Testing Background

- History and Background
  - Problems of past still exist today
    - High cost
      - Productivity = Lines of code / hour (avg 1/hr)
    - Difficult to manage
    - poor reliability
    - lack of user acceptance
    - difficult to maintain
- MTBF = Mean Time Between Failure

- Reliability and Testers
  - Software defects are around 1 defect / LOC
  - Testers vs (Scientific) Pollsters
    - Both paid to make predictions
    - Pollsters have constraints [Money, and time]
    - "exhaustive" ["representative samples"]
- Define common testing terminology (Derived mainly from IEEE and ACM)
  - Reliability: The probability that a software program operates for some given time period without software error
  - Validation: "Are we building the right product?"
  - Verification "Are we building the product right?"
  - Testing: The examination of behavior of the program by executing it on sample data sets
  - Error: Mistake made by a human (ex: forgot about requirement)
  - Defect/Fault: Result of error manifested in the code
  - Failure: Software doesn't do what it is supposed to do (triggered by defects/faults)
- Testing Background Practice Quiz
  - True or False? Verification focuses on if the requirements are meeting customer needs and are what the customer wants.
    - False: Verification looks at building the product right given the assumption that the requirements are correct. Validation looks at building the right product to bring value and meet the customer's requirements.
  - A team of testers are testing software against a given set of requirements. The requirements have already approved by the customer. Is the testing team performing verification or validation of the software, or neither?
    - Verification: Verification looks at building the product right given the assumption that the requirements are correct. Validation looks at building the right product to bring value and meet the customer's requirements.

- Classic Testing Mistakes:
    - The role of testing
        - Thinking that the purpose of testing is to find bugs
        - Starting testing too late (bug detection, not bug reduction)
    - Planning the complete testing effort
        - A testing effort biased toward functional testing
        - An overreliance on beta testing (customer testing)
        - Sticking stubbornly to the test plan
    - Personnel issues
        - Using testing as a transitional job for new programmers
        - A physical separation between developer and testers
        - Believing that programmers can't test their own code
        - Programmers are neither trained nor motivated to test
    - Test automation
        - Attempting to automate all tests
    - Code Coverage
        - Embracing code coverage with the devotion that only simple numbers can inspire
        - Using coverage as a performance goal for testers

Testing Throughout Life Cycle

- Describe how testing is integrated into software development phases
- Waterfall
    - Requirements > Design > Code > Test
    - Do not begin next step until the previous step is done (rigid)
    - Testing occurred at the end, until code is complete
    - Testers typically involved in requirements and each step, review the requirements to ensure that they were indeed testable. Thinking about system level requirements.
- Agile
    - Emphasizes high degree of customer interaction with development process
    - Continuous Integration (at least daily)
        - Static Code Analysis (automated)
        - Compile
        - Unit Test
        - Deploy into Test Environment
        - Integration / Regression Test
- Test Driven Development [TDD] (Red, Green, Refactor Cycle)
    - Emphasis on writing test cases first, use test cases to define behavior that software developer needs to implement
    - Red Phase: Write a minimal test on the behavior needed
    - Green Phase: Write only enough code to make the failing test pass
    - Refactor Phase: Improve code while keeping tests green

- Software Development Process vs Test Development Process
    - Requirement ~ Test objectives (what do we want testing to accomplish? Stress, usability, etc.)
    - Design ~ Test Design (sampling strategy)
    - Code ~ Write Test Cases / Test Scripts
    - Test ~ Executing Tests
    - Maintenance ~ Update tests
- Define the objectives of the different levels of testing
- Testing Levels
    - Unit / Component Testing
        - Individual developer, make sure their unit does what its supposed to, To verify that an individual developer's unit is tested properly. Maybe TDD. High degree of automation with tools like JUnit. Objective is to make sure unit / component does what its supposed to do. Verification.
    - Integration Testing
        - May happen daily or more. Objective is to make sure that new units are working together with existing systems.
    - System Testing
        - Software has been integrated together, but hardware / software integration is a concern. Objective: To test both functional and nonfunctional requirements. Still worried about functional capabilities, but also security, performance, stress, usability, safety, etc. Functional and nonfunctional testing as well.
    - Acceptance Testing
        - Customer testing. Customer needs to be able to install or set up. Might have independence being done or by customer.
    - Beta Testing
        - Some sort of preferred testing. Separate sort of customer and let them do some form of testing. Risks: There might be problems damaging to the company.
- Test Types
    - Functional
        - Verification, validation, making sure the software does what it's supposed to be doing. Pretty much applicable to each level of testing.
    - Non-functional
        - "ilities". secure, performance, usable. Higher levels of testing.
    - Structural
        - White box, making sure code logic is good. Dataflow coverage. Primarily at the unit level.
    - Regression
        - Applicable at all levels. Making sure that new changes don't break the code.

