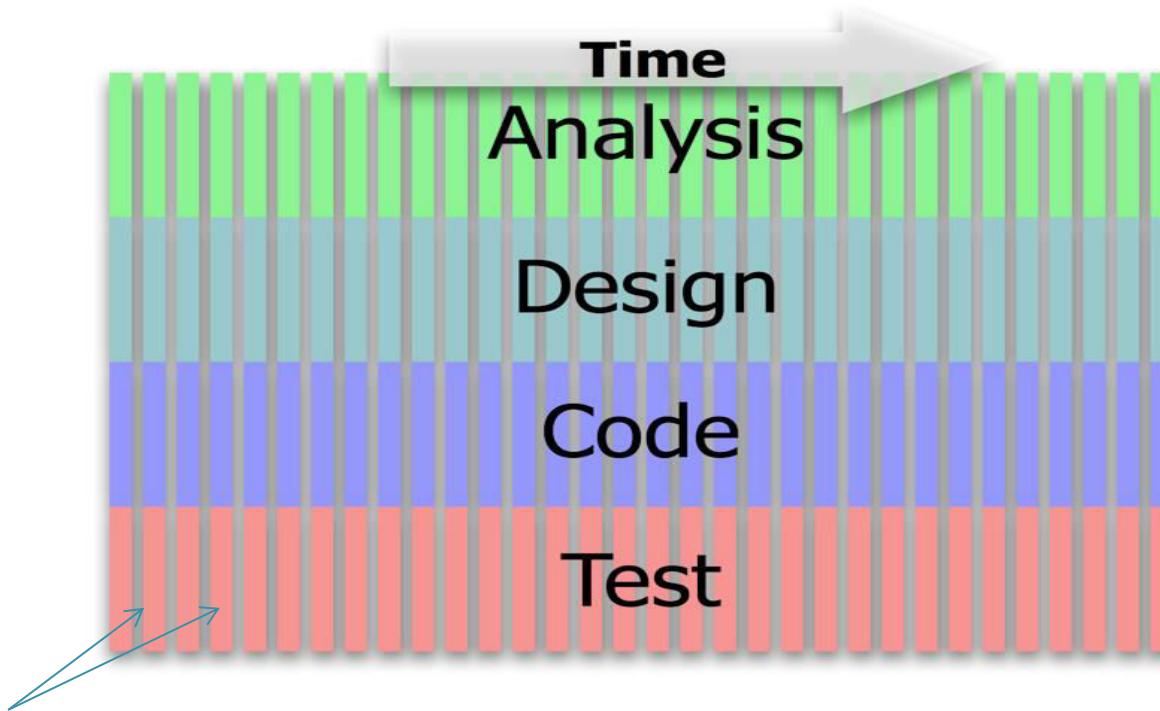
Agile Methodologies

- XP
- Scrum
- Unified process
- Crystal
- DSDM
- Lean

Traditional Software Development



Applying Lean Principles to Software Development ... (a better way of doing the same)



End-to-End small slices of work

Agile Model

- The Agile model puts emphasis on the fact that whole Agile team should be a tightly integrated unit and is composed of
 - Developers
 - Quality assurance members
 - Testers
 - Project owner
 - Customer



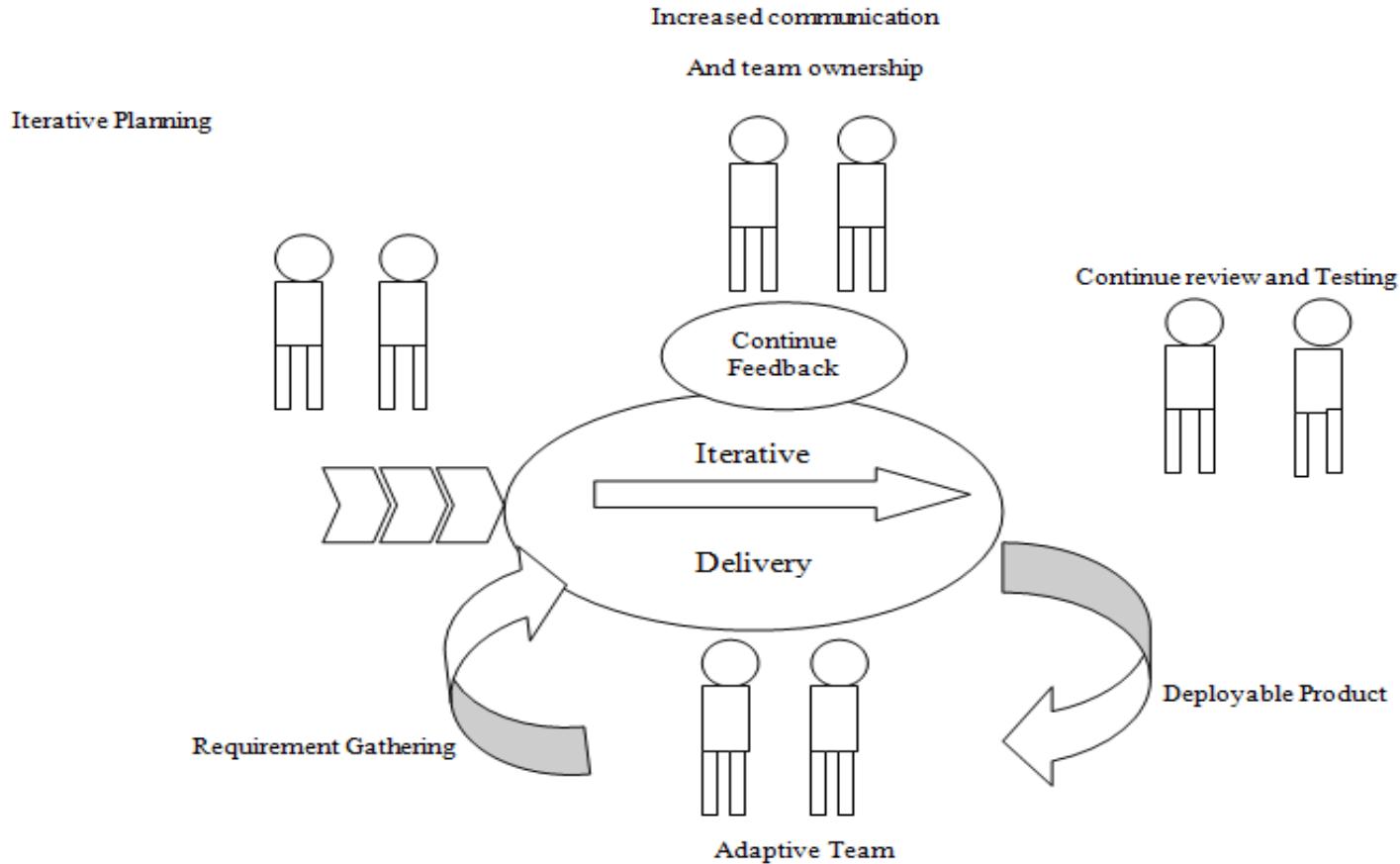
Agile Model

cont ...

- The key feature of ASD is to have effective communication between all team members. For valuable communication and information exchange, daily meetings are held in ASD.
- Another important feature of agile process is iterative delivery.
 - An iteration or delivery cycle in ASD ranges from 1 – 4 weeks.

Agile Model

cont ...



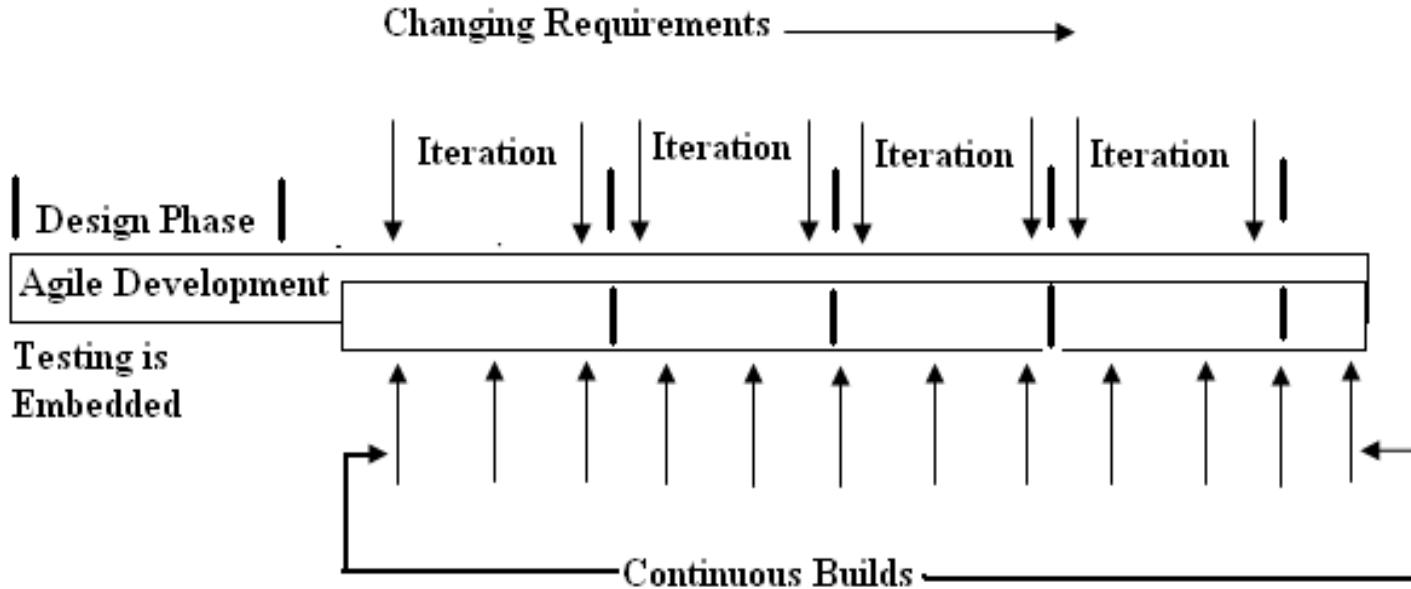
Agile Model: Principal Techniques

- **User stories:**
 - Simpler than use cases.
- **Metaphors:**
 - Based on user stories, developers propose a common vision of what is required.
- **Spike:**
 - Simple program to explore potential solutions.
- **Refactor:**
 - Restructure code without affecting behavior, improve efficiency, structure, etc.

Agile Model: Nitty Gritty

- At a time, only one increment is planned, developed, deployed at the customer site.
 - No long-term plans are made.
- An iteration may not add significant functionality,
 - But still a new release is invariably made at the end of each iteration
 - Delivered to the customer for regular use.

Agile Software Development Life Cycle



Stakeholders in Agile Life Cycle

- Agile SDLC includes specific activities performed by manager (M), developers (D), testers (T), marketing professional (MP) and customer (C).
- Table I lists the activities performed by different stakeholders
- At the time of production of the code or before producing the code, testing is applied by writing failed test cases, unlike the traditional approach of working.



Stakeholders in Agile Life Cycle

- Testing activity begins as soon as user stories (requirements) are finalized and prioritized, and
 - testers try to move business logic into lower levels in order to test with lower effort in the last stage.

Stakeholders in agile life cycle

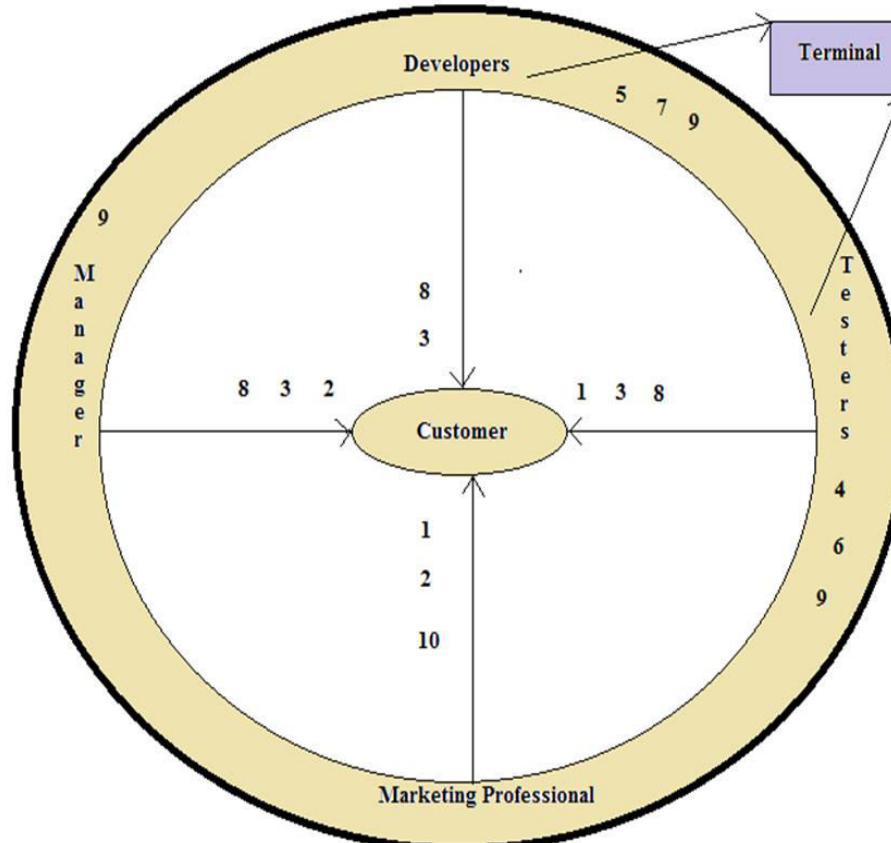


Fig 3: Stakeholders in agile life cycle

Table I :Actor Activity Chart - Agile Life Cycle

S. No.	Actor	Activity
1	C, MP, T	Requirements Gathering
2	M, MP, C	Effort estimation [28], cost, risk, capacity & resource Estimation
3	M, D, T, C	User stories Prioritization
4	T	Designing of Test Cases
5	D	Coding for the user story in the iteration
6	T	Feedback
7	D	Refactoring for the user story
8	C, M, D, T	Review meeting with Demonstration
9	D, T, M, MP	Lesson Learning phase or Retrospective session after the iteration
10	C, MP	Release

Agile software development life cycle

- In agile, a quality(Q) product is delivered by operational teams, and acceptance factor (A) is related to the rate at which customer accepts the delivered product.
- Effectiveness (E) of the team is related to two factors as shown in below equation

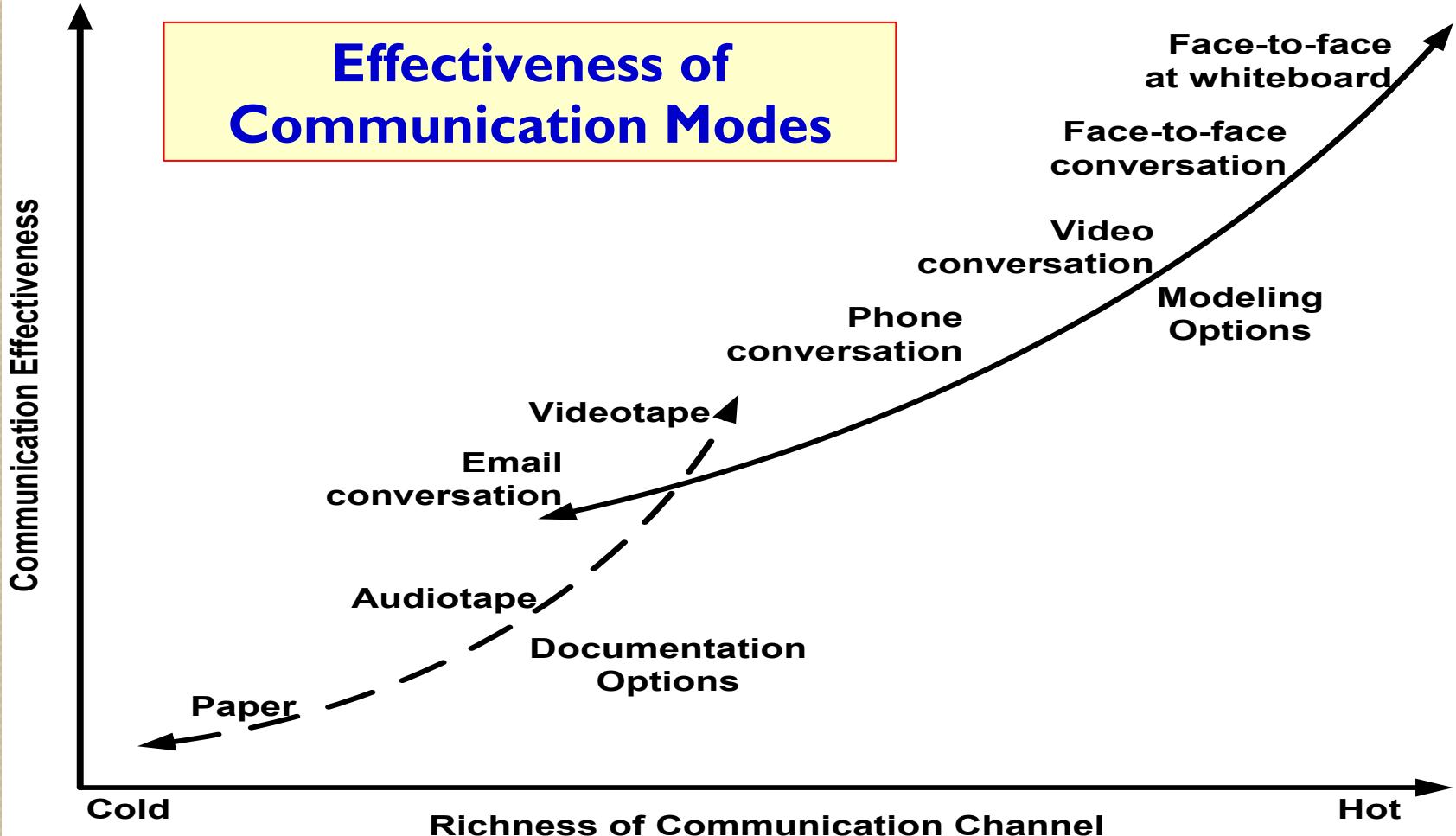
$$E = Q \times A$$

Agile software development life cycle cont...

- Out of these 2 factors, quality is more significant as acceptance is totally dependent on Q.
- Q is more when there is lesser number of backlogs and it is less when backlog items are more.
- Backlogs can be decreased when automation is the preferred approach over a manual way of testing.

Methodology

- Face-to-face communication is favoured over written documents.
- To facilitate face-to-face communication,
 - Development team should share a single office space.
 - Team size is deliberately kept small (5-9 people).
 - This makes the agile model most suited to the development of small projects.



Agile Model: Principles

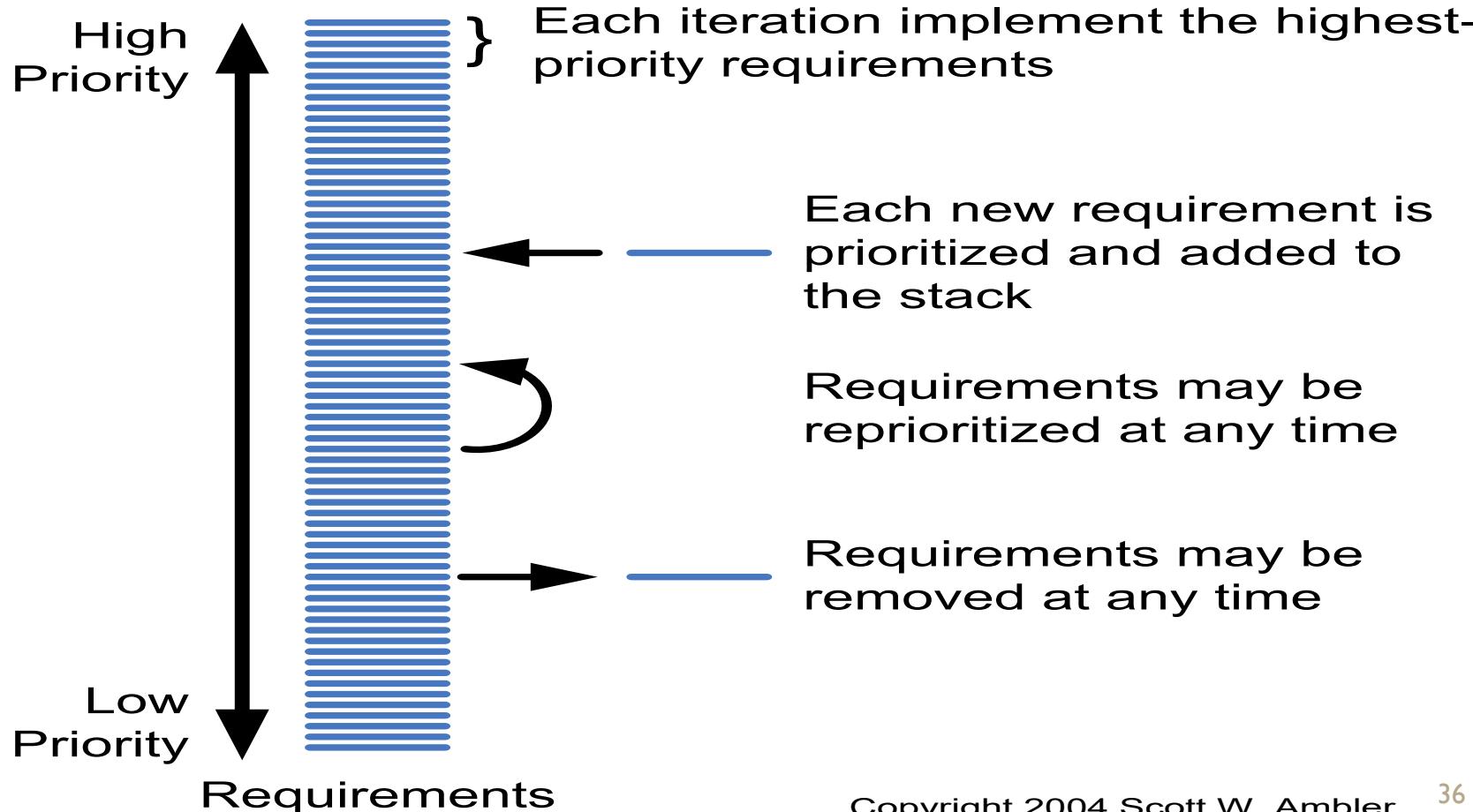
- The primary measure of progress:
 - Incremental release of working software
- Important principles behind agile model:
 - Frequent delivery of versions --- once every few weeks.
 - Requirements change requests are easily accommodated.
 - Close cooperation between customers and developers.
 - Face-to-face communication among team members.



Agile Documentation

- **Travel light:**
 - You need far less documentation than you think.
- **Agile documents:**
 - Are concise
 - Describe information that is less likely to change
 - Describe “good things to know”
 - Are sufficiently accurate, consistent, and detailed
- **Valid reasons to document:**
 - Project stakeholders require it
 - To define a contract model
 - To support communication with an external group
 - To think something through

Agile Software Requirements Management



Adoption Detractors

- Sketchy definitions, make it possible to have
 - Inconsistent and diverse definitions
- High quality people skills required
- Short iterations inhibit long-term perspective
- Higher risks due to feature creep:
 - Harder to manage feature creep and customer expectations
 - Difficult to quantify cost, time, quality.

Agile Model Shortcomings

- Derives agility through developing tacit knowledge within the team, rather than any formal document:
 - Can be misinterpreted...
 - External review difficult to get...
 - When project is complete, and team disperses, maintenance becomes difficult...

Agile Model vs Waterfall Model

- Steps of Waterfall model are a planned sequence:
 - Requirements-capture, analysis, design, coding, and testing .
- Progress is measured in terms of delivered artefacts:
 - Requirement specifications, design documents, test plans, code reviews, etc.
- In contrast agile model sequences:
 - Delivery of working versions of a product in several increments.

Agile Model vs Waterfall Model cont ...

- As regards to similarity:
 - We can say that Agile teams use the waterfall model on a small scale.

Summary

- Discussed the basics of agile software development.
- Highlighted some existing agile methodologies.
- Discussed the agile model in detail.
- Explained the Agile Software Development Life Cycle.
- Presented some of the shortcomings of agile model.
- Compared Agile Model vs Waterfall Model.

References

1. Rajib Mall, Fundamentals of Software Engineering, Fifth Edition, PHI, 2018.
2. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter – 16), Second Edition, Oxford University Press, 2018.



Thank You



Testing Agile Based Software cont ...

Prof. Durga Prasad Mohapatra

Dept. of CSE, NIT Rourkela

India



Scrum



Introduction

- Scrum is an Agile framework which concentrates on how team members should function to produce system flexibly in a constantly changing environment.
- Scrum is concerned with the product owner, project lead, and the team working together in an intensive and interdependent manner.



Introduction

cont ...

- Scrum process includes three phases:
 - Pre-game
 - Development
 - Post-game



Introduction

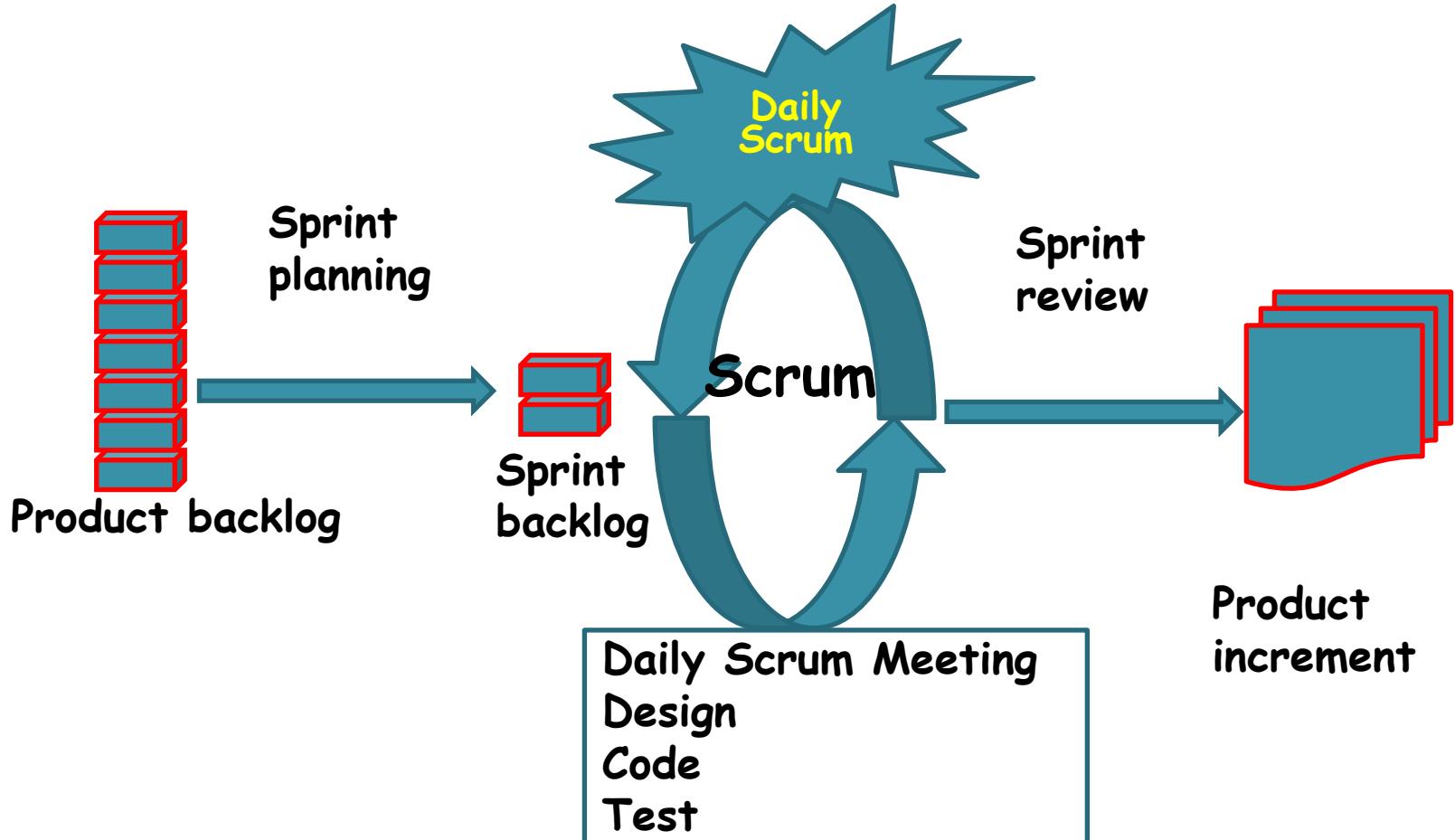
cont ...

- There are six identifiable roles in Scrum that have different tasks and purposes during the process and its practices:
 - Scrum master
 - Product owner
 - Scrum team
 - Customer
 - User
 - Management

Scrum: Characteristics

- Self-organizing teams
- Product progresses in a series of month-long **sprints** (iterations or runs or development cycles)
- Requirements are captured as items in a list, called **product backlog**
- One of the agile processes

Scrum Life Cycle





Sprint

- Scrum projects progress in a series of “sprints”
 - Analogous to XP iterations or time boxes
 - Target duration is one month
- Software increment is designed, coded, and tested during the sprint
- No changes are entertained during a sprint

Scrum Framework

- **Roles:** Product Owner, Scrum Master, Scrum Team, Customer, End User, Management
- **Ceremonies:** Sprint Planning, Sprint Review, Sprint Retrospective, and Daily Scrum Meeting
- **Artifacts:** Product Backlog, Sprint Backlog, and Burndown Chart

Key Roles and Responsibilities in a Scrum Team

- **Product Owner**
 - Represents customers' views and interests.
- **Development Team**
 - Team of 5 - 10 people with cross-functional skill sets.
- **Scrum Master (aka Project Manager)**
 - Facilitates scrum process and resolves impediments at the team and organization level by acting as a buffer between the team and outside interference.



Product Owner

- Defines the features of the product
- Decides on release date and content
- Prioritizes features according to usefulness
- Adjusts features and priority every iteration, as needed
- Accepts or reject work results.



The Scrum Master

- Represents management in the project
- Removes impediments
- Ensures that the team is fully functional and productive
- Enables close cooperation across all roles and functions
- Shields the team from external interferences



Scrum Team

- Typically 5-10 people
- Cross-functional
 - Quality Assurance Engineers, Programmers, UI Designers, etc.
- Teams are self-organizing
- Membership can change only between sprints

Sprint

- Fundamental process flow of Scrum
- It is usually a month-long iteration:
 - during this time an incremental product functionality is completed
- No outside influence is allowed to interfere with the Scrum team during the Sprint
- Each day begins with the Daily Scrum Meeting



Ceremonies

- Sprint Planning Meeting
- Daily Scrum
- Sprint Review Meeting



Sprint Planning

- Goal is to produce Sprint Backlog
- Product owner works with the Team to negotiate what Backlog Items
- Scrum Master ensures that the Team agrees to the realistic goals

Daily Scrum

- Daily
- 15-minutes
- Stand-up meeting
- Not for problem solving
- Three questions:
 - 1.What did you do yesterday?
 - 2.What will you do today?
 - 3.What obstacles are in your way?

Daily Scrum

- Is NOT a problem solving session
- Is NOT a way to collect information about WHO is behind the schedule
- Is a meeting in which team members review what is done and make informal commitments to each other and to the Scrum Master
- Is a good way for a Scrum Master to track the progress of the Team

- Team presents what it accomplished during the sprint
- Typically takes the form of a demo of new features
- Informal
 - 2-hour prep time rule
- Participants
 - Customers
 - Management
 - Product Owner
 - Other team members

Sprint Review Meeting

Product Backlog

- A list of all desired work on the project
 - Usually a combination of
 - story-based work (“let user search and replace”)
 - task-based work (“improve exception handling”)
- List is prioritized by the Product Owner
 - Typically a Product Manager, Marketing, Internal Customer, etc.



Product Backlog

- Requirements for a system, expressed as a prioritized list of Backlog Items
 - Managed and owned by Product Owner
 - Spreadsheet (typically)

Sample Product Backlog

	Item #	Description	Est	By
Very High				
	1	Finish database versioning	16	KH
	2	Get rid of unneeded shared Java in database	8	KH
	-	Add licensing	-	-
	3	Concurrent user licensing	16	TG
	4	Demo / Eval licensing	16	TG
		Analysis Manager		
	5	File formats we support are out of date	160	TG
	6	Round-trip Analyses	250	MC
High				
	-	Enforce unique names	-	-
	7	In main application	24	KH
	8	In import	24	AM
	-	Admin Program	-	-
	9	Delete users	4	JM
	-	Analysis Manager	-	-
	10	When items are removed from an analysis, they should show up again in the pick list in lower 1/2 of the analysis tab	8	TG
	-	Query	-	-
	11	Support for wildcards when searching	16	T&A
	12	Sorting of number attributes to handle negative numbers	16	T&A
	13	Horizontal scrolling	12	T&A
	-	Population Genetics	-	-
	14	Frequency Manager	400	T&M
	15	Query Tool	400	T&M
	16	Additional Editors (which ones)	240	T&M
	17	Study Variable Manager	240	T&M
	18	Haplotypes	320	T&M
	19	Add icons for v1.1 or 2.0	-	-
	-	Pedigree Manager	-	-
	20	Validate Derived kindred	4	KH
Medium				
	-	Explorer	-	-
		Launch tab synchronization (only show queries/analyses for logged in users)	8	T&A
	21	Delete settings (?)	4	T&A



Sprint Backlog

- A subset of Product Backlog Items, which defines the work for a Sprint
 - Created by Team members
 - Each Item has it's own status
 - Updated daily



Sprint Backlog during the Sprint

- Changes occur:
 - Team adds new tasks whenever they need in order to meet the Sprint Goal
 - Team can remove unnecessary tasks
 - But, Sprint Backlog can only be updated by the team
- Estimates are updated whenever there's new information

Burn down Charts

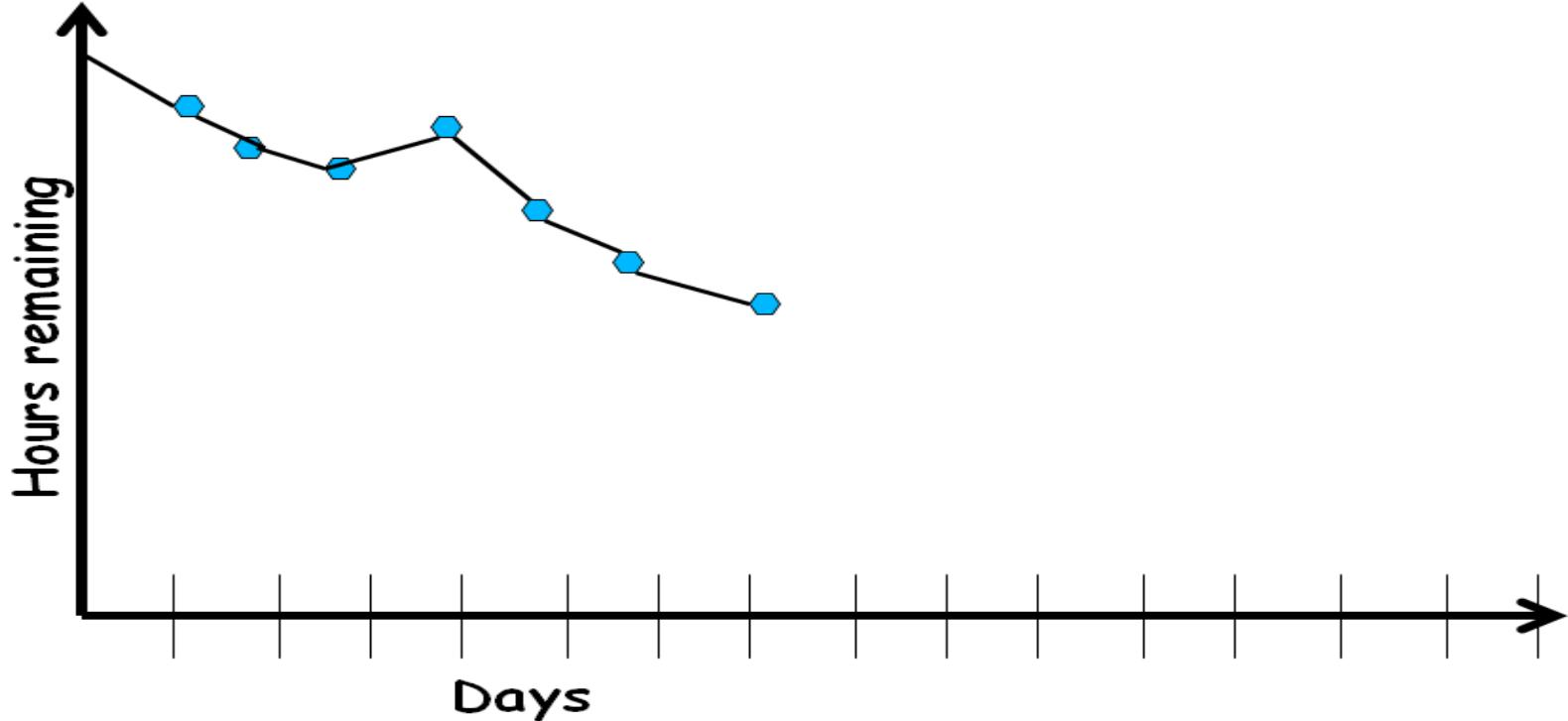
- Are used to represent “work done”.
- Are remarkably simple but effective Information disseminators
- 3 Types:
 - Sprint Burn down Chart (progress of the Sprint)
 - Release Burn down Chart (progress of release)
 - Product Burn down chart (progress of the Product)



Sprint Burn down Chart

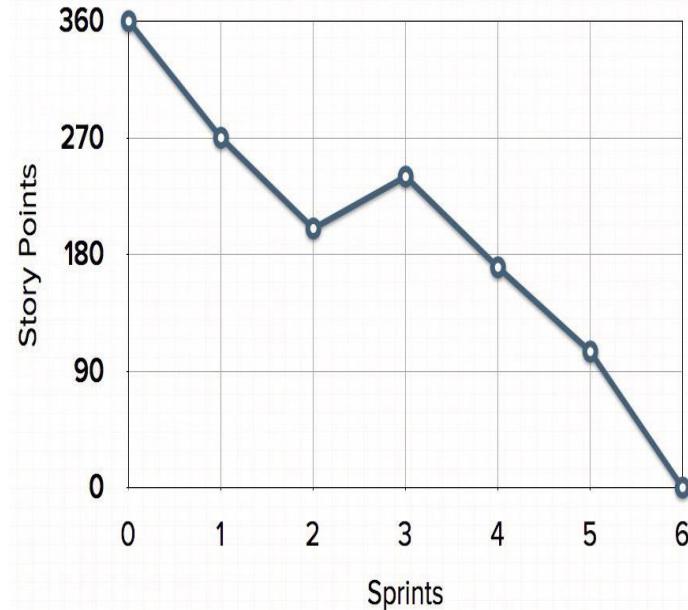
- Depicts the total Sprint Backlog hours remaining per day
- Shows the estimated amount of time to complete
- Ideally should burn down to zero to the end of the Sprint
- Actually is not a straight line

Sprint Burndown Chart



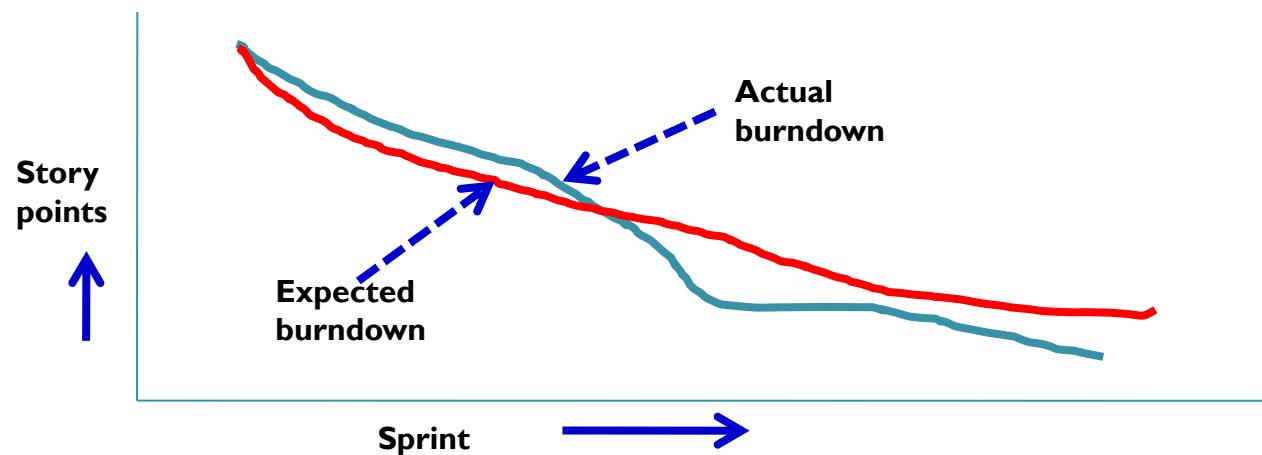
Release Burndown Chart

- Will the release be done on right time?
- How many more sprints?
- X-axis: sprints
- Y-axis: amount of story points remaining



Product Burndown Chart

- It is a “big picture” view of project’s progress (all the releases)





Scalability of Scrum

- A typical Scrum team is 6-10 people
- Jeff Sutherland - up to over 800 people
 - “Scrum of Scrums” or “Meta-Scrum”

Summary

- Discussed the basic concepts of Scrum.
- Presented the characteristics of Scrum.
- Explained the Scrum Life Cycle.
- Discussed the Scrum Framework.
 - Roles
 - Ceremonies
 - Artifacts
- Explained the Burn down Charts.

References

1. Rajib Mall, Fundamentals of Software Engineering, Fifth Edition, PHI, 2018.
2. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter – 16), Second Edition, Oxford University Press, 2018.



Thank You



Testing Agile Based Software cont..

Prof. Durga Prasad Mohapatra

Dept. of CSE, NIT Rourkela

India

Agile Testing Life Cycle

- Agile Testing Life Cycle is based on the '*more is less*' principle which focuses on communication management among stakeholders.
- The adoption of this principle results in quality software product as shown in next figure.
- If communication between the tester and other stakeholder is very frequent and effective, then there would be very few doubts and more clarity.



Agile Testing Life Cycle cont..

- This principle ensures that testing activities along with effective communication and collaboration among major stakeholders, such as business analyst, market evaluator, customer, and developer, result in software products having only few defects.

More is Less Principle

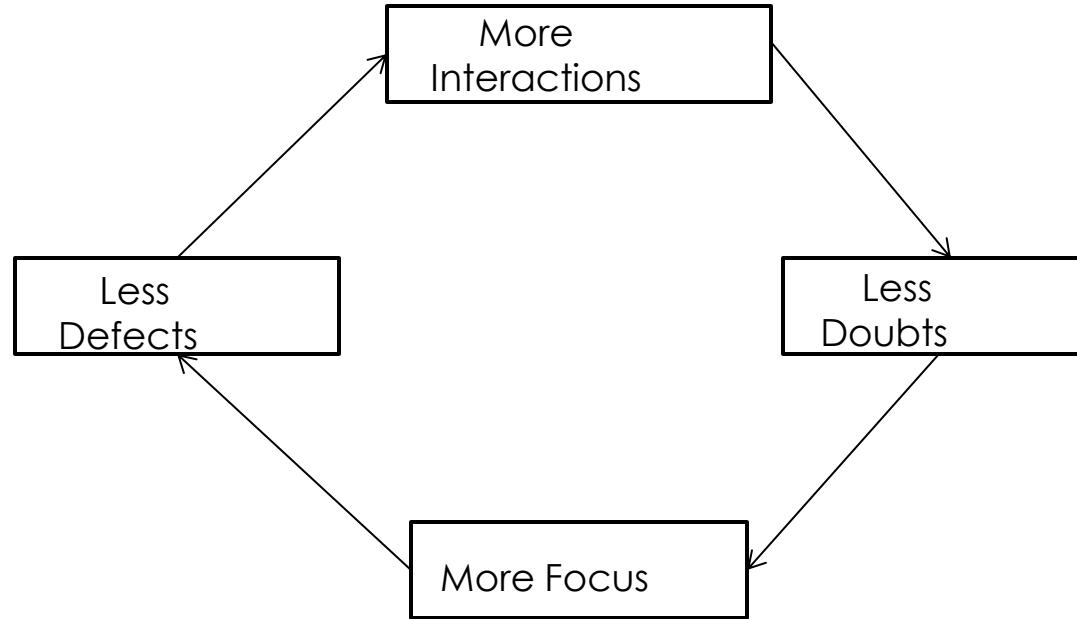


Fig 1: More is less principle

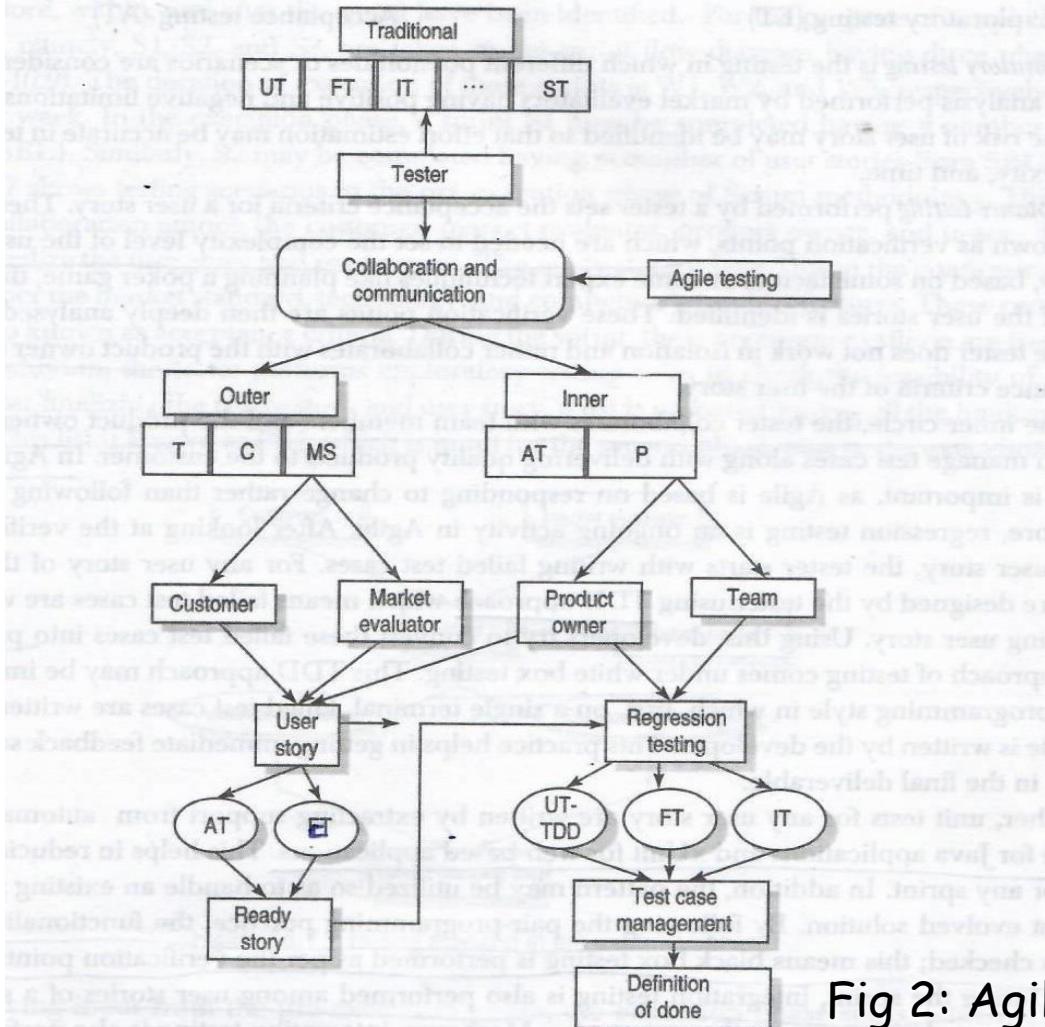


Fig 2: Agile testing life cycle

Agile Testing Abbreviations

Abbreviation	Full Form
UT	Unit Testing
FT	Functional Testing
IT	Integration Testing
ST	System Testing
T	Technology
C	Competitor
MS	Market Standard
AT	Automated Tool
P	Pattern
AT	Acceptance Testing
ET	Exploratory Testing
TDD	Test Driven Development



Agile Testing Life Cycle

cont..

- An agile tester interacts and collaborates with 2 circles, namely an outer circle and inner circle.
- The outer circle is connected to the outside world
- In the outer circle the tester collaborates with customers and market evaluators.
- Then convert the informal requirements into a formal set of requirements known as the *user story*.

Agile Testing Life Cycle cont ...

- Then the tester converts the user story into a ready story.
- This ready story acts like a checklist at the time of verification or acceptance of the user story by the customer.
- This ready story is the outcome of performing 2 types of testing.
 - Exploratory testing(ET)
 - Acceptance testing(AT)



Agile Testing Life Cycle cont..

- Exploratory testing is the testing in which different possibilities or scenarios are considered/explored as per the market analysis performed by market evaluators having positive and negative limitations.
- At the same time, the risk of user story may be identified so that effort estimation may be accurate in terms of effort, complexity, time.



Agile Testing Life Cycle cont..

- Acceptance testing performed by a tester sets the acceptance criteria for a user story. These criteria also known as verification points, which are needed to set the complexity level of the user story. These verification points are then deeply analysed.
- In this case also, the tester does not work in isolation and rather collaborates with the product owner to finalize the acceptance criteria of the user story.



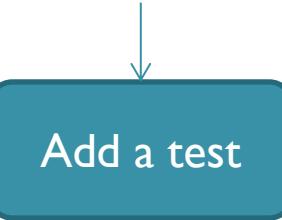
Agile Testing Life Cycle cont..

- In the inner circle, the tester collaborates with team members and product owner.
- In Agile, regression testing is important, as Agile is based on responding to change rather than following a fixed plan.
- So regression testing is an on going activity in Agile.

Agile Testing Life Cycle cont..