- Testing Throughout Life Cycle Practice Quiz
    - How often does testing happen in agile development?
        - At least daily Agile testing follows the theme of continuous integration where code is integrated and tested at least daily.
    - What is the correct phase order in test driven development?
        - Red, Green, Refactor. The red phase is where a small test is written that defines the behavior needed from the code. Then, in the green phase, code is written to ensure the failing test from the previous phase passes. In the refactor phase, the quality of code is improved while all tests are still passing.
    - What is the objective of system testing?
        - To test both functional and nonfunctional requirements. System testing is when other components, such as hardware and software integration testing (functional requirements) and aspects such as stress, usability, and performance testing (nonfunctional requirements) are tested.
    - What is the objective of unit level testing?
        - To verify that an individual developer's unit is tested properly. Unit level testing is where each developer tests their unit (section of code) to ensure it is working properly

Testing Best Practices and Standards

- Testing Principles
    - Principle 1
        - Testing only shows the presence of defects - proof of correctness
    - Principle 2
        - Exhaustive testing is impossible
    - Principle 3
        - Start testing early
    - Principle 4
        - Defects cluster. Individual developer gets sloppy on one section of code or is a novice, some portions of code are more complex than other. Not realistically 1 error per 1000 LOC.
    - Principle 5
        - Testing is context dependent. Test cases may be based on system being in a particular case, maybe what has happened prior to running the test case? Context could be location, time of day.
    - Principle 6
        - Absence-of-errors fallacy.
- Testing Attitude
    - Independence
        - Realize bias. In a perfect world, all testing would be done independently.
    - Customer Perspective

        - Include testing from customer's perspective. Try what customer would try. Sometimes we don't have the big picture of how the software will be used. Validation.
    - Demonstrate that the system works (test intended functionality)
        - Tester's go in and try to break the system.
    - Demonstrate that the system is bullet proof (test unintended functionality)
    - Professionalism
        - Need certifications, ensure that testers are educated and qualified.
- Classic Testing Mistakes
    - Believing the primary objective of system testing is to find bugs
        - Test must concentrate on finding important problems
        - Test must provide an estimate of system quality
    - Not focusing on usability issues
        - Testers not concerned about the success about the product, just what's their job
    - Starting too late
        - Test must help development to avoid problems
    - Delaying stress and performance testing until the end
    - Not testing the documentation [Customer]
    - Not staffing the test team with domain experts
        - Need experts who understand the customer and the customer's perspective
    - Not communicating well with the developers
    - Failing to adequately document and review test designs
- General Testing Standards (ISO/IEC/IEEE 29119 Software Testing)
    - ISE/IEC: Concepts and Definitions
    - ISE/IEC: Test Processes
    - ISE/IEC: Test Documentation
    - ISE/IEC: Test Techniques
    - ISE/IEC: Keyword Driven Testing

- DO-178C Software Considerations in Airborne Systems and Equipment Certification (example)
    - Level A: Catastrophic
    - Level B: Hazardous/Sever
    - Level C: Major
    - Level D: Minor
    - Level E: No Effect
    - Each level would have a test coverage/specific kinds of tests required.
- When to Stop Testing
    - Out of time / money (probably bad answer)
    - No more defects found (probably bad, maybe testing not good anymore)
    - Demonstrated all requirements are met

    - Demonstrated Code coverage
    - Meets reliability objectives
    - Customer is satisfied
- ISTQB Code of Ethics
    - Public:
        - Certified software testers shall act consistently with the public interest
    - Client and Employer
        - Certified software testers shall act in a manner that is in the best intersects of their client and employer, consistent with the public interes
    - Product
        - Certified software testers shall ensure that deleverables they provide meet highest professional standards possible
    - Judgement
        - Certified software testers shall maintain integrity and independence in their professional judgement
    - Management
    - Certified software test managers and leaders shall subscribe to and promote an
    - Profession
    - Colleagues
    - Self
- Testing Principles and Best Practices
- Which group of people should perform system testing
    - An independent test team. A testing attitude is to gain independence to counter bias during testing.
- What is(are) the primary objective(s) of system testing?
    - To find important problems and predict reliability. Finding important problems that affect the customer greatly is more important than finding many minor problems in system testing. Testing must also provide an estimate of reliability.

- When is a reasonable time to stop testing?
    - Once we have met our test objectives. Test objectives typically include testing all of the functional and nonfunctional requirements.
- Why do defects cluster?
    - Because of the complexity of code, programmer skill, etc. Defect clustering happens for many reasons. Some parts of the code are more complex, causing more defects.

- Explain best practices for software testing
- Best Testing Practices
    - Assess software reliability via statistical testing
    - Develop an agile test design
        - Accommodate late changes
        - Emphasis on regression testing (new changes don't break current code)
    - Utilize model-based testing techniques
        - State diagrams
        - Develop a model, and use that model to generate code
    - Develop cross-functional development and test teams
    - Automate Test generation where possible
    - Emphasize usability testing

- Software testing best practices: Review of reading
    - Reviews and inspections.
        - Also review and inspect test cases and test strategies
    - Formal Entry and Exit Criteria
        - What point do start testing like beta testing?
        - When can we stop testing?
    - Multi-platform testing
        - Test code on multiple configurations such as different phones, different OS, etc
    - Automated Test Execution
        - Do automated test where we can
    - In-Process ODC (orthogonal defect classification) feedback loops
        - Measurement method that uses the defect stream to provide precise measurability into product and process.
        - Go in and look at defects, to be able to classify them, their priority, why were they made. Look at that info and have a feedback mechanism in place so we can learn from our mistakes.
    - Requirements for Test Planning
    - Usability Testing
        - Critical
    - Teaming testers with developers, make them work together
    - Code Coverage
        - Use tools to ensure we achieve adequate code coverage, but not excess
    - Automated environment generator
        - Build code based on test cases
    - Testing to help ship on demand
    - State Task Diagram
        - Model based testing as known today
    - Statistical testing
    - Semi-Formal Methods

        - Program verification techniques
    - Bug Bounties
        - Incentive testers to find bugs in the system. Known as crowd testing today. Individuals can be hired and rewarded for finding bugs.
- Automations Role in the Fall of Software Testing
    - Sometimes we put too much focus on trying to automate
    - But it plays a key role in regression testing
    - Sometimes automation leads to erroneous conclusions that we don't need as many testers, or we don't need to spend as much time on testing.
    - Automation tests usually imply shallow testing, easy to create scripts, validate boundary values. But deeper kinds of testing need more involvement usually.
    - Summary: automation has a role, but software development needs testers who have a deep cognitive knowledge to perform different kinds of testing.

Unit 1 Quiz

Question 1

True or False? Reliability is the probability that a software program operates for some given time period without software error.

- True

Question 2

True or False? Testing is the examination of the behavior of the program by executing it on all of the possible data sets.

- False, not **all possible data sets**, just a sample of data

Question 3

In a traditional phase driven waterfall model, when does the test phase start?

- After the coding phase is complete.

Question 4

True or False? Certified software testers should ensure that the deliverables they provide meet the highest professional standards possible.

- True

Question 5

In test driven development, what is the equivalent of gathering requirements during the software development process?

- Gathering testing objectives (What do we want testing to accomplish? Stress, usuability, etc)

Question 6

True or False? The objective of integration testing is to test units and components as a group to determine if they can perform higher level functions and features.

- True

Question 7

True or False? Acceptance testing is typically done by the customer themselves.

- True

Question 8

True or False? Functional testing looks at performance requirements, usability, etc.