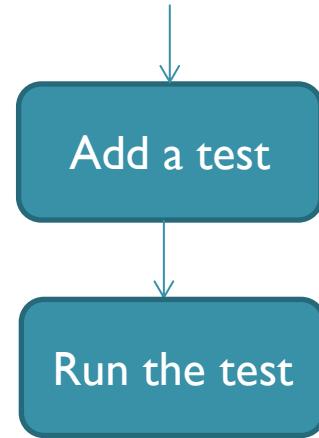
- After looking at the verification points of the user story, the tester starts with writing failed test cases.
- For any user story of the sprint, test cases are designed by the tester using Test Driven Development(TDD) approach which means failed test cases are written for the upcoming user story.

Test Driven Development

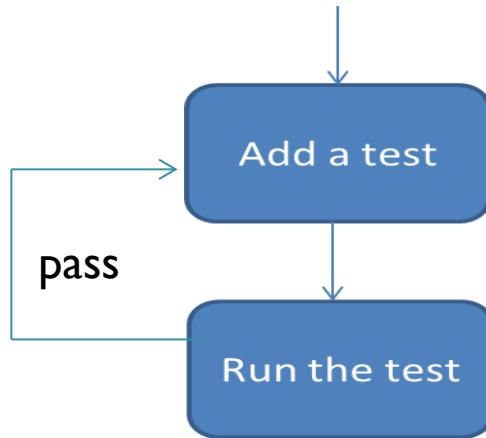


TDD Rhythm-Test, Code ,Refactor

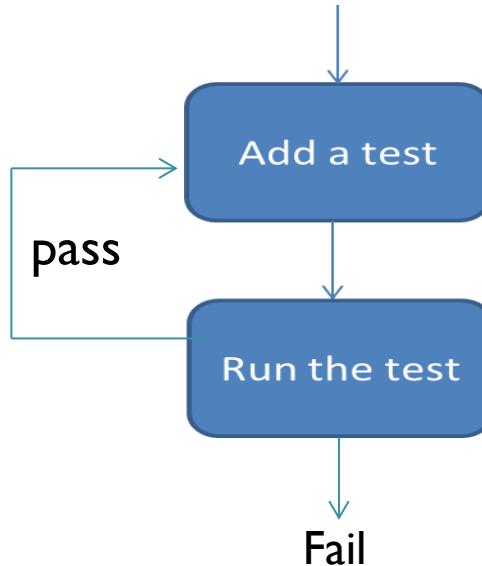
Test Driven Development cont..



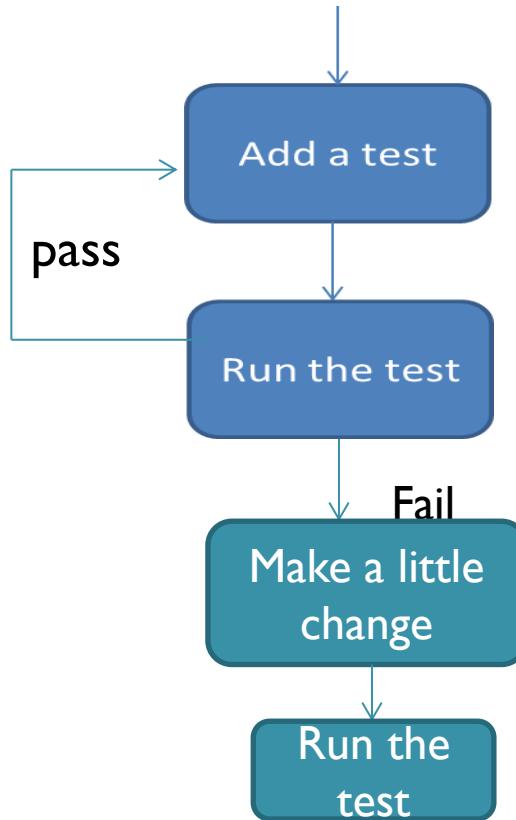
Test Driven Development cont..



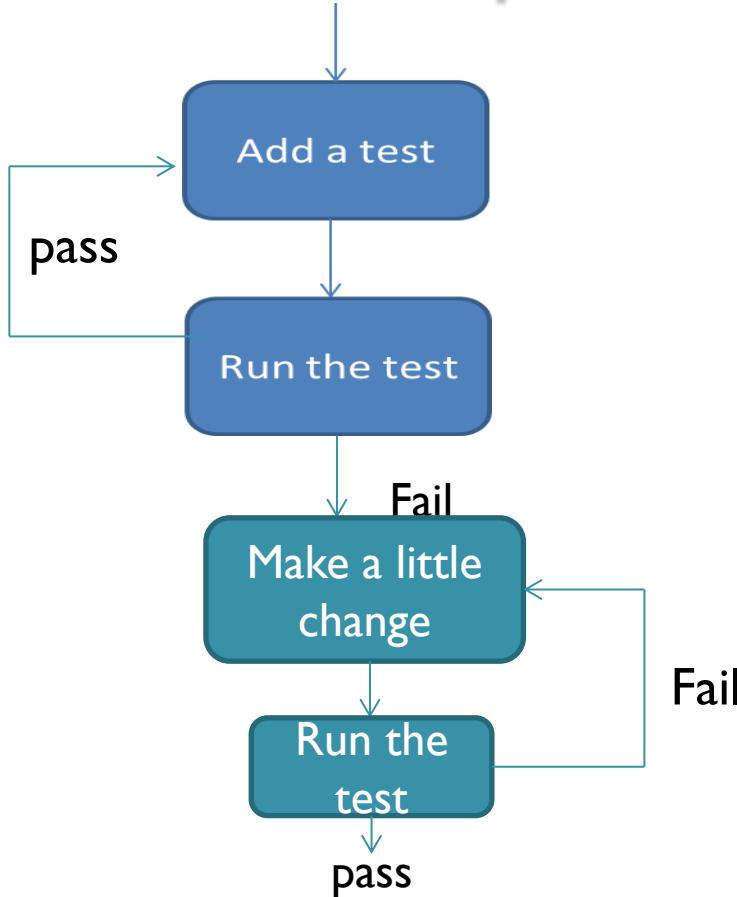
Test Driven Development cont..



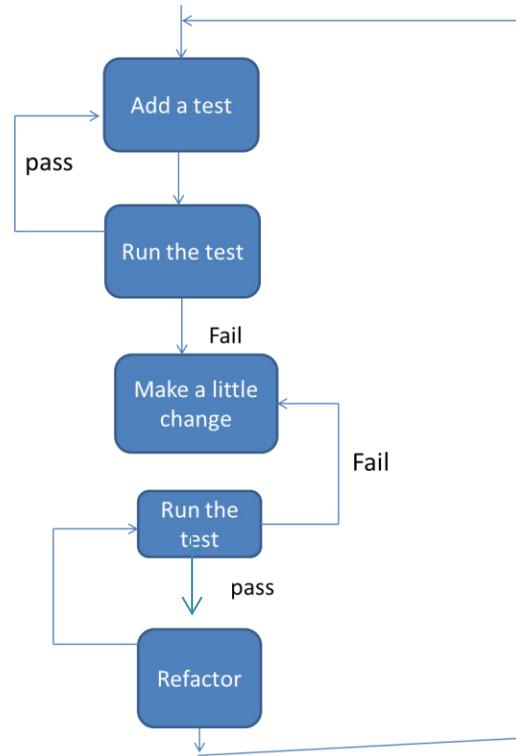
Test Driven Development cont..



Test Driven Development cont..



Test Driven Development cont..





Agile Testing Life Cycle cont..

- Using this, developers try to convert these failed test cases into pass test cases.
- This TDD approach may be implemented in a pair-programming style in which, first, on a single terminal, failed test cases are written, after which the code is written by the developer.
- This practice helps in getting immediate feedback so as to embed quality in final deliverable.



Agile Testing Life Cycle cont..

- Unit tests for any user story are written by extracting support from automated tools like eclipse for Java applications and xUnit for web-based applications.
- This helps in reducing the overall time for any sprint.
- In addition, the pattern may be utilized so as to handle an existing problem with the best evolved solution.



Agile Testing Life Cycle cont..

- During the sprint, integration testing also performed among user stories of a sprint by considering dependencies among the user stories.
- Moreover, integration testing is also performed among user stories of the different sprints.

Agile Testing Life Cycle cont..

- Further, to manage test cases, effective regression techniques, such as regression test selection (RTS) and test case prioritization (TCP) are implemented to run only a subset of test cases out of all the test cases.
- Finally, ‘definition of done’ is declared by the customer after matching the verification points of the ready story with the actual product.

Testing in scrum phases

- Scrum methodology is based upon small duration sprints having small number of user stories listed in the sprint backlog list(SBL), which is a subset of PBL(product backlog list).
- Testing in scrum is divided into 3 phases called
 - Pre-execution phase
 - Execution phase
 - Post-execution phase

Testing in scrum phases cont..

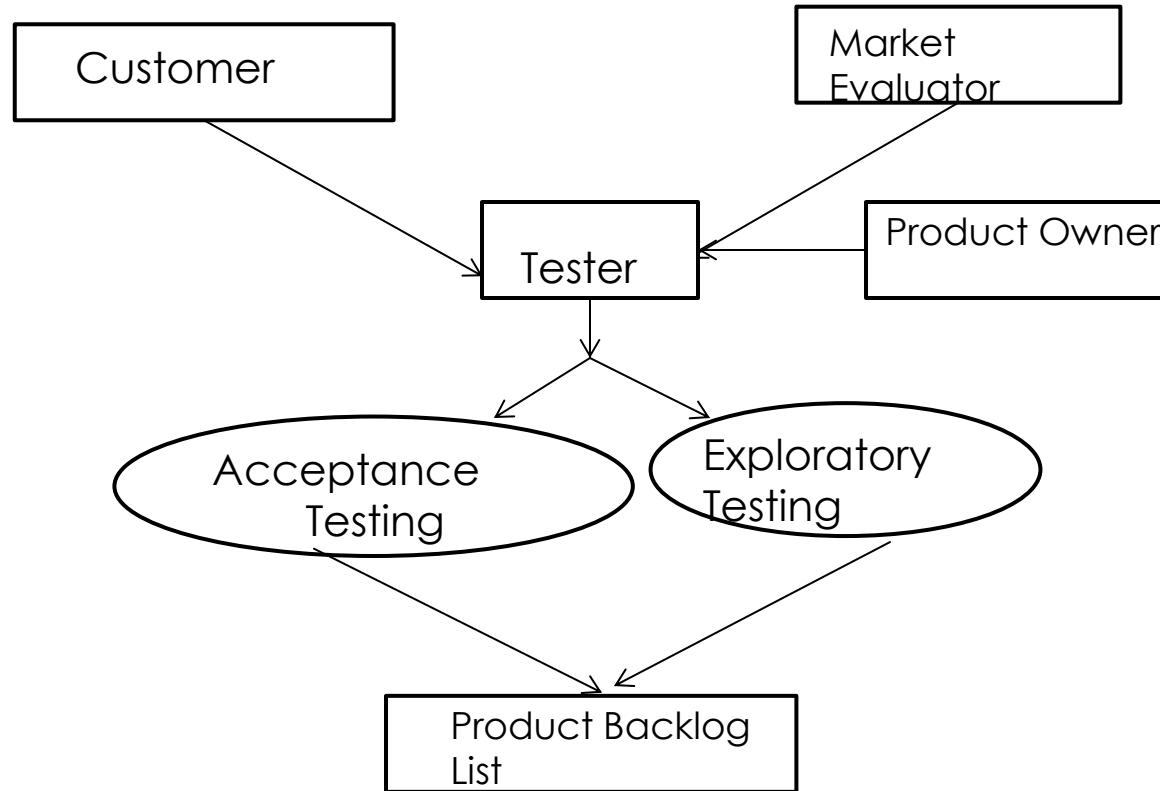
- Here all the testing activities occurring before, within, after the sprint have been identified.
- Here for simplicity, only 3 sprints S1, S2, S3 are taken in sprint flow diagram having 3 phases.
- The duration of execution of these sprints is W1, W2, W3 respectively, where W stands for week.



Testing in scrum phases cont..

- In the execution phase, a sprint S1 may be completed having ‘n’ number of user stories from SBL1.
- Similarly, S2 may be completed having ,m, number of user stories from SBL2.

Testing Scenario in Pre-execution Phase



Testing Scenario in Pre-execution Phase cont..

- This phase starts with collaboration among the customer, market evaluator, product owner and tester.
- They sit together to finalize the user story and ready story.
- The ready story is based upon the conforming points which are as per the market standard, technology and competitor's product features.

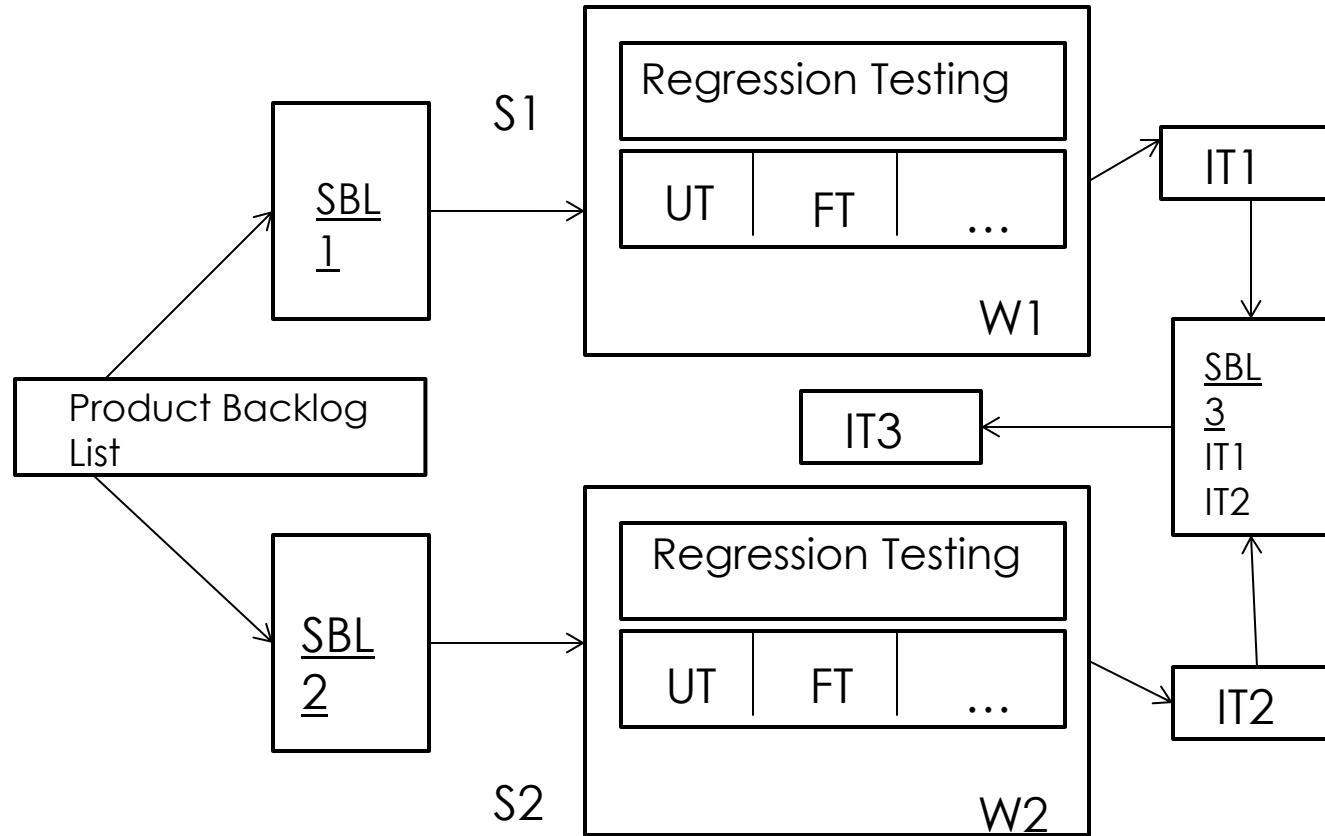
Testing Scenario in Pre-execution Phase cont..

- These conforming points are also known as acceptance criteria.
- During the sprint the acceptance criteria are frequently checked.
- In addition, the tester performs exploratory testing so as to check the feasibility of various scenarios.

Testing Scenario in Pre-execution Phase cont..

- After finalizing the ready story and the user story, a list is prepared having all the finalized set of user stories.
- This list is known as PBL, which is input for the second phase, that is the execution phase.

Testing Scenario in Execution Phase



Testing Scenario in Execution Phase cont..

- After receiving input from the pre-execution phase, the execution phase starts.
- PBL is analyzed by the PO(product owner) and the effort estimation is done for selecting the user stories for SBL1 and SBL2. SBL1 and SBL2 are executed in sprint S1 and S2 respectively.
- In S1 the tester performs unit testing with TDD or white box testing, functional testing or black-box testing, regression testing, integration testing among dependent user stories, and many more depending on the requirements of customer.

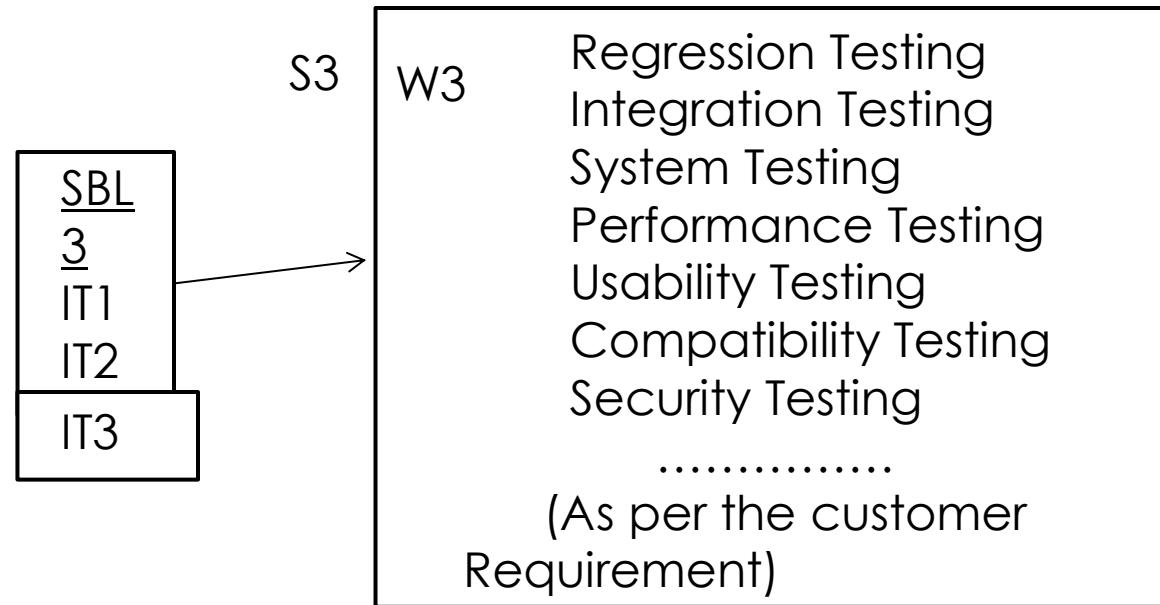
Testing Scenario in Execution Phase cont..

- The output of SI is an integrated set of user stories, IT1 with regression test suite during WI duration.
- Similarly , in sprint S2, the same types of testing are performed--- an integrated set of user stories IT2 with regression test suite during W2 duration.

Testing Scenario in Execution Phase cont..

- Further, these integrated set of user stories IT1 and IT2 are considered user stories In SBL3.
- Further, there may be other user stories which need to be developed in W3 duration which are newly added features in the maintenance time of the product.
- SBL3 is input for the post-execution phase.

Testing Scenario in Post-Execution Phase



Testing Scenario in Post-Execution Phase cont..

- In post-execution phase, user stories are selected from SBL3, based on the priority set by the customer, complexity level, risk level, or any other prioritization factor.
- Various types of testing that are performed in S3 are integration of IT1 and IT2, functional testing, system testing, and regression testing depending on the modification suggested by the customer, if any.

Testing Scenario in Post-Execution Phase cont..

- Other optional testing that may be performed in W3 duration are compatibility testing, security testing, performance testing, usability testing etc.
- Finally, a software product is delivered to the customer.



Regression testing in Agile

- Regression testing in Agile environment is practised under two major categories.
- **Sprint level regression testing (SLRT):** Focuses on testing new functionalities that have been incorporated since the last production release.
- **End-to-end regression testing(EERT):** Refers to the regression testing that incorporates all the fundamental functionalities.



Regression testing in Agile cont..

- Each sprint cycle is followed by a small span of SLRT.
- The completed code goes through further regression cycles but not released into production.
- After few successful sprint cycles-typically 3 to 4 – the application goes through one round of EERT before being released to production.



Challenges to Agile testing

- Frequent changes in agile may have crucial after-effects if necessary steps are not taken, so there is a need to perform regression testing using effective techniques so as to reduce the size of pending backlogs of user stories and the test suite.
- The changes that are introduced later may have several unnoticed effects in the working system. These effects must be controlled in a planned manner by team members to deliver the quality.



Challenges to Agile testing cont..

- In distributed Agile, testing using pair-programming practice may be cumbersome as the first team member of the pair may be at one location and the second member of the pair may be at different location.
- In this scenario, other issues also arise, such as language barrier, cultural barrier, and time zone barrier for effective communication among team members of the sprint. So quality may lag in software products.
- As the number of sprints increases, the test suite size also grows, hence, management of test cases becomes a problem in a distributed environment.



Agile Testing Tool

- TestRail – A Modern Agile Testing Tool to Boost Your Software Testing Efforts.
- Organizes test cases, manages test runs, tracks test results, and measures progress.
- Helps you meet your quality goals and complete your tests on time.

Critical points for agile testing

- Testing is very critical from agile perspective. Typically in scrum, there are many changes and test team must be capable of handling huge regression testing cycle as one progresses from iteration to iteration.
- Understanding of requirements, creation of reusable test cases, integration testing along with regression testing are key factors in successful testing .



Critical points for agile testing cont..

- Competencies/Maturity of Agile Development and Test Team
- Development and Test Process Variability
- Change Management and Communication
- Test Process Flexibility
- Focus on Business Objective
- Stakeholder Maturity/Involvement



Competencies/Maturity of Agile Development and Test Team

- For undertaking agile, one must have the teams, customer and management who psychologically accept agile approach.
- It talks about ability to change very fast, build good working product and communicate with team members and stakeholders effectively as well as efficiently.
- Agile implementation may prefer generalist approach as against specialist approach.
- It needs people with very high maturity as well as technical competence to adapt to changing needs of customer.

Development and Test Process

Variability

- Every process has an inborn variability.
- One may have to attack the generic reasons of variations while there may be some controls to identify special causes of variations.
- One must be able to plot development and test process, and try to remove personal factor from the processes.



Change Management and Communication

- Change is inevitable in agile. It flows from customer to development team and goes back to customer.
- There must be a very close communication between development, test team, customer, and other stakeholders to adapt to changing scenario.
- Requirement change must be welcomed and all people together must decide how customer can be served best.



Test Process Flexibility

- Change is must in agile, and one may have to adapt to the changes. Test process is not an exception to it.
- Different parts of software need different strategies of testing.
- There may be different test plans or one may keep flexibility in a test plan to adapt to these changes. There may be changes in focus in each iteration.
- Initially, there may be heavy unit testing, then it may have integration testing where different iterations come together.
- It may be followed by heavy regression testing.



Focus on Business Objective

- There is always a time pressure in agile development.
- Pressure may come from stakeholders to deliver things faster or it may come from development, if they get delayed.
- One may have to focus on business while defining test process.
- Cost-benefit analysis may be done when it comes to defect fixes and release of software.
- Nobody can find all defects but user must be protected from any accidental failure.
- Testing has to achieve both extremes.



Stakeholder Maturity/Involvement

- Agile development also needs a good maturity from stakeholders.
- Internal and external service providers must understand and work with time pressure.
- Good process of development and testing must be supported by tools and techniques required for agile implementation.
- There may be some specific requirements of stake holders, and these requirements must be rearranged to suite agile development.

Summary

- Discussed the agile testing life cycle in detail.
- Highlighted the test driven development (TDD) approach.
- Discussed agile testing in different scrum phases.
- Explained regression testing in agile.
- Presented some of challenges to agile testing.
- Discussed some critical points for agile testing.



References

- I. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter – 16), Second Edition, Oxford University Press, 2018.



Thank You

Introduction to Android App Testing

Dr. Durga Prasad Mohapatra

Professor

Department of Computer Science & Engineering

N. I. T., Rourkela

Why, what, how, and when to test?

- Early bug detection saves a huge amount of project resources and reduces software maintenance costs. This is the best known reason to write tests for software development project.
- Writing tests will give you a deeper understanding of the requirements and the problem to be solved.
- The reason behind the approach of writing tests is to clearly understand the legacy or third-party code and having the testing infrastructure to confidently change or update the codebase.
- The more the code is covered by your tests, the higher the likelihood of discovering hidden bugs.

What to test

- Developer should test every statement in your code, but this also depends on different criteria and can be reduced to testing the main path of execution or just some key methods.
- Areas of android testing that should be considered are:
 - Activity lifecycle events: You should test whether your activities handle life cycle events correctly. Configuration change events should also be tested as some of these events cause the current activity to be recreated.
 - Database and file system operations: These operations should be tested to ensure that the operations and any errors are handled correctly. These operations should be tested in isolation at lower level, at a higher level or from the application itself.

➤ Physical characteristics of the device: Before shipping your application, you should be sure that all of the different devices it can be run on are supported, or at least you should detect the unsupported situations and take remedial actions. The characteristics of the devices that should be tested are:

- Network capabilities
- Screen densities
- Screen resolutions
- Screen sizes
- Availability of sensors
- Keyboard & other input devices
- GPS
- External storage

- In this respect, an Android emulator can play an important role because it is practically impossible to have access to all of the devices with all of the possible combinations of features, but you can configure emulators for almost every situation.
- However, leave your final tests for actual devices where the real users will run the application so you get feedback from a real environment

Types of tests

- Testing comes in a variety of frameworks with differing levels of support from the Android SDK and your IDE of choice.
- There are several types of tests depending on the code being tested. Regardless of its type, a test should verify a condition and return the result of this evaluation as a single Boolean value that indicates its success or failure.
- Types of tests are:
 - Unit tests
 - Integration tests
 - UI tests
 - Functional or acceptance tests
 - Performance tests
 - System tests

Unit tests

- Unit tests are tests written by programmers for other programmers, and they should isolate the component under tests and be able to test it in a repeatable way.
- Thus, unit tests and mock objects are usually placed together. Mock object is a drop-in replacement for the real object, where you have more control of the object's behavior.
- Mock objects are used to isolate the unit from its dependencies, to monitor interactions, and also to be able to repeat the test any number of times.
- JUnit is the de facto standard for unit tests on Android. It's a simple open source framework for automating unit testing, originally written by Erich Gamma and Kent Beck.

Components used to build up a test case.

- **The `setUp()` method:** This method is called to initialize the fixture (fixture being the test and its surrounding code state). Overriding it, you have the opportunity to create objects and initialize fields that will be used by tests. It's worth noting that this setup occurs before every test.
- **The `tearDown()` method:** This method is called to finalize the fixture. Overriding it, you can release resources used by the initialization or tests. Again, this method is invoked after every test.
- For example, you can release a database or close a network connection here. There are more methods you can hook into before and after your test methods, but these are used rarely, and will be explained as we bump into them.

Components used to build up a test case.

- **Outside the test method:** JUnit is designed in a way that the entire tree of test instances is built in one pass, and then the tests are executed in a second pass.
 - This means that for very large and very long test runs with many Test instances, none of the tests may be garbage collected until the entire test is run.
 - This is particularly important in Android and while testing on limited devices as some tests may fail not because of an intrinsic failure but because of the amount of memory needed to run the application, in addition to its tests exceeding the device limits.
- **Inside the test method:** All public void methods whose names start with test will be considered as a test. As opposed to JUnit 4, JUnit 3 doesn't use annotations to discover the tests; instead, it uses introspection to find their names.
 - There are some annotations available in the Android test framework such as @SmallTest, @MediumTest, or @LargeTest, which don't turn a simple method into a test but organize them in different categories. Ultimately, you will have the ability to run tests for a single category using the test runner.

Components used to build up a test case.

- As a rule of thumb, name your tests in a descriptive way and use nouns and the condition being tested. Also, remember to test for exceptions and wrong values instead of just testing positive cases.
- For example, some valid tests and naming could be:
 - `testOnCreateValuesAreLoaded()`
 - `testGivenIllegalArgumentThenAConversionErrorIsThrown()`
 - `testConvertingInputToStringIsValid()`
- During the execution of the test, some conditions, side effects, or method returns should be compared against the expectations. To ease these operations, JUnit provides a full set of `assert*` methods to compare the expected results from the test to the actual results after running them, throwing exceptions if the conditions are not met.
- Then, the test runner handles these exceptions and presents the results.

Components used to build up a test case.

- These methods, which are overloaded to support different arguments, include:
 - `assertTrue()`
 - `assertFalse()`
 - `assertEquals()`
 - `assertNull()`
 - `assertNotNull()`
 - `assertSame()`
 - `assertNotSame()`
 - `fail()`

Mock objects

- Mock objects are mimic objects used instead of calling the real domain objects to enable testing units in isolation.
- The Android testing framework supports mock objects that you will find very useful when writing tests. You need to provide some dependencies to be able to compile the tests.
- There are also external libraries that can be used when mocking. Several classes are provided by the Android testing framework in the `android.test.mock` package:
 - `MockApplication`
 - `MockContentProvider`
 - `MockContentResolver`
 - `MockContext`
 - `MockCursor`
 - `MockDialogInterface`
 - `MockPackageManager`
 - `MockResources`

Integration tests

- Integration tests are designed to test the way individual components work together. Modules that have been unit tested independently are now combined together to test the integration.
- Usually, Android Activities require some integration with the system infrastructure to be able to run. They need the Activity lifecycle provided by the Activity Manager, and access to resources, the file system, and databases.
- The same criteria apply to other Android components such as Services or Content Providers that need to interact with other parts of the system to achieve their duty.
- In all these cases, there are specialized test classes provided by the Android testing framework that facilitates the creation of tests for these components.

UI tests

- User Interface tests test the visual representation of your application, such as how a dialog looks or what UI changes are made when a dialog is dismissed.
- As you may have already known, only the main thread is allowed to alter the UI in Android. Thus, a special annotation `@UiThreadTest` is used to indicate that a particular test should be run on that thread and it would have the ability to alter the UI.
- On the other hand, if you only want to run parts of your test on the UI thread, you may use the `Activity.runOnUiThread(Runnable r)` method that provides the corresponding Runnable, which contains the testing instructions.

UI tests

- A helper class *TouchUtils* is also provided to aid in the UI test creation, allowing the generation of the following events to send to the Views, such as:

- Click
- Drag
- Long click
- Scroll
- Tap
- Touch

By these means, you can actually remote control your application from the tests.

Functional or acceptance tests

- In agile software development, functional or acceptance tests are usually created by business and Quality Assurance (QA) people, and expressed in a business domain language.
- These are high-level tests to assert the completeness and correctness of a user story or feature.
- They are created ideally through collaboration between business customers, business analysts, QA, testers, and developers. However, the business customers (product owners) are the primary owners of these tests.
- Lately, within acceptance testing, a new trend named *Behavior-driven Development* has gained some popularity, and in a very brief description, it can be understood as a cousin of Test-driven Development.

Functional or acceptance tests

- Behavior-driven Development can be expressed as a framework of activities based on three principles (more information can be found at <http://behaviour-driven.org>):
 - Business and technology should refer to the same system in the same way
 - Any system should have an identified, verifiable value to the business
 - Upfront analysis, design, and planning, all have a diminishing returnTo apply these principles, business people are usually involved in writing test case scenarios in a HLL and use a tool such as *jbehave*.

Test Case Scenario for “jbehave”

- Example:for this scenario is- Given I am using temperature converter.
100 Celsius should be converted to 212 Fahrenheit in respective field.
- *@Given(“I am using the temperature converter”)*

```
public void createTemperatureConverter(){  
// do nothing this is syntactic sugar for readability }  
  
@When(“enter Celsius”) {this.celsius=celsius}  
  
@Then(“ObtainedFahrenheit”)  
{assertEquals(Fahrenheit, TemperatureConverter.celsiusToFahrenheit(celsius)})
```

Performance tests

- Performance tests measure performance characteristics of the components in a repeatable way.
- If performance improvements are required by some part of the application, the best approach is to measure performance before and after a change is introduced.
- As is widely known, premature optimization does more harm than good, so it is better to clearly understand the impact of your changes on the overall performance.
- The introduction of the Dalvik JIT compiler in Android 2.2 changed some optimization patterns that were widely used in Android development.
- Nowadays, every recommendation about performance improvements in the Android developer's site is backed up by performance tests.

System tests

- The system is tested as a whole, and the interaction between the components, software, and hardware is exercised. Normally, system tests include additional classes of tests such as:
 - GUI tests
 - Smoke tests
 - Mutation tests
 - Performance tests
 - Installation tests

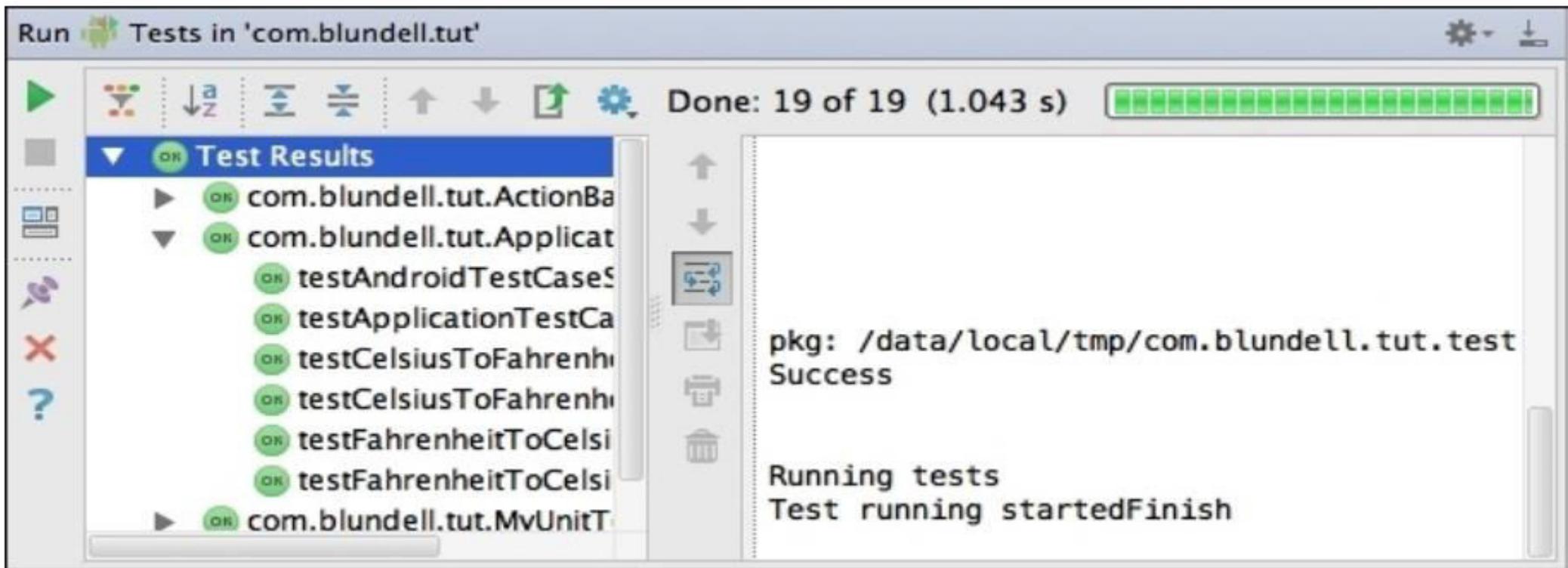
Smoke Testing

- To ensure that system testing would be meaningful.
- It is done before initiating system testing.
- If integrated program cannot pass even the basic tests, its not ready for vigorous testing
- A few test cases are designed to check basic functionalities working
- Ex- In library Automation system, smoke testing may check whether books can be created and deleted and so on.

Android Studio and IDE Support

- Junit is fully supported by this tool and helps in creating the tested android project.
- We can run the tests and analyze the results without leaving the IDE.
- We are able to run the tests from IDE allows us to debug tests that are not behaving correctly.
- Even if we are not developing in an IDE, we can find support to run the tests with gradle (check <http://gradle.org> if we are not familiar with this tool).

Continue...



- Here we see Aside runs 19 unit tests, taking 1.043 seconds, with 0 errors and 0 failures detected.
- Name and duration of test is being displayed. Failure trace is shown on right side of screenshot showing test related information.

Java testing framework

- The Java testing framework is the backbone of Android testing, and sometimes, you can get away without writing Android-specific code.
- Android framework tests to a device, and this has an impact on the speed of our tests, that is, the speed we get feedback from a pass or a fail.
- If you architect your app in a clever way, you can create pure Java classes that can be tested in isolation away from Android.
- The two main benefits of this are increased speed of feedback from test results, and also, to quickly plug together libraries and code snippets to create powerful test suites, you can use the near ten years of experience of other programmers doing Java testing.

Android testing framework

- Android provides a very advanced testing framework that extends the industry standard JUnit library with specific features that are suitable to implement all of the testing strategies and types we mentioned before.
- In some cases, additional tools are needed, but the integration of these tools is, in most of the cases, simple and straightforward.
- Most relevant key features of the Android testing environment include:
 - Android extensions to the JUnit framework that provide access to Android system objects
 - An instrumentation framework that lets the tests control and examine the application
 - Mock versions of commonly used Android system objects
 - Tools to run single tests or test suites, with or without instrumentation
 - Support to manage tests and test projects in Android Studio and at the command line

Instrumentation

- The instrumentation framework is the foundation of the testing framework.
- Instrumentation controls the application under tests and permits the injection of mock components required by the application to run. For example, you can create mock Contexts before the application starts and let the application use it.
- All the interactions of the application with the surrounding environment can be controlled using this approach.
- You can also isolate your application in a restricted environment to be able to predict the results that force the values returned by some methods, or that mock persistent and unchanged data for the ContentProvider's databases or even the file system content.

Instrumentation

- A standard Android project has its instrumentation tests in a correlated source folder called androidTest.
- This creates a separate application that runs tests on your application.
- There is no AndroidManifest here as it is automatically generated. The instrumentation can be customized inside the Android closure of your build.gradle file, and these changes are reflected in the autogenerated AndroidManifest.
- However, you can still run your tests with the default settings if you choose to change nothing.

Instrumentation

- Examples of things you can change are the test application package name, your test runner, or how to toggle performance-testing features:
 - `testApplicationId "com.blundell.something.non.default"`
 - `testInstrumentationRunner "com.blundell.tut.CustomTestRunner"`
 - `testHandleProfiling false`
 - `testFunctionalTest true`
 - `testCoverageEnabled true`
- Here, the Instrumentation package (`testApplicationId`) is a different package to the main application. If you don't change this yourself, it will default to your main application package with the `.test` suffix added.

Instrumentation

- At the moment, `testHandleProfiling` and `testFunctionalTest` are undocumented and unused, so watch out for when we are told what we can do with these.
- Setting `testCoverageEnabled` to `true` will allow you to gather code coverage reports using *Jacoco*.
- Also, notice that both the application being tested and the tests themselves are Android applications with their corresponding APKs installed.
- Internally, they will be sharing the same process and thus have access to the same set of features.

Gradle

- Gradle is an advanced build toolkit that allows you to manage dependencies and define a custom login to build your project.
- The Android build system is a plugin on top of Gradle, and this is what gives you the domain-specific language discussed previously such as setting a testInstrumentationRunner.
- The idea of using Gradle is that it allows you to build your Android apps from the command line for machines without using an IDE such as a continuous integration machine.
- Also, with first line integration of Gradle into the building of projects in Android Studio, you get the exact same custom build configuration from the IDE or command line.

Gradle

- Other benefits include being able to customize and extend the build process; for example, each time your CI builds your project, you could automatically upload a beta APK (Android Application Package) to the Google play store.
- You can create multiple APKs with different features using the same project, for example, one version that targets Google play in an app purchase and another that targets the Amazon app store's coin payments.

Test targets

- During the evolution of your development project, your tests would be targeted to different devices.
- From simplicity, flexibility, and speed of testing on an emulator to the unavoidable final testing on the specific device you are intending your application to be run upon, you should be able to run your application on all of them.
- There are also some intermediate cases such as running your tests on a local JVM virtual machine, on the development computer, or on a Dalvik virtual machine or Activity, depending on the case.
- The emulator is probably the most powerful target as you can modify almost every parameter from its configuration to simulate different conditions for your tests.
- Ultimately, your application should be able to handle all of these situations, so it's much better to discover the problems upfront than when the application has been delivered.

Creating the Android project

- We will create a new Android project. This is done from the ASide menu by going to File | New Project. This then leads us through the wysiwyg guide to create a project.
- In this particular case, we are using the following values for the required component names (clicking on the Next button in between screens):
 - Application name: AndroidApplicationTestingGuide
 - Company domain: blundell.com
 - Form factor: Phone and Tablet
 - Minimum SDK: 17
 - Add an Activity: Blank Activity (go with default names)
- When you click on Finish and the application is created, it will automatically generate the androidTest source folder under the app/src directory, and this is where you can add your instrumented test cases.

Tip

- Alternatively, to create an androidTest folder for an existing Gradle Android project, you can select the src folder and then go to File | New | Directory.
- Then, write androidTest/java in the dialog prompt. When the project rebuilds, the path will then automatically be added so that you can create tests.

Package explorer

- After having created our project, the project view should look like one of the images shown in the following screenshot. This is because ASide has multiple ways to show the project outline.
- On the left, we can note the existence of the two source directories, one colored green for the test source and the other blue for the project source.
- On the right, we have the new Android project view that tries to simplify the hierarchy by compressing useless and merging functionally similar folders.

Creating a test cases

- As described before, we are creating our test cases in the src/androidTest/java folder of the project.
- You can create the file manually by right-clicking on the package and selecting New... | Java Class.
- However, in this particular case, we'll take advantage of ASide to create our JUnit TestCase.
- Open the class under test (in this case, Main Activity) and hover over the class name until you see a light bulb (or press Ctrl/Command + 1). Select Create Test from the menu that appears.

Creating a test cases

- These are the values that we should enter when we create the test case:
 - Testing library: JUnit 3
 - Class name: MainActivityTest
 - Superclass: junit.framework.TestCase
 - Destination package: com.blundell.tut
 - Superclass: junit.framework.TestCase
 - Generate: Select none
- As you can see, you could also have checked one of the methods of the class to generate an empty test method stub. These stub methods may be useful in some cases, but you have to consider that testing should be a behavior-driven process rather than a method-driven one.

Creating a test cases

- The basic infrastructure for our tests is in place; what is left is to add a dummy test to verify that everything is working as expected. We now have a test case template, so the next step is to start completing it to suit our needs.
- To do it, open the recently created test class and add the testSomething() test.
- We should have something like this:

```
package com.blundell.tut;  
import android.test.suitebuilder.annotation.SmallTest;  
import junit.framework.TestCase;  
public class MainActivityTest extends TestCase {  
    public MainActivityTest() {  
        super("MainActivityTest");  
    }  
    @SmallTest  
    public void testSomething() throws Exception {  
        fail("Not implemented yet");  
    }  
}
```

Creating a test cases

- The no-argument constructor is needed to run a specific test from the command line, as explained later using an instrumentation.
- This test will always fail, presenting the message: Not implemented yet. In order to do this, we will use the fail method from the junit.framework.Assert class that fails the test with the given message.

Test annotations

- Looking carefully at the test definition, you might notice that we decorated the test using the `@SmallTest` annotation, which is a way to organize or categorize our tests and run them separately.
- This test will always fail, presenting the message: Not implemented yet. In order to do this, we will use the `fail` method from the `junit.framework.Assert` class that fails the test with the given message.

Test annotations

- There are other annotations that can be used by the tests, such as:

Annotation	Description
@SmallTest	Marks a test that should run as part of the small tests.
@MediumTest	Marks a test that should run as part of the medium tests.
@LargeTest	Marks a test that should run as part of the large tests.
@Smoke	Marks a test that should run as part of the smoke tests. The android.test.suitebuilder.SmokeTestSuiteBuilder will run all tests with this annotation.
@FlakyTest	Use this annotation on the InstrumentationTestCase class' test methods. When this is present, the test method is re-executed if the test fails. The total number of executions is specified by the tolerance, and defaults to 1. This is useful for tests that may fail due to an external condition that could vary with time. For example, to specify a tolerance of 4, you would annotate your test with: @FlakyTest(tolerance=4).

Test annotations

- There are other annotations that can be used by the tests, such as:

Annotation	Description
@UiThreadTest	<p>Use this annotation on the InstrumentationTestCase class' test methods. When this is present, the test method is executed on the application's main thread (or UI thread). As instrumentation methods may not be used when this annotation is present, there are other techniques if, for example, you need to modify the UI and get access to the instrumentation within the same test. In such cases, you can resort to the Activity.runOnUiThread() method that allows you to create any Runnable and run it in the UI thread from within your test:</p> <pre>mActivity.runOnUiThread(new Runnable() { public void run() { // do somethings } });</pre>
@Suppress	<p>Use this annotation on test classes or test methods that should not be included in a test suite. This annotation can be used at the class level, where none of the methods in that class are included in the test suite, or at the method level, to exclude just a single method or a set of methods.</p>

Running the tests

- There are several ways of running our tests, and we will analyze them here.
- Additionally, tests can be grouped or categorized and run together, depending on the situation.
- Running all tests from Android Studio:
 - This is perhaps the simplest method if you have adopted ASide as your development environment. This will run all the tests in the package.
 - Select the app module in your project and then go to Run | (android icon) All Tests. If a suitable device or emulator is not found, you will be asked to start or connect one.
 - The tests are then run, and the results are presented inside the Run perspective, as shown in the following screenshot:

Running the tests

- A more detailed view of the results and the messages produced during their execution can also be obtained in the LogCat view within the Android DDMS perspective, as shown in the following screenshot:
- Running a single test case from your IDE:

- There is an option to run a single test case from ASide, should you need to. Open the file where the test resides, right-click on the method name you want to run, and just like you run all the tests, select Run | (android icon) testMethodName.
- When you run this, as usual, only this test will be executed. In our case, we have only one test, so the result will be similar to the screenshot presented earlier.

Running the tests

- Running from the emulator:
 - The default system image used by the emulator has the Dev Tools application installed, providing several handy tools and settings. Among these tools, we can find a rather long list, as is shown in the following screenshot.
 - Now, we are interested in Instrumentation, which is the way to run our tests. This application lists all of the packages installed that define instrumentation tag tests in their project. We can run the tests by selecting our tests based on the package name, as shown in the following screenshot.

Running the tests

- Running tests from the command line:
 - Finally, tests can be run from the command line too. This is useful if you want to automate or script the process.
 - To run the tests, we use the am instrument command (strictly speaking, the am command and instrument subcommand), which allows us to run instrumentations specifying the package name and some other options.
 - You might wonder what “am” stands for. It is short for Activity Manager, a main component of the internal Android infrastructure that is started by the System Server at the beginning of the boot process, and it is responsible for managing Activities and their life cycle. Additionally, as we can see here, it is also responsible for Activity instrumentation.

Running the tests

➤ The general usage of the am instrument command is:

am instrument [flags] <COMPONENTS> -r -e <NAME> <VALUE> -p <FILE> -w

This table summarizes the most common options:

Option	Description
-r	Prints raw results. This is useful to collect raw performance data.
-e <NAME> <VALUE>	Sets arguments by name. We will examine its usage shortly. This is a generic option argument that allows us to set the <name, value> pairs.
-p <FILE>	Writes profiling data to an external file.
-w	Waits for instrumentation to finish before exiting. This is normally used in commands. Although not mandatory, it's very handy, as otherwise, you will not be able to see the test's results.

Running the tests

- Running all tests: This command line will open the adb shell and then run all tests with the exception of performance tests:

```
$: adb shell
```

```
#: am instrument -w
```

```
com.blundell.tut.test/android.test.InstrumentationTestRunner
```

```
com.blundell.tut.MainActivityTest: Failure in testSomething:
```

```
junit.framework.AssertionFailedError: Not implemented yet at
```

```
com.blundell.tut.MainActivityTest.testSomething(MainActivityTest.java:15) at
```

```
java.lang.reflect.Method.invokeNative(Native Method) at
```

```
android.test.AndroidTestRunner.runTest(AndroidTestRunner.java:191) at
```

```
android.test.AndroidTestRunner.runTest(AndroidTestRunner.java:176) at
```

```
android.test.InstrumentationTestRunner.onStart
```

```
(InstrumentationTestRunner.java:554) at
```

```
android.app.Instrumentation$InstrumentationThread.run
```

```
(Instrumentation.java:1701) Test results for InstrumentationTestRunner=.F Time:
```

```
0.002 FAILURES!!! Tests run: 1, Failures: 1, Errors: 0 Note that the package you
```

```
declare with -w is the package of your instrumentation tests, not the package of the application under test.
```

Running the tests

- Running tests from a specific test:

➤ To run all the tests in a specific test case, you can use:

\$: adb shell

```
#:am instrument -w -e class com.blundell.tut.MainActivityTest  
com.blundell.tut.test/android.test.InstrumentationTestRunner
```

- Running a specific test by name :

➤ Additionally, we have the alternative of specifying which test we want to run in the command line:

\$: adb shell

```
#: am instrument -w -e class com.blundell.tut.MainActivityTest\#testSomething  
com.blundell.tut.test/android.test.InstrumentationTestRunner
```

➤ This test cannot be run in this way unless we have a no-argument constructor in our test case; that is the reason we added it before.

Running the tests

- Running specific tests by category:

- As mentioned before, tests can be grouped into different categories using annotations (Test Annotations), and you can run all tests in this category.
- The following options can be added to the command line:

Option	Description
-e unit true	This runs all unit tests. These are tests that are not derived from InstrumentationTestCase (and are not performance tests).
-e func true	This runs all functional tests. These are tests that are derived from InstrumentationTestCase.
-e perf true	This includes performance tests.
-e size {small medium large}	This runs small, medium, or large tests depending on the annotations added to the tests.
-e annotation <annotation-name>	This runs tests annotated with this annotation. This option is mutually exclusive with the size option.