- False. Functional testing looks at Verification, validation, making sure the software does what it's supposed to be doing. **Nonfunctional testing** looks at "ilities". secure, performance, usable. Higher levels of testing.

Question 9

Testing only shows the:

- Presence of errors, not the presence of correctness or the absence of errors.

Question 10

True or False? Does an absence of errors in a program show that the right product was built (validation)?

- False. The absence of errors in a program show that the software built the product right (verification). This is an example of verification vs validation. Verification: "Are we building the product right?". Validation: "Are we building the right product?". The absence of errors shows that we built the software correctly, but not necessarily that we solved the customer's problem.

Unit 2 Notes

- Contrast requirement-based versus scenario-based testing
- Apply the equivalence partitioning testing technique
- Apply cause-effect testing technique
- Test asynchronous events
- Apply state-based testing technique
- Describe model-based testing strategies
- Apply the boundary value testing technique

Input Sampling Techniques Part 1

- Contrast requirement-based versus scenario-based testing
    - Find in slides

Scenario based testing: Review of reading

- Use cases.
- Can have a normal scenario, and multiple different scenarios.
- Each scenario will have a test cases, maybe more than one
- Characteristics of good scenarios
    - 5 key characteristics. It is a story that is motivating, credible, complex, and easy to evaluate
- Twelve ways to Create Good Scenarios
    - Designing scenario tests is much like doing a requirements analysis, but is not requirements analysis. They rely on similar information but use it differently.
        - The requirements analyst tries to foster agreement about the system to be built. The tester exploits disagreements to predict problems with the system
        - The tester doesn't have to reach conclusions or make recommendations about how the product should work. Her task is to expose credible concerns to the stakeholders
        - The tester doesn't have to make the product design tradeoffs. She exposes the consequences of those tradeoffs, especially unanticipated or more serious consequences than expected.
        - The tester doesn't have to respect prior agreements. (Caution: testers who belabor the wrong issues lose credibility.)
        - The scenario tester's work need not be exhaustive, just useful.
    - Consider disfavored users: how do they want to abuse your system?
        - As Gause and Weinberg point out, some users are disfavored. For example, consider an accounting system and an embezzling employee. This user's interest is to get more money. His objective is to use this system to steal the money. This is disfavored: the system should make this harder for the disfavored user rather than easier
    - List "system events." How does the system handle them?
        - An event is any occurrence that the system is designed to respond to. In Mastering the Requirements Process, Robertson and Robertson write

about business events, events that have meaning to the business, such as placing an order for a book or applying for an insurance policy. As another example, in a real-time system, anything that generates an interrupt is an event. For any event, you'd like to understand its purpose, what the system is supposed to do with it, business rules associated with it, and so on. Robertson and Robertson make several suggestions for finding out this kind of information.