Running the tests

- In our example, we annotated the test method `testSomething()` with `@SmallTest`. So this test is considered to be in that category, and is thus run eventually with other tests that belong to that same category, when we specify the test size as small.
- This command line will run all the tests annotated with `@SmallTest`:

```
$: adb shell
```

```
#: am instrument -w -e size small
```

```
com.blundell.tut.test/android.test.InstrumentationTestRunner
```

Running the tests

- Running tests using Gradle:

➤ Your gradle build script can also help you run the tests and this will actually do the previous commands under the hood. Gradle can run your tests with this command:

gradle connectedAndroidTest

Creating a custom annotation

In case you decide to sort the tests by a criterion other than their size, a custom annotation can be created and then specified in the command line.

As an example, let's say we want to arrange our tests according to their importance, so we create an annotation `@VeryImportantTest`, which we will use in any class where we write tests (`MainActivityTest` for example):

Running the tests

```
package com.blundell.tut;  
/**  
 * Marker interface to segregate important tests  
 */  
@Retention(RetentionPolicy.RUNTIME)  
public @interface VeryImportantTest {  
}
```

Following this, we can create another test and annotate it with `@VeryImportantTest`:

```
@VeryImportantTest  
public void testOtherStuff() {  
    fail("Also not implemented yet");  
}
```

So, as we mentioned before, we can include this annotation in the am instrument command line to run only the annotated tests:

```
$: adb shell  
#: am instrument -w -e annotation com.blundell.tut.VeryImportantTest  
com.blundell.tut.test/android.test.InstrumentationTestRunner
```

Running the tests

- Running performance tests:

We will be reviewing performance test details in Chapter 8, Testing and Profiling Performance, but here, we will introduce the available options to the am instrument command.

To include performance tests on your test run, you should add this command line option:

-e perf true: This includes performance tests

- Dry run

Sometimes, you might only need to know what tests will be run instead of actually running them.

This is the option you need to add to your command line:

-e log true: This displays the tests to be run instead of running them

This is useful if you are writing scripts around your tests or perhaps building other tools.

Debugging tests

- You should assume that your tests might have bugs too. In such a case, usual debugging techniques apply, for example, adding messages through LogCat.
- If a more sophisticated debugging technique is needed, you should attach the debugger to the test runner.
- In order to do this without giving up on the convenience of the IDE and not having to remember hard-to-memorize command-line options, you can Debug Run your run configurations. Thus, you can set a breakpoint in your tests and use it. To toggle a breakpoint, you can select the desired line in the editor and left-click on the margin.
- Once it is done, you will be in a standard debugging session, and the debug window should be available to you.
- It is also possible to debug your tests from the command line; you can use code instructions to wait for your debugger to attach.

Other command line options

- The am instrument command accepts other <name, value> pairs beside the previously mentioned ones:

Name	Value
debug	true. Set break points in your code.
package	This is a fully qualified package name of one or several packages in the test application.
class	A fully qualified test case class to be executed by the test runner. Optionally, this could include the test method name separated from the class name by a hash (#).
coverage	true. Runs the EMMA code coverage and writes the output to a file that can also be specified. We will dig into the details about supporting EMMA code coverage for our tests in Chapter 9, Alternative Testing Tactics.

References

1. Paul Blundell, Diego Torres Milano, Learning Android Application Testing, PACKT Publisher, 2015.

Thank You.

Black-box testing techniques

Durga Prasad Mohapatra
Professor
Dept. of CSE
NIT Rourkela

Boundary Value Analysis

Boundary Value Analysis

- Some typical programming errors occur:
 - at boundaries of equivalence classes
 - might be purely due to psychological factors.
- Programmers often fail to see:
 - special processing required at the boundaries of equivalence classes.

Boundary Value Analysis

- Programmers may improperly use < instead of <=
- Boundary value analysis:
 - select test cases at the boundaries of different equivalence classes.

Example

- For a function that computes the square root of an integer in the range of 1 and 5000:
 - test cases must include the values {0,1,5000,5001} along with the values obtained from Equivalence partitioning.



BOUNDARY VALUE ANALYSIS (BVA)

- BVA offers several methods to design test cases. Following are the few methods used:
- **I. BOUNDARY VALUE CHECKING (BVC)**
- **2. ROBUSTNESS TESTING METHOD**
- **3. WORST-CASE TESTING METHOD**
- **4. ROBUST WORST-CASE TESTING METHOD**

BOUNDARY VALUE CHECKING (BVC)

- In this method, the test cases are designed by holding one variable at its extreme value and other variables at their nominal values in the input domain.
- The variable at its extreme value can be selected at:

BOUNDARY VALUE CHECKING (BVC)

- (a) Minimum value (Min)
- (b) Value just above the minimum value (Min+)
- (c) Maximum value (Max)
- (d) Value just below the maximum value (Max−)

BOUNDARY VALUE CHECKING (BVC)

- Let us take the example of two variables, A and B.
- If we consider all the above combinations with nominal values, then following test cases (see Fig. I) can be designed:
 - 1.Anom, Bmin
 - 2.Anom, Bmin+
 - 3.Anom, Bmax
 - 4.Anom, Bmax-
 - 5.Amin, Bnom
 - 6.Amin+, Bnom
 - 7.Amax, Bnom
 - 8.Amax-, Bnom
 - 9.Anom, Bnom

BOUNDARY VALUE CHECKING (BVC)

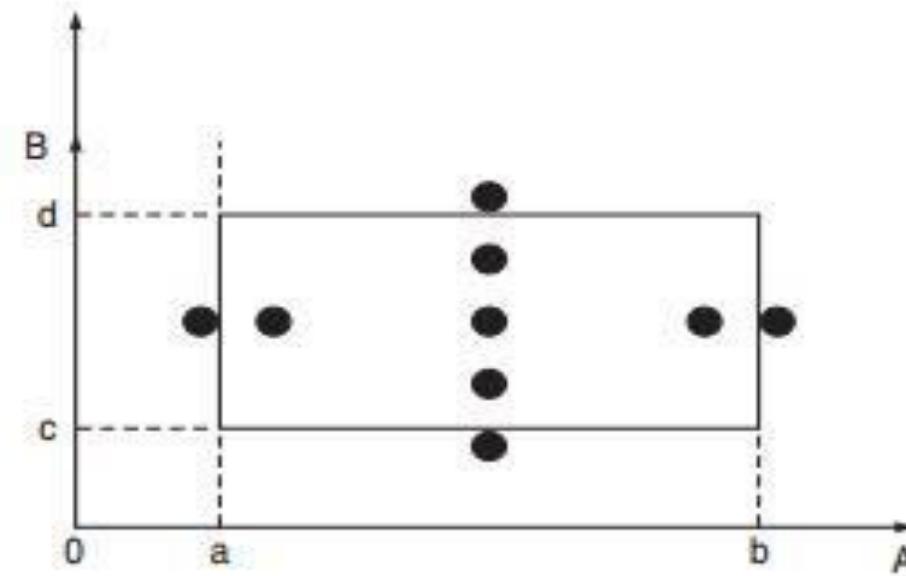


Fig 1: Boundary Value Checking

BOUNDARY VALUE CHECKING (BVC)

- It can be generalized that for n variables in a module, $4n + 1$ test cases can be designed with boundary value checking method.

ROBUSTNESS TESTING METHOD

- The idea of BVC can be extended such that boundary values are exceeded as: □
 1. A value just greater than the Maximum value (Max+)
 2. A value just less than Minimum value (Min-)

ROBUSTNESS TESTING METHOD

- When test cases are designed considering the above points in addition to BVC, it is called robustness testing.
- Let us take the previous example again. Add the following test cases to the list of 9 test cases designed in BVC:
 - I0.Amax+, Bnom I1.Amin-, Bnom
 - I2.Anom, Bmax+ I3.Anom, Bmin-

ROBUSTNESS TESTING METHOD

- It can be generalized that for n input variables in a module, $6n + 1$ test cases can be designed with robustness testing.

WORST-CASE TESTING METHOD

- We can again extend the concept of BVC by assuming more than one variable on the boundary.
- It is called worst-case testing method.
- Again, take the previous example of two variables, A and B. We can add the following test cases to the list of 9 test cases designed in BVC as:

WORST-CASE TESTING METHOD

- | | |
|----------------|-----------------|
| 10.Amin, Bmin | 11.Amin+, Bmin |
| 12.Amin, Bmin+ | 13.Amin+, Bmin+ |
| 14.Amax, Bmin | 15.Amax-, Bmin |
| 16.Amax, Bmin+ | 17.Amax-, Bmin+ |
| 18.Amin, Bmax | 19.Amin+, Bmax |
| 20.Amin, Bmax- | 21.Amin+,Bmax- |
| 22.Amax, Bmax | 23.Amax-, Bmax |
| 24.Amax, Bmax- | 25.Amax-,Bmax- |

WORST-CASE TESTING METHOD

- It can be generalized that for n input variables in a module, 5^n test cases can be designed with worst-case testing.

ROBUST WORST-CASE TESTING METHOD

- In the previous method, the extreme values of a variable considered are of BVC only.
- The worst case can be further extended if we consider robustness also, that is,
- in worst case testing if we consider the extreme values of the variables as in robustness testing method covered in Robustness Testing

ROBUST WORST-CASE TESTING METHOD

- Again take the example of two variables, A and B. We can add the following test cases to the list of 25 test cases designed in previous section.
- 26.Amin-, Bmin-
- 27.Amin-, Bmin
- 30.Amin+, Bmin-
- 28.Amin, Bmin-
- 29.Amin-, Bmin+
- 31.Amin-, Bmax

- 32. Amax, Bmin-
- 34. Amax-, Bmin-
- 36. Amax+, Bmin
- 38. Amax+, Bmin+
- 40. Amax+, Bmax
- 42. Amax+, Bmax-
- 44. Amax+, Bnom
- 46. Amin-, Bnom
- 48. Amax+, Bmin-
- 33. Amin-, Bmax-
- 35. Amax+, Bmax+
- 37. Amin, Bmin+
- 39. Amax+, Bmax+
- 41. Amax, Bmax+
- 43. Amax-, Bmax+
- 45. Anom, Bmax+
- 47. Anom, Bmin-
- 49. Amin-, Bmax+

Example

A program reads an integer number within the range [1,100] and determines whether it is a prime number or not. Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Test cases using BVC

- Since there is one variable, the total number of
- test cases will be $4n + 1 = 5$.
- In our example, the set of minimum and maximum values is shown below:

- Min value = 1
- Min+ value = 2
- Max value = 100
- Max– value = 99
- Nominal value = 50–55

- Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	53	Prime number

Test cases using robust testing

- Since there is one variable, the total number of test cases will be $6n + 1 = 7$. The set of boundary values is shown below:

- Min value = 1
- Min- value = 0
- Min+ value = 2
- Max value = 100
- Max- value = 99
- Max+ value = 101
- Nominal value = 50–55

- Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	0	Invalid input
2	1	Not a prime number
3	2	Prime number
4	100	Not a prime number
5	99	Not a prime number
6	101	Invalid input
7	53	Prime number

Test cases using worst-case testing

- Since there is one variable, the total number of test cases will be $5^n = 5$.
- Therefore, the number of test cases will be same as BVC.

Example

- A program computes a^b where a lies in the range $[1,10]$ and b within $[1,5]$.
- Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Test cases using BVC

- Since there are two variables, a and b, the total number of test cases will be $4n + 1 = 9$. The set of boundary values is shown below:

	a	b
Min value	1	1
Min+ value	2	2
Max value	10	5
Max- value	9	4
Nominal value	5	3

Using these values, test cases can be designed as shown below:

Test Case ID	a	b	Expected Output
1	1	3	1
2	2	3	8
3	10	3	1000
4	9	3	729
5	5	1	5
6	5	2	25
7	5	4	625
8	5	5	3125
9	5	3	125

Test cases using robust testing

- Since there are two variables, a and b, the total number of test cases will be $6n + 1 = 13$.
- The set of boundary values is shown below:

	a	b
Min value	1	1
Min- value	0	0
Min+ value	2	2
Max value	10	5
Max+ value	11	6
Max- value	9	4
Nominal value	5	3

Using these values, test cases can be designed as shown below:

Test Case ID	a	b	Expected output
1	0	3	Invalid input
2	1	3	1
3	2	3	8
4	10	3	1000
5	11	3	Invalid input
6	9	3	729
7	5	0	Invalid input
8	5	1	5
9	5	2	25
10	5	4	625
11	5	5	3125
12	5	6	Invalid input
13	5	3	125

Test cases using worst-case testing

- Since there are two variables, a and b, the total number of test cases will be $5^n = 25$.
- The set of boundary values is shown below:

	a	b
Min value	1	1
Min+ value	2	2
Max value	10	5
Max- value	9	4
Nominal value	5	3

There may be more than one variable at extreme values in this case.
 Therefore, test cases can be designed as shown below :

Test Case ID	a	b	Expected Output
1	1	1	1
2	1	2	1
3	1	3	3
4	1	4	1
5	1	5	1
6	2	1	2
7	2	2	4
8	2	3	8
9	2	4	16
10	2	5	32
11	5	1	5
12	5	2	25
13	5	3	125
14	5	4	625
15	5	5	3125
16	9	1	9
17	9	2	81
18	9	3	729
19	9	4	6561
20	9	5	59049
21	10	1	10
22	10	2	100
23	10	3	1000
24	10	4	10000
25	10	5	100000

Summary

- We discussed black-box test case design using:
 - **boundary value analysis**
- Explained BVA with some examples.

References

1. Rajib Mall, Fundamentals of Software Engineering, (Chapter – 10), Fifth Edition, PHI Learning Pvt. Ltd., 2018.
2. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter – 4), Second Edition, Oxford University Press, 2016.



Thank You



Cause-Effect Graphing

Dr. Durga Prasad Mohapatra

Professor

Department of CSE

NIT Rourkela

Cause-Effect Graphing

- Cause-effect graphing, also known as *dependency modeling*,
 - focuses on modelling dependency relationships amongst
 - program input conditions, known as *causes*, and
 - output conditions, known as *effects*.
- The relationship is expressed visually in terms of a cause-effect graph.
- The graph is a visual representation of a logical relationship amongst inputs and outputs that can be expressed as a Boolean expression.

Cause-Effect Graphing (Contd..)

- The graph allows selection of various combinations of input values as tests.
- The combinatorial explosion in the number of tests is avoided by using certain heuristics during test generation.

Cause-Effect Graphing (Contd..)

- A cause is any condition in the requirements that may effect the program output.
- An effect is the response of the program to some combination of input conditions.
 - For example, it may be
 - An error message displayed on the screen
 - A new window displayed
 - A database updated.

Cause-Effect Graphing (Contd..)

- An effect need not be an “output” visible to the user of the program.
- Instead, it could also be an internal *test point* in the program that can be probed during testing to check if some intermediate result is as expected.
 - For example, the intermediate test point could be at the entrance into a method to indicate that indeed the method has been invoked.

Example

- Consider the requirement “Dispense food only when the DF switch is ON”
 - Cause is “DF switch is ON”.
 - Effect is “Dispense food”.
- This requirement implies a relationship between the “DF switch is ON” and the effect “Dispense food”.
- Other requirements might require additional causes for the occurrence of the “Dispense food” effect.

Cause and Effect Graphs

- Testing would be a lot easier:
 - if we could automatically generate test cases from requirements.
- Work done at IBM:
 - Can requirements specifications be systematically used to design functional test cases?

Cause and Effect Graphs

- Examine the requirements:
 - restate them as logical relation between inputs and outputs.
 - The result is a Boolean graph representing the relationships
 - called a **cause-effect graph**.

Cause and Effect Graphs

- Convert the graph to a decision table:
 - each column of the decision table corresponds to a test case for functional testing.

Steps to create cause-effect graph

- Study the functional requirements.
- Mark and number all causes and effects.
- Numbered causes and effects:
 - become nodes of the graph.

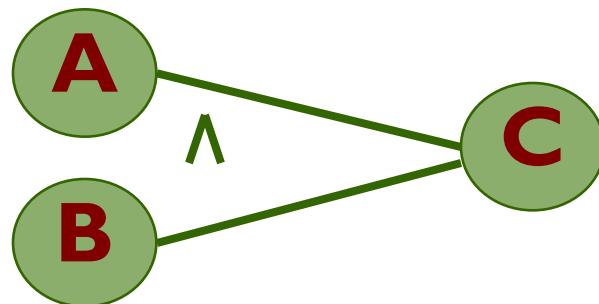
Steps to create cause-effect graph

- Draw causes on the LHS
- Draw effects on the RHS
- Draw logical relationship between causes and effects
 - as edges in the graph.
- Extra nodes can be added
 - to simplify the graph

Drawing Cause-Effect Graphs

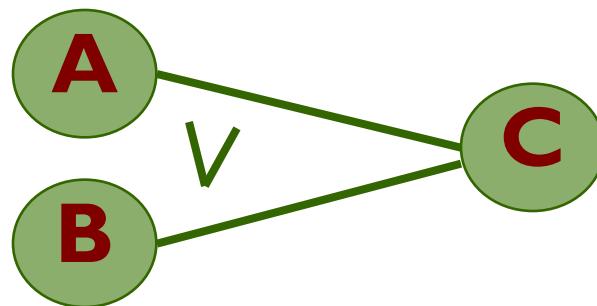


If A then B

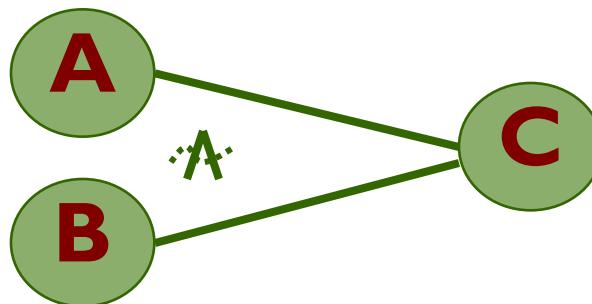


If (A and B)then C

Drawing Cause-Effect Graphs

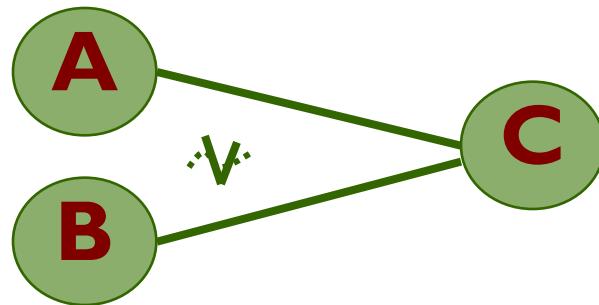


If (A or B)then C

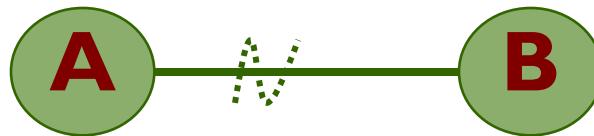


**If (not(A and B))then
C**

Drawing Cause-Effect Graphs



If $(\neg(A \vee B))$ then C



If $(\neg A)$ then B

Cause effect graph- Example

- A water level monitoring system
 - used by an agency involved in flood control.
 - **Input:** level(a,b)
 - a is the height of water in dam in meters
 - b is the rainfall in the last 24 hours in cms

Cause effect graph- Example

- Processing
 - The function calculates whether the level is safe, too high, or too low.
- Output
 - message on screen
 - level=safe
 - level=high
 - invalid syntax

Cause effect graph- Example

- We can separate the requirements into 5 causes:

- 1
 - first five letters of the command is “level”
- 2
 - command contains exactly two parameters
 - separated by comma and enclosed in parentheses

Cause effect graph- Example

- Parameters a and b are real numbers:
 - such that the water level is calculated to be low

3

4

- or safe.

- The parameters a and b are real numbers:
 - such that the water level is calculated to be high.

5

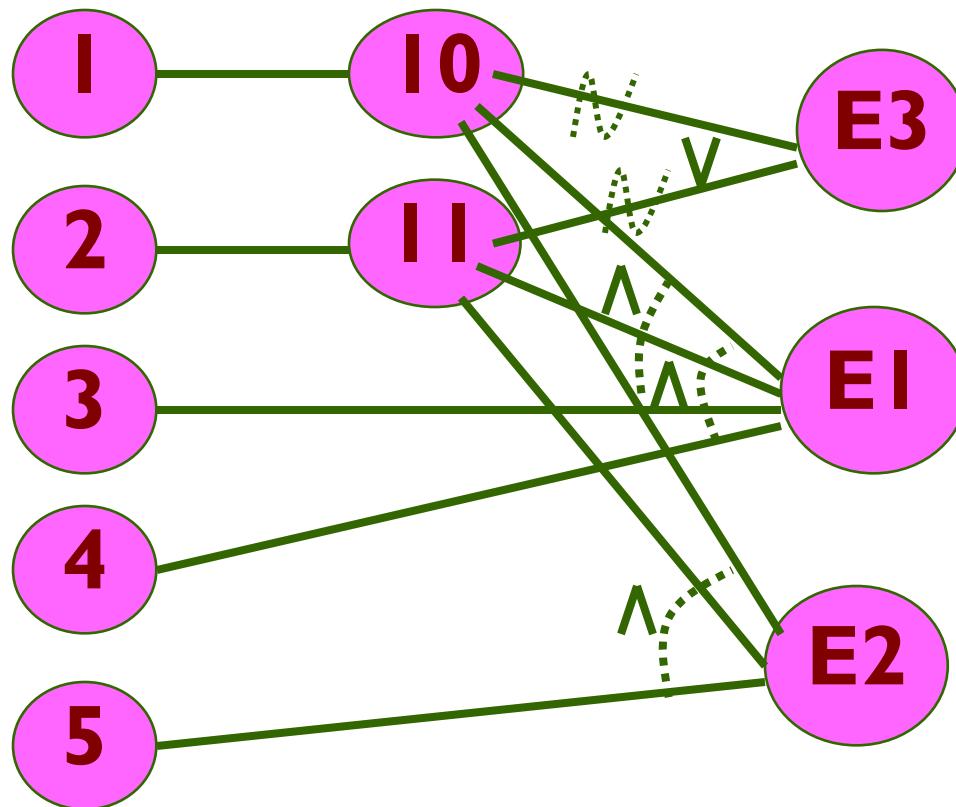
Cause effect graph- Example

- 10 ◦ Command is syntactically valid
- 11 ◦ Operands are syntactically valid.

Cause effect graph- Example

- Three effects
 - level = safe E1
 - level = high E2
 - invalid syntax E3

Cause effect graph- Example



Cause effect graph- Decision table

	Test 1	Test 2	Test 3	Test 4	Test 5	
Cause 1	I	I	I	S	I	
Cause 2	I	I	I	X	S	I = Invoked x = don't care
Cause 3	I	S	S	X	X	
Cause 4	S	I	S	X	X	s = suppressed
Cause 5	S	S	I	X	X	
Effect 1	P	P	A	A	A	P = present
Effect 2	A	A	P	A	A	A = absent
Effect 3	A	A	A	P	P	

Cause effect graph- Example

- Put a row in the decision table for each cause or effect:
 - in the example, there are five rows for causes and three for effects.

Cause effect graph- Example

- The columns of the decision table correspond to test cases.
- Define the columns by examining each effect:
 - list each combination of causes that can lead to that effect.

Cause effect graph- Example

- We can determine the number of columns of the decision table
 - by examining the lines flowing into the effect nodes of the graph.

Cause effect graph- Example

- Theoretically we could have generated $2^5=32$ test cases.
 - Using cause effect graphing technique reduces that number to 5.

Cause effect graph

- Not practical for systems which:
 - include timing aspects
 - feedback from processes is used for some other processes.

Procedure used for the generation of tests

- Identify causes and effects by reading the requirements. Each cause and effect is assigned a unique identifier. Note that an effect can also be a cause for some other effect.
- Express the relationship between causes and effects using a cause-effect graph.
- Transform the cause-effect graph into a limited entry decision table, hereafter referred to as decision table.
- Generate tests from the decision table.

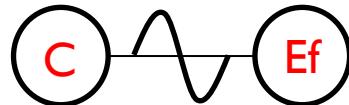
Basic elements of a cause-effect graph

- implication
- not (\sim)
- and (\wedge)
- or (\vee)

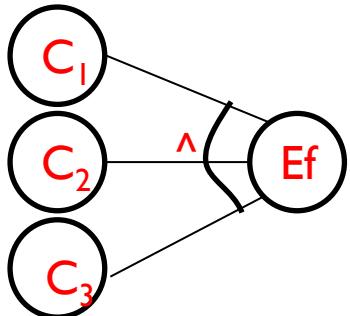
C implies Ef



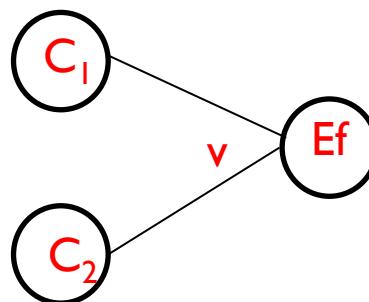
not C implies Ef



Ef when C_1 and C_2 and C_3



Ef when C_1 or C_2



- C, C_1, C_2, C_3 denote causes.
- Ef denotes an effect.

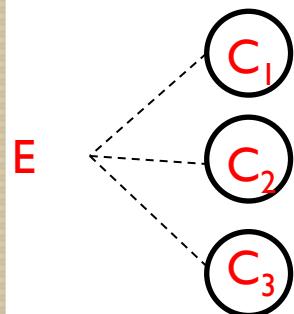
Semantics of basic elements

- C implies Ef : $\text{if}(C) \text{ then } Ef;$
- not C implies Ef : $\text{if}(\neg C) \text{ then } Ef;$
- Ef when C_1 and C_2 and C_3 : $\text{if}(C_1 \& \& C_2 \& \& C_3) \text{ then } Ef;$
- Ef when C_1 or C_2 : $\text{if}(C_1 \mid\mid C_2) \text{ then } Ef;$

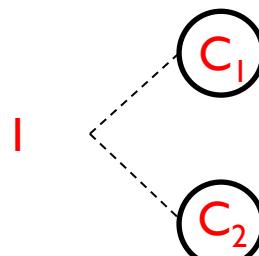
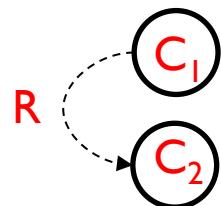
Constraints amongst causes (E,I,O,R)

- Constraints show the relationship between the causes.
- Exclusive (E)
- Inclusive (I)
- Requires (R)
- One and only one (O)

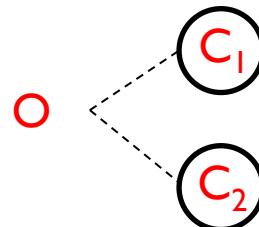
Exclusive: either C_1 or C_2 or C_3 Inclusive: at least C_1 or C_2



C_1 requires C_2



One and only one, of C_1 and C_2



Constraints amongst causes (E,I,O,R)

- Exclusive (E) constraint between three causes C_1 , C_2 and C_3 implies that exactly one of C_1 , C_2 , C_3 can be true.
- Inclusive (I) constraint between two causes C_1 and C_2 implies that at least one of the two must be present.
- Requires (R) constraint between C_1 and C_2 implies that C_1 requires C_2 .
- One and only one (O) constraint models the condition that one, and only one, of C_1 and C_2 must hold.

Possible values of causes constrained by E, I, R, O

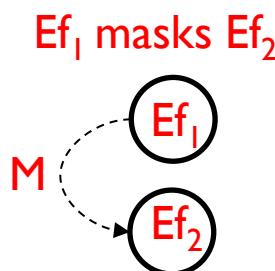
- A 0 or 1 under a cause implies that the corresponding condition is, respectively, false and true.
- The arity of all constraints, except R, is greater than or equal to 2, i.e., all except the R constraint can be applied to two or more causes; the R constraint is applied to two causes.
- A condition that is false (true) is said to be in the “0-state” (1 state).
- Similarly, an effect can be “present” (1 state) or “absent” (0 state).

Possible values of causes constrained by E, I, R, O

Constraint	Arity	Possible values		
		C1	C2	C3
$E(C_1, C_2, C_3)$	$n \geq 2$	0	0	0
			0	0
		0		0
		0	0	
$I(C_1, C_2)$	$n \geq 2$		0	-
		0		-
				-
$R(C_1, C_2)$	$n=2$			-
		0	0	-
		0		-
$O(C_1, C_2, C_3)$	$n \geq 2$		0	0
		0		0
		0	0	

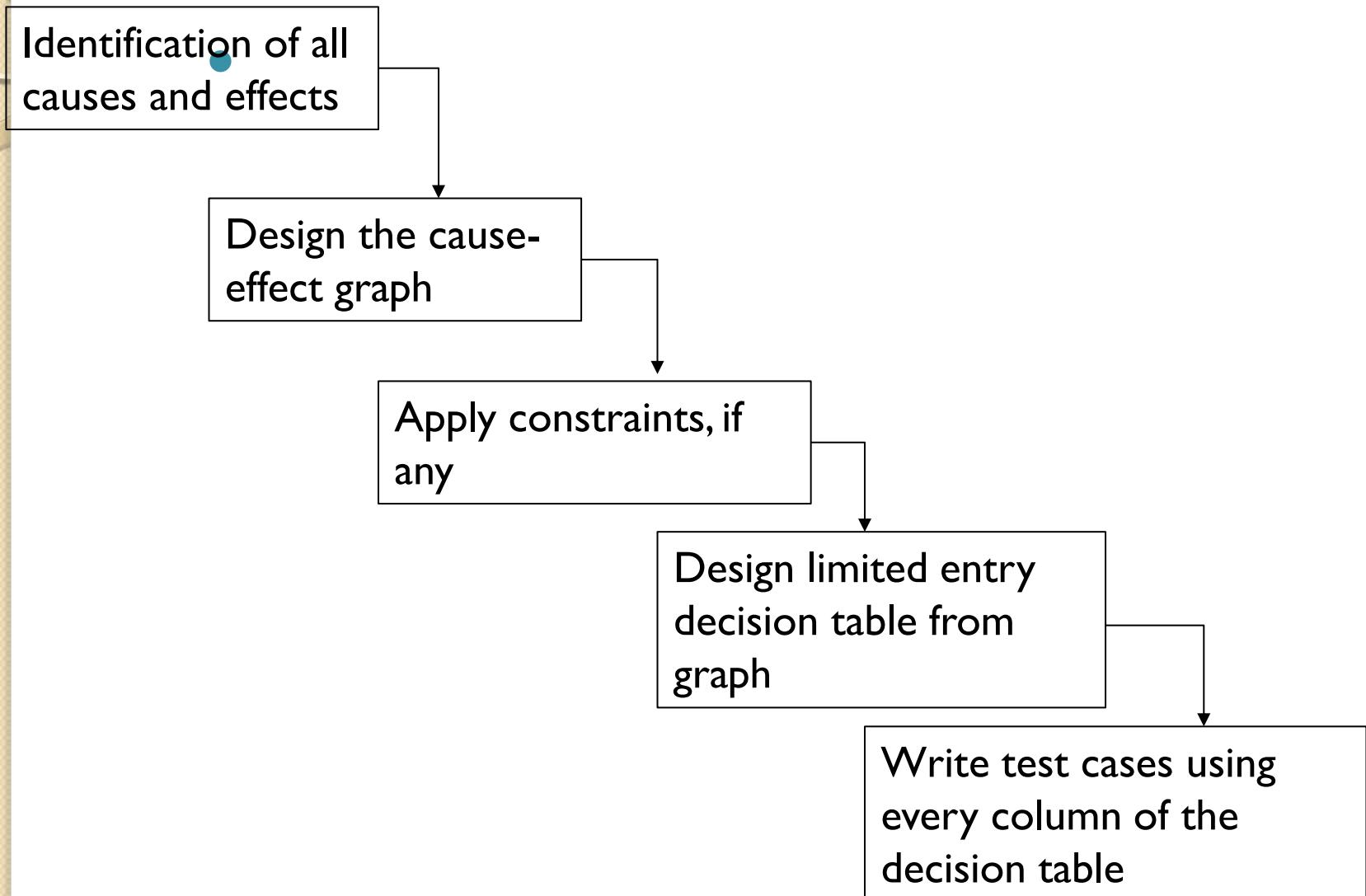
Constraint amongst effects

- Masking (M)



- Masking (M) constraint between two effects Ef_1 and Ef_2 implies that if Ef_1 is present, then Ef_2 is forced to be absent.

Steps for generating test cases using Cause-Effect Graph



Creating Cause-Effect Graph

- The process of creating a cause-effect graph consists of two major steps.
- The causes and effects are identified by a careful examination of the requirements.
 - This process also exposes the relationships amongst various causes and effects as well as constraints amongst the causes and effects.
 - Each cause and effect is assigned a unique identifier for ease of reference in the cause-effect graph.

Creating Cause-Effect Graph

- The cause-effect graph is constructed to
 - express the relationships extracted from the requirements.
- When the number of causes and effects is large, say over 100 causes and 45 effects,
 - it is appropriate to use an incremental approach.

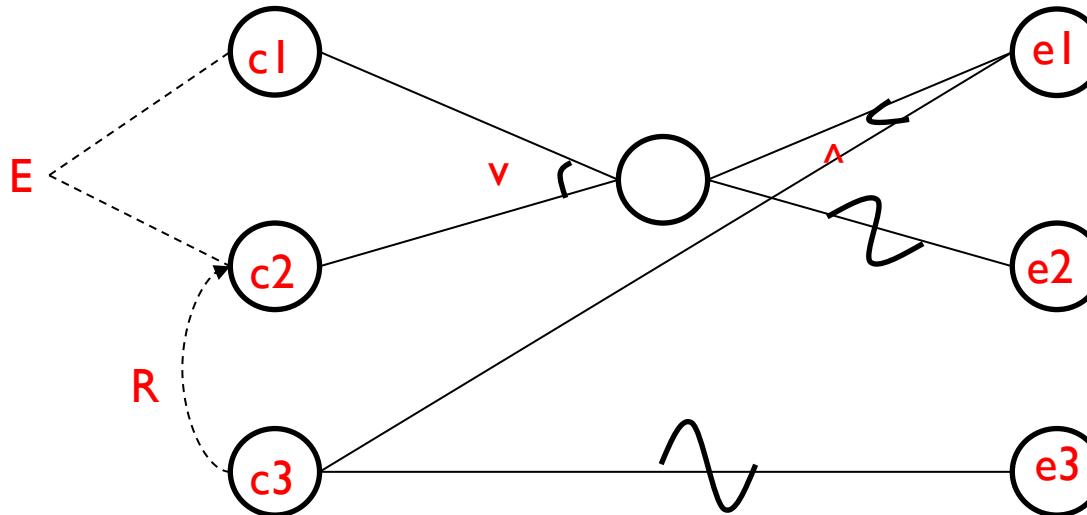
Another example

- Consider the example of keeping the record of marital status and number of children of a citizen.
- The value of marital status must be 'U' or 'M'.
- The value of the number of children must be digit or null in case a citizen is unmarried.
- If the information entered by the user is correct then an update is made.
- If the value of marital status of the citizen is incorrect, then the error message 1 is issued.
- Similarly, if the value of the number of children is incorrect, then the error message 2 is issued.

Answer

- Causes are
 - c1: marital status is U
 - c2: marital status is M
 - c3: number of children is a digit
- Effects are
 - e1: updation made
 - e2: error message 1 is issued
 - e3: error message 2 is issued

Answer



- There are two constraints
 - Exclusive (between $c1$ and $c2$) and
 - Requires (between $c3$ and $c2$)
- Causes $c1$ and $c2$ cannot occur simultaneously.
- For cause $c3$ to be true, cause $c2$ has to be true.

Decision Table from cause-effect graph

- Each column of the decision table represents a combination of input values, and hence a test.
- There is one row for each condition and effect.
- Thus the table decision table can be viewed as an $N \times M$ matrix with
 - N being the sum of the number of conditions and effects and
 - M the number of tests.
- Each entry in the decision table is a 0 or 1
 - depending on whether or not the corresponding condition is false or true, respectively.
- For a row corresponding to an effect, an entry is 0 or 1
 - if the effect is not present or present, respectively.

Test generation from a decision table

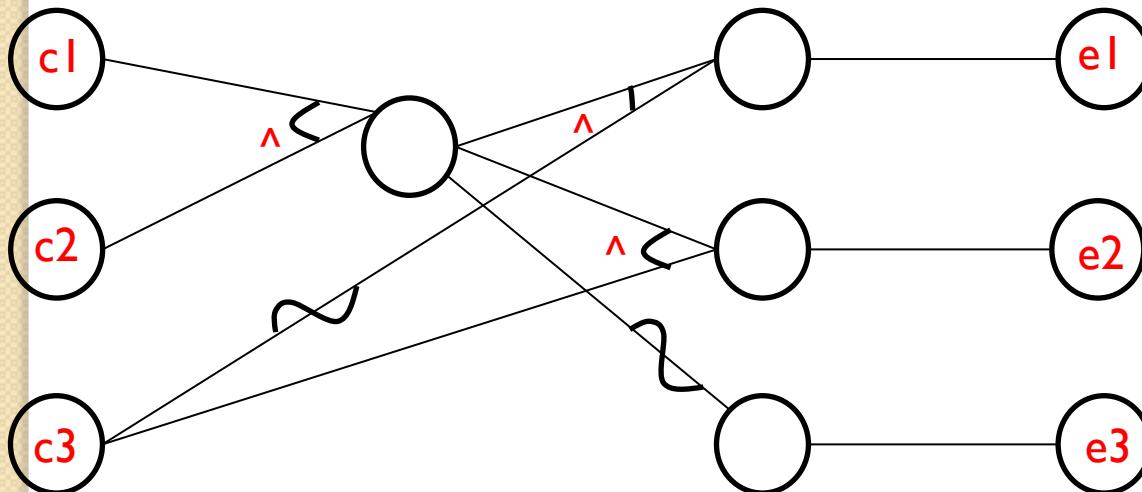
- Test generation from a decision table is relatively forward.
- Each column in the decision table generates at least one test input.
- Note that each combination might be able to generate more than one test when a condition in the cause-effect graph can be satisfied in more than one way.
- For example, consider the following cause:
 $C: x < 99$
- The condition above can be satisfied by many values such as $x=1$ and $x=49$.
- Also, C can be made false by many values of x such as $x=100$ and $x=999$.
- Thus, one might have a choice of values of input variables while generating tests using columns from a decision table

Example

- A tourist of age greater than 21 years and having a clean driving record is supplied a rental car.
- A premium amount is also charged if the tourist is on business,
- Otherwise, it is not charged.
- If the tourist is less than 21 year old, or does not have a clean driving record,
 - The system will display the following message: “Car cannot be supplied”.

Answer

- Causes are
 - c1: Age is over 21
 - c2: Driving record is clean
 - c3: Tourist is on business
- Effects are
 - e1: Supply a rental car without premium charge
 - e2: Supply a rental car with premium charge
 - e3: Car cannot be supplied



Decision Table and Test Cases

	1	2	3	4
c1: Over 21?	F	T	T	T
c2: Driving record clean?	-	F	T	T
c3: On business?	-	-	F	T
e1: Supply a rental car without premium charge			X	
e2: Supply a rental car with premium charge				X
e3: Car cannot be supplied	X	X		

Test Case	Age	Driving_record_clean	On_business	Expected Output
1	20	Yes	Yes	Car cannot be supplied
2	26	No	Yes	Car cannot be supplied
3	62	Yes	No	Supply a rental car without premium charge
4	62	Yes	Yes	Supply a rental car with premium charge

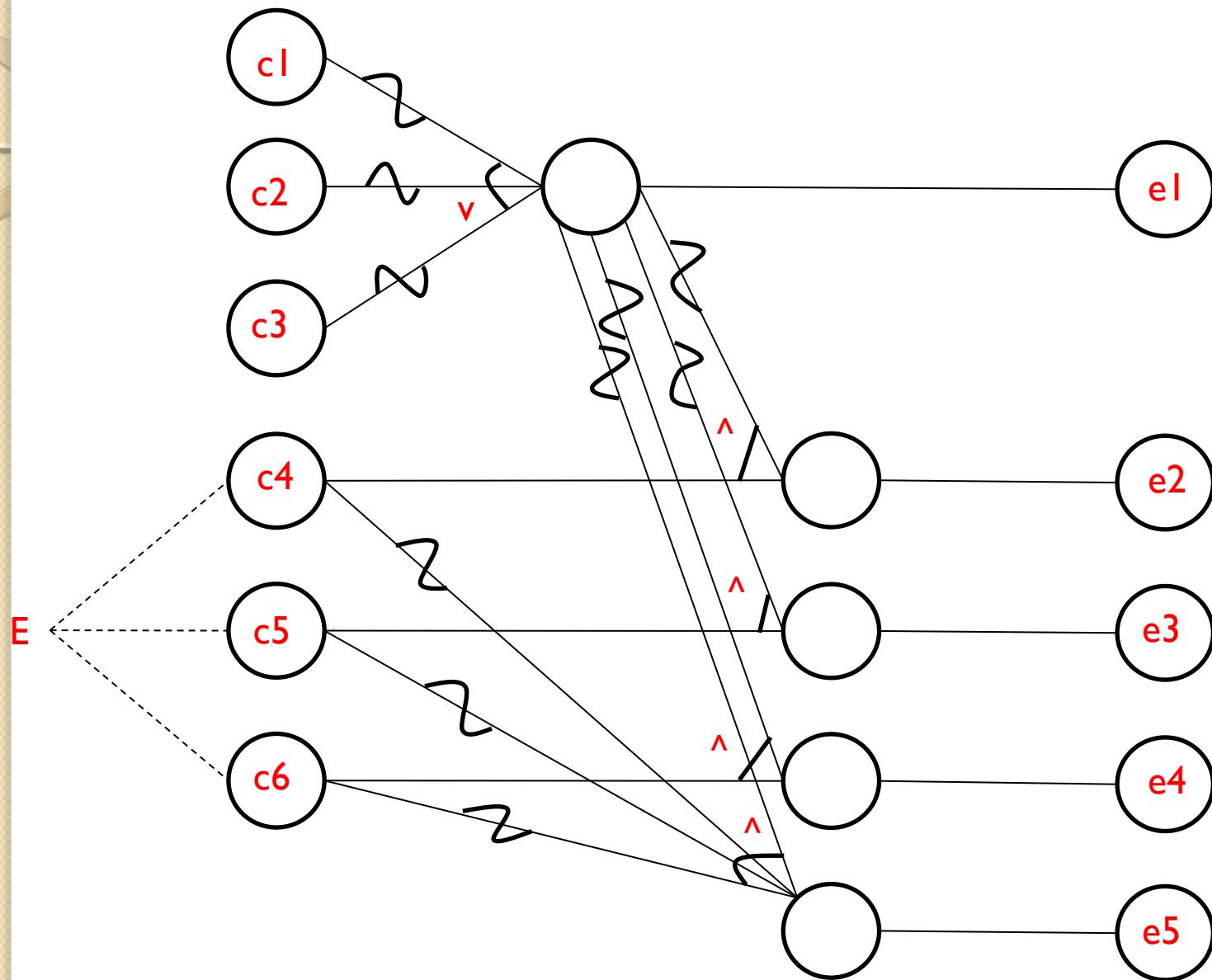
Example 2: Triangle Classification Problem

- Consider a program for classification of a triangle.
- Its input is a triple of positive integers (say a, b and c) and the input values are greater than zero and less than or equal to 100.
- The triangle is classified according to the following rules:
 - Right angled triangle: $c^2=a^2+b^2$ or $a^2=b^2+c^2$ or $b^2=c^2+a^2$
 - Obtuse angled triangle: $c^2>a^2+b^2$ or $a^2>b^2+c^2$ or $b^2>c^2+a^2$
 - Acute angled triangle: $c^2< a^2+b^2$ or $a^2< b^2+c^2$ or $b^2< c^2+a^2$
 - The program output may have one of the following words: [Acute angled triangle, Obtuse angled triangle, Right angled triangle, Invalid triangle, Input values are out of range]

Answer

- Causes are:
 - c1: side “a” is less than the sum of sides “b” and “c”.
 - c2: side “b” is less than the sum of sides “a” and “c”.
 - c3: side “c” is less than the sum of sides “a” and “b”.
 - c4: square of side “a” is equal to the sum of squares of sides “b” and “c”.
 - c5: square of side “a” is greater than the sum of squares of sides “b” and “c”.
 - c6: square of side “a” is less than the the sum of squares of sides “b” and “c”.
- Effects are:
 - e1: Invalid triangle
 - e2: Right angle triangle
 - e3: Obtuse angled triangle
 - e4: Acute angled triangle
 - e5: Impossible stage

Cause-Effect Graph



Decision Table

	1	2	3	4	5	6	7	8	9	10	11
c1: $a < b+c$	F	T	T	T	T	T	T	T	T	T	T
c2: $b < a+c$	-	F	T	T	T	T	T	T	T	T	T
c3: $c < a+b$	-	-	F	T	T	T	T	T	T	T	T
c4: $a^2 = b^2 + c^2$	-	-	-	T	T	T	T	F	F	F	F
c5: $a^2 > b^2 + c^2$	-	-	-	T	T	F	F	T	T	F	F
c6: $a^2 < b^2 + c^2$	-	-	-	T	F	T	F	T	F	T	F
e1: Invalid triangle	X	X	X								
e2: Right angle triangle							X				
e3: Obtuse angled triangle								X			
e4: Acute angled triangle									X		
e5: Impossible					X	X	X	X			X



Thank You

Concolic Testing: A modern software testing technique

**Dr. Durga Prasad Mohapatra
Professor**

Department of CSE, NIT Rourkela



Seminar Outline

1 [Introduction](#)

2 [Fundamental Ideas](#)

3 [Survey of Related works](#)



Introduction

Software Testing is an important phase in SDLC.

- Helps to achieve software dependability and improve quality.
 - Time consuming: Approximately 40% of software development time is devoted to testing.
-
- Testing can be done in two ways – Manual vs. Automated.
 - **An automation tool** for test case generation can effectively reduce the time required in testing.
 - **Automation tool** can be designed to generate test cases for different test coverage criteria.



Reliability and Coverage Model -By Prof. Aditya Mathur

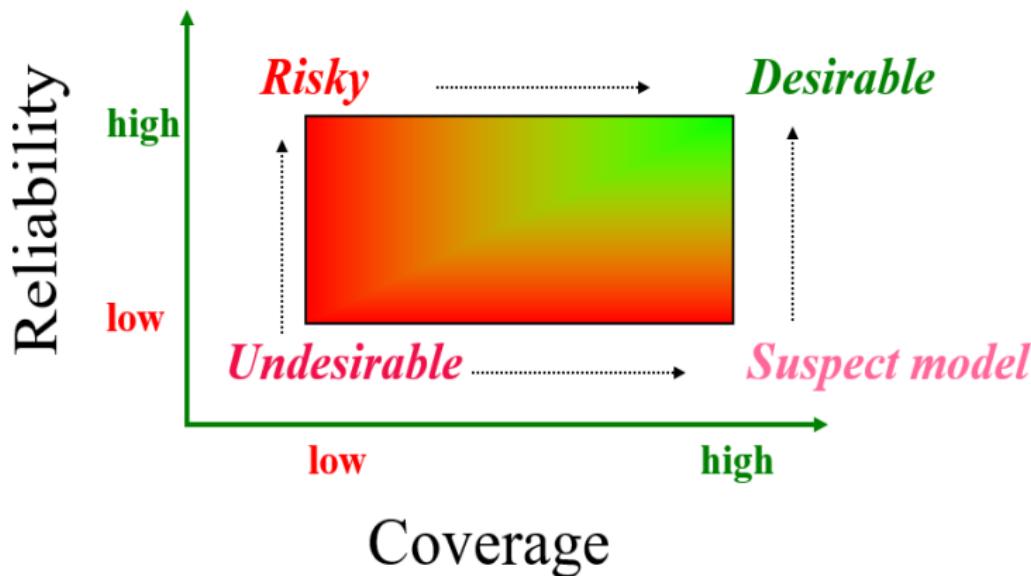


Figure 1: Reliability and Coverage



Software Testing

“No product of human intellect comes out right the first time. We rewrite sentences, rip out knitting stitches, replant gardens, remodel houses, and repair bridges. Why should software be any different?”

- By Wiener, Ruth: Digital Woes, Why We Should Not Depend on Software.



Software Testing

- The purpose of the verification process is to detect and report errors that have been introduced in the development process.
- The verification process must ensure that the produced software implements intended function completely and correctly, while avoiding unintended function.
- Verification is an integral process, which is coupled with every development step. Testing quality at the end of the life cycle is impractical.



Software Testing: Coverage

- Coverage refers to the extent to which a given verification activity has satisfied its objectives: in essence, providing an exit criteria for when to stop. That is, what is “enough” is defined in terms of coverage.
- Coverage is a measure, not a method or a test. As a measure, coverage is usually expressed as the percentage of an activity that is accomplished.
- **Our goal, then, should be to provide enough testing to ensure that the probability of failure due to hibernating bugs is low enough to accept. “Enough” implies judgment.**



RTCA/DO178-B/C standards

- RTCA stands for Radio Technical Commission for Aeronautics.
- DO-178B (and DO-278) are used to assure safety of avionics software.
- DO-178C includes the coverage analysis of Object-Oriented Programs used in safety of avionics software.
- These documents provide guidance in the areas of SW development, configuration management, verification and the interface to approval authorities (e.g., FAA, EASA).



Levels of Software

- Different failure conditions require different software conditions → 5 levels

Failure Condition	Software Level
Catastrophic	Level A
Hazardous/Severe - Major	Level B
Major	Level C
Minor	Level D
No Effect	Level E

Figure 2: Levels of Software



Relation of Coverages

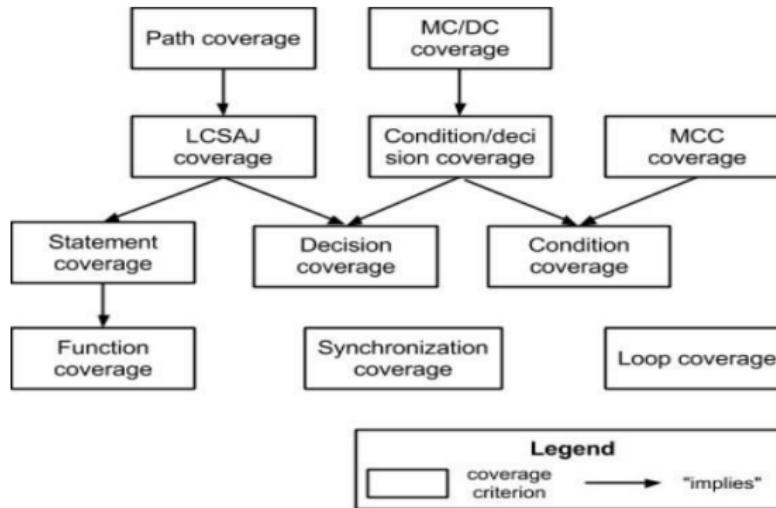


Figure 3: Relation of Coverages



LCSAJ Coverage

- **LCSAJ** stands for Linear Code Sequence and Jump.
- It is a white box **testing** technique to identify the **code coverage**.
- It begins at the start of the program or branch and ends at the end of the program or the branch.
- **LCSAJ** is ordinarily equivalent to **statement coverage**.



Types of Structural Coverage

Coverage Criteria	Statement Coverage	Decision Coverage	Condition Coverage	Condition/Decision Coverage	MC/DC	Multiple Condition Coverage
Every point of entry and exit in the program has been invoked at least once		•	•	•	•	•
Every statement in the program has been invoked at least once	•					
Every decision in the program has taken all possible outcomes at least once		•		•	•	•
Every condition in a decision in the program has taken all possible outcomes at least once			•	•	•	•
Every condition in a decision has been shown to independently affect that decision's outcome					•	• ^b
Every combination of condition outcomes within a decision has been invoked at least once						•

weakest → strongest



Figure 4: Types

Definitions

Predicate / Boolean Expression

A predicate is an expression that evaluates to a boolean value, and which is required for our approach.

A simple example is: $((a > b) \vee C) \wedge p(x)$. Predicates may contain boolean variables, non-boolean variables that are compared with relational operators, and calls to function that return a boolean value, all three of which may be joined with logical operators.



Definitions

Predicate Coverage

For each $p \in P$, Test Requirement (TR) for predicate coverage contains two requirements: p evaluates to true, and p evaluates to false.



Definitions

Clause / Atomic Condition

A clause is a predicate that does not contain any of the logical operators.

The predicate $((a > b) \vee C) \wedge p(x)$ contains three clauses; a relational expression $(a > b)$, a boolean variable C and a boolean function call $p(x)$.



Definitions

Clause Coverage

For each $c \in C$, TR for clause coverage contains two requirements: c evaluates to true, and c evaluates to false.



Definitions

Statement Coverage

- The statement coverage based strategy aims to design the test cases so as to execute every statement in a program at least once.
- The principle idea governing the statement coverage strategy is that unless a statement is executed, there is no way to determine whether an error exists in that statement.



Definitions

Branch Coverage

Each decision should take all possible outcomes at least once either true or false.

For example if($m > n$), the test cases are (1) $m \leq n$, (2) $m > n$



Example

```
1 int computeGCD(x,y)
2     int x,y;
3     {
4         while (x!=y){
5             if(x>y) then
6                 x=x-y;
7             else
8                 y=y-x;
9         }
10    return x;
11 }
```

Figure 5: Euclid's GCD computation program



Test cases for Statement Coverage

- To design the test cases for the statement coverage, the conditional expression of the while statements needs to be made true and the conditional expression of the if statement needs to be made both true and false.
- By choosing the test set $\{(x=3,y=3), (x=4,y=3), (x=3,y=4)\}$, all the statements of the program would be executed at-least once.



Test cases for Statement Coverage

- For the GCD program, the test cases for branch coverage can be $\{(x=3,y=3), (x=3,y=2), (x=4,y=3),(x=3,y=4)\}$.



Observation

- It is easy to show that branch coverage based testing is a stronger testing than statement coverage-based testing.
- We can prove this by showing that branch coverage ensures statement coverage, but not vice-versa.



Definitions

Concolic Testing

- The concept of CONCOLIC testing combines the CONConcrete constraints execution and symbOLIC constraints execution to automatically generate test cases for full path coverage.
- This testing generates test suites by executing the program with random values.
- At execution time both concrete and symbolic values are saved for execution path.
- During execution, the variables are stored in some symbolic values such as x_0 , and y_0 , instead of x and y.
- The next iteration of the process forces the selection of different paths.



Definitions

Concolic Testing

- The tester selects a value from the path constraints and negates the values to create a new path value. Then the tester finds concrete constraints to satisfy the new path values.
- The selection of values is responsible by the Constraint Solver which is a part of Concolic tester.
- These constraints are inputs for all next executions. This concolic testing is performed iteratively until exceeds the threshold value or sufficient code coverage is obtained.



Concolic Testing Example

Concolic Testing

Example

```
1: x = input();
2: y = input();
3: if (x > y) {
4:     assert(x != 100);
5: }
```



Concolic Testing Example

Concolic Testing

Example

```
1: x = input();
2: y = input();
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Reads input values

We want to violate this



Concolic Testing Example

Concolic Testing

Example

```
1: x = input();
2: y = input();
3: if (x > y) {
4:     assert(x != 100);
5: }
```



Concolic Testing Example

Concolic Testing

Example

```
1: x = input();      ← 5
2: y = input();      ← 10
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Assign Random Values



Concolic Testing Example

Concolic Testing

Example

```
1: x = input(); ← 5
2: y = input(); ← 10
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x		
y		

Execution Tree



Concolic Testing Example

Concolic Testing

Example

```
1: x = input();      ← 5
2: y = input();      ← 10
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	5	i_0
y		

Execution Tree



Concolic Testing Example

Concolic Testing

Example

```
1: x = input();      ← 5
2: y = input();      ← 10
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	5	i_0
y	10	i_1

Execution Tree



Concolic Testing Example

Concolic Testing

Example

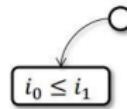
```
1: x = input();      ← 5
2: y = input();      ← 10
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	5	i_0
y	10	i_1

Execution Tree



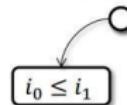
Concolic Testing Example

Concolic Testing

Example

```
1: x = input();
2: y = input();
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Execution Tree



Concolic Testing Example

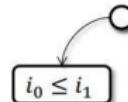
Concolic Testing

Example

```
1: x = input();
2: y = input();
3: if (x > y) {
4:     assert(x != 100);
5: }
```



Execution Tree



Concolic Testing Example

Concolic Testing

Example

```
1: x = input();
2: y = input();
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Negate Constraint:
Generate i_0 and i_1 s.t. $i_0 > i_1$

Execution Tree

$$i_0 \leq i_1$$



Concolic Testing Example

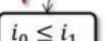
Concolic Testing

Example

```
1: x = input(); ← 20
2: y = input(); ← 3
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Negate Constraint:
Generate i_0 and i_1 s.t. $i_0 > i_1$

Execution Tree



Concolic Testing Example

Concolic Testing

Example

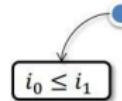
```
1: x = input(); ← 20
2: y = input(); ← 3
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x		
y		

Execution Tree



Concolic Testing Example

Concolic Testing

Example

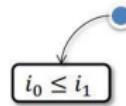
```
1: x = input(); ← 20
2: y = input(); ← 3
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	20	i_0
y		

Execution Tree



Concolic Testing Example

Concolic Testing

Example

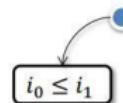
```
1: x = input();      ← 20
2: y = input();      ← 3
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	20	i_0
y	3	i_1

Execution Tree



Concolic Testing Example

Concolic Testing

Example

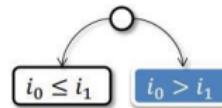
```
1: x = input();      ← 20
2: y = input();      ← 3
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	20	i_0
y	3	i_1

Execution Tree



Concolic Testing Example

Concolic Testing

Example

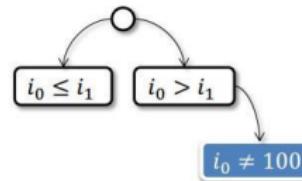
```
1: x = input(); ← 20
2: y = input(); ← 3
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	20	i_0
y	3	i_1

Execution Tree



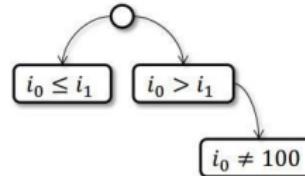
Concolic Testing Example

Concolic Testing

Example

```
1: x = input();
2: y = input();
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Execution Tree



Concolic Testing Example

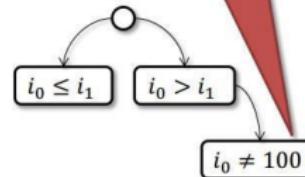
Concolic Testing

Example

```
1: x = input();  
2: y = input();  
3: if (x > y) {  
4:     assert(x != 100);  
5: }
```

Negate Constraint:
Generate i_0 and i_1 s.t. $i_0 > i_1$
and $i_0 = 100$

Execution Tree



Concolic Testing Example

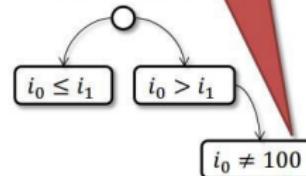
Concolic Testing

Example

```
1: x = input(); ← 100
2: y = input(); ← 4
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Negate Constraint:
Generate i_0 and i_1 s.t. $i_0 > i_1$
and $i_0 = 100$

Execution Tree



Concolic Testing Example

Concolic Testing

Example

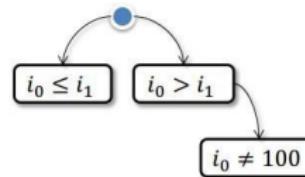
```
1: x = input(); ← 100
2: y = input(); ← 4
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x		
y		

Execution Tree



Concolic Testing Example

Concolic Testing

Example

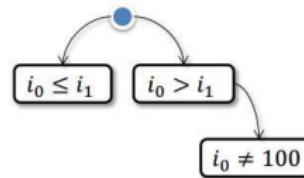
```
1: x = input(); ← 100
2: y = input(); ← 4
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	100	i_0
y		

Execution Tree



Concolic Testing Example

Concolic Testing

Example

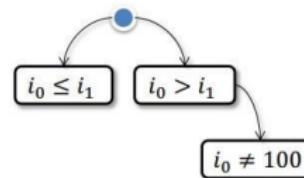
```
1: x = input(); ← 100
2: y = input(); ← 4
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	100	i_0
y	4	i_1

Execution Tree



Concolic Testing Example

Concolic Testing

Example

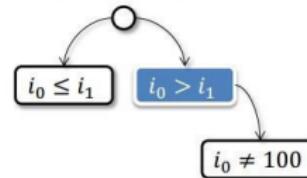
```
1: x = input(); ← 100
2: y = input(); ← 4
3: if (x > y) {
4:     assert(x != 100);
5: }
```

Concolic Execution

Memory

	Concrete	Symbolic
x	100	i_0
y	4	i_1

Execution Tree



Concolic Testing Example

Concolic Testing

Example

```
1: x = input(); ← 100
2: y = input(); ← 4
3: if (x > y) {
4:     assert(x != 100);
5: }
```

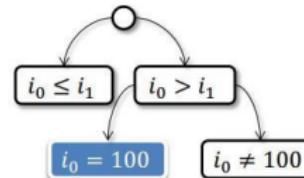


Concolic Execution

Memory

	Concrete	Symbolic
x	100	i_0
y	4	i_1

Execution Tree



Definitions

Concolic Testing: Final report

- Concolic testing technique explored total 3 paths for the example program.
- It has generated total three test cases for the variables x and y. These are shown in Table 1.

Table 1: Test cases

Test cases	TC1	TC2	TC3
x	5	20	100
y	10	3	4



Definitions

Distributed Concolic Testing

- *Distributed Concolic Testing (DCT)* is an extension of the original concolic testing approach that uses several computing nodes in a distributed manner.
- It significantly reduces the time to generate test cases with better efficiency.
- Concolic testing is the combination of concrete and symbolic testing. In addition, distributed concolic testing is scalable and achieves a linear speedup by using a large number of computing nodes for test case generation.



Concolic Testers

In Table 1, we have compared concolic testing tools with respect to the following parameters/features: 1) variables types; 2) pointers; 3) native calls; 4) non-linear arithmetic operations; 5) bitwise operations; 6) array offsets and 7) function pointers.

The abbreviation used in Table 1 are the following: “Y

- “means the tool supports the feature. “N”means
- the tool does not support the feature.
- “P”means the tool can partially support the feature.
- “NA”means unknown.



Concolic Testers

Table 2: Summary of concolic tester with their properties.

Tool Name	Supporting Language	Supporting Platform	Support ConstraintsSolver	Support for float/double	Support for pointer	Support for nativecall	Support for nonlinear arithmetic op.	Support for binseep.	Support for offset	Support for functionpointer
DART	C	NA	LPSOLVER	N	N	N	NA	NA	N	N
SMART	C	LINUX	LPSOLVER	N	N	N	NA	NA	N	N
CUTE	C	LINUX	LPSOLVER	N	Y	N	NA	NA	N	N
jCUTE	JAVA	LINUXWINDOWS	NA	N	-	N	NA	NA	N	N
CREST	C	LINUX	YICES	N	N	N	P	P	N	N
EIE	C	LINUX	SIP	N	Y	N	Y	Y	Y	N
KLEE	C	LINUX	SIP	N	Y	P	Y	Y	Y	NA
RASET	C	LINUX	SIP	N	Y	N	Y	Y	Y	NA
FUZZ	JAVA	LINUX	BULTONFF	N	NA	N	N	N	NA	NA
PATHCRAWLER	C	NA	NA	NA	NA	N	NA	NA	NA	NA
PEX	.NET	WINDOWS	Z3	N	NA	N	NA	NA	NA	NA
SAGE	MACHINECODE	WINDOWS	DECOLVER	NA	N	Y	NA	NA	NA	NA
APOLLO	PHP	WINDOWS	CHOCO	NA	NA	N	NA	N	NA	NA
SCORE	C	LINUX	Z3SMT Solver	Y	N	N	Y	N	NA	NA



Comparison of related works

Table 3: Summary of different work on concolic testing.

S.No	Authors	Testing Type	FrameWork Type	Input Type	Output Type
1	Das et al. [16]	Concolic Testing, MC/DC	BCT, CREST, CA	C-Program	MC/DC %
2	Bokil et al. [24]	SC, DC, BC, MC/DC	AutoGen	C-Program	Test data, Time
3	Kim et al. [20]	HCT	SMT Solver, CREST	Flash storage Platform Software	Reduction Ratio
4	Majumdar et al. [17]	HCT, BC	CUTE	Editor in C-Language	Test Cases
5	Burnim et al. [21]	Heuristics Concolic Testing, BC	CREST	Software Application in C	Branch Covered
6	Kim et al. [23]	Concolic Testing	CREST	Embedded C Application	Branch Covered
7	Kim et al. [22]	Concolic Testing	CONBOL	Embedded Software	BC%, Time
8	Kim et al. [19, 25]	Distributed Concolic Testing	SCORE	Embedded C Program	BC%, Effectiveness
9	Sen et al. [26]	Concolic Testing, BC	CUTE, JCUTE	C and Java Programs	Test Cases, BC%, Time



Characteristics of different approaches

Table 4: Characteristics of different approaches on concolic testing.

SNo	Authors	Generated Test Cases	Measuring Coverage%	Determined Time Constraints	Computed Speed
1	Dasetal. [16]	C	C	X	X
2	Bokletal. [24]	C	X	C	X
3	Kmetetal. [20]	C	X	X	X
4	Majumdar etal. [17]	C	X	X	X
5	Burimetal. [21]	C	X	X	X
6	Kmetetal. [23]	C	X	X	X
7	Kmetetal. [22]	C	C	C	X
8	Kmetetal. [19,25]	C	C	X	C
9	Sasetal. [26]	C	C	C	X



References >> I

- 1 Palacios, M., Garc'ia-Fanjul, J., Tuya, J. and Spanoudakis, G., 2015. Coverage-based testing for service level agreements. *IEEE Transactions on Services Computing*, 8(2), pages 299-313.
- 2 Fraser, G. and Arcuri, A., 2014. A large-scale evaluation of automated unit test generation using EvoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2), page 8.
- 3 Jones, JA. and Harrold, MJ. 2013. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*. 29(3), pages 195-209.
- 4 Baluda, M., Braione, P., Denaro, G. and Pezz'e, M., 2011. Enhancing structural software coverage by incrementally computing branch executability. *Software Quality Journal*, 19(4), pages 725-751.



References >> II

- 5 Jiang B, Tse TH, Grieskamp W, Kicillof N, Cao Y, Li X, Chan WK. 2011. *Assuring the model evolution of protocol software specifications by regression testing process improvement*. Software: Practice and Experience. 41(10). pages 1073-1103.
- 6 McMinn, P. 2004. Search-based software test data generation: a survey: Research articles. *Software Testing, Verification and Reliability*, 14(2), pages 105–156.
- 7 Harman M., Hu L., Hierons R., Wegener J., Sthamer H., Baresel A., and Roper M., 2004. Testability Transformation. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 30(1), pages 1-14.
- 8 Wegener, J., Baresel, A. and Sthamer, H., 2001. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14), pages 841-854.
- 9 Duran, J.W. and Ntafos, S.C., 1984. An evaluation of random testing. *IEEE transactions on software engineering*, (4), pages 438-444.



References >> III

- 10 Miller, W., and Spooner, DL. 1976. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223-226,
- 11 Kuhn, DR. 1999. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering Methodology*, 8(4), pages 411-424.
- 12 Ferguson, R. and Korel, B., 1996. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(1), pages. 63-86.
- 13 DeMillo, R., and Offutt, J., 1993. Experimental results from an Automatic Test Case Generation. *ACM Transaction on Software Engineering Methodology*, 2(2), pages 109-175.
- 14 Ntafos, S.C., 1988. A comparison of some structural testing strategies. *IEEE Transactions on software engineering*, 14(6), pages 868-874.
- 15 Chilenski, J., and Miller. S. 1994 Application of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal*, pages 193-200.



References >> IV

- [16] Das A., and Mall R., 2013. Automatic Generation of MC/DC Test Data. *International Journal of Software Engineering, Acta Press* 2(1).
- [17] Majumder R., and Sen K., 2007. Hybrid Concolic Testing. *Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA* pages 416-426.
- [18] Hayhurst, KJ., Veerhusen DS., Chilenski, JJ., Rierson, LK. 2001. A Practical Tutorial on Modified Condition/Decision Coverage, *NASA/TM-2001-210876*.
- [19] Kim Y., and Kim M., 2011. SCORE: a scalable concolic testing tool for reliable embedded software. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* pages 420-423.
- [20] Kim, M., Kim, Y. and Choi, Y., 2012. Concolic testing of the multi-sector read operation for flash storage platform software. *Formal Aspects of Computing*, 24(3), pages 355-374.



References >> V

- [21] Burnim J., and Sen K., 2008. Heuristics for scalable dynamic test generation. In *Proc. ASE*, pages 443-446, Washington, D.C., USA.
- [22] Kim Y., Kim Y., Kim T., Lee G., Jang Y., and Kim M., 2013. Automated unit testing of large industrial embedded software using concolic testing. *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)* pages 519-528.
- [23] Kim M., and Kim Y.,and Jang Y., 2012. Industrial application of concolic testing on embedded software: Case studies. *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 390-399.
- [24] Bokil, P., Darke, P., Shrotri, U., and Venkatesh, R., 2009. Automatic Test Data Generation for C Programs. In *proceedings 3rd IEEE International Conference on Secure Software Integration and Reliability Improvement*.



References >> VI

- 25 Kim M., Kim Y., and Rothermel G., 2012. A Scalable Distributed Concolic Testing Approach: An Empirical Evaluation. *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 340-349.
- 26 Sen K., and Agha G., 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools (Tools Paper). *DTIC Document*.



Thank You!





Data Flow Testing

Dr. Durga Prasad Mohapatra

Professor

CSE Department

NIT Rourkela

Introduction

- In path coverage, the emphasis was to cover a path using statement or branch coverage.
- However, data and data integrity are as important as code and code integrity of a module.
- We have checked every possibility of the control flow of a module. But what about the data flow in the module?
- These questions can be answered,, if we consider data objects in the control flow of a module.

Introduction

cont...

- Data flow testing is a white-box technique that can be used to detect improper use of data values due to coding errors.
- Errors may be unintentionally introduced in a program by programmers,
 - e.g. a programmer might use a variable without defining it.
- Data flow testing gives a chance to look out for
 - inappropriate data definition, their use in predicate, computations, and termination.

Introduction

cont...

- It identifies the potential bugs
 - by examining the patterns in which that piece of data is used.
- Example: If an out-of-scope data is being used in a computation, then it is a bug. There may be several patterns like this which indicate data anomalies.

Introduction

cont...

- To examine the patterns, the control flow graph of a program is used.
- This test strategy selects the paths in the module's control flow such that various sequences of data objects can be chosen.
- The major focus is on the points at which the data receives values and the places at which the data initialized has been referenced.
- Thus, we have to choose enough paths in the control flow to ensure that every data is initialized before use and all the defined data have been used somewhere.

Data Flow-Based Testing

- Selects test paths of a program:
 - According to the locations of
 - Definitions and Uses of different variables in a program.

Example

```
1 X(){  
2 int a=5; /* Defines variable a */  
....  
3 While(c>5) {  
4 if (d<50)  
5     b=a*a; /*Uses variable a */  
6     a=a-1; /* Defines variable a */  
....  
7 }  
8 print(a); } /*Uses variable a */
```

Data Flow-Based Testing cont ...

- For a statement numbered S,
 - $\text{DEF}(S) = \{X / \text{statement } S \text{ contains a definition of } X\}$
 - $\text{USES}(S) = \{X / \text{statement } S \text{ contains a use of } X\}$
 - Example: 1: **a=b;** $\text{DEF}(1)=\{a\}$, $\text{USES}(1)=\{b\}$.
 - Example: 2: **a=a+b;** $\text{DEF}(1)=\{a\}$, $\text{USES}(1)=\{a,b\}$.

Data Flow-Based Testing cont ...

- A variable X is said to be **live** at statement S1, if
 - X is defined at a statement S, and
 - there exists a path from S to S1 not containing any definition of X.

DU Chain Example

```
1 X(){  
2     int a=5; /* Defines variable a */  
3     While(c>5) {  
4         if (d<50)  
5             b=a*a; /*Uses variable a */  
6             a=a-1; /* Defines variable a */  
7     }  
8     print(a); } /*Uses variable a */
```

Definition-use chain (DU chain)

- $[X, S, S_1]$,
 - S and S_1 are statement numbers,
 - $X \in \text{DEF}(S)$,
 - $X \in \text{USES}(S_1)$, and
 - the definition of X in the statement S is **live** at statement S_1 .

Data Flow-Based Testing Strategy

- One simple data flow testing strategy:
 - **Every DU chain in a program be covered at least once.**
- Data flow testing strategies:
 - Useful for selecting test paths of a program containing nested if and loop statements.

Example

```
1 X(){  
2   B1; /* Defines variable a */  
3   While(C1) {  
4     if (C2)  
5       if(C4) B4; /*Uses variable a */  
6     else B5;  
7     else if (C3) B2;  
8     else B3;    }  
9   B6 }
```

Example cont ...

- [a,1,5]: a DU chain.
- Assume:
 - $\text{DEF}(X) = \{\text{B1}, \text{B2}, \text{B3}, \text{B4}, \text{B5}\}$
 - $\text{USES}(X) = \{\text{B2}, \text{B3}, \text{B4}, \text{B5}, \text{B6}\}$
 - There are 25 DU chains.
- However only 5 paths are needed to cover these chains.

Data Flow Testing

cont...

- It also closely examines the state of the data in the CFG resulting in a richer test suite
 - than the one obtained from CFG based path testing strategies like statement coverage, branch coverage, etc.

States of a Data Object

- Defined (d):
- Killed / Undefined / Released (k):
- Usage (u):
- Computational use (c-use) or
- Predicate use (p-use).

State of a Data Object cont ...

A data object can be in the following states:

- **Defined (d)** A data object is called defined when it is initialized, i.e., when it is on the left side of an assignment statement. Defined state can also be used to mean that a file has been opened, a dynamically allocated object has been allocated, something is pushed onto the stack, a record written, and so on.

State of a Data Object

cont...

- ***Kill/Undefined/Released (k)***
- When the data has been reinitialized or the scope of a loop control variable finishes, i.e., exiting the loop or memory is released dynamically or a file has been closed.

State of a Data Object cont...

- **Usage (*u*)** When the data object is on the right side of assignment or the control variable in a loop, or in an expression used to evaluate the control flow of a case statement, or as a pointer to an object, etc.
- In general, we say that the usage is either computational use (c-use) or predicate use (p-use).

Data-Flow Anomalies

- Data-flow anomalies represent the patterns of data usage which may lead to an incorrect execution of the code.
- An anomaly is denoted by a two-character sequence of actions.

Data-Flow Anomalies

cont...

- Example: ‘dk’ means a variable is defined and killed without any use, which is a **potential bug**.
- There are nine possible two-character combinations out of which only four are data anomalies, as shown in next Table.

Table I: Two-character data-flow anomalies

Anomaly	Explanation	Effect of Anomaly
du	Define-use	Allowed, Normal case.
dk	Define-Kill	Potential bug. Data is killed without use after definition.
ud	Use-define	Data is used and then redefine. Allowed, Usually not a bug because the language permits reassignment at almost any time.
uk	Use-Kill	Allowed, Normal situation.
ku	Kill-use	Serious bug because the data is used after being killed.
kd	Kill-define	Data is Killed and then redefined, Allowed
dd	Define-define	Redefining a variable without using it . Harmless bug, but not allowed.
uu	Use-use	Allowed Normal case.
kk	Kill-kill	Harmless bug, but not allowed.

- Not all data-flow anomalies are harmful, but most of them are suspicious and indicate that an error can occur.
- There may be single-character data anomalies also.
- To represent these types of anomalies, we take the following conventions:
 - $\sim x$: indicates all prior actions are not of interest to x .
 - $x\sim$: indicates all post actions are not listed of interest to x .

Table 2: Single-character data-flow anomalies

Anomaly	Explanation	Effect of Anomaly
$\sim d$	First definition	Normal situation, Allowed.
$\sim u$	First Use	Data is used without defining it. Potential bug.
$\sim k$	First Kill	Data is killed before defining it, Potential bug.
$D\sim$	Define last	Potential bug.
$U\sim$	Use last	Normal case, Allowed.
$K\sim$	Kill last	Normal case, Allowed.

Some Terminologies

Suppose P is a program that has a graph G (P) and a set of variables V. The graph has a single entry and exit node.

- **Definition node** Defining a variable means assigning value to a variable for the very first time in a program. For example, input statements, assignment statements, loop control statements, procedure calls, etc.

Some Terminologies

contd...

- **Usage node** It means the variable has been used in some statement of the program. Node n that belongs to $G(p)$ is usage node of variable v , if the value of variable v is used at the statements corresponding to node n.

Some Terminologies

contd...

- A usage node can be of the following two types:
 - 1) Predicate usage Node: If usage node n is a predicate node, then n is a predicate usage node.
 - 2) Computation Usage Node: If usage node n corresponds to a computation statement in a program other than predicate, then it is called a computation usage node.

Some Terminologies

contd...

- **Loop-free path segment** It is a path segment for which every node is visited once at most.
- **Simple path segment** It is a path segment in which at most one node is visited twice. A simple path segment is either loop-free or if there is a loop, only one node is involved.
- **Definition-use path (du-path)** A du-path with respect to a variable v is a path between the definition node and usage node of that variable, Usage node can either be a p-usage or a c-usage node.

Some Terminologies

contd...

- **Definition-clear path (dc-path)** A dc-path with respect to a variable v is a path between the definition node and the usage node such that no other node in the path is a defining node of variable v .
- The du paths which are not dc paths are important, as these are potential spots for testing persons.
- Those du-paths which are definition-clear are easy to test in comparison to du-paths which are not dc-paths.
- The du-paths which are not dc-paths need more attention.

Static Data Flow Testing

With static analysis, the source code is analysed without executing it.

EXAMPLE:

Consider a program for calculating the gross salary of an employee in an organization. If his basic salary < 1500, then HRA=10% of the Basic and DA=90% of basic. If his salary >= 1500, then HRA=500 and DA=98% of basic. Calculate the gross salary.

```
main()
{
1. float bs, gs, da, hra=0;
2. printf("Enter basic salary");
3. scanf("%f",&bs);
4. if(bs < 1500)
5. {
6.     hra=bs * 10/100;
7.     da= bs * 90/100;
8. }
9. else
10. {
11.     hra = 500;
12.     da= bs * 98/100;
13. }
14. gs= bs+ hra+ da;
15. printf("Gross Salary = Rs. %f", gs);
16. }
```

Find out the define-use-kill patterns for all the variables in the program

Solution

Pattern	Line Number	Explanation
~d	3	Normal case. Allowed
du	3-4	Normal case. Allowed
uu	4-6,6-7,7-12,12-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

Define-use-kill patterns for variable ‘bs’

Solution

cont...

Pattern	Line Number	Explanation
~d	14	Normal case. Allowed
du	14-15	Normal case. Allowed
uk	15-16	Normal case. Allowed
K~	16	Normal case. Allowed

Define-use-kill patterns for variable ‘gs’

Solution

cont...

Pattern	Line Number	Explanation
~d	7	Normal case. Allowed
du	7-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

Define-use-kill patterns for variable '**da**'

Solution

cont...

Pattern	Line Number	Explanation
~d	1	Normal case. Allowed
dd	1-6 or 1-11	Double definition. Not allowed. Harmless bug.
du	6-14 or 11-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

Define-use-kill patterns for variable 'hra'

From the above static data flow testing, only one bug is found, i.e in variable HRA of double definition.

Summary

- Discussed the basic concepts of data flow testing.
- Explained DU Chain.
- Presented the different states of a data object.
- Explained the different data-flow anomalies.
- Explained static data flow testing with an example.

References

1. Rajib Mall, Fundamentals of Software Engineering, (Chapter – 10), Fifth Edition, PHI Learning Pvt. Ltd., 2018.
2. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter – 5), Second Edition, Oxford University Press, 2016.



Thank you



Data Flow Testing

cont...

Dr. Durga Prasad Mohapatra
Professor
CSE Department
NIT Rourkela

Static Analysis is not Enough

- All anomalies using static analysis cannot be determined and this is an unsolvable problem.
- For example, if the variable is used in an array as the index, we cannot determine its state by static analysis.
- Or it may be the case that the index is generated dynamically during execution, therefore we cannot guarantee what the state of the array element is referenced by that index.

Dynamic Data Flow Testing

- The test cases are designed in such a way that every definition of data variable to each of its use is traced to each of its definition.
- Various strategies are employed for the creation of test cases.
- All these strategies are defined below.

Dynamic Data Flow Testing

cont...

- **All-du Paths (ADUP)** It states that every definition of every variable to every use of that definition should be exercised under some test. It is the strongest data flow testing strategies.

Dynamic Data Flow Testing

cont...

- **All-uses (AU)** This states that for every use of the variable, there is a path from the definition of that variable (nearest to the use in backward direction) to the use.

Dynamic Data Flow Testing

cont...

- **All-p-uses/Some-c-uses (APU + C)** This strategy states that for every variable and every definition of that variable, include at least one dc-path from the definition to every predicate use. If there are definitions of the variable with no p-use following it, then add computational use (c-use) test cases as required to cover every definition.
- Note: A dc-path (definition-clear path) with respect to a variable v is a path between the definition node & the usage node s.t. that no other node in the path is a defining node of variable v .

Dynamic Data Flow Testing

cont...

- **All-c-uses/Some-p-uses (ACU + P)** This strategy states that for every variable and every definition of that variable, include at least one dc-path from the definition to every computational use. If there are definitions of the variable with no c-use following it, then add predicate use (p-use) test cases as required to cover every definition.

Dynamic Data Flow Testing

cont...

- **All-Predicate-uses (APU)** It is derived from the APU + C strategy and states that for every variable, there is a path from every definition to every p-use of that definition. If there is a definition with no p-use following it, then it is dropped from contention.

Dynamic Data Flow Testing

cont...

- **All-Computational-Uses (ACU)** It is derived from the strategy ACU + P strategy and states that for every variable, there is a path from every definition to every c-use of that definition. If there is a definition with no c-use following if, then it is dropped from contention.

Dynamic Data Flow Testing

cont...

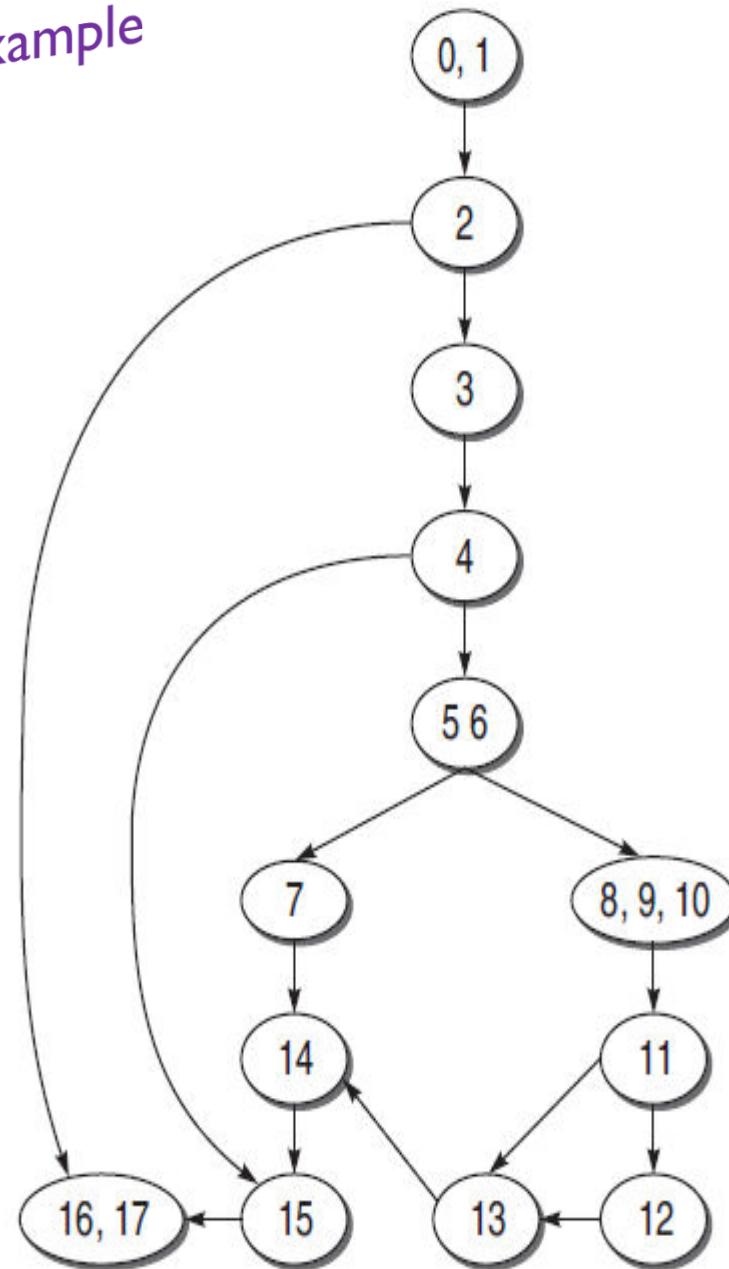
- **All-Definition (AD)** It states that every definition of every variable should be covered by at least one use of that variable, be that a computational use or a predicate use.

Example

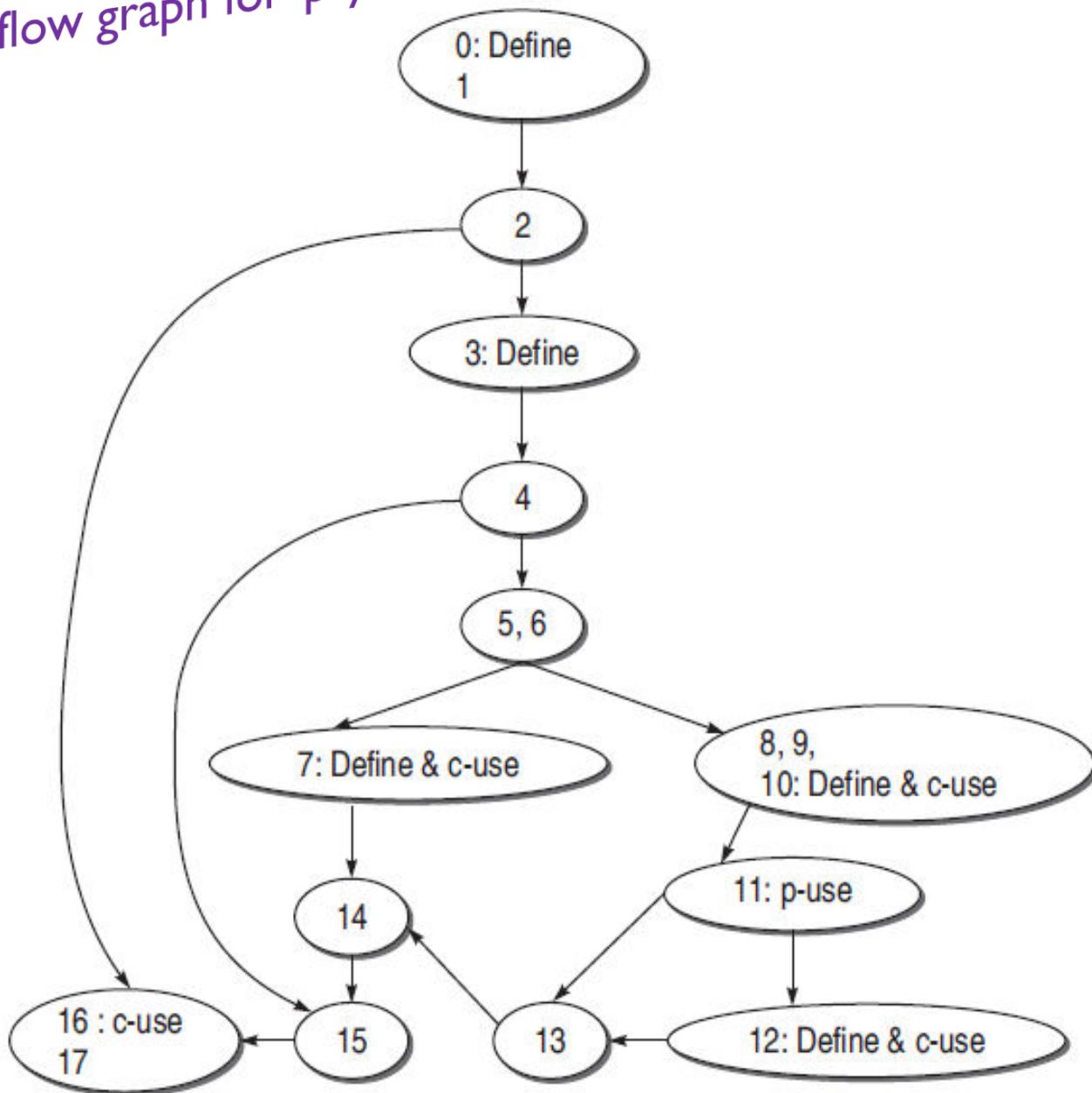
- Consider a program given below. Draw its control flow graph and data flow graph for each variable used in the program, and derive data flow testing paths with all the strategies discussed above.

```
main()
{
    int work;
0.    double payment =0;
1.    scanf("%d", work);
2.    if (work > 0) {
3.        payment = 40;
4.        if (work > 20)
5.        {
6.            if(work <= 30)
7.                payment = payment + (work - 25) * 0.5;
8.            else
9.            {
10.                payment = payment + 50 + (work -30) * 0.1;
11.                if (payment >= 3000)
12.                    payment = payment * 0.9;
13.                }
14.            }
15.        }
16.        printf("Final payment", payment);
```

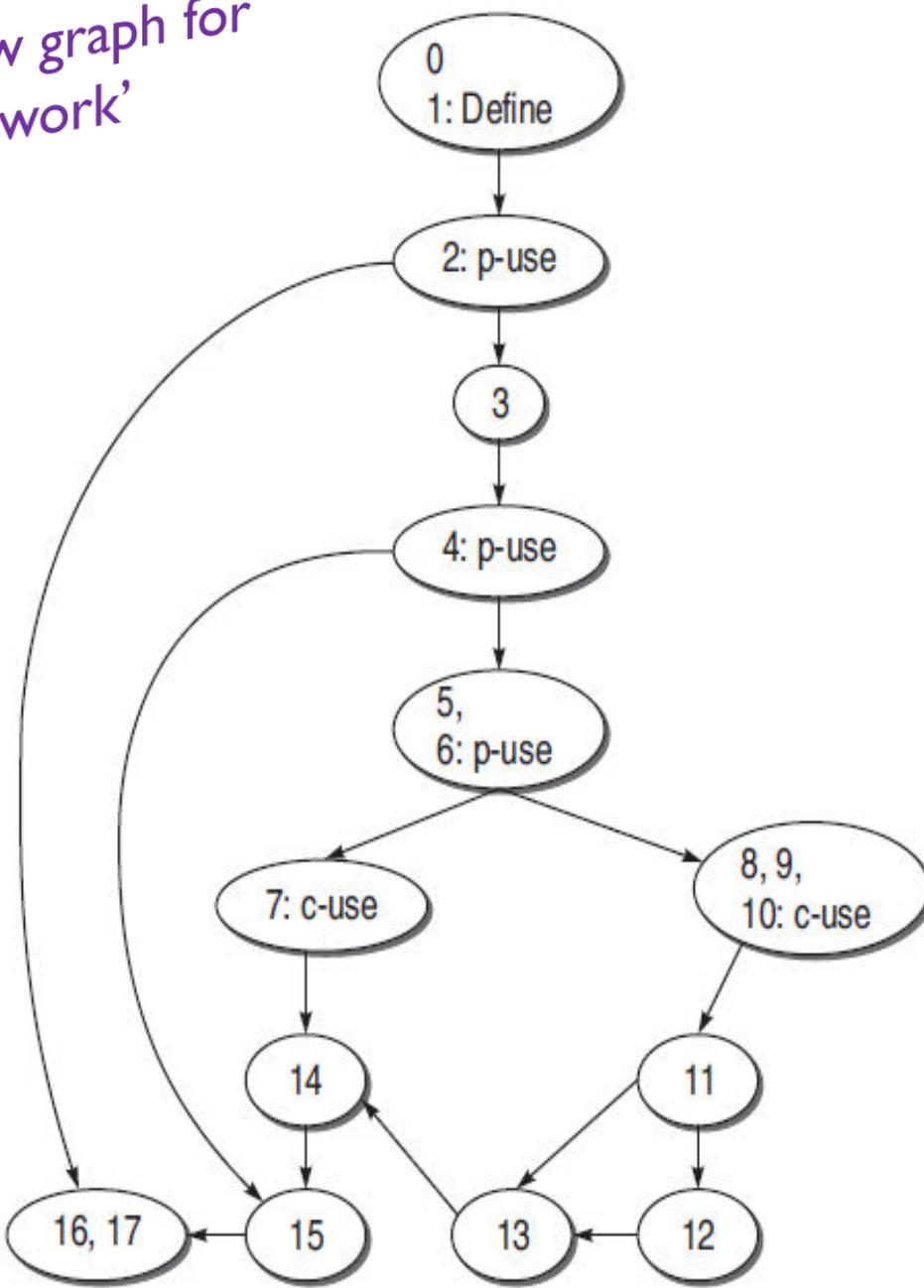
CFG for the Example



Data flow graph for 'payment'



Data flow graph for variable 'work'



- Prepare a list of all the definition nodes and usage nodes for all the variables in the program.

Variable	Defined At	Used At
Payment	0,3,7,10,12	7,10,11,12,16
Work	1	2,4,6,7,10

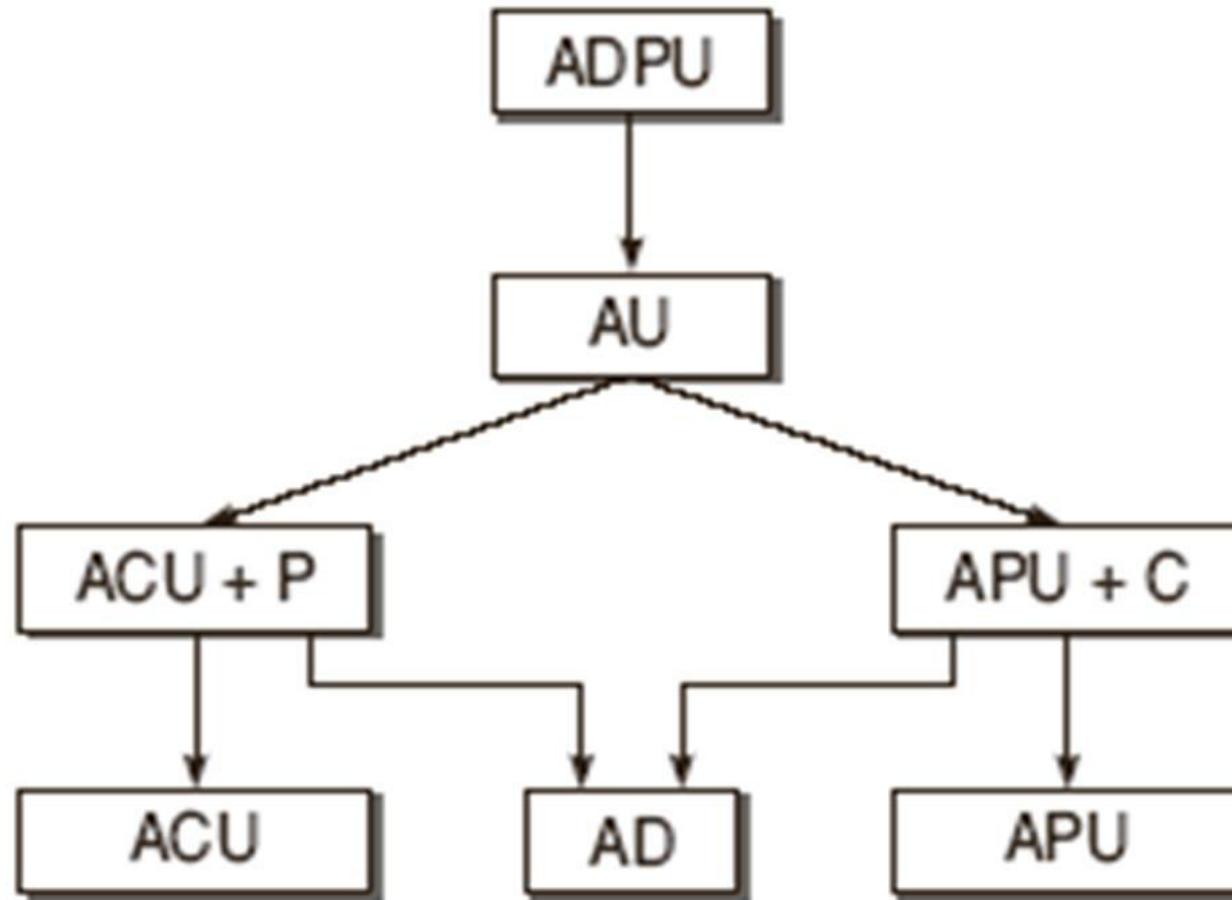
Strategy	Payment	Work
All Uses(AU)	3-4-5-6-7 10-11 10-11-12 12-13-14-15-16 3-4-5-6-8-9-10	1-2 1-2-3-4 1-2-3-4-5-6 1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10
All p-uses(APU)	0-1-2-3-4-5-6-8-9-10-11	1-2 1-2-3-4 1-2-3-4-5-6
All c-uses(ACU)	0-1-2-16 3-4-5-6-7 3-4-5-6-8-9-10 3-4-15-16 7-14-15-16 10-11-12 10-11-13-14-15-16 12-13-14-15-16	1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10

Strategy	Payment	Work
All-p-uses / Some-c-uses (APU + C)	0-1-2-3-4-5-6-8-9-10-11	1-2
	10-11-12	1-2-3-4
	12-13-14-15-16	1-2-3-4-5-6
		1-2-3-4-5-6-8-9-10
All-c-uses / Some-p-uses (ACU + P)	0-1-2-16	1-2-3-4-5-6-7
	3-4-5-6-7	1-2-3-4-5-6-8-9-10
	3-4-5-6-8-9-10	1-2-3-4-5-6
	3-4-15-16	
	7-14-15-16	
	10-11-12	
	10-11-13-14-15-16	
	12-13-14-15-16	
	0-1-2-3-4-5-6-8-9-10-11	

Strategy	Payment	Work
All-du-paths (ADUP)	0-1-2-3-4-5-6-8-9-10-11	1-2
	0-1-2-16	1-2-3-4
	3-4-5-6-7	1-2-3-4-5-6
	3-4-5-6-8-9-10	1-2-3-4-5-6-7
	3-4-15-16	1-2-3-4-5-6-8-9-10
	7-14-15-16	
	10-11-12	
	10-11-13-14-15-16	
	12-13-14-15-16	
All Definitions (AD)	0-1-2-16	1-2
	3-4-5-6-7	
	7-14-15-16	
	10-11	
	12-13-14-15-16	

Ordering of data flow testing strategies

- While selecting a test case, we need to analyse the relative strengths of various data flow testing strategies.



Ordering of data flow testing strategies

cont...

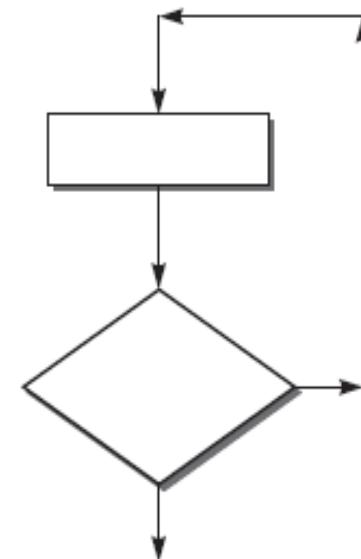
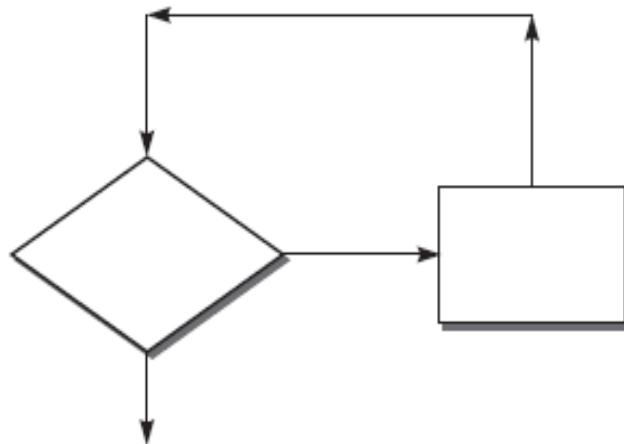
- The previous Figure depicts the relative strength of the data flow strategies.
- In this figure, the relative strength of testing strategies reduces along the direction of the arrow.
- It means that all-du-paths (ADUP) is the strongest criterion for selecting the test cases.

Loop testing

- Loop testing can be viewed as an extension to branch coverage.
- Loops are important in the software from the testing view point. If loops are not tested properly, bugs can go undetected.
- Loop testing can be done effectively while performing development testing (unit testing by the developer) on a module.
- Sufficient test cases should be designed to test every loop thoroughly.
- There are four different kinds of loops. How each kind of loop is tested, is discussed below.

Simple loops

- Simple loops mean, we have a single loop in the flow, as shown below.



Testing Simple Loops

- Check whether you can bypass the loop or not. If the test case for bypassing the loop is executed and, still you enter inside the loop, it means there is a bug.
- Check whether the loop control variable is negative.
- Write one test case that executes the statements inside the loop.

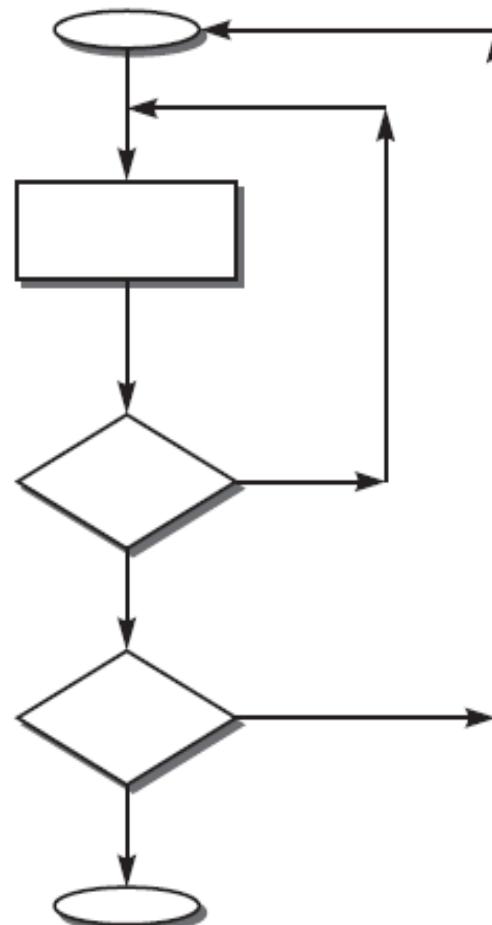
Testing Simple Loops cont ...

- Write test cases for a typical number of iterations through the loop.
- Write test cases for checking the boundary values of the maximum and minimum number of iterations defined (say max and min) in the loop. It means we should test for min, min+1, min-1, max-1, max, and max+1 number of iterations through the loop.

Nested loops

- When two or more loops are embedded, it is called a nested loop, as shown in next slide. If we have nested loops in the program, it becomes difficult to test.

Nested loops



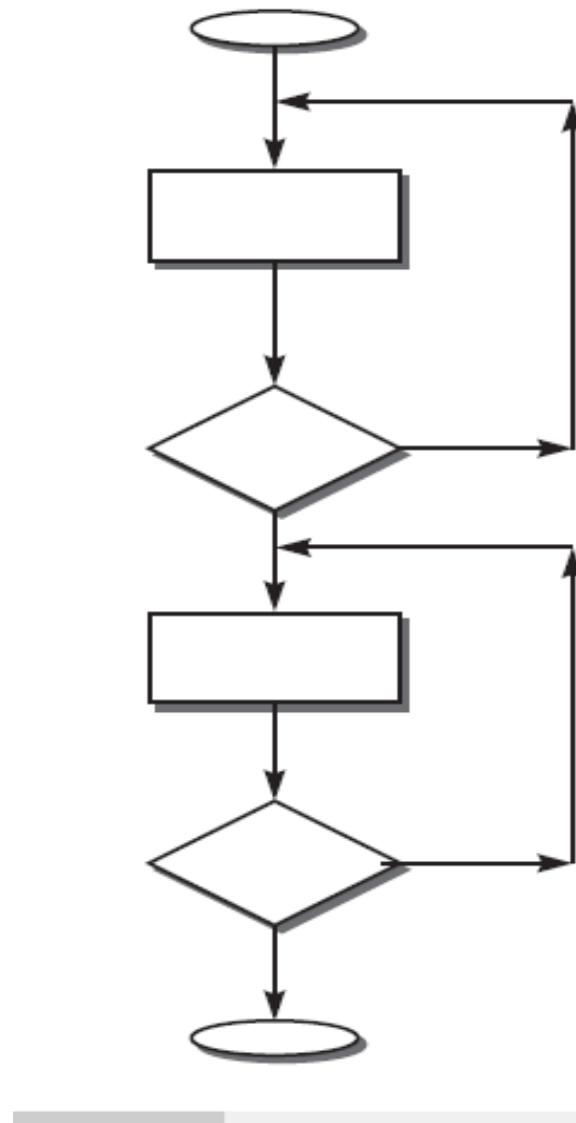
Testing Nested Loops

- If we adopt the approach of simple tests to test the nested loops, then the number of possible test cases grows geometrically.
- Thus, the strategy is to start with the innermost loops while holding outer loops to their minimum values. Continue this outward in this manner until all loops have been covered

Concatenated loops

- The loops in a program may be concatenated (shown in next slide).
- Two loops are concatenated if it is possible to reach one, after exiting the other, while still on a path from entry to exit.
- If the two loops are not on the same path, then they are not concatenated.
- The two loops on the same path may or may not be independent.
- If the loop control variable for one loop is used for another loop, then they are concatenated, but nested loops should be treated like nested only.

Concatenated loops



Testing Concatenated Loops

- The concatenated loop may be treated as a sequence of two or more numbers of simple loops.
- So, the strategy for testing of simple loops may be extended to testing of concatenated loops.

Unstructured loops

- This type of loops is really impractical to test.
- They must be redesigned or at least converted into simple or concatenated loops.

Summary

- Explained dynamic data flow testing strategies with an example.
- Presented the ordering of different data flow testing strategies.
- Discussed different loop testing strategies.

References

- I. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter – 5), Second Edition, Oxford University Press, 2016.



Thank you



Software Debugging

Dr. Durga Prasad Mohapatra

Professor

National Institute of Technology

Rourkela



Debugging

- Once errors are identified:
 - it is necessary to identify the precise location of the errors and to fix them.
- Each debugging approach has its own advantages and disadvantages:
 - each is useful in appropriate circumstances.



Some Debugging Approaches

- Brute Force method
- Symbolic Debugger
- Backtracking
- Cause-Elimination Method
- Program Slicing

Brute-force method

- This is the most common method of debugging:
 - least efficient method.
 - program is loaded with print statements
 - print the intermediate values
 - hope that some of printed values will help identify the error.

Symbolic Debugger

- Brute force approach becomes more systematic:
 - with the use of a symbolic debugger,
 - symbolic debuggers get their name for historical reasons
 - early debuggers let you only see values from a **program dump**:
 - determine which variable it corresponds to.

Symbolic Debugger

- Using a symbolic debugger:
 - values of different variables can be easily checked and modified
 - single stepping to execute one instruction at a time
 - **break points** and **watch points** can be set to test the values of variables.



Backtracking

- This is a fairly common approach.
- Beginning at the statement where an error symptom has been observed:
 - source code is traced backwards until the error is discovered.

Example