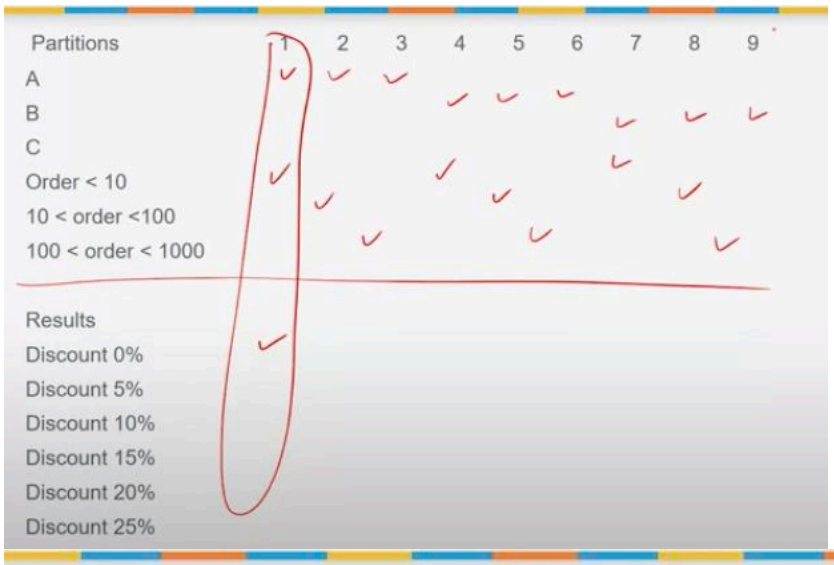
- Interview users about famous challenges and failures of the old system.
  - Meet with users (and other stakeholders) individually and in groups. Ask them to describe the basic transactions they're involved with. Get them to draw diagrams and explain how things work. As they warm up, encourage them to tell you the system's funny stories, the crazy things people tried to do with the system. If you're building a replacement system, learn what happened with the predecessor. Along with the funny stories, collect stories of annoying failures and strange things people tried that the system couldn't handle gracefully. Later, you can sort out how "strange" or "crazy" these attempted uses of the system were. What you're fishing for are special cases that had memorable results but were probably not considered credible enough to mention to the requirements analyst. Hans Buwalda talks about these types of interviews (www.stickyminds.com).
- Try converting real-life data from a competing or predecessor application.
  - Running existing data (your data or data from customers) through your new system is a time-honored technique.
  - A benefit of this approach is that the data include special cases, allowances for exceptional events, and other oddities that develop over a few years of use and abuse of a system.
  - A big risk of this approach is that output can look plausible but be wrong. Unless you check the results very carefully, the test will expose bugs that you simply don't notice. According to Glen Myers, The Art of Software Testing, 35% of the bugs that IBM found in the field had been exposed by tests but not recognized as bugs by the testers. Many of them came from this type of testing.