```
int main( ){
    int i, j, s;
    i=1;
    while(i<=10){
        s=s+i;
        i++;
        j=j++;}
    printf("%d",s);
}
```

Backtracking

- Unfortunately, as the number of source lines to be traced back increases,
 - the number of potential backward paths increases
 - becomes unmanageably large for complex programs.



Cause-elimination method

- In this method, once a failure is observed, the symptoms of the failure (e.g. certain variable is having a negative value though it should be positive) are noted.
- Determine a list of causes:
 - which could possibly have contributed to the error symptom.
 - tests are conducted to eliminate each.
- A related technique of identifying errors by examining error symptoms:
 - **software fault tree analysis.**



Program Slicing

- This technique is similar to back tracking.
- However, the search space is reduced by defining slices.
- A slice is defined for a particular variable at a particular statement:
 - set of source lines preceding this statement which can influence the value of the variable.



Program Slicing cont ...

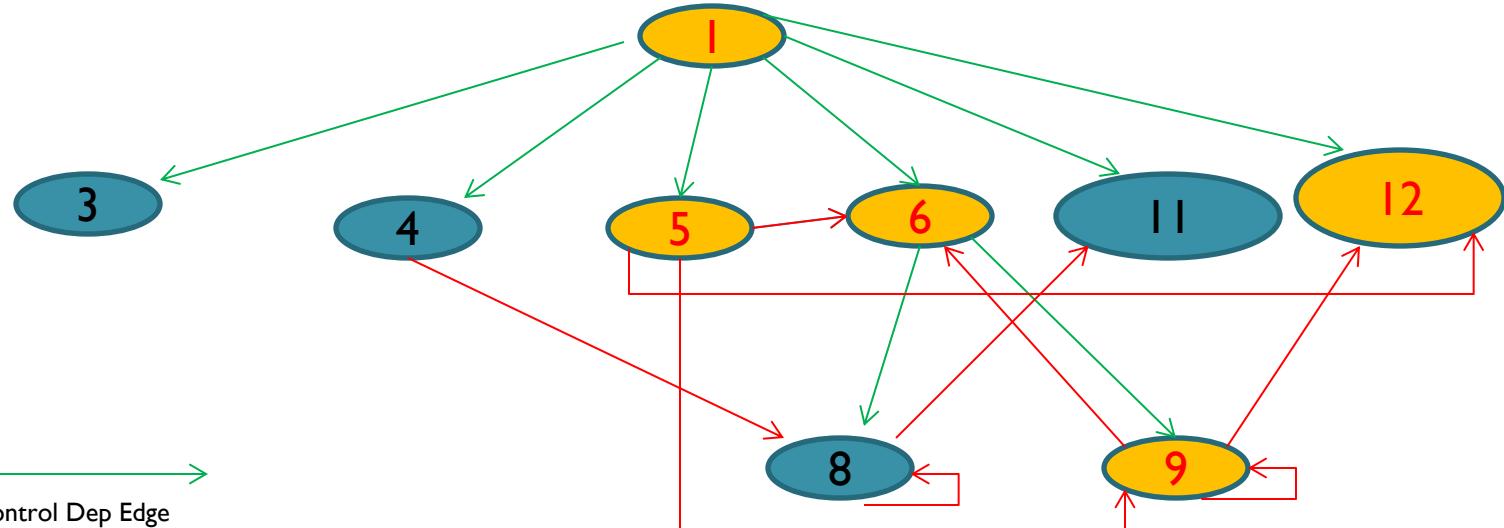
- Slice of a program w.r.t. program point p and variable x :
 - All statements and predicates that might affect the value of x at point p .
- $\langle p, x \rangle$ known as slicing criterion.

Example

```
1 main( )
2 {
3     int i, sum;
4     sum = 0;
5     i = 1;
6     while(i <= 10)
7     {
8         sum = sum + i;
9         ++ i;
10    }
11    printf("%d", sum);
12    printf("%d", i);
13 }
```

An Example Program & its backward slice w.r.t. $\langle 12, i \rangle$

Program Dependence Graph



Control Dep Edge

Data Dep Edge

Example

An Example Program

& its slice w.r.t. $\langle 9, i \rangle$

```
1. int main( ){  
2.     int i, s;  
3.     i=1;  
4.     s=1;  
5.     while(i<=10){  
6.         s=s+i;  
7.         i++;}  
8.     printf("%d",s);  
9.     printf("%d",i);  
10. }
```

Types of Slices

Static Slice: Statements that may affect value of a variable at a program point for *all possible executions*.

Dynamic Slice: Statements that actually affect value of a variable at a program point for *that particular execution*.

Backward Slice: Statements that *might have* affected the variable at a program point.

Forward Slice: Statements that *might be* affected by the variable at a program point.

Example of Forward Slice

```
1 main( )
2 {
3     int i, sum;
4     sum = 0;
5     i = 1;
6     while(i <= 10)
7     {
8         sum = sum + i;
9         ++ i;
10    }
11    printf("%d", sum);
12    printf("%d", i);
13 }
```

An Example Program & its forward slice w.r.t. $\langle 5, i \rangle$

Types of Slices

cont ...

- **Intra-Procedural Slice:** for programs having only one procedure
 - Not applicable for OOPs
- **Inter-Procedural Slice:** for programs having more than one procedure
 - Applicable for OOPs

Applications of Slicing

- Debugging
- Program understanding
- Testing
- Software maintenance
- Complexity measurement
- Program integration
- Reverse engineering
- Software reuse



Debugging Guidelines

- Debugging usually requires a thorough understanding of the program design.
- Debugging may sometimes require full redesign of the system.
- A common mistake novice programmers often make:
 - not fixing the error but the error symptoms.



Debugging Guidelines

- Be aware of the possibility:
 - an error correction may introduce new errors.
- After every round of error-fixing:
 - regression testing must be carried out.

Program Analysis Tools

- An automated tool:
 - takes program source code as input
 - produces reports regarding several important characteristics of the program,
 - such as size, complexity, adequacy of commenting, adherence to programming standards, etc.

Program Analysis Tools

- Some program analysis tools:
 - produce reports regarding the adequacy of the test cases.
- There are essentially two categories of program analysis tools:
 - **Static analysis tools**
 - **Dynamic analysis tools**



Static Analysis Tools

- Static analysis tools:
 - Assess properties of a program without executing it.
 - Analyze the source code
 - provide analytical conclusions.



Static Analysis Tools

- Whether coding standards have been adhered to?
 - Commenting is adequate?
- Programming errors such as:
 - uninitialized variables
 - mismatch between actual and formal parameters.
 - Variables declared but never used, etc.

Static Analysis Tools

- Code walk through and inspection can also be considered as static analysis methods:
 - however, the term static program analysis is generally used for automated analysis tools.



Dynamic Analysis Tools

- Dynamic program analysis tools require the program to be executed:
 - its behaviour recorded.
 - Produce reports such as, extent of coverage achieved, adequacy of test cases, etc.

Summary

- Discussed different debugging approaches.
 - Brute Force method
 - Symbolic Debugger
 - Backtracking
 - Cause-Elimination Method
 - Program Slicing
- Presented some debugging guidelines.
- Explained the Program Analysis Tools
 - Static analysis tools
 - Dynamic analysis tools

References

- I. Rajib Mall, Fundamentals of Software Engineering, (Chapter – 10), Fifth Edition, PHI Learning Pvt. Ltd., 2018.



Thank You



Efficient Test Suite Management

Prof. Durga Prasad Mohapatra
Professor
Dept.of CSE, NIT Rourkela



Why do test suites grow?

- Test cases in an existing test suite can often be used to test a modified program.
- However, if the test suite is inadequate for retesting, new test cases may be developed and added to the test suite.
- There may be unnecessary test cases in the test suite including both obsolete and redundant test cases.



Why do test suites grow? (contd..)

- A change in a program causes a test case to become obsolete by removing the reason for the test case's inclusion in the test suite.
- A test case is redundant if other test cases in the test suite provide the same coverage of the program.
- Thus due to obsolete and redundant test cases, the size of a test suite continues to grow unnecessarily.



Why test suite minimization is important?

- release date of the product is near
- limited staff to execute all the test cases
- limited test equipment or unavailability of testing tools.



Advantages of minimizing a test suite

- redundant test cases will be eliminated
- reduces the cost of the project by reducing a test suite to a minimal subset.
- decreases both the overhead of maintaining the test suite and the number of test cases that must be rerun after changes are made to the software, thereby reducing the cost of regression testing.
- so it is a great practical advantage to reduce the size of test cases.

Defining test suite minimization problems

Harrold et al. have defined the problem of minimizing test suite as given below.

- **Given:** a test suite TS; a set of test case requirements r_1, r_2, \dots, r_n that must be satisfied to provide the desired testing coverage of the program; and subsets of TS: T_1, T_2, \dots, T_n , one associated with each of the r_i 's such that any one of the test cases t_j belonging to T_i can be used to test r_i .



Defining test suite minimization problems (contd..)

- **Problem:** find a representative set of test cases from TS that satisfies all the r_i 's
 - a representative set of test cases that satisfies the r_i 's must contain at least one test case from each T_i ,
 - such a set is called a **hitting set** of the group of sets, T_1, T_2, \dots, T_n .

Defining test suite minimization problems (contd..)

- The r_i 's can represent either all the test case requirements of a program or those requirements related to program modifications.
- maximum reduction is achieved by finding the smallest representative of test cases,
- however this subset of the test suite is the minimum cardinality hitting set of the T_i 's.
- the problem of finding the minimum cardinality hitting set is NP-complete.
- thus minimization techniques resort to heuristics.



Test Suite Prioritization

- The reduction process can be understood if the cases in a test suite are prioritized in some order.
- The purpose of prioritization is to reduce the set of test cases based on some rational, non-arbitrary criteria, while aiming to select the most appropriate tests.
- For example, the following priority categories can be determined for the test cases.

Test Suite Prioritization cont...

- **Priority I** The test cases must be executed, otherwise there may be worse consequences after the release of the product.
- For example, if the test cases for this category are not executed, then critical bugs may appear.

Test Suite Prioritization cont...

- **Priority 2** *The test cases may be executed, if time permits.*
- **Priority 3** *The test case is not important prior to the current release.*
- It may be tested shortly after the release of the current version of the software.

Test Suite Prioritization cont...

- **Priority 4** *The test case is never important, as its impact is nearly negligible.*
- In the prioritization scheme, the main guideline is to ensure that low-priority test cases do not cause any severe impact on the software.
- There may be several goals of prioritization.

Test Suite Prioritization cont...

- These goals can become the basis for prioritizing the test cases. Some of them are discussed here:
 - I. Testers or customers may want to get some critical features tested and presented in the first version of the software.
 - Thus, the important features become the criteria for prioritizing the test cases.
 - But the consequences of not testing some low-priority features must be checked.
 - So, risk factor should be analyzed for every feature in consideration.

Test Suite Prioritization cont...

2. Prioritization can be on the basis of the functionality advertised in the market.

It becomes important to test those functionalities on a priority basis, which the company has promised to its customers.

3. The rate of fault detection of a test suite can reveal the likelihood of faults earlier.

Test Suite Prioritization cont...

4. To increase the coverage of coverable code in the system under test at a faster rate, allowing a code coverage criterion to be met earlier in the test process.
5. To increase the rate at which high-risk faults are detected by a test suite, thus locating such faults earlier in the testing process.



Test Suite Prioritization cont...

6. To increase the likelihood of revealing faults related to specific code changes, earlier in the regression testing process.



Types of Test Case Prioritization

- General Test Case Prioritization
- Version-Specific Test Case Prioritization



General Test Case Prioritization

- In this prioritization, we prioritize the test cases that will be useful over a succession of subsequent modified versions of P, **without any knowledge of the modified version.**
- Thus, a general test case prioritization can be performed following the release of a program version during off-peak hours, and the cost of performing the prioritization is amortized over the subsequent releases.



Version-Specific Test Case Prioritization

- Here, we prioritize the test cases such that they will be useful on a specific version P' of P .
- Version-specific prioritization is performed after a set of changes have been made to P and prior to regression testing P' , **with the knowledge of the changes that have been made.**



Prioritization Techniques

- Prioritization techniques schedule execution of test cases in an order that attempts to increase their effectiveness at meeting some performance goal.
- Various prioritization techniques may be applied to a test suite with the aim of meeting that goal.
- Prioritization can be done at two levels:
 - Prioritization for regression test suite
 - Prioritization for system test suite



Types of Prioritization Techniques

- Coverage-based Test Case Prioritization
- Risk-Based Prioritization
- Prioritization Based on Operational Profiles
- Prioritization using Relevant Slices
- Prioritization Based on Requirements

Coverage-based Test Case Prioritization

- a) Total statement Coverage Prioritization
- b) Additional Statement Coverage Prioritization
- c) Total Branch Coverage Prioritization
- d) Additional Branch Coverage Prioritization
- e) Total Fault-Exposing-Potential(FEP) Prioritization

Total statement Coverage Prioritization

- This prioritization orders the test cases based on the total number of statements covered.
- It counts the number of statements covered by the test cases and orders them in a descending order.
- If multiple test cases cover the same number of statements, then a random order may be used.
- For example, if T_1 covers 5 statements, T_2 covers 3, and T_3 covers 12 statements; then according to this prioritization, the order will be T_3, T_1, T_2 .



Additional statement coverage prioritization

- Total statement coverage prioritization schedules the test cases based on the total statements covered.
- However, it will be useful if it can execute those statements as well that have not been covered yet.
- It iteratively selects a test case T_1 , that yields the greatest statement coverage, and then selects a test case which covers a statement not covered by T_1 .
- Repeat this process until all statements covered by at least one test case have been covered.



Additional statement coverage prioritization cont...

- For example, if we consider Table I, according to total statement coverage criteria, the order is 2, 1, 3.
- But additional statement coverage selects test case 2 first and next, it selects test case 3, as it covers statement 4 which has not been covered by test case 2. Thus, the order according to addition coverage criteria 2, 3, 1.

Table I. Statement coverage

Statement	Statement Coverage		
	Test Case 1	Test Case 2	Test Case 3
1	X	X	X
2	X	X	X
3		X	X
4			X
5			
6		X	
7	x	X	
8	X	X	
9	X	x	

Total branch coverage prioritization

- In this prioritization, the criterion to order is to consider condition branches in a program instead of statements.
- Thus, it is the coverage of each possible outcome of a condition in a predicate.
- The test case which will cover maximum branch outcomes will be ordered first.
- For example, in Table 2, the order will be 1, 2, 3.

Total branch coverage prioritization cont...

Table 2. Branch coverage

Branch statements	Branch Coverage		
	Test case 1	Test case 2	Test case 3
Entry to while	X	X	X
2-true	X	X	x
2-false	X		
3-true		X	
3-false	X		

Additional Branch Coverage Prioritization

- The idea is the same as in additional statement coverage of first selecting the test case with the maximum coverage of branch outcomes and
 - then, selecting the test case which covers the branch outcome not covered by the previous one.

Total Fault-Exposing-Potential (FEP) Prioritization

- Statement and branch coverage prioritization ignore a fact about test cases and faults:
 - Some bugs/faults are more easily uncovered than other faults.
 - Some test cases have the proficiency to uncover particular bugs as compared to other test cases.

Total Fault-Exposing-Potential (FEP) Prioritization

- Thus, the ability of a test case to expose a fault is called the **fault exposing potential**.
- It depends not only on whether test cases cover a faulty statement, but also on the probability that a fault in that statement will also cause a failure for that test case.
- To obtain an approximation of the FEP of a test case, an approach was adopted using mutation analysis.

This approach is discussed below.

- Given program P and test suite T,
- First create a set of mutants $N = \{n1, n2, \dots, nm\}$ for P, noting which statement s_j in P contains each mutant.
- Next, for each test case $t_i \in T$, execute each mutant version n_k of P on t_i , noting whether t_i kills that mutant.
- Calculate $FEP(s,t) = \text{Mutants of } s_j \text{ killed} / \text{Total number of mutants of } s_j$.
- To perform total FEP prioritization, given these $FEP(s,t)$ values, calculate an *award value* for each test case $t_i \in T$, by summing the $FEP(sj, ti)$ values for all statements s_j in P. Given these award values, we prioritize test cases by sorting them in order of descending award value.

The next table shows FEP estimates of a program. Total FEP prioritization outputs the test order as (2, 3, 1).

Statement	FEP(s,t) values		
	Test case 1	Test case 2	Test case 3
1	0.5	0.5	0.3
2	0.4	0.5	0.4
3		0.01	0.4
4		1.3	
5			
6	0.3		
7	0.6		0.1
8		0.8	0.2
9			0.6
Award values	1.8	3.11	2.0



Summary

- Discussed why do test suites grow.
- Explained why test suite minimization is important.
- Explained the basics of test suite prioritization.
- Presented the different types of prioritization techniques.
- Discussed in detail the different types of coverage-based test case prioritization techniques.

References

- I. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter – 12), Second Edition, Oxford University Press, 2016.



Thank you



Efficient Test Suite Management

cont ...

Prof. Durga Prasad Mohapatra
Professor
Dept. of CSE, NIT Rourkela



Risk-Based Prioritization

- It is a well defined process that prioritizes modules for testing.
- Uses risk analysis to highlight potential problem areas, whose failure have adverse consequences.
- Testers use this risk analysis to select the most crucial tests.

Risk-Based Prioritization cont ...

- This technique is used to prioritize the test cases based on *some potential problems* which may occur during the project. It uses:
 - **Probability of occurrence/fault likelihood:** It indicates the probability of occurrence of a problem.
 - **Severity of impact/failure impact:** If the problem has occurred, how much impact does it have on the software.

Risk-Based Prioritization

cont ...

- Risk analysis uses these two components by first listing the potential problems and then, assigning a probability and severity value for each identified problem, as shown in next Table.
- By ranking the results in this table in the form of risk exposure, testers can identify the potential against which the software needs to be tested and executed first.

Risk-Based Prioritization

cont ...

A risk analysis table consists of the following columns:

- **Problem ID:** A unique identifier to facilitate referring to a risk factor.
- **Potential problem:** Brief description of the problem.
- **Uncertainty factor:** It is the probability of occurrence of the problem. Probability values are on a scale of 1(low) to 10(high).
- **Severity of impact:** Severity values on a scale of 1(low) to 10(high).
- **Risk exposure:** Product of probability of occurrence and severity of impact.

Table I .A sample risk analysis table

Problem ID	Potential Problem	Uncertainty Factor	Risk Impact	Risk Exposure
P1	Specification ambiguity	2	3	6
P2	Interface Problem	5	6	30
P3	File corruption	6	4	24
P4	Databases not synchronized	8	7	56
P5	Unavailability of modules for integration	9	10	90

Inference from the table

The problems / modules given in the previous table can be prioritized in the order of P5, P4, P2, P3, P1, based on the risk exposure values.



Prioritization Based on Operational Profiles

- In this approach, the test planning is done based on the **operational profiles** of the important functions which are of use to the customer.
- An **operational profile** is a set of tasks performed by the system and their probabilities of occurrence.
- After estimating the operational profiles, testers decide the total number of test cases, keeping in view the costs and resource constraints.



Prioritization using Slices

- During regression testing, the modified program is executed on all existing regression test cases to check that it still works the same way as the original program, except where a change is expected.
- But re-running the test suite for every change in the software makes regression testing a time-consuming process.

Prioritization using Slices cont ...

- If we can find the portion of the software which has been affected with the change in software, then we can prioritize the test cases based on this information.
- This is called the **slicing technique**.

Execution Slice

- The set of statements executed under a test case is called the **execution slice** of the program.
- Please refer to the following program.
- Table 2 shows the test cases for the given program.

```

Begin
S1: read (basic, empid);
S2: gross = 0;
S3: if (basic > 5000 || empid > 0)
{
    S4:     da = (basic*30)/100;
    S5:     gross = basic + da;
}
S6: else
{
    S7:     da = (basic*15)/100;
    S8:     gross = basic + da;
}
S9: print (gross, empid);
End

```

Fig 1. Example program for execution

Table 2 Test cases

Test Case	Basic	Empid	Gross	Empid
T1	8000	100	10400	100
T2	2000	20	2300	20
T3	10000	0	13000	0

Prioritization using Slices cont ...

- T1 and T2 produce correct results.
- On the other hand, T3 produces an incorrect result.
- Syntactically, it is correct, but an employee with the empid '0' will not get any salary, even if his basic salary is read as input.
- So it has to be modified.

Prioritization using Slices cont ...

- Suppose S3 is modified as [if(basic>5000 && empid>0)].
- So, for T1, T2, and T3, the program would be rerun to validate whether the change in S3 has introduced new bugs or not.
- But, if there is a change in S7,[da = (basic*25)/100; instead of da = (basic*15)/100;], then only T2 will be rerun.

Prioritization using Slices cont ...

- So in the execution slice, we will have less number of statements.
- The execution slice is highlighted in the given code segment.

Example

```
Begin
    S1: read (basic, empid);
    S2: gross=0;
    S3: if(basic > 5000 || empid > 0)
        {
        S4:     da = (basic*30)/100;
        S5:     gross = basic + da;
        }
    S6: else
        {
        S7:     da = (basic*15)/100;
        S8:     gross = basic + da;
        }
    S9: print(gross, empid);
End
```

Dynamic Slice

The set of statements under a test case having an effect on the program output is called the **dynamic slice** of the program with respect to the out-put variables.

Example

```
Begin
    S1: read (a,b);
    S2: sum=0;
    S2.1: I=0;
    S3: if (a==0)
        {
            S4:      print(b);
            S5:      sum+=b;
        }
    S6: else if(b==0)
        {
            S7:      print(a);
            S8:      sum+=a;
        }
    S9: else
        {
            S10:     sum=a+b+sum;
            S10.1    I=25;
            S10.2    print(I);
        }
    S11:endif
    S12:print(sum);
End
```

Test Cases

Table 3: Test cases for the given program.

Test Case	a	b	sum
T1	0	4	4
T2	67	0	67
T3	23	23	46

Dynamic Slice cont ...

- T1, T2 and T3 will run correctly but if some modification is done in S10.I [say I = 50], then this change will not affect the output variable.
- So, there is no need to rerun any of the test cases.
- On the other hand, if S10 is changed [say, sum =a* b+sum], then this change will affect the output variable 'sum'.
- So there is a need to rerun T3.
- The dynamic slice is highlighted in the code segment (S1, S2,S10,S12).

```
Begin
    S1: read (a,b);
    S2: sum=0;
    S2.1: I=0;
    S3: if (a==0)

        {
        S4:      print(b);
        S5:      sum+=b;
        }
    S6: else if(b==0)
        {
        S7:      print(a);
        S8:      sum+=a;
        }
    S9: else
        {
        S10:     sum=a+b+sum;
        S10.1    I=25;
        S10.2    print(I);
        }
    S11:endif
    S12:print(sum);
End
```

Relevant Slice

- The set of statements that were executed under a test case and did not affect the output, **but have the potential to affect the output produced by a test case**, is known as the **relevant slice** of the program.
- For example, consider the example given in previous figure.



Relevant Slice cont ...

- Statements S3 and S6 have the potential to affect the output, if modified.
- On the basis of relevant slices, we can prioritize the test cases.
- This technique is helpful for prioritizing the regression test suite which saves time and effort for regression testing.



Prioritization Based on Requirements

- This technique is used for prioritization of the system test cases.
- The system test cases also become too large in number, as this testing is performed on many grounds. Since system test cases are largely dependent on the requirements, **the requirements** can be analysed to prioritize the test cases.
- This technique does not consider all the requirements on the same level. Some requirements are more important and critical as compared to others, and these test cases having more weight are executed earlier.

PORT

- Hema srikanth et al. have proposed a requirements based test case prioritization technique.
- It is known as PORT (prioritization of requirements for test).
- They have taken the following four factors for analyzing and measuring the criticality of requirements.



Factors for analyzing & measuring criticality of requirements.

- **Customer-assigned priority of requirements:** Based on priority, the customer assigns a weight (on scale of 1 to 10) to each requirement.
- **Requirement volatility:** This is a rating based on the frequency of change of a requirement.
- **Developer-perceived implementation complexity:** The developer gives more weight to a requirement which he thinks is more difficult to implement.
- **Fault proneness of requirements:** This factor is identified based on the previous versions of system. If a requirement in an earlier version of the system has more bugs, i.e. it is error-prone, then this requirement in the current version is given more weight. This factor cannot be considered for a new software.

Prioritization Factor Value

- Based on these four factor values, a prioritization factor value (PFV) is computed as given below.

$$PFVi = \sum(FVij \times FWj)$$

where $FVij$ = Factor value which is the value of factor j corresponding to requirement i, and FWj = Factor weight which is the weight given to factor j.

- PFV is then used to produce a prioritized list of system test cases.

Measuring Effectiveness of a prioritized test suite

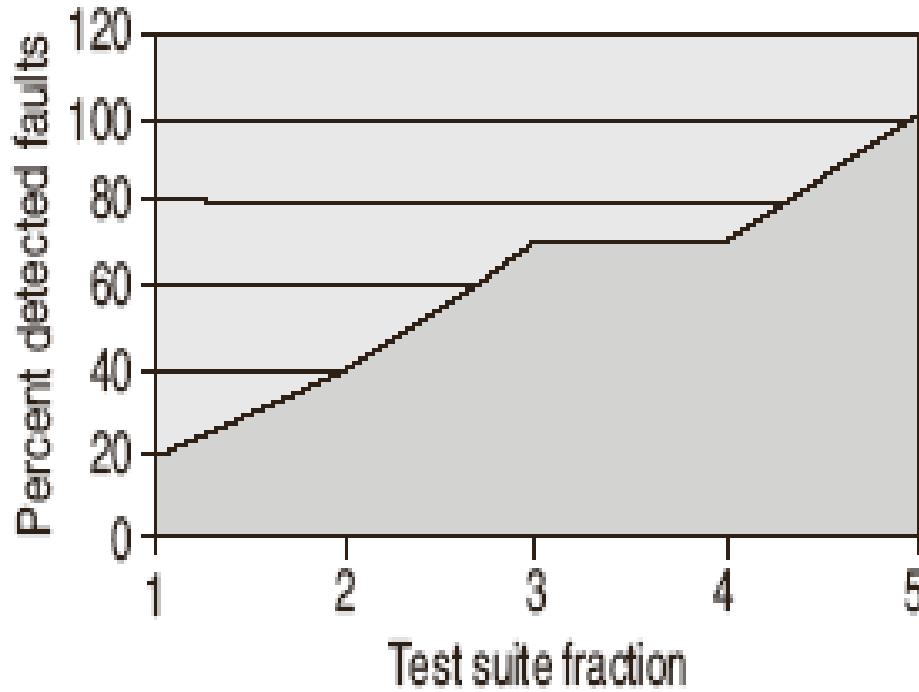
- Elbaum et al. developed **APFD** (average percentage of faults detection) metric that measures the weighted average of percentage of faults detected during the execution of a test suite.
- Its value ranges from 0 to 100, where a higher value means a faster fault-detection rate.

APFD Metric

APFD is a metric to detect how quickly a test suite identifies the faults. It is defined as follows:

$$APFD = 1 - ((TF_1 + TF_2 + \dots + TF_m) / nm) + 1/2n$$

where TF_i is the position of the first test in test suite T that exposes fault i , m is the total number of faults exposed in the system or module under T and n is the total number of test cases in T .



Example

Consider a program with 10 faults & test suite of 10 test cases, as shown in below table.

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
F1					x			x		
F2		x	x	x		x				
F3	x	x	x	x			x	x		
F4						x				x
F5	x		x		x	x		x		x
F6					x	x	x		x	
F7									x	
F8		x		x		x			x	x
F9	x									x
F10			x	x				x	x	

- Let us consider the order of test suite as (T1,T2, T3,T4,T5,T6,T7,T8,T9,T10).
- Then calculate the APFD for this program as follows:

$$APFD = 1 - \frac{(5+2+1+6+1+5+9+2+1+3)}{10*10} + \frac{1}{2 \times 10}$$

$$= 0.65 + 0.05$$

$$= 0.7$$

- All the bugs detected are not of the same level of severity.
- One bug may be more critical as compared to others.
- Moreover, the cost of executing the test cases also differs.
- One test case may take more time as compared to others.
- Thus, APFD does not consider the **severity level of the bugs** and **the cost of executing the test cases in a test suite**.

Cost-cognizant APFD

- Elbaum *et al.* modified their APFD metric and considered these two factors to form a new metric which is known as *cost-cognizant* APFD and denoted as **APFDc**.
- In APFDc, the total cost incurred in all the test cases is represented on x-axis and the total fault severity detected is taken on y-axis.
- Thus, it measures the unit of fault severity detected per unit test case cost.

Summary

- Discussed in detail the following types of test case prioritization techniques.
 - Risk-Based Prioritization
 - Prioritization Based on Operational Profiles
 - Prioritization using Relevant Slices
 - Prioritization Based on Requirements
- Discussed a metric, called **APFD**, to measure the effectiveness of the prioritization technique.
- Also, discussed a variance of APFD, called *cost-cognizant* APFD, which is denoted as **APFDc**.

References

- I. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter – 12), Second Edition, Oxford University Press, 2016.



Thank you



Efficient Test Suite Management

cont ...

Prof. Durga Prasad Mohapatra
Professor
Dept. of CSE, NIT Rourkela



Overview of last class

- Discussed a technique for prioritizing system test cases based on requirements, proposed by **Srikanth et. al.**
- This technique is known as PORT (prioritization of requirements for test).
- They have considered the following four factors for analyzing and measuring the criticality of requirements.

Overview of last class

cont ...

- **Customer-assigned priority of requirements:** Based on priority, the customer assigns a weight (on scale of 1 to 10) to each requirement.
- **Requirement volatility:** This is a rating based on the frequency of change of a requirement.
- **Developer-perceived implementation complexity:** The developer gives more weight to a requirement which he thinks is more difficult to implement.
- **Fault proneness of requirements:** This factor is identified based on the previous versions of system. If a requirement in an earlier version of the system has more bugs, i.e. it is error-prone , then this requirement in the current version is given more weight.This factor cannot be considered for a new software.

Overview of last class cont ...

- Based on these four factor values, a prioritization factor value (PFV) is computed as given below.

$$PFVi = \sum(FVij \times FWj)$$

where $FVij$ = Factor value is the value of factor j corresponding to requirement i, and FWj = Factor weight is the weight given to factor j.

- PFV is then used to produce a prioritized list of system test cases.

An Approach for Prioritization of Regression Test Cases

Kavitha & Kumar proposed an approach for prioritization of regression test cases by considering the following factors:

1. Customer-assigned priority of requirements
2. Changes in requirements
3. Developer-perceived code implementation complexity
4. Fault impact of requirements
5. Completeness
6. Traceability
7. Execution time

Based on these factors, a weightage is assigned to each test case and the test cases are prioritized according to the weightages.

Another Approach for Test Case Prioritization Based on Requirements

Kumar & Chauhan proposed a hierarchical test case prioritization approach, where prioritization is performed at three levels:

- Prioritize the requirements on the basis of 12 factors by assigning a weightage to each requirement.
- Map the prioritized requirements to their corresponding modules to get the prioritized modules.
- Rank (prioritize) the test cases of the prioritized modules for execution.

Prioritization of requirements

- The process of prioritization of requirements is performed on the basis of 12 factors.
- These factors are in accordance with every phase of SDLC.
- All these factors have been assigned a priority value between 0 and 10.
- These priority values are assigned by various stakeholders of the project.
- Table in next slide shows these factors.

Factors considered for requirement prioritization

S.NO	Factors	Phase of SDLC	Priority Value Assigned by
1	Requirement volatility	Requirement analysis	Customer
2	Customer assigned priority	Requirement analysis	Customer
3	Implementation complexity	Design	Developer
4	Fault proneness of requirements	Design	Developer
5	Developer assigned priority	Requirement analysis	Developer
6	Show stopper requirements	Design	Developer
7	Frequency of execution of requirements	Requirement analysis	Developer
8	Expected faults	Coding	Developer
9	Cost	Requirement analysis	Analyst
10	Time	Requirement analysis	Analyst
11	Penalty	Requirement analysis	Customer
12	Traceability	Testing	Tester



Requirement Prioritization Factor Value

- For each requirement, based on these 12 factors, a Requirement Prioritization Factor Value (RPFV) is calculated using below Eq.

$$RPFV = \sum_{j=1}^n (pfvalue_{ij} X pfweight_j)$$

- i represents the number of requirements, & j represents the number of factors
- RPFV represents the prioritization factor value for a requirement which is the summation of the product of priority value of a factor and the project factor weight.

Requirement Prioritization Factor Value

cont ...

- pfvalue is the value of factor for the i^{th} requirement
- pfweight is the factor weight for the j^{th} factor for a particular project.
- By using the previous Eq. the weight prioritization factor RPFV for every requirement can be calculated.
- The table in next slide shows the prioritization of 4 sample requirements on the basis of RPFV for each requirement.

Requirements prioritization

Factor	R1	R2	R3	R4	Weight factor
Customer assigned priority	8	10	9	9	0.02
Developer assigned priority	8	9	9	8	0.08
Requirements volatility	3	0	0	2	0.1
Fault proneness	0	0	0	0	0.15
Expected faults	2	3	4	2	0.10
Implementation complexity	3	4	5	3	0.10
Execution frequency	5	10	9	6	0.05
Traceability	0	0	0	0	0.05
Show stopper requirements	0	9	6	0	0.2
Penalty	1	4	3	3	0.05
Time	3	6	5	4	0.05
Cost	4	7	6	6	0.05
RPFV	2.25	4.77	4.15	2.47	1.0

Inference

- In this table, R2 has the highest RPFV among all requirements.
- So, the prioritization order of these requirements is R2, R3, R4, and R1.
- The value of RPFV depends on the values of pfvalue and pfweight.

Prioritization of modules

- It is a process of mapping between prioritized requirements and their corresponding modules.
- If there is more than one module then the modules are prioritized.
- It uses criteria like Cyclometric complexity and non-dc path.

Prioritization of modules cont ...

- Definition-clear path (dc-path): A dc-path with respect to a variable v is a path between the definition node and the usage node s.t. no other node in the path is a defining node of variable v . Non-dc paths are more error prone.
- The test cases of higher priority modules are prioritized first and executed.
- For each module, a module prioritize value (MPV) is calculated by adding cyclometric complexity and the number of non-dc paths.

Example

Factors	M1	M2	M3	M4
Cyclomatic complexity	8	4	4	5
Non-dc path	7	5	6	3
MPV	15	9	10	8

Example

- The table shows the prioritization of 4 sample modules on the basis of MPV for each module.
- The order of prioritization of modules on the basis of MPV is M1, M3, M2 and M4.

Test case prioritization process

- It is used to prioritize and schedule the test cases corresponding to prioritized modules.
- Some weight factors are used for test case prioritization such as
 - test case complexity,
 - requirements coverage,
 - dependency of the test cases and
 - test impact.



Test Case Complexity

- It shows how difficult it is to execute a test case.
- It also shows how much effort is required to execute the test case.
- This factor is assigned a value between 1 and 10.

Requirements Coverage

- It shows how many requirements are covered by executing the test case.
- This factor is scaled between 1 to 10.
- A higher value shows maximum requirements are covered by the test case.
- Higher the number of requirements coverage, higher is the priority of the test case to be executed first.

Dependency

- It shows the dependency of test cases on some pre-requisites.
- It shows how many pre-requisites are required for each test case before execution of the test case.
- This factor is assigned a value between 1 and 10.

Test Impact

- It is the most critical factor in test case prioritization.
- It shows the impact of test case on a system if it is not executed.
- This factor assesses importance of the test case.
- This factor is assigned a value between 1 and 10.

Test Case Weight Prioritization

- TCWP is calculated as follows

$$\text{TCWP} = \sum_{j=1}^n (f\text{value } ij \times f\text{weight } j)$$

where,

- TCWP = weight prioritization for each test case calculated from the four factors.
- fvalue = value assigned to each test case.
- fweight = weight assigned to each factor.

Test Case Weight Prioritization cont ...

- After calculating the value of each test case, test cases are ordered by TCWP such that maximum TCWP gives a test case the highest priority and executed.
- Suppose, a set of four test cases TC1,TC2,TC3, and TC4 are to be prioritized. For these test cases TCWP is calculated by using the above equation and are prioritized on the basis of values of TCWP.

Example

S. No.	Factor	TC1	TC2	TC3	TC4	Weight
1	Test impact	4	8	7	9	0.4
2	Test case complexity	8	7	5	9	0.3
3	Requirement coverage	6	2	4	4	0.2
4	Dependency	7	6	6	8	0.1
	TCWP	5.90	6.30	5.70	7.90	1.0

So, the final prioritized order of test cases is TC4, TC2, TC1, TC3

Reference

- I. Naresh Chauhan, Software Testing: Principles and Practices, Second Edition, (Chapter 12), Oxford University Press, 2018.



Thank you



Efficient Test Suite Management

cont...

Dr. Durga Prasad Mohapatra
Professor
Dept. of CSE, NIT Rourkela

Data Flow Based Test Case Prioritization

- This is a white box testing technique used to detect improper use of data values due to coding errors.
- Data usage for a variable affects the white box testing and thereby the regression testing.
- If the prioritization of regression test suite is based on this concept, the rate of detection of faults will be high and critical bugs can be detected earlier.

Data Flow Based Test Case Prioritization cont...

J.Rummel et al. proposed an approach to regression test prioritization that leverages the all-du's (definition-use) test adequacy criterion that focuses on the definition and use of variables within the program under test.

Data Flow Based Test Case Prioritization cont...

- Kumar et al. proposed an approach for test case prioritization using *du* path as well as definition clear (dc) paths.

Idea: *du* paths which may not be *dc*, may be very problematic as non-*dc* paths may be subtle source of errors.

Data Flow Based Test Case Prioritization cont...

- In another work, it was indicated that for testing modified programs, the prioritized test suite for original program may not work because after modification, new *du* paths may get introduced and some of these *du* paths may also not be definition clear (dc).
- These paths may also be more prone to errors.

Data Flow Based Test Case Prioritization cont...

- So, a list of all such newly introduced non-dc paths is prepared and test cases corresponding to these paths at the highest priority are placed in the test suite for a modified program. This set of test cases is referred to as '**Set-1**'.
- Because of modification in the program, some existing dc paths may become *non-dc*. The set of test cases corresponding to these paths is taken at the next priority and this set of test cases may be referred as '**Set-2**'.

Data Flow Based Test Case Prioritization cont...

- Further, there may be some *non-dc* paths of the original program, which are still *non-dc* after modification of the program. The set of test cases for such paths may be referred to as '**Set-3**'.
- Finally, there are some test cases which do not cover non-dc paths, i.e., these test cases cover dc-paths. This may be referred to as '**Set-4**'.
- Using these 4 concepts (sets), an algorithm has been designed for prioritizing the test suite.

Algorithm for prioritizing the test suite

Step 1:

If a test case covers more number of paths in Set-1

Then place it at the highest priority.

Step 2:

If 2 test cases (in Step1)cover an equal number of paths in Set-1

Then the test case which covers more number of paths in Set-2 in the modified program is placed at the next priority.

Step 3:

If 2 test cases (in Step 2) cover equal number of paths in Set-2 in the modified program,

Then the test case which covers maximum number of lines of code in the modified program is placed at the next priority.



Module Coupling Slice-Based Test Case Prioritization

- This technique is based on dependence and coupling between the modules, and proceeds in 2 steps.

Steps for Module Coupling Slice-Based Test Case Prioritization

Step 1: Find the highly affected module whenever there is a change in a module.

Step 2: Prioritize the test cases of the affected module identified in Step 1.

- In both steps, coupling information between modules is considered.

Module Coupling Slice-Based Test Case Prioritization

cont...

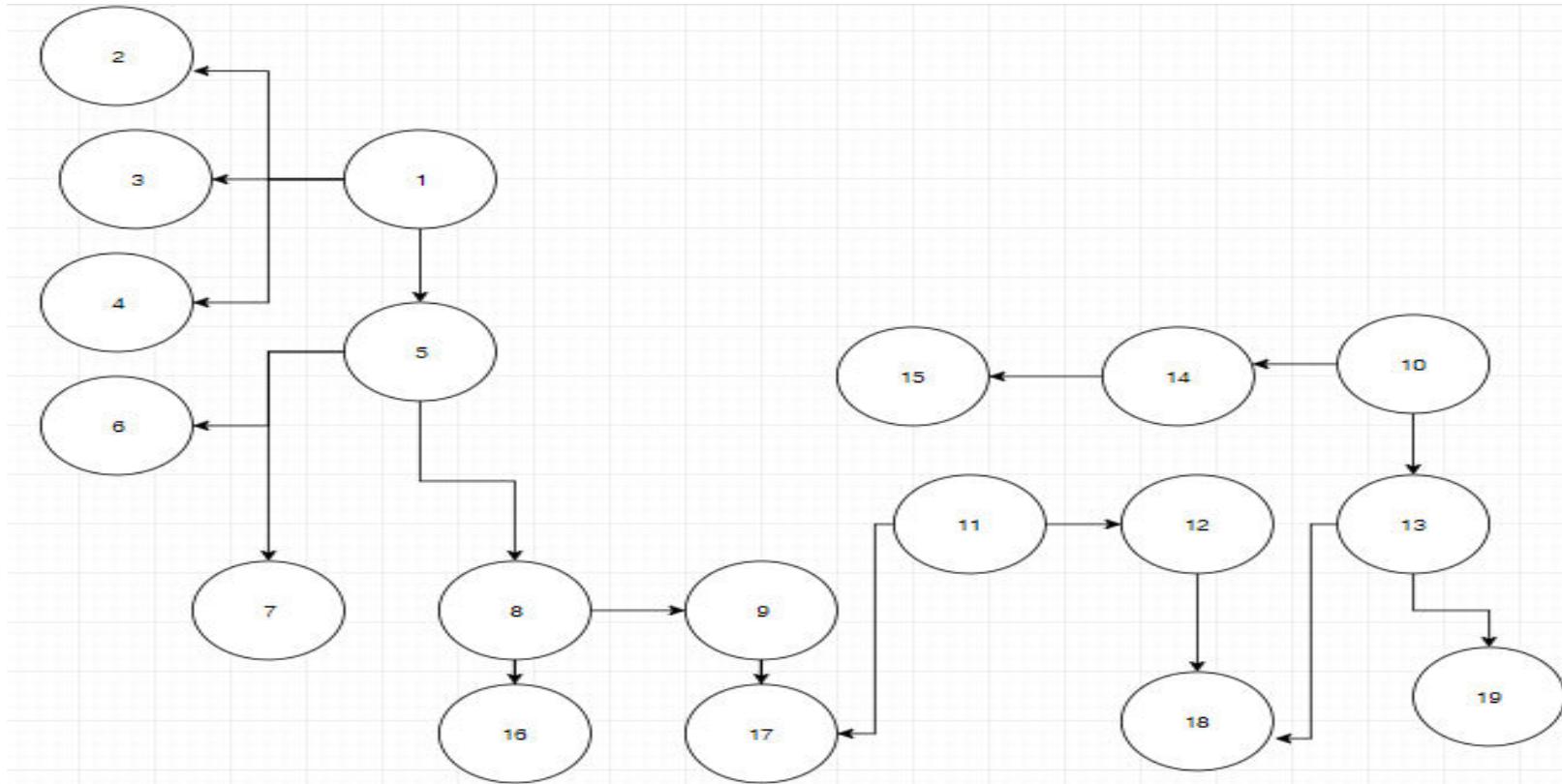
- When there is a change in a module, there will be some effect on other modules which are coupled with this module.
- Based on the coupling information between the modules, the highly affected module can be identified.



Module-Coupling Effect

- The effect is worse if there is high coupling between the modules causing high probability of errors. This is called as *Module-Coupling Effect*.
- If regression test case prioritization is done based on this module coupling effect, there will be high percentage of detecting **critical errors** that have been propagated to other modules due to any change in a module.

Example Call Graph





Module-Coupling Effect cont...

- Modules 17 and 18 are being called by multiple modules.
- If there is any change in these modules 17 & 18, then modules 9,11 and 12,13 will be affected respectively.
- If there is no prioritization, then according to regression testing, all test cases of all the affected modules will be executed, thus increasing the testing time and effort.
- If the coupling type between modules is known, then a prioritization scheme can be developed based on this coupling information.



Module Coupling Slice-Based Test Case

Prioritization cont...

- Modules having worst type of coupling will be prioritized over other modules and their test cases.
- Module dependence can be identified by coupling and cohesion.
- A quantitative measure of the dependence of modules will be useful to find out the stability of the design.
- The work is based on the premise that different values can be assigned for various types of module coupling and cohesion.

Module Coupling Slice-Based Test Case Prioritization cont...

Table 1: Coupling types and their values

Coupling Types	Value
Content	0.95
Common	0.70
External	0.60
Control	0.50
Stamp	0.35
Data	0.20

Module Coupling Slice-Based Test Case Prioritization

cont...

Table 2: Cohesion types and their values

Cohesion Types	Value
Coincidental	0.95
Logical	0.40
Temporal	0.60
Procedural	0.40
Communicational	0.25
Sequential	0.20
Functional	0.20

Module Coupling Slice-Based Test Case Prioritization

cont...

- A matrix can be obtained by using the 2 tables, which gives the dependence among all the modules in a program.
- This dependence matrix describes the probability of having to change module i, given that module j has been changed.
- Module Dependence Matrix is derived using the following 3 steps:

Steps for Module Dependence Matrix

Step I:

- (i) Determine the coupling among all the modules in the program.
- (ii) Construct an $m \times m$ coupling matrix, where m is the number of modules. Using Table 1, fill the elements of matrix. Element C_{ij} represents coupling between modules i and j .
- (iii) This matrix is symmetric, i.e. $C_{ij} = C_{ji}$ for all i and j .
- (iv) Elements on the diagonal are all 1 ($C_{ii} = 1$ for all i).

Steps for Module Dependence Matrix cont...

- **Step 2:**
 - (i) Determine strength of each module in the program.
 - (ii) Using Table 2, record the corresponding numerical values of cohesion in module cohesion matrix (**S**).

Steps for Module Dependence Matrix cont...

- **Step 3:**

- (i) Construct the Module Dependence Matrix \mathbf{D} using the following equation:

$$D_{ij} = 0.15 (S_i + S_j) + 0.7 C_{ij}, \text{ where } C_{ij} \neq 0,$$
$$= 0, \text{ where } C_{ij} = 0 \quad D_{ii} = 1 \text{ for all } i.$$

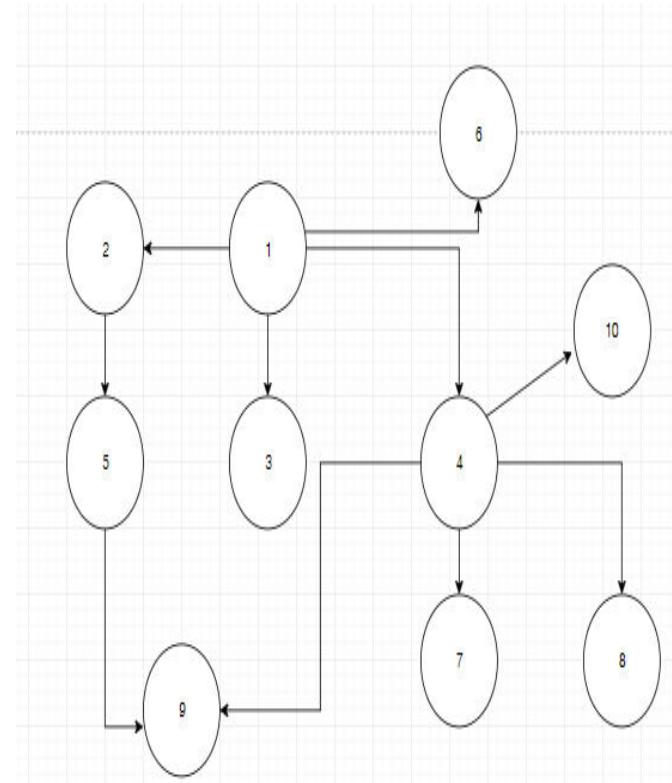
Prioritization of modules can be done by comparing non zero entries of matrix \mathbf{D} .

Steps for Module Dependence Matrix cont...

- For example, if module i has been modified, find all the existing parent modules(j,k,l,...) of that changed module(i) and after that compare first-order dependence matrix entries for particular links, like i-j, i-k,i-l and so on.
- The link having the highest module dependence matrix value will get the highest priority and the link with the lowest module dependence matrix value will get the lowest priority.

Example

A Software consists of 10 modules. The coupling and cohesion information of these modules are given in tables shown in next slide. Let us find out the badly affected module in this software.



Call Graph

Example

Coupling information

Type of Coupling	No. of Modules in Relation	Examples
Data Coupling	3	1-2,1-4,1-6
Stamp Coupling	1	1-3
Control Coupling	4	4-7,4-8, 4-9,4-10
Common Coupling	2	2-5,5-9
Message (Content) Coupling	0	-

Cont...

Cohesion information

Module Number	Cohesion Type
1	Coincidental
2	Functional
3	Communicational
4	Logical
5	Procedural
6	Functional
7	Functional
8	Functional
9	Functional
10	Functional

Example cont...

By using coupling values among different modules, a module coupling matrix is given below:

1.0	0.2	0.35	0.2	0.0	0.2	0.0	0.0	0.0	0.0
0.2	1.0	0.0	0.0	0.70	0.0	0.0	0.0	0.0	0.0
0.35	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.2	0.0	0.0	1.0	0.0	0.0	0.50	0.50	0.50	0.50
0.0	0.2	0.0	0.0	1.0	0.0	0.0	0.0	0.70	0.0
0.2	0.2	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	0.0	0.2	0.50	0.0	0.0	1.0	0.0	0.0	0.0
0.0	0.0	0.0	0.50	0.0	0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.0	0.50	0.70	0.0	0.0	0.0	1.0	0.0
0.0	0.0	0.0	0.50	0.0	0.0	0.0	0.0	0.0	1.0

Example

cont...

By using the value of cohesion among different modules a Cohesion Matrix (**S**) is designed as shown below:

0.95	0.2	0.25	0.4	0.4	0.2	0.2	0.2	0.2	0.2
------	-----	------	-----	-----	-----	-----	-----	-----	-----

Example

cont...

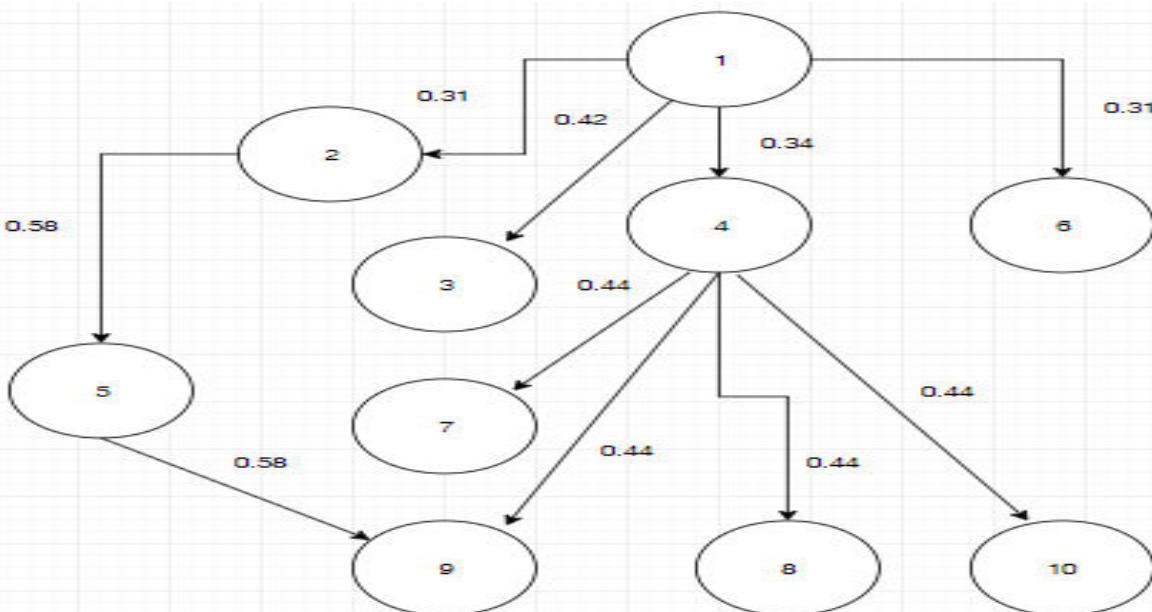
A Module Dependence Matrix (**D**) is designed, (using the previous equation).

1.0	0.31	0.42	0.34	0.0	0.31	0.0	0.0	0.0	0.0
0.31	1.0	0.0	0.0	0.58	0.0	0.0	0.0	0.0	0.0
0.42	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.34	0.0	0.0	1.0	0.0	0.0	0.44	0.44	0.44	0.44
0.0	0.58	0.0	0.0	1.0	0.0	0.0	0.0	0.58	0.0
0.31	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.44	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.44	0.0	0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.0	0.44	0.58	0.0	0.0	0.0	1.0	0.0
0.0	0.0	0.0	0.44	0.0	0.0	0.0	0.0	0.0	1.0

Example

cont...

From the module dependence values obtained from call graph as shown below and from Module Dependence matrix, we conclude that change in module 4 propagates to modules 7,8,9 and 10.



Example

cont...

- Modules 7,8,9 and 10 have the same module dependence value(0.44), so the order of prioritization of test cases for these modules is the same.
- Similarly, the change in module 1 propagates to modules 2,3,4 and 6.
- The module dependence values for these modules show that module 3 is the more affected module when compared to modules 2,4 and 6.
- So, the test cases for module 3 have to be prioritized first as compared to modules 2,4 and 6.

Module Coupling Slice-Based Test Case Prioritization

cont...

- After identifying the highly affected module due to a change in a module, there is a need to execute the test cases corresponding to this affected module.
- However, there may be a large number of test cases in this module.
- Therefore, there is a need to prioritize these test cases so that critical bugs can be found easily.
- Based on the coupling information between the changed module and the affected module, a **coupling slice** can be prepared that helps in prioritizing the test cases.

Module Coupling Slice-Based Test Case Prioritization

cont...

- All the statement numbers present in the execution history of a program comprise the execution slice of a program.
- Coupling information can be helpful to decide which variables are affected in the caller module.
- Depending upon this information the statement numbers of affected variables can be found. This may be called a **coupling slice**.

Module Coupling Slice-Based Test Case Prioritization cont...

- At last, these statement numbers are matched in the execution slice.
- The test cases for these statements will be given high priority and executed first.

THANK YOU

Examples on Equivalence Class Partition-Based Testing

**Durga Prasad Mohapatra
NIT Rourkela**

Example-1: Intersection of two straight lines

- Q. Design B-B test cases for the following program:

The program checks if 2 straight lines intersect and print their point of intersection.

Ans: The straight line may be represented as: $y = mx + c$

Straight lines are given in the form of (m_1, c_1) and (m_2, c_2) .

Equivalent classes are as follows:

- ✓ Case 1: The lines are parallel i.e. $m_1 = m_2$

So points are (1, 2) and (1, 5)

- ✓ Case 2: Coincident lines i.e. $m_1 = m_2$ and $c_1 = c_2$

And points are (2, 3) and (2, 3)

Example-1 Contd.

- ✓ Case 3: Lines intersecting at one point i.e. $m_1 \neq m_2$

The points may be (2, 5) and (3, 6).

So there are 3 valid equivalent classes. You may include one more valid equivalent classes

There are no boundary values here.

So, the resultant test suite is:

$\{(1, 2), (1, 5)\}, \{(2, 3), (2, 3)\} \{(2, 5), (3, 6)\}.$

Example-2: Solving quadratic equations

- Q. Design B-B test cases for the following program:
The program solves quadratic equations of the form

$$ax^2 + bx + c$$

It accepts 3 floating point values as input and gives the roots,
e.g. The input may be (7.7, 3.3 and 4.5).

Ans: Equivalent Classes are as follows:

- I. $b^2 = 4ac$ inputs are: a= 2.0, b=4.0 and c=2.0
- II. $b^2 > 4ac$ inputs are a=2.0, b= 5.0 and c=2.0
- III. $b^2 < 4ac$ inputs are a=2.0, b= 3.0 and c=2.0
- IV. Invalid Equation inputs are a=0, b= 0 and c=10.0

Write down the test cases accordingly.

Example-3: Solving linear equations

- Q. Design B-B test cases for the following program:

The program solves linear equations in upto 10 independent variables

$$\text{e.g. } 5x + 6y + z = 5$$

$$10x + 2y + 5z = 20$$

.....

.....

Example-3 Contd.

- Ans: Equivalent Classes are as follows:

I. Valid Equivalent Classes

- Many Solution ($\# \text{ var} < \# \text{eqns}$)
- No Solution ($\# \text{ var} > \# \text{eqns}$)
- Unique Solution ($\# \text{ var} = \# \text{eqns}$)

II. Invalid Equivalent Classes

- Too many Variables ($\# \text{ var} > 10$)
- Invalid Equation ($\# \text{ var} = 0$)

Write down the test cases accordingly.

Example-4: Intersection of two circles

- Q. Design B-B test cases for the following program:

The program finds the points of intersection of 2 circles

- Ans: Equivalent Classes are as follows:

- I. $r_1+r_2 <$ distance between (x_1, y_1) and (x_2, y_2) i.e. not intersecting
- II. $r_1+r_2 =$ distance i.e. touching at 1 point
- III. $r_1+r_2 >$ distance i.e. intersecting at 2 points
- IV. Distance=0 and $r_1 = r_2$ i.e. overlapping
- V. Distance=0 and $r_1 \neq r_2$
- VI. Invalid circles

Write down the test cases accordingly.

Example-5: Query Book Option in LIS

Q. Design B-B test cases for the option **Query Book** using a Keyword (e.g. Author name or title).

Ans: The equivalent classes and the corresponding test cases are as follows:

- Not present in catalogue (SE, not present)
- Present in catalogue (SE, present, 15 issued, not available)
- Present in catalogue (SE, present, 10 issued, 5 available)

References

1. Fundamentals of Software Engineering,
Rajib Mall, Fifth Edition, PHI, 2018.

Thank You

Black-box testing techniques

Durga Prasad Mohapatra
Professor
Dept. of CSE
NIT Rourkela

Black-box Testing

- Test cases are designed using only **functional specification** of the software:
 - without any knowledge of the internal structure of the software.
- For this reason, black-box testing is also known as **functional testing**.

Errors detected by black box testing

- Black-box testing attempts to find errors in the following categories:
 1. To test the modules independently .
 2. To test the functional validity of the software so that incorrect or missing functions can be recognized.
 3. To look for interface errors. □

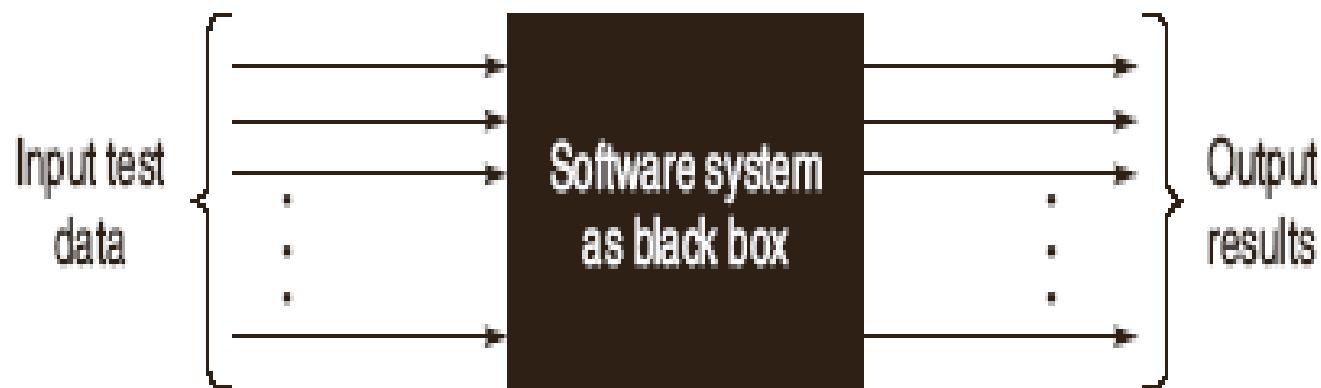
Errors detected by black box testing

4. To test the system behavior and check its performance
5. To test the maximum load or stress on the system.
6. To test the software such that the user/customer accepts the system within defined acceptable limits.

Black-box Testing Techniques

- There are many approaches to design black box test cases:
 - Equivalence class partitioning
 - Boundary value analysis
 - State table based testing
 - Decision table based testing
 - Cause-effect graph based testing
 - Orthogonal array testing
 - Positive-negative testing

Black-box Testing



Equivalence Class Partitioning

- Input values to a program are partitioned into equivalence classes.
- Partitioning is done such that:
 - program behaves in similar ways to every input value belonging to an equivalence class.

Why define equivalence classes?

- Test the code with just one representative value from each equivalence class:
 - as good as testing using any other values from the equivalence classes.

Equivalence Class Partitioning

- How do you determine the equivalence classes?
 - examine the input data.
 - few general guidelines for determining the equivalence classes can be given

Equivalence Class Partitioning

- If the input data to the program is specified by a **range of values**:
 - e.g. numbers between 1 to 5000.
 - one valid and two invalid equivalence classes are defined.

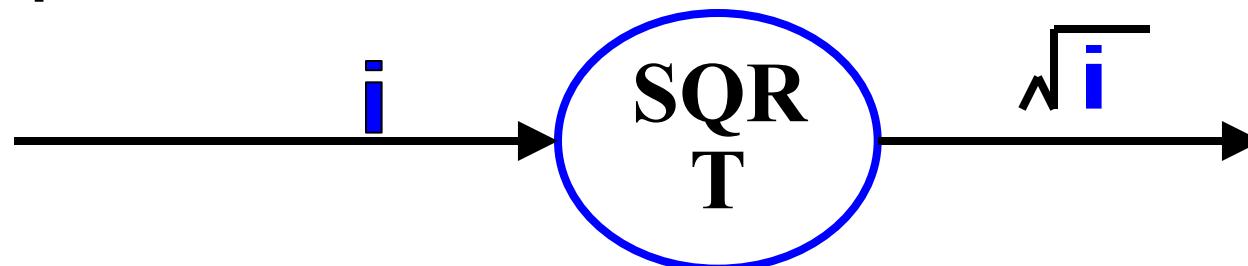


Equivalence Class Partitioning

- If input is an enumerated set of values:
 - e.g. {a,b,c}
 - one equivalence class for valid input values
 - another equivalence class for invalid input values should be defined.

Example - I

- A program reads an input value in the range of 1 and 5000:
 - computes the square root of the input number



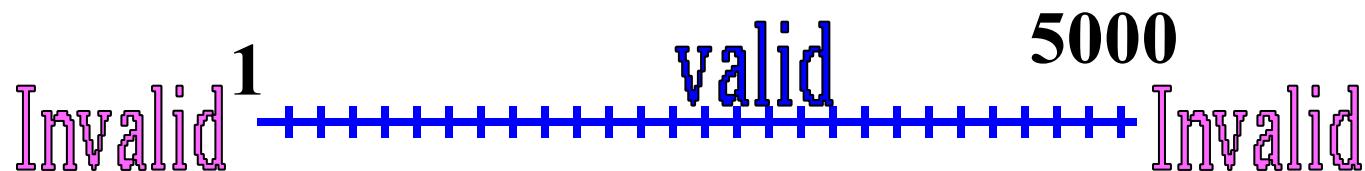
Example - I (cont.)

- There are three equivalence classes:
 - the set of negative integers,
 - set of integers in the range of 1 and 5000,
 - integers larger than 5000.



Example - I (cont.)

- The test suite must include:
 - representatives from each of the three equivalence classes:
 - a possible test suite can be:
 $\{-5,500,6000\}$.



Example - 2

- A program reads three numbers, A, B, and C, with a range [1, 50] and prints the largest number. Design test cases for this program using equivalence class testing technique.

Solution

I. First we partition the domain of input as valid input values and invalid values, getting the following classes:

- $I_1 = \{<A, B, C> : I \leq A \leq 50\}$
- $I_2 = \{<A, B, C> : I \leq B \leq 50\}$
- $I_3 = \{<A, B, C> : I \leq C \leq 50\}$
- $I_4 = \{<A, B, C> : A < I\}$

Solution

- I5 = { $\langle A, B, C \rangle : A > 50$ }
- I6 = { $\langle A, B, C \rangle : B < 1$ }
- I7 = { $\langle A, B, C \rangle : B > 50$ }
- I8 = { $\langle A, B, C \rangle : C < 1$ }
- I9 = { $\langle A, B, C \rangle : C > 50$ }

Solution

- Now the test cases can be designed from the above derived classes, taking
 - one test case from each class such that the test case covers maximum valid input classes, and
 - separate test cases for each invalid class.

Solution

- The test cases are shown below:

Test case ID	A	B	C	Expected result	Classes covered by the test case
1	13	25	36	C is greatest	I_1, I_2, I_3
2	0	13	45	Invalid input	I_4
3	51	34	17	Invalid input	I_5
4	29	0	18	Invalid input	I_6
5	36	53	32	Invalid input	I_7
6	27	42	0	Invalid input	I_8
7	33	21	51	Invalid input	I_9

Solution

2. We can derive another set of equivalence classes based on some possibilities for three integers, A, B, and C. These are given below:
- I1 = { $\langle A, B, C \rangle : A > B, A > C \}$
 - I2 = { $\langle A, B, C \rangle : B > A, B > C \}$
 - I3 = { $\langle A, B, C \rangle : C > A, C > B \}$

Solution

- I4 = { $\langle A, B, C \rangle : A = B, A \neq C \}$
- I5 = { $\langle A, B, C \rangle : B = C, A \neq B \}$
- I6 = { $\langle A, B, C \rangle : A = C, C \neq B \}$
- I7 = { $\langle A, B, C \rangle : A = B = C \}$

Solution

Test case ID	A	B	C	Expected Result	Classes Covered by the test case
1	25	13	13	A is greatest	I_1, I_5
2	25	40	25	B is greatest	I_2, I_6
3	24	24	37	C is greatest	I_3, I_4
4	25	25	25	All three are equal	I_7

Example - 3

A program determines the next date in the calendar. Its input is entered in the form of with the following range:

- $1 \leq mm \leq 12$
- $1 \leq dd \leq 31$
- $1900 \leq yyyy \leq 2025$

Example

- Its output would be the next date or an error message ‘invalid date.’ Design test cases using equivalence class partitioning method.

Solution

First, we partition the domain of input in terms of valid input values and invalid values, getting the following classes:

- $I1 = \{ \langle m, d, y \rangle : 1 \leq m \leq 12 \}$
- $I2 = \{ \langle m, d, y \rangle : 1 \leq d \leq 31 \}$
- $I3 = \{ \langle m, d, y \rangle : 1900 \leq y \leq 2025 \}$
- $I4 = \{ \langle m, d, y \rangle : m < 1 \}$

Solution

- $I_5 = \{<m, d, y> : m > 12\}$
- $I_6 = \{<m, d, y> : d < 1\}$
- $I_7 = \{<m, d, y> : d > 31\}$
- $I_8 = \{<m, d, y> : y < 1900\}$
- $I_9 = \{<m, d, y> : y > 2025\}$

Solution

- The test cases can be designed from the above derived classes,
 - taking one test case from each class such that the test case covers maximum valid input classes, and
 - separate test cases for each invalid class.
- The test cases are shown in next slide.

Solution

Test case ID	mm	dd	yyyy	Expected result	Classes covered by the test case
1	5	20	1996	21-5-1996	I_1, I_2, I_3
2	0	13	2000	Invalid input	I_4
3	13	13	1950	Invalid input	I_5
4	12	0	2007	Invalid input	I_6
5	6	32	1956	Invalid input	I_7
6	11	15	1899	Invalid input	I_8
7	10	19	2026	Invalid input	I_9

Example - 4

- A program takes an angle as input within the range [0, 360] and determines in which quadrant the angle lies. Design test cases using equivalence class partitioning method.

Solution

I. First, we partition the domain of input as valid and invalid classes as follows:

- I1 = {<Angle> : $0 \leq \text{Angle} \leq 360$ }
- I2 = {<Angle> : $\text{Angle} < 0$ }
- I3 = {<Angle> : $\text{Angle} > 360$ }

Solution

- The test cases designed from these classes are shown below:

Test Case ID	Angle	Expected results	Classes covered by the test case
1	50	I Quadrant	l_1
2	-1	Invalid input	l_2
3	361	Invalid input	l_3

Solution

2. The classes can also be prepared based on the output criteria as shown below:

- O1 = {<Angle>: First Quadrant, if $0 \leq \text{Angle} \leq 90\}$
- O2 = {<Angle>: Second Quadrant, if $91 \leq \text{Angle} \leq 180\}$
- O3 = {<Angle>: Third Quadrant, if $181 \leq \text{Angle} \leq 270\}$

Solution

- O4 = {<Angle>: Fourth Quadrant, if $270 \leq \text{Angle} \leq 360\}$ }
- O5 = {<Angle>: Invalid Angle};
- However, O5 is not sufficient to cover all invalid conditions this way. Therefore, it must be further divided into equivalence classes as shown in next slide:

Solution

- O₅₁={<Angle>: Invalid Angle, if Angle > 360}
- O₅₂={<Angle>: Invalid Angle, if Angle < 0}

Solution

- Now, the test cases can be designed from the above derived classes as shown below:

Test Case ID	Angle	Expected results	Classes covered by the test case
1	50	I Quadrant	O_1
2	135	II Quadrant	O_2
3	250	III Quadrant	O_3
4	320	IV Quadrant	O_4
5	370	Invalid angle	O_{51}
6	-1	Invalid angle	O_{52}

Summary

- Discussed the errors detected by black-box test testing techniques.
- Explained equivalence partitioning technique with some examples.

References

1. Rajib Mall, Fundamentals of Software Engineering, (Chapter – 10), Fifth Edition, PHI Learning Pvt. Ltd., 2018.
2. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter – 4), Second Edition, Oxford University Press, 2016.



Thank You



Integration Testing

Prof. Durga Prasad Mohapatra
Professor
Dept.of CSE, NIT Rourkela

Introduction

- In the modular design of a software system where the system is composed of different modules, integration is the activity of combining the modules together when all the modules have been prepared.
- Integration of modules is done according to the design of software specified earlier.
- Integration aims at constructing a working software system.
- But a working software demands full testing and thus, integration testing comes into the picture.

Introduction

cont ...

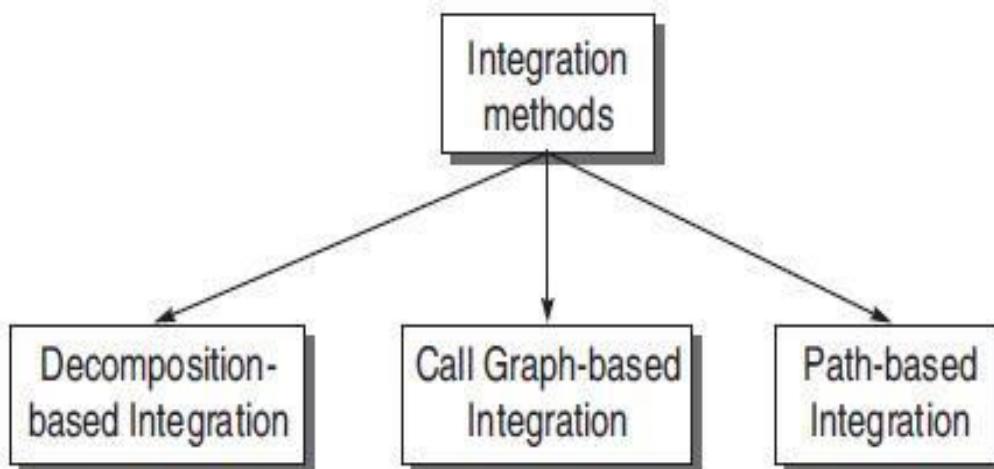
- Why do we need integration testing? When all modules have been verified independently, then why is integration testing necessary?
- As discussed earlier, modules are not standalone entities.
- They are a part of a software system which comprises of many interfaces. Even if a single interface is mismatched, many modules may be affected.

Thus, integration testing is necessary for the following reasons:

1. Integration testing exposes inconsistency between the modules such as improper call or return sequences.
2. Data can be lost across an interface.
3. One module when combined with another module may not give the desired result.
4. Data types and their valid ranges may mismatch between the modules.

Approaches for integration testing

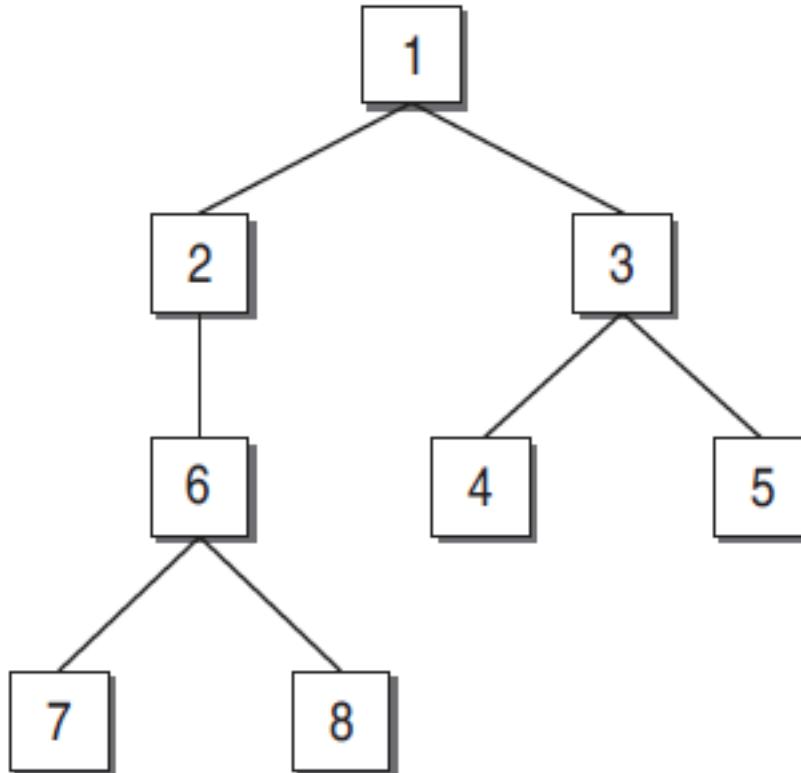
- Thus, integration testing focuses on bugs caused by interfacing between the modules while integrating them.
- There are three approaches for integration testing:



Decomposition-based integration

- Idea is based on decomposition into functional modules
- Functional decomposition is shown as a tree
- Classified into
 - Non-incremental
 - incremental

Integration Testing



- Non incremental Approach

- Also known as big-bang integration

- Discarded for the following reasons:

- Big-bang requires more work

- With increase in the number of modules, more number of drivers and stubs will be required to test the modules independently.

- Actual modules are not interfaced directly until the end of the software system.

- It will be difficult to localize the errors since the exact location of bugs cannot be found easily.

- Incremental Approach
 - It is beneficial for the following reasons:
 - Does not require many drivers and stubs.
 - Interfacing errors are uncovered earlier.
 - It is easy to localize the errors since modules are combined one by one, thus debugging is easier.
 - Incremental testing is a more thorough testing



Types of incremental integration testing

- I. Top-down integration
2. Bottom-up integration

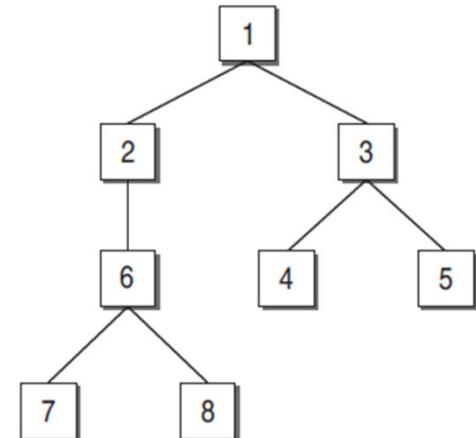


Top-down integration Testing

- The strategy in top-down integration is to look at the design hierarchy from top to bottom.
- Start with the high-level modules and move downward through the design hierarchy.
- Modules subordinate to the top module are integrated in the following two ways:
 - Depth first integration
 - Breadth first integration

Depth first integration

- In this type, all modules on a major control path of the design hierarchy are integrated first.
- In the example shown in below figure, modules 1, 2, 6, 7/8 will be integrated first. Next, modules 1, 3, 4/5 will be integrated.



Breadth first integration

- In this type, all modules directly subordinate at each level, moving across the design hierarchy horizontally, are integrated first.
- In the example shown in previous figure, modules 2 and 3 will be integrated first.
- Next, modules 6, 4, and 5 will be integrated. Modules 7 and 8 will be integrated last.

However, in practice, these two sequences of top-down integration cannot be used every time. In general, there is no best sequence, but the following guidelines can be considered:

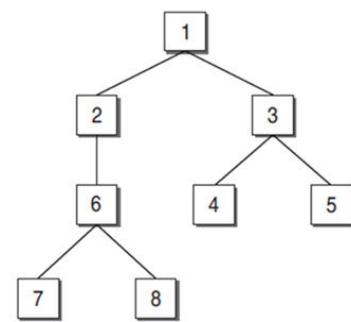
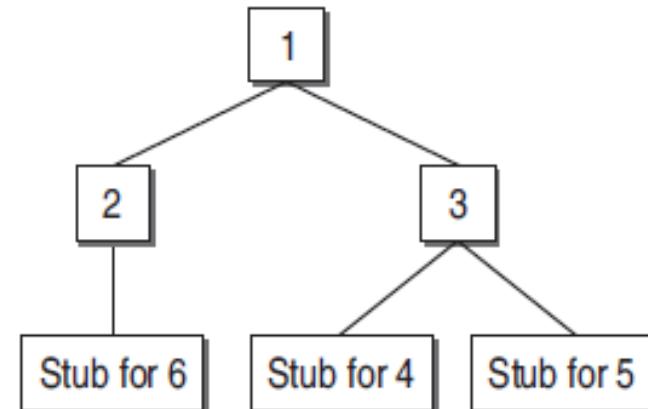
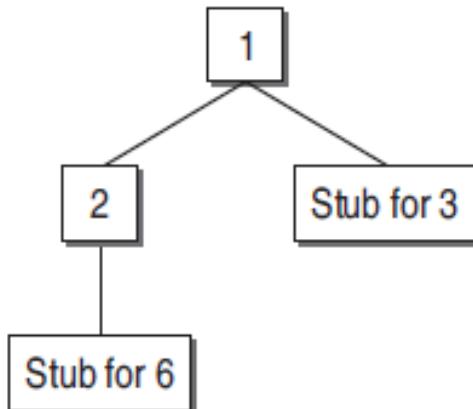
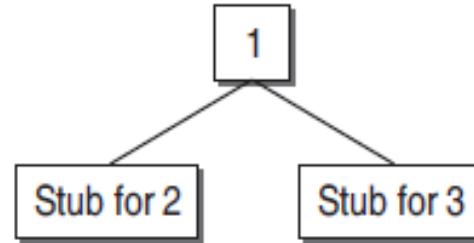
1. In practice, the availability of modules matters the most. The module which is ready to be integrated, will be integrated and tested first. We should not wait to test it according to depth first or breadth first sequence, but use the availability of modules.

-
2. If there are critical sections of the software, design the sequence such that these sections will be added and tested as early as possible. A critical section might be a complex module, a module with a new algorithm or a module suspected to be error prone.
3. Design the sequence such that the I/O modules are added as early as possible, so that all interface errors will be detected earlier.

Top-Down Integration Procedure

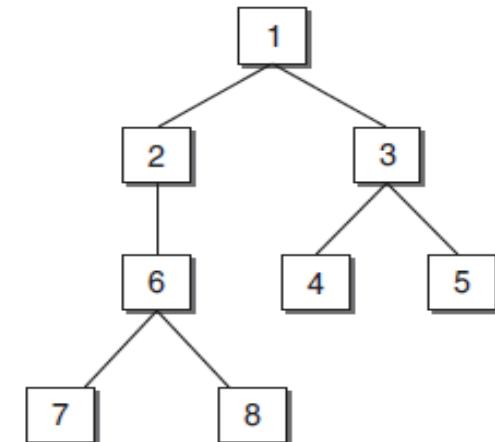
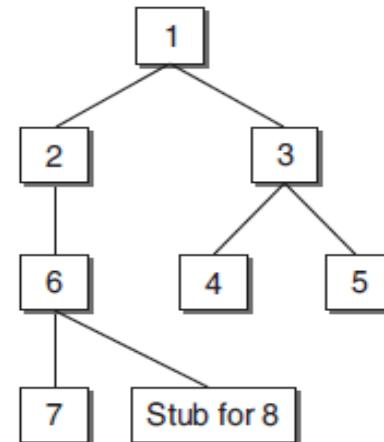
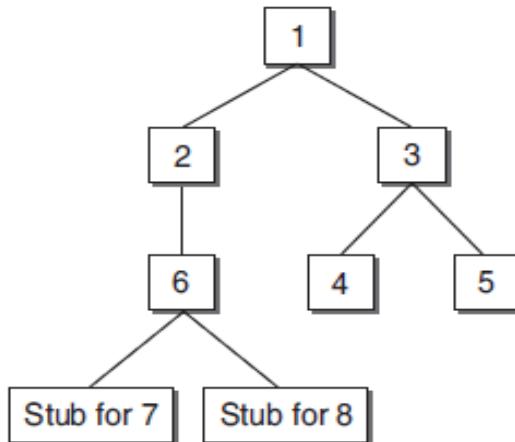
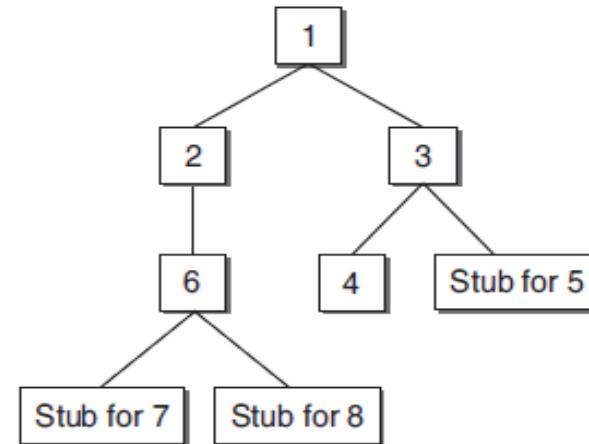
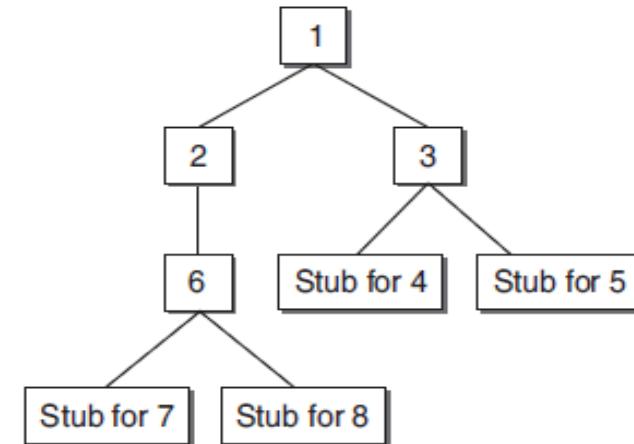
- I. Start with the top or initial module in the software. Substitute the stubs for all the subordinate modules of top module. Test the top module.
2. After testing the top module, stubs are replaced one at a time with the actual modules for integration.
3. Perform testing on this recent integrated environment.
4. Regression testing may be conducted to ensure that new errors have not appeared.
5. Repeat steps 2-4 for the whole design hierarchy.

Example



Example

cont...



Drawbacks of top-down integration

- Stubs must be prepared as required
- Stubs are often more complicated than they appear
- Before the I/O functions are added, the representation of test cases in stubs can be difficult

Bottom-up Integration Testing

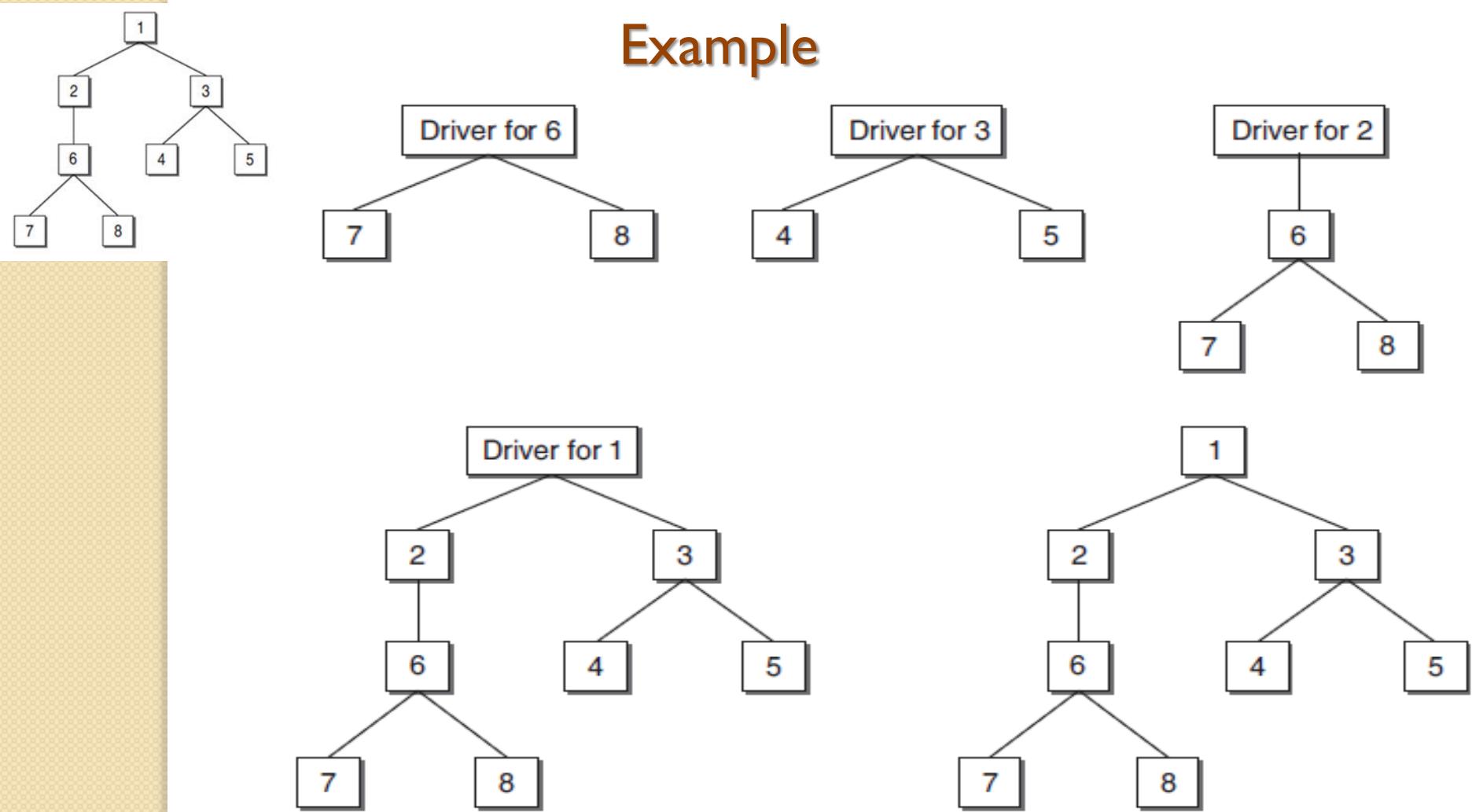
- The bottom-up strategy begins with the terminal or modules at the lowest level in the software structure. After testing these modules, they are integrated and tested moving from bottom to top level.
- Since the processing required for modules subordinate to a given level is always available, stubs are not required in this strategy.

- Bottom-up integration can be considered as the opposite of top-down approach.
- Unlike top-down strategy, this strategy does not require the architectural design of the system to be complete.
- Thus, bottom-up integration can be performed at an early stage in the developmental process.
 - It may be used where the system reuses and modifies components from other systems.

Steps in bottom-up integration

1. Start with the lowest level modules (modules from which no other module is being called), in the design hierarchy.
2. Look for the super-ordinate module which calls the module selected in Step 1. Design the driver module for this super-ordinate module.
3. Test the module selected in Step 1 with the driver designed in Step 2.
4. The next module to be tested is any module whose subordinate modules (modules it calls) have all been tested.
5. Repeat Steps 2 to 4 and move up in the design hierarchy.
6. Whenever, the actual modules are available, replace stubs and drivers with the actual one and test again.

Example



Comparison between top-down and bottom-up testing

Issue	Top-Down Testing	Bottom-Up Testing
Architectural Design	It discovers errors in high-level design, thus detects errors at an early stage.	High-level design is validated at a later stage.
System Demonstration	Since we integrate the modules from top to bottom, the high-level design slowly expands as a working system. Therefore, feasibility of the system can be demonstrated to the top management.	It may not be possible to show the feasibility of the design. However, if some modules are already built as reusable components, then it may be possible to produce some kind of demonstration.
Test Implementation	(nodes – 1) stubs are required for the subordinate modules.	(nodes – leaves) test drivers are required for super-ordinate modules to test the lower-level modules.

Practical Approach for Integration Testing

- There is no single strategy adopted for industry practice.
- For integrating the modules, one cannot rely on a single strategy.
- There are situations depending on the project in hand which will force to integrate the modules by combining top-down and bottom-up techniques.
- This combined approach is sometimes known as **sandwich integration testing**.

- Selection of an integration testing strategy depends on software characteristics and sometimes project schedules.
- In general, sandwich testing strategy that uses top-down tests for upper levels of the program structure with bottom-up tests for subordinate levels is the best compromise.
- The practical approach for adopting sandwich testing is driven by the following factors:

–Priority

- First putting together those subsystems with more important requirements.
- Follow top-down approach if the module has high level of control on its sub-ordinate modules.
- Modules with more user interfaces should be tested first, as they are more error prone.
- Module having high cyclomatic complexity should be tested first.

–Availability

- The module that is ready to be integrated. Will be tested first.

Pros and Cons of Decomposition Tech.

- Debugging is easy in decomposition based integration
- Better for monitoring the progress of integration.
- But, more effort is required as stubs and drivers are needed.



Integration Testing Effort

- The integration testing effort is computed as the number of test sessions.
- A test session is one set of test cases for a specific configuration.
- The total number of test sessions in a decomposition-based integration is computed as:
 - Number of test sessions = nodes - leaves + edges



Summary

- Discussed basics of different approaches for integration testing.
- Discussed decomposition-based integration in detail.
 - Big bang integration
 - Top-down integration
 - Breadth First Integration
 - Depth First Integration
 - Bottom-up integration
 - Sandwich integration
- Explained how to compute the integration testing effort .

References

1. Rajib Mall, Fundamentals of Software Engineering, (Chapter – 10), Fifth Edition, PHI Learning Pvt. Ltd., 2018.
2. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter – 7), Second Edition, Oxford University Press, 2016.



Thank You



Integration Testing cont ...

Prof. Durga Prasad Mohapatra
Professor
Dept.of CSE, NIT Rourkela

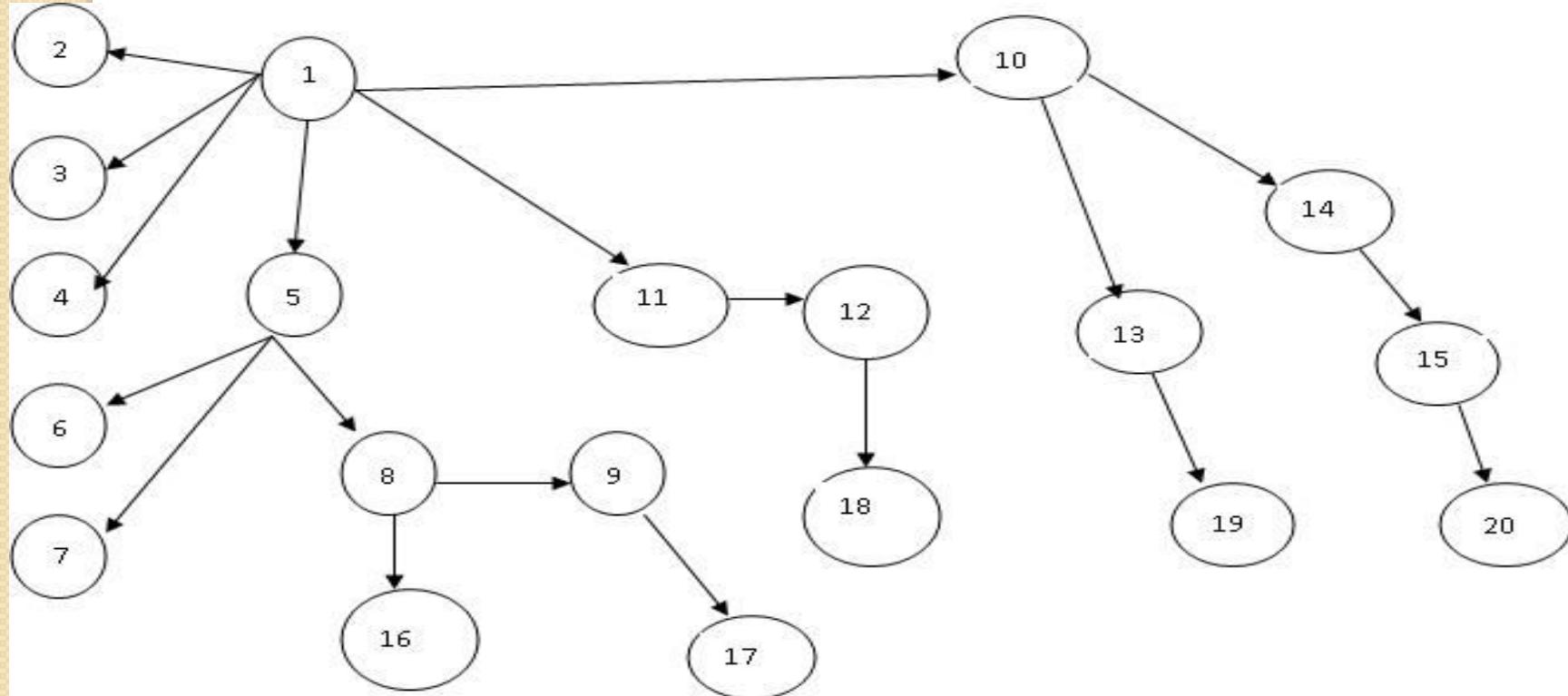
Call Graph-Based Integration

- It is assumed that integration testing detects bugs which are structural.
- However, it is also important to detect some behavioral bugs.
- If we can refine the functional decomposition tree into a form of module calling graph, then we are moving towards behavioral testing at the integration level.
- This can be done with the help of a *call graph* as given by Jorgensen.

Call Graph-Based Integration

- A call graph is a directed graph, wherein the nodes are either modules or units, and a directed edge from one node to another means one module has called another module.
- The call graph can be captured in a matrix form which is known as the adjacency matrix.

Example Call Graph

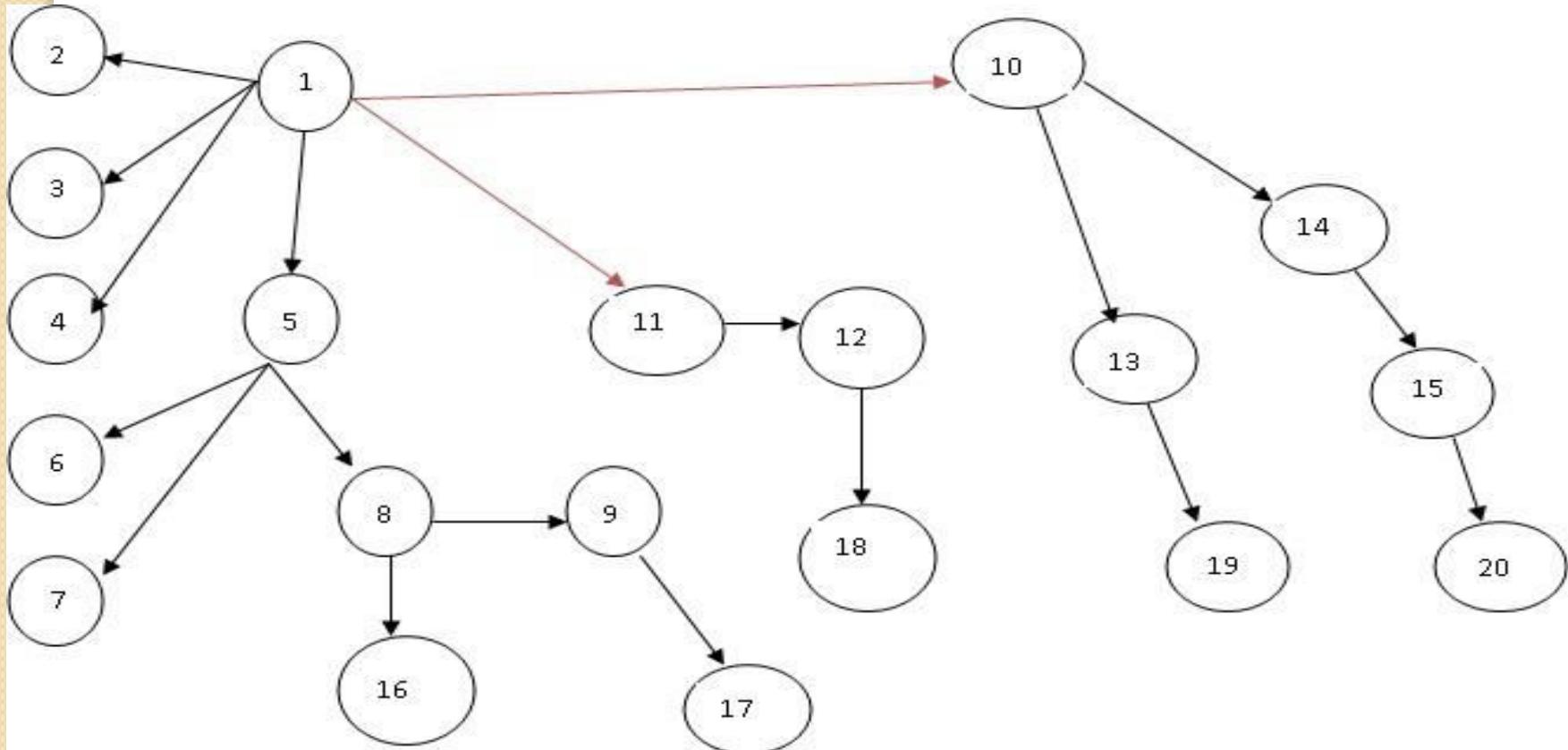


Adjacency Matrix

- The idea behind using a call graph for integration testing is to avoid the efforts made in developing the stubs and drivers.
- If we know the calling sequence, and if we wait for the called or calling function, if not ready, then call graph based integration can be used.

- There are two types of integration testing based on call graph:
 - Pair-wise Integration
 - If we consider only one pair of calling and called modules, then we can make a set of pairs for all such modules.
 - The resulting set will be the total test sessions which will be equal to the sum of all edges in the call graph.
 - Neighborhood Integration

Pair wise Integration



Test sessions = No. of edges = 19

Neighborhood Integration

- There is not much reduction in the number of test sessions in pair-wise integration as compared to the decomposition-based integration.
- If we consider neighborhoods of a node in the call graph, then number of test sessions may reduce.
- Neighborhood of a node is the immediate predecessor as well as the immediate successor of the node.
- So, neighborhood of a node can be defined as the set of nodes that are one edge away from the given node

Node	Neighbourhoods	
	Predecessors	Successors
1	2,3,4,5,10,11
5	1	6,7,8
8	5	9,16
9	8	17
10	1	13,14
11	1	12
12	11	18
13	10	19
14	10	15
15	14	20

The total test sessions = nodes – sink nodes= **20 – 10 = 10**

sink node is an instruction in a module at which execution terminates.

Path Based Integration

- In call graph, when a module or unit executes, some path of the source instructions is executed.
- It is possible that in the path execution, there may be a call to another unit.
- At that point, the control is passed from the calling unit to the called unit.

Path Based Integration

- This passing of control from one unit to another unit is important for integration testing.
- It can be done with path based integration.
- We need to understand the following definitions for path-based integration:

–Source node

- It is an instruction in the module at which the execution starts or resumes.
- The nodes where the control is being transferred after calling the module are also source nodes.

–Sink node

- It is an instruction in the module at which the execution terminates.
- The nodes from which the control is transferred are also sink nodes.

–Module execution path (MEP)

- It is a path consisting of set of executable statements within a module like in a flow graph.

—Message

- When the control is transferred from one unit to another, then the programming language mechanism used to do this is known as a message.
- For example, a function call (message from one unit to another unit)

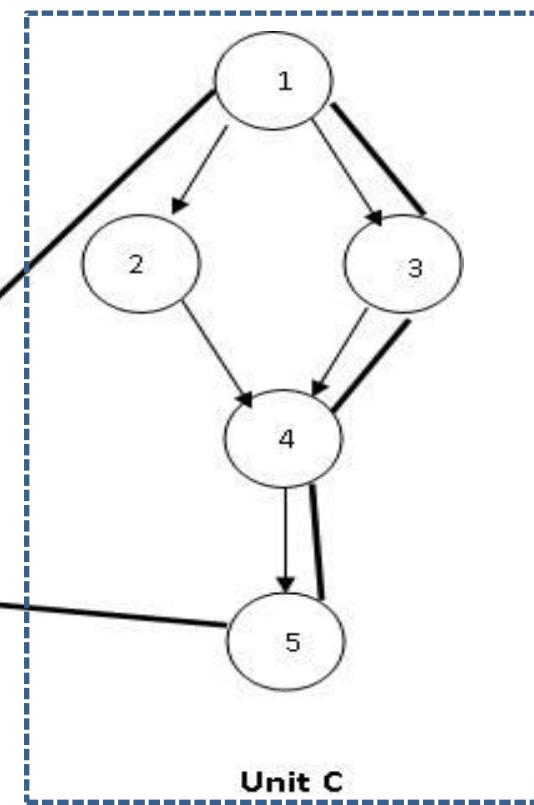
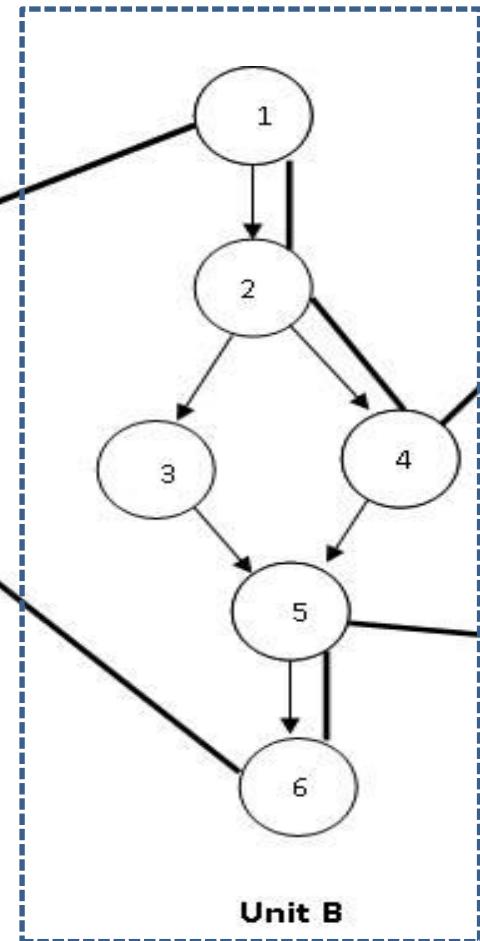
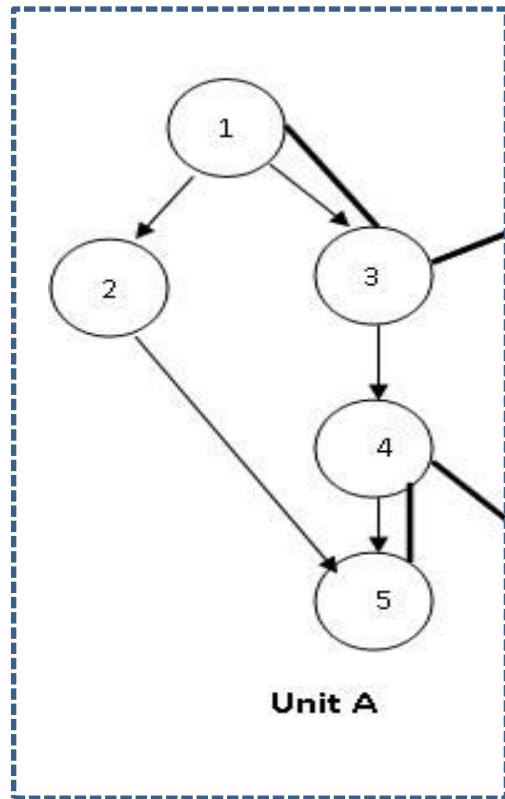
—MM-path

- It is a path consisting of MEPs and messages.
- The path shows the sequence of executable statements; it also crosses the boundary of a unit when a message is followed to call another unit.
- In other words, MM-path is a set of MEPs & transfer of control among different units in the form of messages.

– MM-Path Graph

- It can be defined as an extended flow graph where nodes are MEPs and edges are messages.
- It returns from the last called unit to the first unit where the call was made.
- In this graph (shown in next slide), messages are highlighted with thick lines.

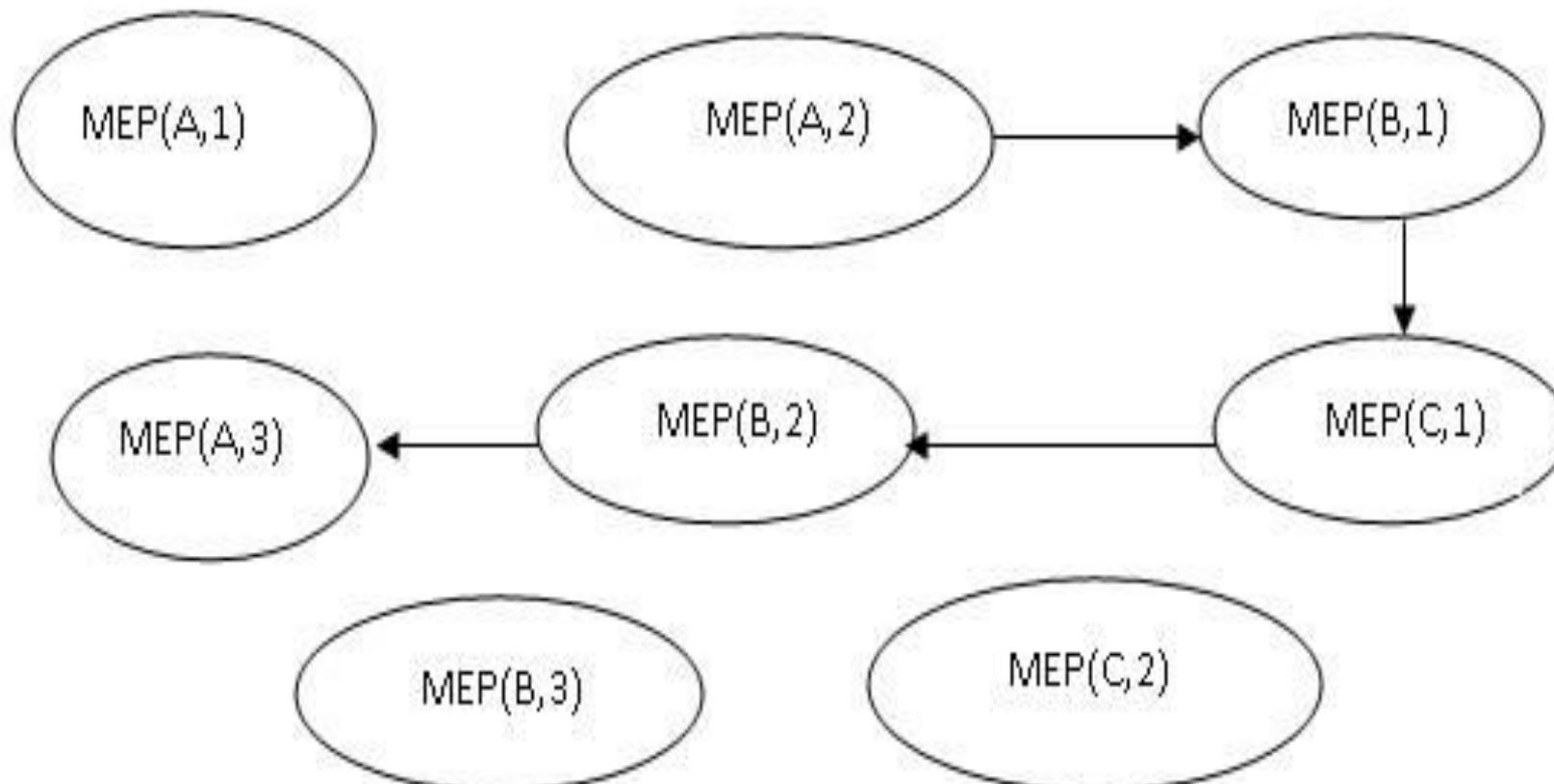
MM-Path



MM-Path Details

	Source nodes	Sink nodes	MEPs
Unit A	1,4	3,5	MEP(A,1)=<1,2,5> MEP(A,2)=<1,3> MEP(A,3)=<4,5>
Unit B	1,5	4,6	MEP(B,1)=<1,2,4> MEP(B,2)=<5,6> MEP(B,3)=<1,2,3,5,6>
Unit C	1	5	MEP(C,1)=<1,3,4,5> MEP(C,2)=<1,2,4,5>

MM-Path Graph



Function Testing

- When an integrated system is tested, all its specified functions and external interfaces are tested on the software.
- Every functionality of the system specified in the functions is tested according to its external specifications.
- An external specification is a precise description of the software behavior from the viewpoint of the outside world.

- Kit has defined function testing as the process of attempting to detect discrepancies between the functional specifications of a software and its actual behavior.
- The objective of the function test is to measure the quality of the functional components of the system.
- Tests verify that the system behaves correctly from the user/business perspectives and functions according to the requirements, models or any other design paradigm.

- The function test must determine if each component or business event:
 - Performs in accordance to the specifications,
 - Responds correctly to all conditions that may present themselves by incoming events/data,
 - Moves data correctly from one business event to the next, and
 - Is initiated in the order required to meet the business objectives of the system

- Function testing can be performed after unit and integration testing, or whenever the development team thinks that the system has sufficient functionality to execute some tests.
- The test cases are executed such that the execution of the given test case against the software will exercise external functionality of certain parts.
- To keep a record of function testing, a function coverage metric is used.

- Function coverage can be measured with a function coverage matrix.
- It keeps track of those functions that exhibited the greatest number of errors.
- An effective function test cycle must have a defined set of processes and deliverables.
- The primary processes/ deliverables for requirement based function test are as follows:

–Test Planning

- The test leader with assistance from the test team defines the scope, schedule, and deliverables for the function test cycle.
- He delivers a test plan and a test schedule- these often undergo several revisions during the testing cycle.

–Partitioning/ functional Decomposition

- It is the process of breaking down a system into its functional components or functional areas.
- Another group in the organization takes responsibility for the functional decomposition of the system.
- If decompositions are deemed insufficient, then testing organization takes up the responsibility of decomposition.

–Requirement definition

- The testing organization needs specified requirements in the form of proper documents to proceed with the function test.
- These requirements need to be itemized under an appropriate functional partition.

–Test case design

- A tester designs and implements a test case to validate that the product performs in accordance with the requirements.
- These requirements need to be itemized under an appropriate functional partition and mapped to the requirements being tested.

–Traceability matrix formation

- Test cases need to be mapped back to the appropriate requirement.
- A function coverage matrix is prepared.
- This matrix is a table, listing specific functions to be tested, the priority for testing each function, and test cases required to test each function.
- Once all the aspects of the function have been tested by one or more test cases, then the test design activity for that function can be considered complete.

Function coverage matrix

Functions/Features	Priority	Test Cases
F1	3	T2, T4, T6
F2	1	T1, T3, T5

–Test case execution

- In all phases of testing, an appropriate set of test cases need to be executed and the result of those test cases recorded.
- So, the test case to be executed should be defined in the test plan.
- If the current application does not support the testing, then it should be deferred.

Summary

- Discussed call graph-based integration testing in detail.
 - Pair-wise Integration
 - Neighborhood Integration
- Explained path-based integration testing.
- Explained Function Testing in detail.
- Described the primary processes/ deliverables for requirement based function test.

References

1. Rajib Mall, Fundamentals of Software Engineering, (Chapter – 10), Fifth Edition, PHI Learning Pvt. Ltd., 2018.
2. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter – 7), Second Edition, Oxford University Press, 2016.



The slide features a decorative vertical bar on the left side. This bar has a textured, light beige background with a subtle diamond pattern. It is adorned with three concentric, semi-transparent circles in a light blue-grey color. A small, solid blue-grey circle is positioned at the bottom of the bar, just above the text area.

Thank You

Introduction to Testing

Dr. Durga Prasad Mohapatra
Professor
Department Of CSE
NIT, Rourkela.

Defect Reduction Techniques

- Review
- Testing
- Formal verification
- Development process
- Systematic methodologies

Why Test?

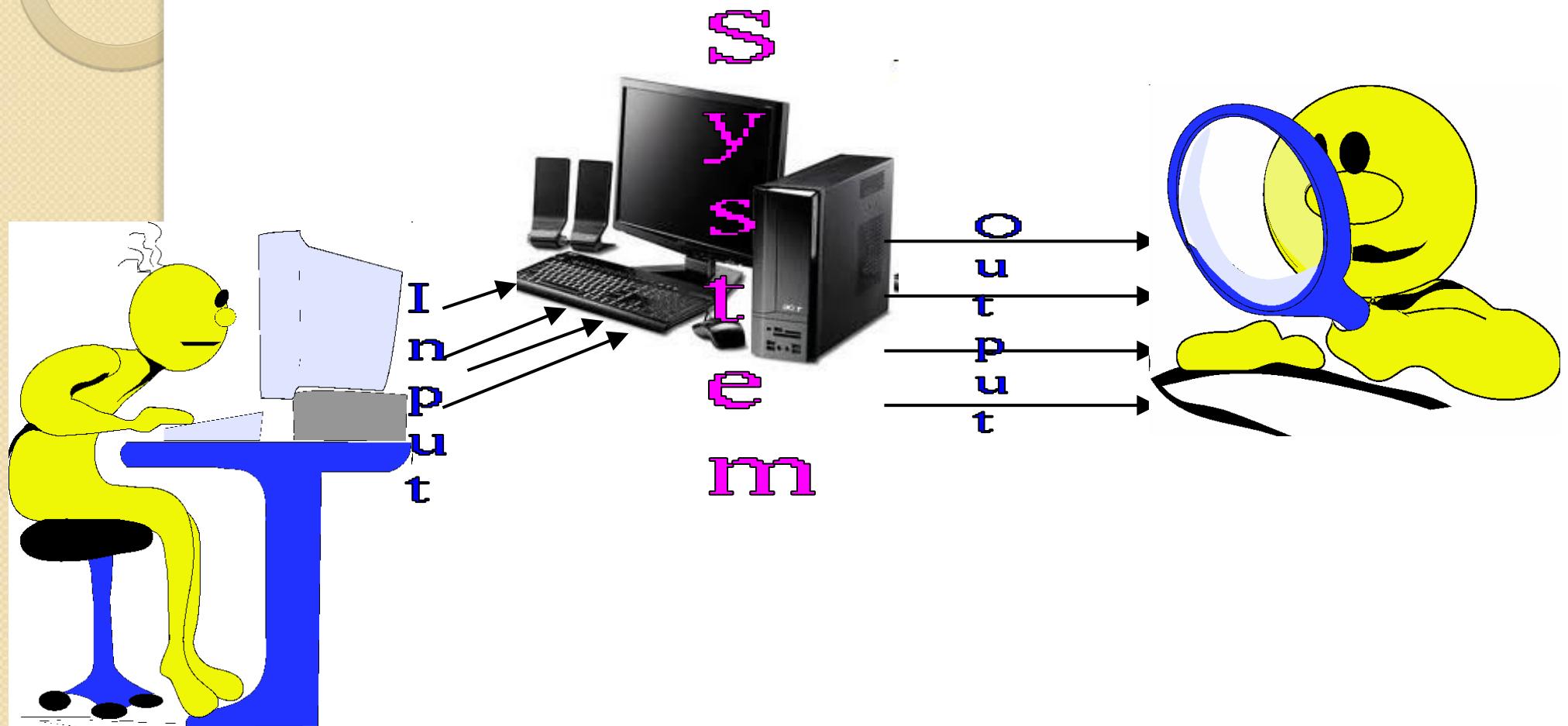


- Ariane 5 rocket self-destructed 37 seconds after launch
- Reason: A control software bug that went undetected
 - Conversion from 64-bit floating point to 16-bit signed integer value had caused an exception
 - The floating point number was larger than 32767
 - Efficiency considerations had led to the disabling of the exception handler.
- Total Cost: over \$1 billion

How Do You Test a Program?

- Input test data to the program.
- Observe the output:
 - Check if the program behaved as expected.

How Do You Test a Program?



How Do You Test a Program?

- If the program does not behave as expected:
 - Note the conditions under which it failed.
 - Later debug and correct.

What's So Hard About Testing ?

- Consider `int proc1(int x, int y)`
- Assuming a 64 bit computer
 - Input space = 2^{128}
- Assuming it takes 10secs to key-in an integer pair
 - It would take about a billion years to enter all possible values!
 - Automatic testing has its own problems!

Testing Facts

- **Consumes largest effort among all phases**
 - Largest manpower among all other development roles
 - Implies more job opportunities
- **About 50% development effort**
 - But 10% of development time?
 - How?

Testing Facts

- Testing is getting more complex and sophisticated every year.
 - Larger and more complex programs
 - Newer programming paradigms

Overview of Testing Activities

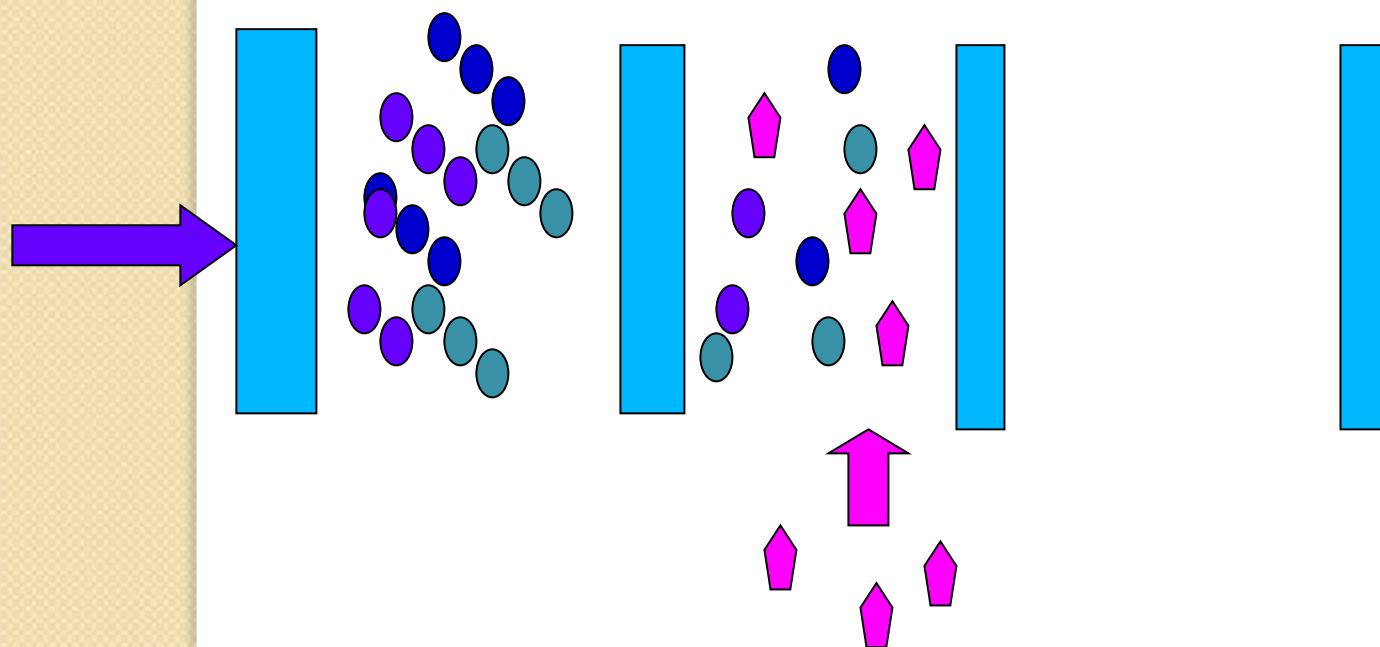
- Test Suite Design
- Run test cases and observe results to detect failures.
- Debug to locate errors
- Correct errors.

Error, Faults, and Failures

- A failure is a manifestation of an error (also defect or bug).
 - Mere presence of an error may not lead to a failure.

Pesticide Effect

- Errors that escape a fault detection technique:
 - Can not be detected by further applications of that technique.



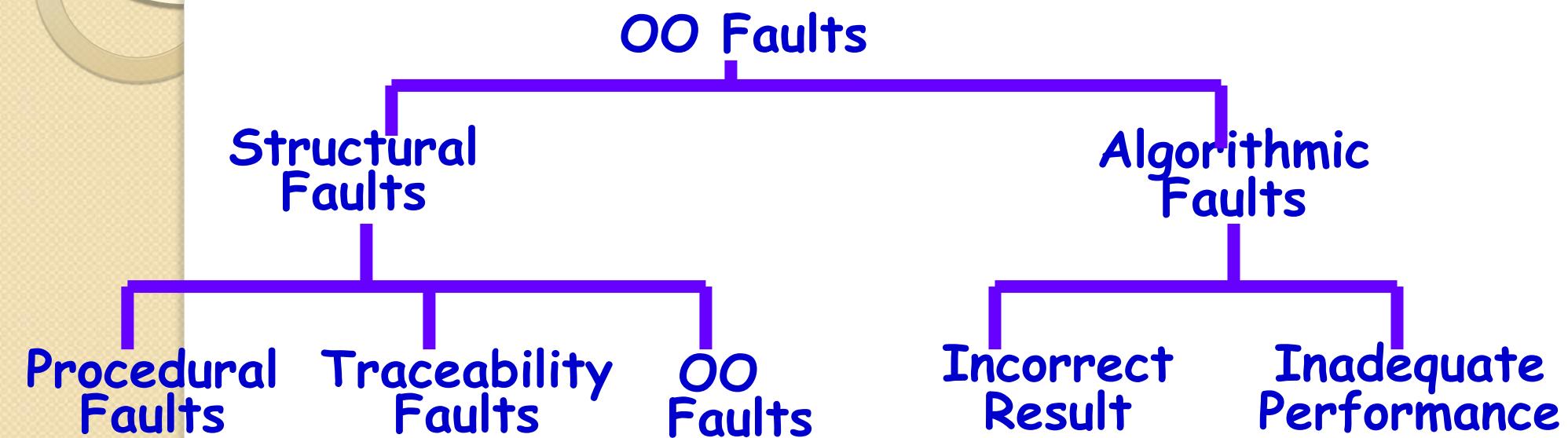
Pesticide Effect

- Assume we use 4 fault detection techniques and 1000 bugs:
 - Each detects only 70% bugs
 - How many bugs would remain
 - $1000 * (0.3)^4 = 81$ bugs

Fault Model

- Types of faults possible in a program.
- Some types can be ruled out
 - Concurrency related-problems in a sequential program

Fault Model of an OO Program



Hardware Fault-Model

- Simple:
 - Stuck-at 0
 - Stuck-at 1
 - Open circuit
 - Short circuit
- Simple ways to test the presence of each
- Hardware testing is fault-based testing



Software Testing

- Each test case typically tries to establish correct working of some functionality
 - Executes (covers) some program elements
 - For restricted types of faults, fault-based testing exists.

Test Cases and Test Suites

- Test a software using a set of carefully designed test cases:
 - The set of all test cases is called the test suite

Test Cases and Test Suites

- A **test case** is a triplet [I,S,O]
 - I is the data to be input to the system,
 - S is the state of the system at which the data will be input,
 - O is the expected output of the system.

Aim of Testing

- The aim of testing is to identify all defects in a software product.
- However, in practice even after thorough testing:
 - one cannot guarantee that the software is error-free.

Aim of Testing

- The input data domain of most software products is very large:
 - it is not practical to test the software exhaustively with each input data value.

Aim of Testing

- Testing does however expose many errors:
 - testing provides a practical way of reducing defects in a system
 - increases the users' confidence in a developed system.

Aim of Testing

- Testing is an important development phase:
 - requires the maximum effort among all development phases.
- In a typical development organization:
 - maximum number of software engineers can be found to be engaged in testing activities.

Aim of Testing

- Many engineers have the wrong impression:
 - testing is a secondary activity
 - it is intellectually not as stimulating as the other development activities, etc.

Aim of Testing

- Testing a software product is in fact:
 - as much challenging as initial development activities such as specification, design, and coding.
- Also, testing involves a lot of creative thinking.

Levels of Testing

- Software products are tested at three levels:
 - Unit testing
 - Integration testing
 - System testing

Unit testing

- During unit testing, modules are tested in isolation:
 - If all modules were to be tested together:
 - it may not be easy to determine which module has the error.

Unit testing

- Unit testing reduces debugging effort several folds.
 - Programmers carry out unit testing immediately after they complete the coding of a module.



Integration testing

- After different modules of a system have been coded and unit tested:
 - modules are integrated in steps according to an integration plan
 - partially integrated system is tested at each integration step.



System Testing

- System testing involves:
 - validating a fully developed system against its requirements.

Verification versus Validation

- Verification is the process of determining:
 - Whether output of one phase of development conforms to its previous phase.
- Validation is the process of determining:
 - Whether a fully developed system conforms to its SRS document.

Verification versus Validation

- Verification is concerned with phase containment of errors,
 - Whereas the aim of validation is that the final product be error free.

Design of Test Cases

- Exhaustive testing of any non-trivial system is impractical:
 - Input data domain is extremely large.
- Design an **optimal test suite**:
 - Of reasonable size and
 - Uncovers as many errors as possible.

Design of Test Cases

- If test cases are selected randomly:
 - Many test cases would not contribute to the significance of the test suite,
 - Would not detect errors not already being detected by other test cases in the suite.
- Number of test cases in a randomly selected test suite:
 - Not an indication of effectiveness of testing.

Design of Test Cases

- Testing a system using a large number of randomly selected test cases:
 - Does not mean that many errors in the system will be uncovered.
- Consider following example:
 - Find the maximum of two integers x and y .

Design of Test Cases

- The code has a simple programming error:

```
If (x>y) max = x;  
else max = x;
```
- Test suite $\{(x=3,y=2);(x=2,y=3)\}$ can detect the error,
- A larger test suite $\{(x=3,y=2);(x=4,y=3);(x=5,y=1)\}$ does not detect the error.

Design of Test Cases

- Systematic approaches are required to design an **optimal test suite**:
 - Each test case in the suite should detect different errors.

Design of Test Cases

- There are essentially three main approaches to design test cases:
 - **Black-box approach**
 - **White-box (or glass-box) approach**
 - **Grey-box (or model based) approach**

Black-Box Testing

- Test cases are designed using only **functional specification** of the software:
 - Without any knowledge of the internal structure of the software.
- For this reason, black-box testing is also known as **functional testing**.

Black-box Testing Techniques

- There are many approaches to design black box test cases:
 - Equivalence class partitioning
 - Boundary value analysis
 - State table based testing
 - Decision table based testing
 - Cause-effect graph based testing
 - Orthogonal array testing
 - Positive-negative testing

White-box Testing

- Designing white-box test cases:
 - Requires knowledge about the internal structure of software.
 - **White-box testing is also called structural testing.**

White-Box Testing Techniques

- There exist several popular white-box testing methodologies:
 - Statement coverage
 - Branch coverage
 - Path coverage
 - Condition coverage
 - MC/DC coverage
 - Mutation testing
 - Data flow-based testing

Coverage-Based Testing Versus Fault-Based Testing

- Idea behind coverage-based testing:
 - Design test cases so that certain program elements are executed (or covered).
 - Example: statement coverage, path coverage, etc.
- Idea behind fault-based testing:
 - Design test cases that focus on discovering certain types of faults.
 - Example: Mutation testing.

Why Both BB and WB Testing?

Black-box

- Impossible to write a test case for every possible set of inputs and outputs
- Some code parts may not be reachable
- Does not tell if extra functionality has been implemented.

White-box

- Does not address the question of whether or not a program matches the specification
- Does not tell you if all of the functionality has been implemented
- Does not discover missing program logic

Grey Box / Model Based Testing

- In grey box testing, test cases are designed from design documents / models, such as UML diagrams.
- Grey-box testing is also called model based testing.
- Mainly used for testing of O-O systems.

Summary

- Discussed importance of testing and the basic concepts of testing.
- Presented the levels of testing.
 - Unit testing
 - Integration testing
 - System testing
- Discussed the fundamentals of black box testing, white box testing and grey box testing.

References

1. R. Mall, Fundamentals of Software Engineering, (Chapter - 10), Fifth Edition, PHI Learning Pvt. Ltd., 2018.



Thank You

Introduction to Testing

Dr. Durga Prasad Mohapatra
Professor
Department Of CSE
NIT, Rourkela.

Defect Reduction Techniques

- Review
- Testing
- Formal verification
- Development process
- Systematic methodologies

Why Test?

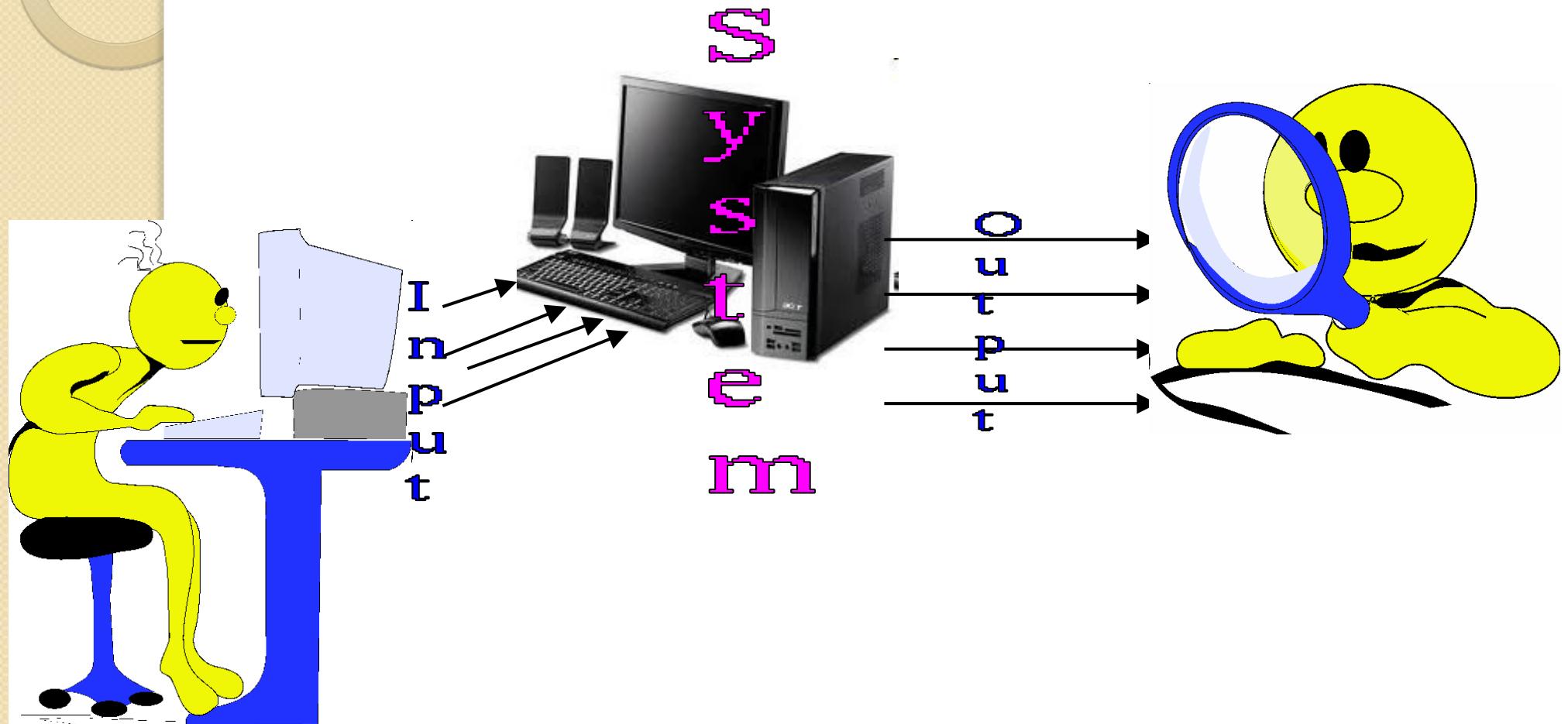


- Ariane 5 rocket self-destructed 37 seconds after launch
- Reason: A control software bug that went undetected
 - Conversion from 64-bit floating point to 16-bit signed integer value had caused an exception
 - The floating point number was larger than 32767
 - Efficiency considerations had led to the disabling of the exception handler.
- Total Cost: over \$1 billion

How Do You Test a Program?

- Input test data to the program.
- Observe the output:
 - Check if the program behaved as expected.

How Do You Test a Program?



How Do You Test a Program?

- If the program does not behave as expected:
 - Note the conditions under which it failed.
 - Later debug and correct.

What's So Hard About Testing ?

- Consider `int proc1(int x, int y)`
- Assuming a 64 bit computer
 - Input space = 2^{128}
- Assuming it takes 10secs to key-in an integer pair
 - It would take about a billion years to enter all possible values!
 - Automatic testing has its own problems!

Testing Facts

- **Consumes largest effort among all phases**
 - Largest manpower among all other development roles
 - Implies more job opportunities
- **About 50% development effort**
 - But 10% of development time?
 - How?

Testing Facts

- Testing is getting more complex and sophisticated every year.
 - Larger and more complex programs
 - Newer programming paradigms

Overview of Testing Activities

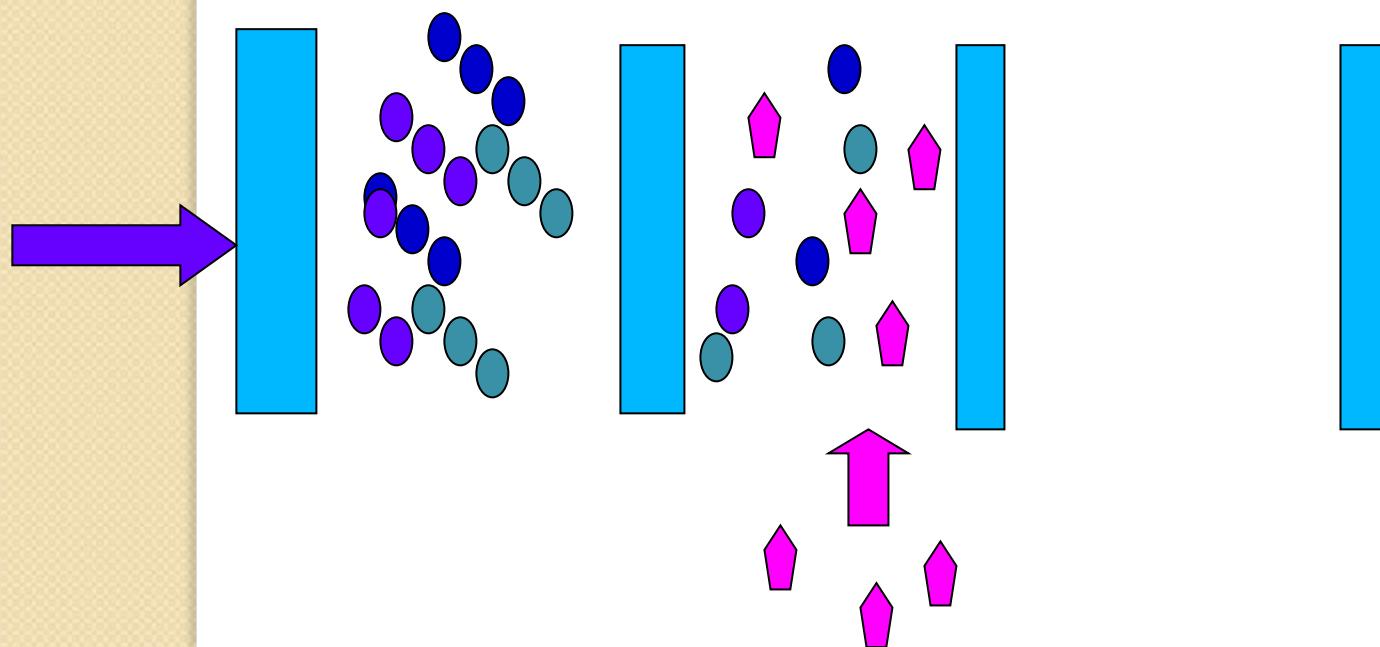
- Test Suite Design
- Run test cases and observe results to detect failures.
- Debug to locate errors
- Correct errors.

Error, Faults, and Failures

- A failure is a manifestation of an error (also defect or bug).
 - Mere presence of an error may not lead to a failure.

Pesticide Effect

- Errors that escape a fault detection technique:
 - Can not be detected by further applications of that technique.



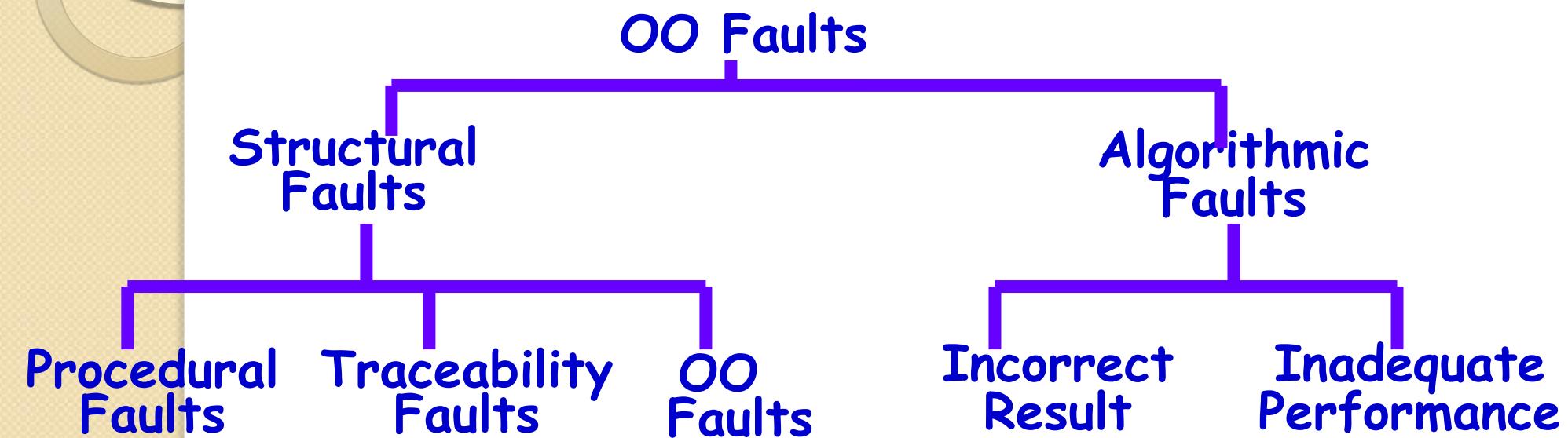
Pesticide Effect

- Assume we use 4 fault detection techniques and 1000 bugs:
 - Each detects only 70% bugs
 - How many bugs would remain
 - $1000 * (0.3)^4 = 81$ bugs

Fault Model

- Types of faults possible in a program.
- Some types can be ruled out
 - Concurrency related-problems in a sequential program

Fault Model of an OO Program



Hardware Fault-Model

- Simple:
 - Stuck-at 0
 - Stuck-at 1
 - Open circuit
 - Short circuit
- Simple ways to test the presence of each
- Hardware testing is fault-based testing



Software Testing

- Each test case typically tries to establish correct working of some functionality
 - Executes (covers) some program elements
 - For restricted types of faults, fault-based testing exists.

Test Cases and Test Suites

- Test a software using a set of carefully designed test cases:
 - The set of all test cases is called the test suite

Test Cases and Test Suites

- A **test case** is a triplet [I,S,O]
 - I is the data to be input to the system,
 - S is the state of the system at which the data will be input,
 - O is the expected output of the system.

Aim of Testing

- The aim of testing is to identify all defects in a software product.
- However, in practice even after thorough testing:
 - one cannot guarantee that the software is error-free.

Aim of Testing

- The input data domain of most software products is very large:
 - it is not practical to test the software exhaustively with each input data value.

Aim of Testing

- Testing does however expose many errors:
 - testing provides a practical way of reducing defects in a system
 - increases the users' confidence in a developed system.

Aim of Testing

- Testing is an important development phase:
 - requires the maximum effort among all development phases.
- In a typical development organization:
 - maximum number of software engineers can be found to be engaged in testing activities.

Aim of Testing

- Many engineers have the wrong impression:
 - testing is a secondary activity
 - it is intellectually not as stimulating as the other development activities, etc.

Aim of Testing

- Testing a software product is in fact:
 - as much challenging as initial development activities such as specification, design, and coding.
- Also, testing involves a lot of creative thinking.

Levels of Testing

- Software products are tested at three levels:
 - Unit testing
 - Integration testing
 - System testing

Unit testing

- During unit testing, modules are tested in isolation:
 - If all modules were to be tested together:
 - it may not be easy to determine which module has the error.

Unit testing

- Unit testing reduces debugging effort several folds.
 - Programmers carry out unit testing immediately after they complete the coding of a module.



Integration testing

- After different modules of a system have been coded and unit tested:
 - modules are integrated in steps according to an integration plan
 - partially integrated system is tested at each integration step.

System Testing

- System testing involves:
 - validating a fully developed system against its requirements.

Verification versus Validation

- Verification is the process of determining:
 - Whether output of one phase of development conforms to its previous phase.
- Validation is the process of determining:
 - Whether a fully developed system conforms to its SRS document.

Verification versus Validation

- Verification is concerned with phase containment of errors,
 - Whereas the aim of validation is that the final product be error free.

Design of Test Cases

- Exhaustive testing of any non-trivial system is impractical:
 - Input data domain is extremely large.
- Design an **optimal test suite**:
 - Of reasonable size and
 - Uncovers as many errors as possible.

Design of Test Cases

- If test cases are selected randomly:
 - Many test cases would not contribute to the significance of the test suite,
 - Would not detect errors not already being detected by other test cases in the suite.
- Number of test cases in a randomly selected test suite:
 - Not an indication of effectiveness of testing.

Design of Test Cases

- Testing a system using a large number of randomly selected test cases:
 - Does not mean that many errors in the system will be uncovered.
- Consider following example:
 - Find the maximum of two integers x and y .

Design of Test Cases

- The code has a simple programming error:

```
If (x>y) max = x;  
else max = x;
```
- Test suite $\{(x=3,y=2);(x=2,y=3)\}$ can detect the error,
- A larger test suite $\{(x=3,y=2);(x=4,y=3);(x=5,y=1)\}$ does not detect the error.

Design of Test Cases

- Systematic approaches are required to design an **optimal test suite**:
 - Each test case in the suite should detect different errors.

Design of Test Cases

- There are essentially three main approaches to design test cases:
 - **Black-box approach**
 - **White-box (or glass-box) approach**
 - **Grey-box (or model based) approach**

Black-Box Testing

- Test cases are designed using only **functional specification** of the software:
 - Without any knowledge of the internal structure of the software.
- For this reason, black-box testing is also known as **functional testing**.

Black-box Testing Techniques

- There are many approaches to design black box test cases:
 - Equivalence class partitioning
 - Boundary value analysis
 - State table based testing
 - Decision table based testing
 - Cause-effect graph based testing
 - Orthogonal array testing
 - Positive-negative testing

White-box Testing

- Designing white-box test cases:
 - Requires knowledge about the internal structure of software.
 - **White-box testing is also called structural testing.**

White-Box Testing Techniques

- There exist several popular white-box testing methodologies:
 - Statement coverage
 - Branch coverage
 - Path coverage
 - Condition coverage
 - MC/DC coverage
 - Mutation testing
 - Data flow-based testing

Coverage-Based Testing Versus Fault-Based Testing

- Idea behind coverage-based testing:
 - Design test cases so that certain program elements are executed (or covered).
 - Example: statement coverage, path coverage, etc.
- Idea behind fault-based testing:
 - Design test cases that focus on discovering certain types of faults.
 - Example: Mutation testing.

Why Both BB and WB Testing?

Black-box

- Impossible to write a test case for every possible set of inputs and outputs
- Some code parts may not be reachable
- Does not tell if extra functionality has been implemented.

White-box

- Does not address the question of whether or not a program matches the specification
- Does not tell you if all of the functionality has been implemented
- Does not discover missing program logic

Grey Box / Model Based Testing

- In grey box testing, test cases are designed from design documents / models, such as UML diagrams.
- Grey-box testing is also called model based testing.
- Mainly used for testing of O-O systems.

Summary

- Discussed importance of testing and the basic concepts of testing.
- Presented the levels of testing.
 - Unit testing
 - Integration testing
 - System testing
- Discussed the fundamentals of black box testing, white box testing and grey box testing.

References

1. R. Mall, Fundamentals of Software Engineering, (Chapter - 10), Fifth Edition, PHI Learning Pvt. Ltd., 2018.



Thank You

Mutation Testing - A White Box Testing Technique

Dr. Durga Prasad Mohapatra

Dept. of Computer Science & Engineering

NIT Rourkela

White-box Testing

- Designing white-box test cases:
 - Requires knowledge about the internal structure of software.
 - White-box testing is also called structural testing.
 - In this unit we will study white-box testing.

White-Box Testing Methodologies

- There exist several popular white-box testing methodologies:
 - Statement coverage
 - Branch coverage
 - Condition coverage
 - MC/DC coverage
 - Path coverage
 - Data flow-based testing
 - Mutation testing

Introduction

- Mutation testing is the process of mutating some segment of code (putting some error in the code) and then, testing this mutated code with some data.
- If the test data is able to detect the mutations in the code, then the test data is quite good, otherwise we must focus on the quality of the test data.

Introduction cont...

- Mutation testing helps a user create test data by interacting with user to iteratively strengthen quality of test data.
- During mutation testing, faults are introduced into a program.
- Test data are used to execute these faulty programs with the goal of causing each faulty program to fail.
- Faulty programs are called **mutants** of the original program.

Introduction cont...

- A mutant is said to be killed when a **test case causes it to fail**.
- When this happens, the mutant is considered dead and no longer needs to remain in the testing process, since the faults represented by that mutant have been detected.

Introduction cont...

- The software is first tested:
 - using an initial testing method based on white-box strategies.
- After the initial testing is complete,
 - mutation testing is taken up.

Main Idea

- The idea behind mutation testing:
 - make a few arbitrary small changes to a program at a time.

Main Idea cont ...

- Insert faults into a program:
 - Check whether the test suite is able to detect these.
 - This either validates or invalidates the test suite.

Main Idea cont ...

- Each time the program is changed,
 - it is called a mutated program
 - the change is called a mutant.

Main Idea cont ...

- A mutated program:
 - Tested against the full test suite of the program.
- If there exists at least one test case in the test suite for which:
 - A mutant gives an incorrect result,
 - Then the mutant is said to be dead.

Main Idea cont ...

- If a mutant remains alive:
 - even after all test cases have been exhausted,
 - the test suite is enhanced to kill the mutant.
- The process of generation and killing of mutants:
 - can be automated by predefining a set of primitive changes that can be applied to the program.

Primitive changes

- The primitive changes can be:
 - Deleting a statement
 - Altering an arithmetic operator,
 - Changing the value of a constant,
 - Changing a data type, etc.

Traditional Mutation Operators

- Deletion of a statement
- Boolean:
 - Replacement of a statement with another
 - eg. == and >=, < and <=
 - Replacement of boolean expressions with true or false
 - eg. a || b with true
 - Replacement of arithmetic
 - eg. * and +, / and -
 - Replacement of a variable (ensuring same scope/type)

Underlying Hypotheses

- Mutation testing is based on the following two hypotheses:
 - **The Competent Programmer Hypothesis**
 - **The Coupling Effect**

Both of these were proposed by DeMillo et al., 1978

The Competent Programmer Hypothesis

- Programmers create programs that are close to being correct:
 - Differ from the correct program by some simple errors.

The Coupling Effect

- Complex errors are caused due to several simple errors.
- It therefore suffices to check for the presence of the simple errors

Types of mutants

- Primary Mutant
- Secondary Mutant

Primary Mutant

When the mutants are **single modification** of the initial program using some operators, they are called **primary mutants**.

Example

```
if(a>b)
    x=x + y;
else
    x = y;
Printf("%d", x);
.....
```

We can consider following mutant for this example;

- M1: x=x-y;
- M2: x=x/y;
- M3: x=x+1;
- M4: printf("%d", y);

Example cont ...

The results of the initial program and its mutant

Test Data	x	y	Initial program result	Mutant Result
TD1	2	2	4	0(M1)
TD2(x and y# 0)	4	3	7	1.4(M2)
TD3(y #1)	3	2	5	4(M3)
TD4(y #0)	5	2	7	2(M4)

Secondary Mutants

- When **multiple levels** of mutation are applied on the initial program, then, this class of mutant is called **secondary Mutant**.
- In this case, it is very difficult to identify the initial program form its mutants.

Example

If($a < b$)

$c = a;$

Mutants for this code may be as follows:

M1 : if($a \leq b - 1$)

$c = a;$

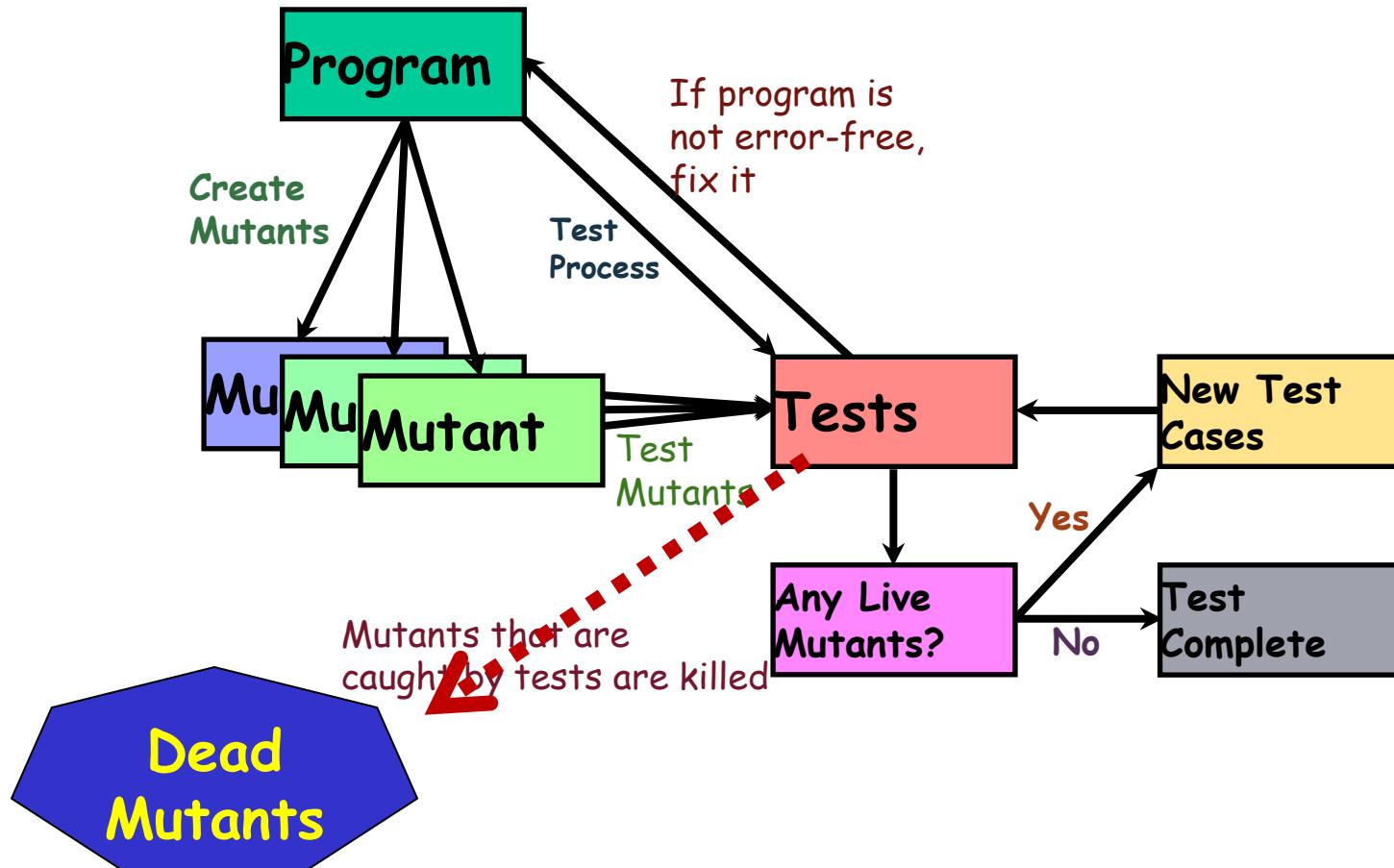
M2 : if($a + 1 \leq b$)

$c = a;$

M3 : if($a == b$)

$c = a + 1;$

Mutation Testing Process



Mutation Testing Process cont...

- Construct the mutants of a test program.
- Add test cases to the mutation system and check the output of the program on each test case to see if it is correct.
- If the output is incorrect, a fault has been found and the program must be modified and the process restarted.
- If the output is correct, that test case is executed against each live mutant.
- If the output of a mutant differs from that of the original program on the same test case, the mutant is assumed to be incorrect and is killed.

Mutation Testing Process cont...

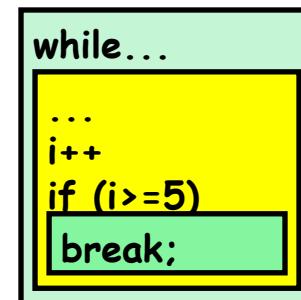
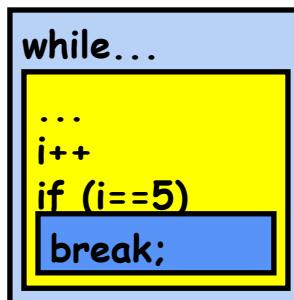
- After each test case has been executed against each live mutant, each remaining mutant falls into one of the following two categories:
 - ✓ One, the mutant is functionally equivalent to the original program. An equivalent mutant always produces the same output as the original program, so no test case can kill it.
 - ✓ Two, the mutant is killable, but the set of test cases is insufficient to kill it. In this case, new test cases need to be created, and the process iterates until the test set is strong enough to satisfy the tester.

Mutation Testing Process cont...

- The mutation score for a set of test data is the percentage of non-equivalent mutants killed by that data.
- If the mutation score is 100%, then the test data is called mutation adequate.

Equivalent Mutants

- There may be surviving mutants that **cannot be killed**,
 - These are called **Equivalent Mutants**
- Although syntactically different:
 - These mutants are **indistinguishable** through testing.
- Therefore have to be checked 'by hand'



Disadvantages of Mutation Testing

- Equivalent mutants
- Computationally very expensive.
 - A large number of possible mutants can be generated.
- Certain types of faults are very difficult to inject.
 - Only simple syntactic faults introduced

Quiz 1

- Identify one advantage and one disadvantage of the mutation test technique.

Quiz 1: Solution

- Identify two advantages and two disadvantages of the mutation test technique.
- **Adv:**
 - Can be automated
 - Helps effectively strengthen black box and coverage-based test suite
- **Disadv:**
 - Equivalent mutants

Summary

- Presented basic concepts of mutation testing.
- Explained types of mutants.
- Discussed the mutation testing process.
- Presented some limitations of mutation testing.

References

1. Rajib Mall, Fundamentals of Software Engineering, (Chapter - 10), Fifth Edition, PHI Learning Pvt. Ltd., 2018.
2. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter - 5), Second Edition, Oxford University Press, 2016.

Thank You

Example on Mutation Testing

Dr. D. P. Mohapatra

Associate Professor

Department of CSE
National Institute of Technology, Rourkela-769008
Odisha



Example

Consider the following program. Generate five mutants and design test cases taking into account each mutant, using mutation testing. Calculate mutation score of your test suite.

```
main(){
float x,y,z;
printf("Enter values of three variables x,y,z");
scanf("%f%f%f",&x,&y,&z);
if(x>y){
if(x>z)
printf("%d is greatest",x);
else
printf("%d is greatest",z);
}
else{
if(y>z)
printf("%d is greatest",y);
else
printf("%d is greatest",z);
}
}
```



Solution

The instrumented program is given below.

```
1 main(){
2     float x,y,z;
3     printf("Enter values of three variables x,y,z");
4     scanf("%f%f%f",&x,&y,&z);
5     if(x>y){
6         if(x>z)
7             printf("%d is greatest",x);
8         else
9             printf("%d is greatest",z);
10    }
11 }
```



Test suite

Consider the test suite given in Table 1.

Table 1: Test suite for Q ??

Sl. No.	x	y	z	Expected Output
1	6	10	2	10
2	10	6	2	10
3	6	2	10	10
4	6	10	20	20



Mutated Statements

Table 2: Mutated statements

Mutant No.	Line No.	Original line	Modified line
M1	4	<code>if($x > y$)</code>	<code>if($x < y$)</code>
M2	4	<code>if($x > y$)</code>	<code>if($x > (y + z)$)</code>
M3	5	<code>if($x > z$)</code>	<code>if($x < z$)</code>
M4	8	<code>if($y > z$)</code>	<code>if($y = z$)</code>
M5	7	<code>printf("%d is greatest",z);</code>	<code>printf("%d is greatest",y);</code>



Actual Output of mutant M1

The mutated line numbers and changed lines are given in Table 2. The actual output obtained by executing the mutants M1-M5 is shown in Tables 3-7.

Table 3: Actual output of mutant M1

Test case	x	y	z	Expected output	Actual output
1	6	10	2	10	6
2	10	6	2	10	6
3	6	2	10	10	10
4	6	10	20	20	20



Actual output of mutant M2

Table 4: Actual output of mutant M2

Test case	x	y	z	Expected output	Actual output
1	6	10	2	10	10
2	10	6	2	10	10
3	6	2	10	10	10
4	6	10	20	20	20



Actual output of mutant M3

Table 5: Actual output of mutant M3

Test case	x	y	z	Expected output	Actual output
1	6	10	2	10	10
2	10	6	2	10	2
3	6	2	10	10	6
4	6	10	20	20	20



Actual output of mutant M4

Table 6: Actual output of mutant M4

Test case	x	y	z	Expected output	Actual output
1	6	10	2	10	10
2	10	6	2	10	10
3	6	10	2	10	10
4	6	10	20	20	10



Actual output of mutant M5

Table 7: Actual output of mutant M5

Test case	x	y	z	Expected output	Actual output
1	6	10	2	10	10
2	10	6	2	10	10
3	6	2	10	10	2
4	6	10	20	20	20



Additional test case

Table 8: Additional test case

Test case	x	y	z	Expected Output
5	10	5	6	10



Output of added test case

Table 9: Output of added test case

Test case	x	y	z	Expected output	Actual output
5	10	5	6	10	6



Revised Test suite

Table 10: Revised Test suite

Sl. No.	x	y	z	Expected Output
1	6	10	2	10
2	10	6	2	10
3	6	2	10	10
4	6	10	20	20
5	10	5	6	10



Mutation Score

Mutation score=Number of mutants killed/Total number of mutants=4/5=0.8

Higher the mutant score, better is the effectiveness of the test suite. The mutant M2 is live in the example. We may have to write a specific test case to kill this mutant. The additional test case is given in Table 8.

Now, when we execute test case 5, the actual output will be different from the expected output (Table 9), hence the mutant will be killed. This test case is very important and should be added to the given test suite. Therefore, the revised test suite is given in Table 10.



Testing Object-Oriented Programs

Dr. Durga Prasad Mohapatra

Professor

Department of Computer Science and Engineering
National Institute of Technology, Rourkela, India

Plan of the Talk

- Introduction
- Challenges in testing OO programs
- Test suite design using UML models
- Test design patterns
- Conclusion

Introduction

- More than 50% of development effort is being spent on testing.
- Quality and effective reuse of software depend to a large extent:
 - on thorough testing.
- It was expected during initial years that OO would help substantially reduce testing effort as object-orientation incorporates several good programming features:
 - But, as we find it out today --- it only complicates testing.

Complicacy in testing

- Soon it was realized that satisfactory testing object oriented programs is much more difficult.
- Requires much more cost and effort in comparison to procedural programs.
 - as the various object-oriented features introduce additional complications and scope of new type of bugs.
- Additional test cases are needed to be designed to detect the bugs.

Today's Focus

- Most reported research on OO paradigms focus on:
 - Analysis and design.
- We discuss some important issues in testing object-oriented systems.

Fault Model

- Different types of faults in a program:
 - Infinite for practical purposes .
- A fault model:
 - A map of possible types of faults.
 - Necessary to guide any rational testing strategy.
- A fault model may be constructed from analysis of failure data.

Places To Look For Faults

- Some places (error types) can trivially be omitted from a fault model:
 - I Cannot make English grammar mistakes when I am writing a C program.
 - Cannot commit errors due to side effects (change global variables) while writing a Java program.
- A **test strategy**:
 - Yields a test suite when applied to a unit under test.

Challenges in OO Testing

- What is an appropriate unit for testing?
- Implications of OO features:
 - Encapsulation
 - Inheritance
 - Polymorphism & Dynamic Binding, etc.
- State-based testing
- Test coverage analysis
- Integration strategies
- Test process strategy

What is a Suitable Unit for Testing?

- What is the fundamental unit of testing conventional programs?
 - A function.
- However, as far as OO programs are concerned:
 - Methods are not the basic unit of testing.

Weyukar's Anticomposition axiom

- Any amount of testing of individual methods can not ensure that a class has been satisfactorily tested.

Weyukar's Anticomposition axiom

- The main justification for anticomposition axiom is that
 - a method operates in the scope of the data and other methods of its object.
- So, it is necessary to test a method in the context of these.
- As objects can have significant states, the behavior of a method can be different based on the **state** of corresponding object.

Basic unit of testing

- Therefore a method has to be tested with all other methods and data of the corr. object.
- Moreover, a method needs to be tested at all states that the object can assume.
- So, it is improper to consider a method as the basic unit of testing an OOP.
- An object is the basic unit of testing of object-oriented programs.
- In object oriented testing, unit testing is conducted by testing each object in isolation.

Suitable Unit for Testing OO Programs

- **Class level:**
 - Testing interactions between attributes and methods must be addressed.
 - State of the object must be considered.

Suitable Unit of Testing

cont...

- **Cluster level:**
 - Tests the interactions among a group of cooperating classes.
- A sequence of interactions is typically required to implement a visible behavior (i.e. a use case).
- **System level:**
 - Tests an entire operational system.

Encapsulation

- Encapsulation is not a source of errors:
 - However, an obstacle to testing.
 - It prevents accessing attribute values by a debugger.
- While testing:
 - Precise current state information is necessary.

Solving Encapsulation-Related Problems

Several solutions are possible:

- Built-in or inherited state reporting methods.
- Low level probes to manually inspect object attributes.
- Proof-of-correctness technique (formal).

Solving Encapsulation-Related Problems

- Most feasible way: State reporting methods.
- Reliable verification of state reporting methods is a problem.

Inheritance

- Should inherited methods be retested?
 - Retesting of inherited methods is the rule, rather than an exception.
- Retesting required:
 - Because a new context of usage results when a subclass is derived.
- Correct behavior at an upper level:
 - Does not guarantee correct behavior at a lower level.

Inheritance --- Overriding

- In case of method overriding:
 - Need to retest the classes in the context of overriding.
 - An overridden method must be retested even when only minor syntactic changes are made.

Deep Inheritance Hierarchy

- A subclass at the bottom of a deep hierarchy:
 - may have only one or two lines of code,
 - but may inherit hundreds of features.
- This situation creates fault hazards:
 - similar to unrestricted access to global data in procedural programs.
- **Inheritance weakens encapsulation.**

Deep Inheritance Hierarchy cont...

- A deep and wide inheritance hierarchy can defy comprehension:
 - Lead to bugs and reduce testability.
 - Incorrect initialization and forgotten methods can result.
 - Class flattening may increase understandability.
- Multiple Inheritance:
 - increases number of contexts to test.

Abstract and Generic Classes

- Unique to OO programming:
 - Provide important support for reuse.
- Must be extended and instantiated to be tested.
- May never be considered fully tested:
 - Since need retesting when new subclasses are created.

Polymorphism

- Each possible binding of a polymorphic component requires separate testing:
 - Often difficult to find all bindings that may occur.
 - Increases the chances of bugs .
 - An obstacle to reaching coverage goals.

Polymorphism

cont...

- Polymorphism complicates integration planning:
 - many server classes may need to be integrated before a client class can be tested.

Dynamic Binding

- Dynamic binding implies:
 - The code that implements a given function is unknown until run time.
 - Static analysis cannot be used to identify the precise dependencies in a program.
- It becomes difficult to identify all possible bindings and test them.

Object states

- Objects store data parmently and also they have significant states.
- The behavior of a object is usually different in different states.
- Hence the object has to be tested at all its possible states.

Object states

- All state transition functions also needs to be tested thoroughly.
- There should be no extra transitions or extra states other than those defined in the state model.

Why are traditional testing techniques considered unsatisfactory for testing OOPs?

- In procedural programs, procedures are the basic units where in OOPs objects are the basic units of testing.
- Statement coverage based testing is not meaningful for testing OOPs as the inherited methods have to be retested in the derived class.
- In fact, the different O-O features require different test cases to be designed compared to traditional testing.

Why are traditional testing techniques considered unsatisfactory for testing OOPs?

- These O-O features are explicit in design models, and it is usually difficult to extract from an analysis of the source code.
- Hence test cases are designed based on the design model.
- So, this approach is considered to be intermediate between white box and black box approach & is called **grey box** approach, which is important for testing OOPs.

Test Process Strategy

- Object-oriented development tends toward:
 - shorter incremental cycles.
- Development characterized as:
 - design a little, code a little, test a little.
- The distance between OO specification and implementation, therefore:
 - typically small compared to conventional systems.

Test Process Strategy

cont...

- The gap between white-box/black-box test is diminished.
 - Therefore lower importance of code-based testing.
 - Model-based testing has assumed importance (also called grey box testing).
- Conventional white-box testing can be used for testing individual methods.

Testing Based on UML Models

- UML has become the *de facto* standard for OO modeling and design:
- Though UML is intended to produce rigorous models:
 - Does incorporate many flexibilities.
 - incomplete, inconsistent, and ambiguous models often result.
- Never the less:
 - UML models are an important source of information for test design.

Test Approaches for UML-Based Designs

- Two Approaches:
 - 1 Interpret the generic test strategy to develop the test suite.
 - 2 Apply the related test design pattern.

Development of test suite by interpreting the generic test strategy

Use Case-based Testing

- Use cases roughly capture system level requirements.
- A collection of use cases defines complete functionality of the system.
- Each use case (UC) consists of a mainline scenario (sequence) and several alternate scenarios (sequences).
- For each UC, the mainline & all alternate scenarios (sequences) are tested to check, if any errors show up.

Use Case-based Testing

cont...

- Several general kinds of tests can be derived from use cases:
- **Scenario Coverage:**
 1. Test cases for basic courses-- "the expected flow of events" **mainline sequence**.
 2. Test cases of other courses-- "all other flows of events" **alternate sequences**.
- Also, test cases for the features described in user documentation, traceable to each use case, can be generated from use cases.

Use Case Diagram

cont...

- Generic test requirements include:
 - At least one test case should exercise:
 - Every Use Case when actor communicates with Use Case.
 - Every **extension** combination such as Use Case 1 **extends** Use Case 2.
 - Every **uses** combination such as Use Case 1 **uses** Use Case 2.

Class Diagram-based Testing

- Class diagram documents the structure of a system. It represents the entities and their inter-relationships.
- **Testing derived classes:** All derived classes have to be instantiated & tested. In addition to the new methods defined in the derived class, the **inherited methods** must be retested.
- The test cases should also exercise:
 - Each association relation (**association testing**)
 - Independent creation & destruction of the objects (container & components) in an aggregation relation (**aggregation testing**).

Testing Relations Among Classes

- Relations among classes are:
 - Inheritance, Association, Composition, and Dependency.
- Each relationship has corresponding generic test requirements
 - For example, a generalization relation is:
 - not Reflexive(R) and not Symmetric(S) but is Transitive(T)

Testing Relations

cont . . .

- For any relation, we can ask
 - which of the three (R, S, T) properties is required, excluded, or irrelevant.
- The answers leads to a simple but effective test suite.

Testing Relations

cont . . .

- For example, when reflexivity is excluded,
 - Any x that would make xRx true should be rejected.
- Example: Student is member of Dept-Library and Dept-Library is a part of Dept.
 - The relation *is member of* is not R , not S but is T .

State model-based testing

- State charts can be used to represents state-based behavior of :
 - a class, subsystem, or system.
- Statechart model:
 - provides most of the information necessary for class-level state-based testing.

State model-based testing

- The concept of control flow of a conventional program :
 - Does not map readily to an OO program.
- In a state model:
 - We specify how the object's state would change under certain conditions.

State model-based testing

- Flow of control in OO programs:
 - Message passing from one object to another.
 - Causes the receiving object to perform some operation, can lead to an alteration of its state.
- **State Coverage:** Each method of an object is tested at each state of the object.

State model-based testing

- The state model defines the allowable transitions at each state.
- States can be constructed:
 - Using equivalence classes defined on the instance variables.
- Jacobson's OOSE advocates:
 - Design test cases to cover all state transitions.

State model-based testing

- State transitions coverage: It is tested whether all transitions depicted in the state model work satisfactorily.
- State transition path coverage: All transition paths in the state model are tested.

State model-based testing

- Test cases can be derived from the state machine model of a class:
 - Methods result in state transitions.
 - Test cases are designed to exercise each transition at a state.
- However, the transitions are tied to user-selectable activation sequences:
 - Use Cases

Difficulty with state model-based testing

- The locus of state control is distributed over an entire OO application.
 - Cooperative control makes it difficult to achieve system state and transition coverage.
- A global state model becomes too complex for practical systems.
 - Rarely constructed by developers.
 - A global state model is needed to show how classes interact.

Activity diagram-based testing

- Activity diagrams are based on:
 - flow charts, state transition diagrams, flow graphs and Petri nets.
- Can be considered to be a flow chart that can represent two or more threads of concurrent execution.
- Supports all features of a basic flow graph,
 - Can be used to develop test models for control flow based techniques.

Activity diagram-based testing

- Activity diagram can be used to construct:
 - Decision tables
 - Composite control flow graph representing a collection of sequence diagrams.
 - Control flow graph at method scope:
 - this may be useful to analyze path coverage.

Activity diagram-based testing

- Generic test requirements:
 - At least one test case should exercise each different path:
 - Action 1 is followed by Action 2
 - Synch Point is followed by Action
 - Action is started after Signal
 - Synch Point is reached after Action

Sequence diagram-based Testing

- A sequence diagram (SD) documents message exchanges in time dimension.
- Generic tests include:
 - **Message coverage:** At least one test case should exercise each message.
 - **Message path coverage:** All end-to-end message paths in the SD should be identified and exercised.

Difficulties with Sequence Diagram-based testing

- Representing complex control is difficult:
 - **For example:** notation for selection and iteration is clumsy.
 - It becomes difficult to determine whether all scenarios are covered.
- Dynamic binding and the consequent superclass/subclass behavior :
 - Cannot be represented directly.

Difficulties with Sequence Diagram-based testing cont...

- A Sequence diagram shows only a single collaboration.
- Different bindings are shown on separate sheets.

Collaboration diagram-based testing

- Collaboration diagram represents interactions among objects.
- A collaboration diagram specifies:
 - The implementation of a use case.
- May also depict the participants in design patterns.

Collaboration diagram-based testing

- Each collaboration diagram represents only one slice of the interaction.
- A composite collaboration diagram would be necessary:
 - To develop a complete test suite for an implementation.
- Generic tests include:
 - **Message coverage:** At least one test case should exercise each message.
 - Other coverage like **Condition Coverage**⁵⁶

Component diagram-based testing

- Component diagram shows the dependency relationships among:
 - components,
 - physical containment of components,
 - interfaces and calling components.
- The basic testing practice with respect to component diagrams:
 - All call paths should be identified and exercised.

Deployment diagram-based testing

- Deployment diagram represents the hardware, software, and network architecture.
 - Useful in integration planning.
- When a component runs on a node,
 - test cases should exercise whether a component can be loaded and run on each designated host node.
- When a node communicates with another node,
 - test cases should exercise open, transmit, and close communication for each remote component.

Test Coverage

- Test coverage analysis:
 - helps determine the “thoroughness” of testing achieved.
- Several coverage analysis criteria for traditional programs have been proposed:
 - What is a coverage criterion?
- Tests that are adequate w.r.t a criterion:
 - Cover all elements of the domain determined by that criterion.

Test Coverage Criterion cont...

- But, what are the elements that appropriately characterize an object-oriented program?
 - Certainly different from procedural programs.
 - **For example:** Statement coverage is not appropriate due to inheritance and polymorphism.
- Appropriate test coverage criteria are needed.

Test Coverage Criteria Based on UML Models

- Generalization criterion
- Class attribute criterion
- Condition coverage criterion
- Full predicate coverage criterion
- Each message on link criterion
- All message paths criterion
- Collection coverage criterion

Generalization (GN) Criterion

- Given a test set T and a system model SM ,
 T must cause:
 - every specialization defined in a generalization relationship to be created.
- GN criterion fault model:
 - likely to reveal faults that can arise from violation of the substitutability principle.

Substitutability Principle

"An instance of a subclass can be used anywhere an instance of its superclass is expected."

Class Attribute Criterion

- Given a test set T , a system model SM , and a class C , T must cause
 - a set of representative attribute value combinations in each instance of class C must be created.
- The value space of attributes provides an opportunity to develop test criteria.
- The value space of an attribute are usually restricted by OCL constraints.
- Example:** age attribute in Customer class can have a constraint $age > 18$

Condition Coverage Criterion

- Each message in a collaboration diagram occurs only under certain conditions.
- Given a test set T and collaboration diagram CD , T must cause:
 - Each condition in each decision to evaluate to both TRUE and FALSE.

Full Predicate Coverage Criterion

- Given a test set T and collaboration diagram CD , T must cause:
 - Each clause in every condition in CD to take the values of TRUE and FALSE.
- A condition may consist of more than one clause connected by Boolean operators (e.g., AND, OR).

Message On Each Link Criterion

- Given a test set T and collaboration diagram CD , T must cause:
 - Message on each link connecting two objects in CD to be exercised at least once.
- This criterion ensures:
 - All possible messages between two objects occur during tests.

All Message Paths Criterion

- Given a test set T and collaboration diagram CD , T must cause:
 - Each possible message path in CD to be taken at least once.
- A message path is a sequence of messages.
- All message paths is a stronger criterion than message on each link criterion.

Collection Coverage Criterion

- Given a test set T and collaboration diagram CD , T must test:
 - each interaction with a collection of objects of various representative sizes at least once.
- **Example:** An object *Staff* can be an aggregation of *Office staff*, *Faculty* and *Technical Staff*.

Development of test suite by applying test design patterns

Test Design Pattern

- A design pattern is a generalized (reusable) solution to a recurring problem.
- It is based on common sense and some good practices.
- It provides a concise description of:
 - common elements,
 - context, and
 - essential requirements for a solution.
- Test design patterns are to testing what software design patterns are to programming.

Test Design Pattern

cont...

Patterns developed under different scope of testing:

Scope	Example Pattern
Method scope	Polymorphic Message Test
Class Scope	Modal Class Test
Reusable Components	Abstract Class Test
Subsystem	Round-trip Scenario Test
Integration	Collaboration Integration
Application Scope	Extended Use Case Pattern
Regression Test	Retest Within Firewall

Extended Use Case (EUC)Test Pattern

- Difficulties with use case :
 - No variable definitions required
 - No business rules required
 - No input/output constraints
 - Narrative can be ambiguous
- In a test-ready model we need to
 - Define all input/output relationships
 - Define all operational variables

EUC Intent

- Model the input output relationships in a use case using a decision table.
- It then becomes possible to develop test cases:
 - by considering specific inputs for a use case and corresponding expected results.

EUC Context

cont...

- Unless test points are systematically selected based on a use case's implicit constraints and relationships:
 - all fundamental relationships would not be exercised.
- **Example:** A Use Case "Pay-Fees" can be a generalization of "Pay-by-Credit-Card" and "Pay-by-Cash" Use Cases

EUC Fault Model

- What kind of bugs can EUCs help detect?
 - Domain bugs.
 - Logic bugs.
- Example: Customer registers with a Supermarket:
 - Domain error: Incorrect Pin code
 - Logic Error: Sales registered against an invalid customer code.

EUC Fault Model

cont...

- Generic system-scope faults:
 - Incorrect output.
 - Abnormal termination.
 - Inadequate response time.
 - Omitted capability.
 - Extra capability.

EUC Test Model

- An Extended Use Case consists of the following:
 - A complete inventory of **operational variables**.
 - A complete specification of domain constraints for each operational variable.
 - An operational relation for each use case
 - The relative frequency of each use case (optional).

EUC Test Procedure

- A test suite is developed in 4 Steps
 - 1. Identify the Operational Variables.
 - 2. Define the domains of the operational variables.
 - 3. Develop the Operational Relations.
 - 4. Develop Test cases.

1. Identify Operational Variables

- Operational variables:
 - Factors which vary from one scenario to the next.
 - Determine different system responses.
- Operational variables include:
 - Explicit inputs and outputs.
 - Environmental conditions which result in significantly different behavior.
 - Abstractions of the state of the system.

2. Define the Domains of the Operational Variables

- The domains are developed by defining:
 - The valid and invalid values for each variable.

3. Develop the Operational Relation

- Operational variables:
 - May determine distinct classes of system responses.
- Express them in the form of a decision table:
 - When all the conditions in a row are true, the expected action is to be produced.

4. Develop Test Cases

- Every variant is made true once and false once.
- Oracle:
 - Expected results are typically developed by inspection.

EUC Test Example

Use Case	Actor	Possible Input/Output Combinations
Establish Session	Bank Customer	(1) Wrong PIN entered once, corrected PIN entered. Display menu. (2) PIN ok, customer's bank not online. Display "Try later." (3) PIN ok, customer's accounts are closed. Display "Call your bank." (4) Stolen card inserted, valid PIN entered. Retain card.
Cash Withdrawal	Bank Customer	(1) Requests \$50, account open, balance 1,234.56, \$50 dispensed. (2) Requests \$100, account closed. (3) Requests \$155.39, account open. \$150 dispensed.
Cash Replenishment	ATM Operator with Armed Guard	(1) ATM opened, Cash dispenser is empty, \$15,000 is added. (2) ATM opened, Cash dispenser is full.

Some Use Cases and Scenarios for an Automatic Teller Machine

Variant	Operational Variables				Expected Result	
	Card PIN	Entered PIN	Customer Bank Response	Customer Account Status	Message	Card Action
1	Invalid	DC	DC	DC	Insert an ATM card	Eject
2	Valid	Matches Card PIN	Bank Acknowledges	Closed	Contact your bank.	Eject
3	Valid	Matches Card PIN	Bank Acknowledges	Open	Select a transaction	None
4	Valid	Matches Card PIN	Bank Does Not Acknowledge	DC	Please try later	Eject
5	Valid	Doesn't Match	DC	DC	Reenter PIN	None
6	Revoked	DC	Bank Acknowledges	DC	Card invalid	Retain
7	Revoked	DC	Bank Does Not Acknowledge	DC	Card invalid	Eject

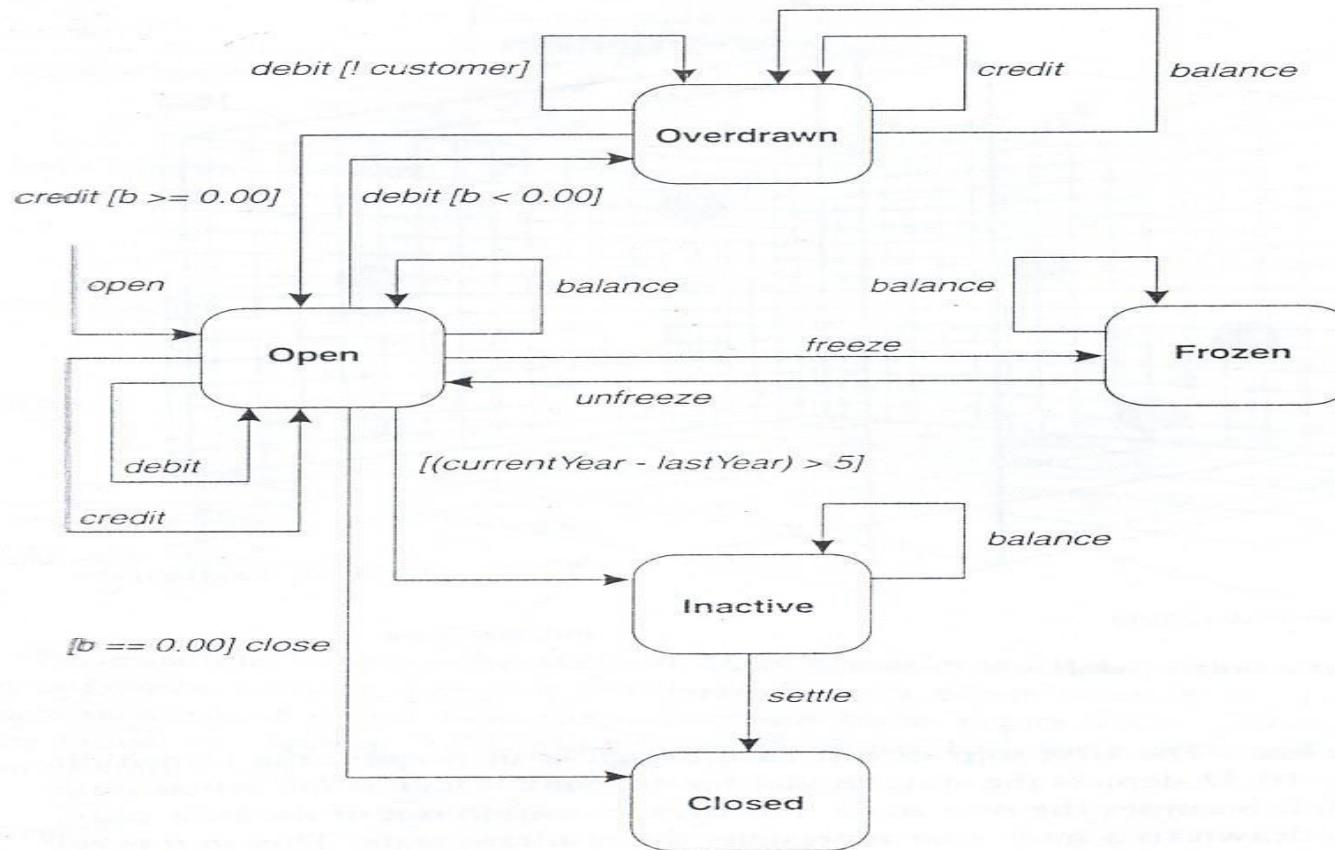
ATM Operational Relation for the *Establish Session* Use Case

Modal Class Test (MCT) Pattern

- Some classes can accept any message in any state:
 - But others restrict based on past messages received or current values (domain).
- A modal class is a special class :
 - Places domain constraint as well as sequence constraint while accepting messages.
- **Domain Constraint:** An object of class Account will not accept a withdrawal message if the balance is less than 0.
- **Sequence Constraint:** In the frozen state, an Account object can accept a credit or debit message only after an unfreeze message.

Modal Class Test (MCT)

cont...



State transition diagram for class Account.

MCT Fault Model

- A class may fail to implement its state model in five ways:
 1. **Missing transition:** A valid message is rejected in a valid state.
 2. **Incorrect action:** A wrong response for an accepting state and valid message.
 3. **Invalid resultant state:** A message results in transition to a wrong state.
 4. **Invalid state:** An invalid state is produced.
 5. **Sneak path:** A sneak path allows a wrong message to be accepted.

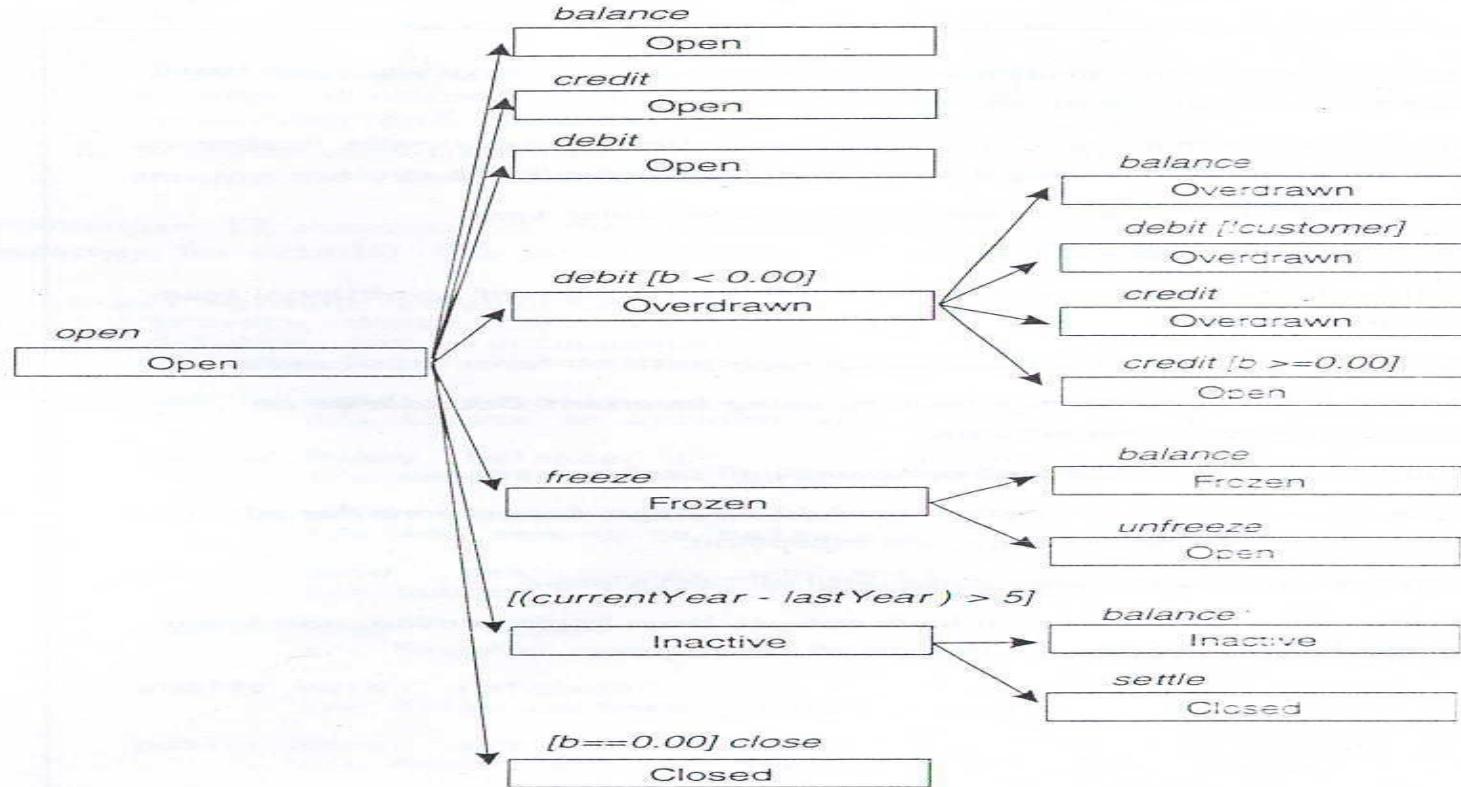
MCT Test Strategy

- The behavior of a class is represented using a state model.
- Generate the transition tree.
- Perform conditional transition test.
- Tabulate events and actions along each path to form test cases.
- Perform illegal transitions and sneak path test.

MCT Transition Tree

- For each transition out of the root state ,
 - A branch is drawn to a node that represents the resultant state.
 - This step is repeated for each resultant state node.
- A node is marked terminal :
 - if no transition to a new state (yet to be drawn) exists.
- Each branch in the tree (an event path) becomes a test case.

MCT Transition Tree Testing: Example



MCT Conditional Transition Test

- Each conditional transition:
 - Analyzed to identify additional test cases.
- For each conditional transition :
 - Develop a truth table for the variables in the condition.
- Develop an additional test case:
 - For each entry in the truth table that is not already exercised.

MCT Sneak Path Test

- Sneak Path is a bug that allows an illegal message to be accepted,
 - Resulting in an illegal transition.
- Illegal Transition is present:
 - When a valid state of CUT accepts a message not specified for that state.
- Illegal Message:
 - Results in an illegal transition.

MCT Sneak Path

cont... .

- To test sneak paths send illegal messages:
 - **Example:** In the overdrawn state, messages: open, freeze, close etc can lead to possible sneak paths.
- To confirm that no sneak paths exist:
 - Attempt each type of illegal message.
- The expected response:
 - The message should be rejected.
 - Also, the state of the object should be unchanged after rejecting the illegal message.₉₃

Collaboration Integration Pattern(CIP)

- CIP chooses the order of integration:
 - according to collaborations and dependencies.
- Classes to be integrated are tested by testing one collaboration at a time.
- A system consists of many collaborations.
 - Their sequence is determined by dependencies and sequential activation constraints.
- Integration by collaboration:
 - Exercises interfaces between the participants of a collaboration.

CIP Test Procedure

- 1. Develop a dependency tree for the SUT.
- 2. Map collaborations onto tree until all components and interfaces are covered.
- 3. Choose a sequence in which to apply the collaborations.
- 4. Several heuristics are available for choosing a sequence.

CIP Test Procedure cont..

Example Heuristics:

1. Begin with the simplest and finish with the most complex.
2. Begin with the collaboration that requires the fewest stubs .
3. Test in order of risk of disruption of system testing.

CIP Test Procedure

cont...

4. Develop test suite for each collaboration
5. Run the test suite and debug until first collaboration passes
6. Continue until all collaborations have been exercised

Integration Strategies

- Essentially two approaches exist
 - Thread-based
 - Use-based

Thread-based integration Strategies

- A thread consists of:
 - All the classes needed to respond to a single external input.
- Each class is unit tested,
 - then each thread is exercised.

Use Case-based Integration Strategies

- Use case-based integration:
 - Begins by testing classes that use services of none (or a few) of other classes.
 - Next, classes that use the first group of classes are tested.
 - Followed by classes that use the second group, and so on.

Research Challenge 1: Testing Based on Precode Artifacts

- Precode artifacts:
 - Design,
 - Requirements,
 - Architecture specifications.
- Software architecture involves description of elements from which systems are built,
 - Interactions among those elements,
 - Patterns that guide their composition,
 - Constraints on these patterns.

Research Challenge 2: Testing Evolving Software

- Regression testing is used to test software that undergoes evolution due to:
 - Technology changes,
 - New or modified components.
- Regression testing remains one of the most expensive activities performed during a software lifecycle.

Research Challenge 3: Measuring Effectiveness Of Testing Techniques

- Quantitative values of effectiveness of a test-set design in revealing faults:
 - Analytical, statistical, or empirical.
- We also need to understand the classes of faults for which the different criteria are useful.

Research Challenge 4: Test Data Generation

- Generating test data (inputs for test cases) is often a labour-intensive process.
- To date, available techniques generating test data automatically work for toy systems:
 - Do not scale to large systems.
- Also, it should be possible to automatically execute test cases:
 - The test output should also be automatically analyzed.

Research Challenges 5: Methods and Tools

- Along with fundamental research and empirical methods for testing:
 - tool development is also needed.
- An important requirement:
 - Methods and tools be scalable to large systems.
 - Many of the tools and methods developed so far work only on toy systems.

Summary

- Discussed introduction to O-O S/W testing
- Challenges in testing OO programs
- Test suite design using UML models
- Test design patterns
- Current research challenges

References

1. R. Mall, Fundamentals of Software Engineering, Fifth Edition, (Chapter – 10), PHI, 2018.
2. R. S. Pressman, Software Engineering, McGraw-Hill, 2018.
3. N. Chauhan, Software Testing; Principles and Practices, Second Edition, (Chapter – 14), Oxford University Press, 2018.

Thank You