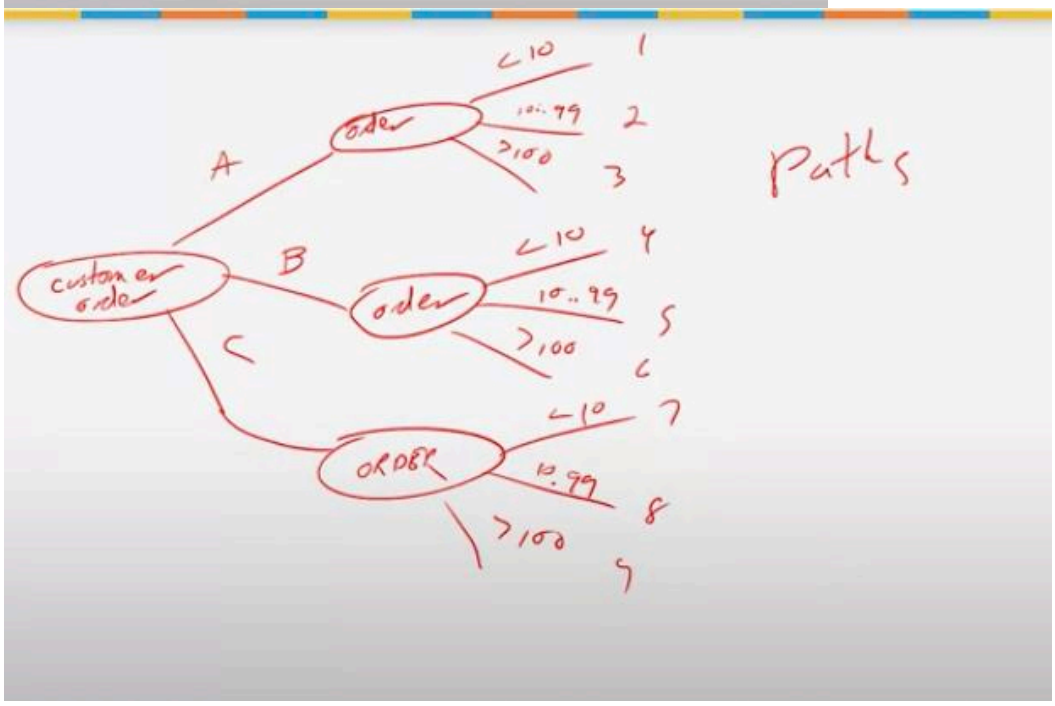
Input Sampling Techniques Part 2

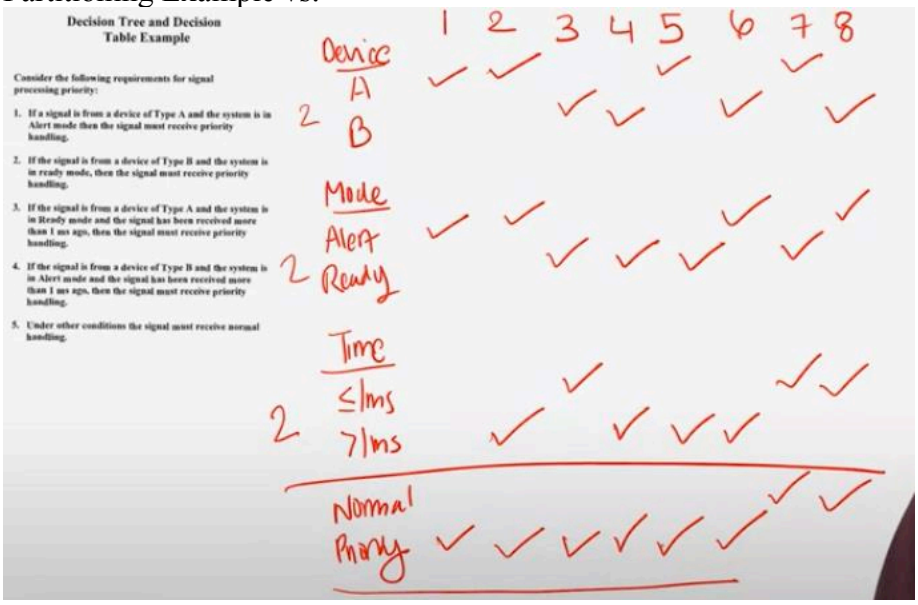
- Equivalence Partitioning Testing
  - See slides
- Equivalence Partitioning Testing Knowledge check
  - True or False? Equivalence partitioning is a good technique to utilize when there are multiple independent inputs.
    - True. Equivalence partitioning must be applied with independent inputs. In the lecture example of abs(x), it is necessary to test a negative value, 0, and a positive value which are all independent inputs.
  - What is NOT an equivalence partitioning step?
    - Write test cases covering as many of the uncovered invalid equivalence partitions as possible. There must only be one test case that covers each uncovered invalid equivalence partition at a time. Since equivalence

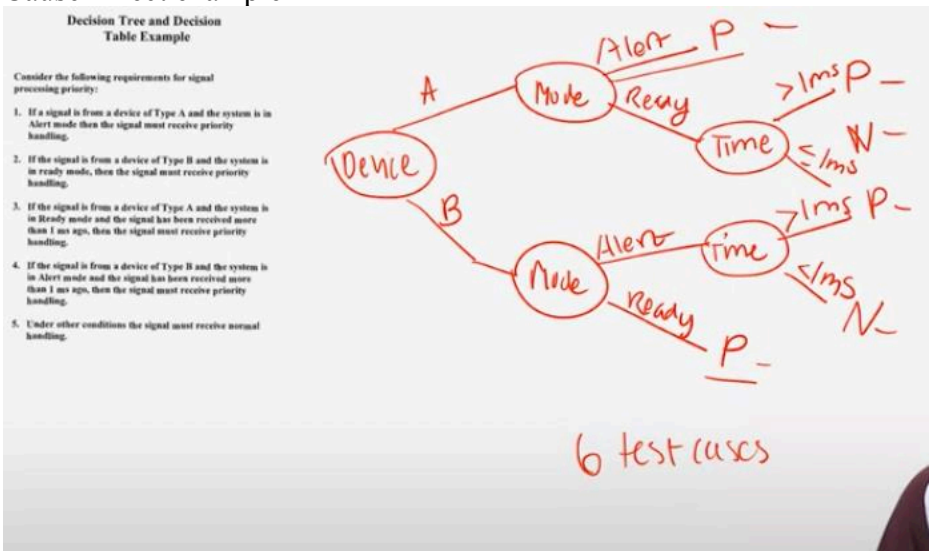
partitioning is used during black box testing, if there are multiple invalid partitions tested, it is not known which invalid partition caused the error.

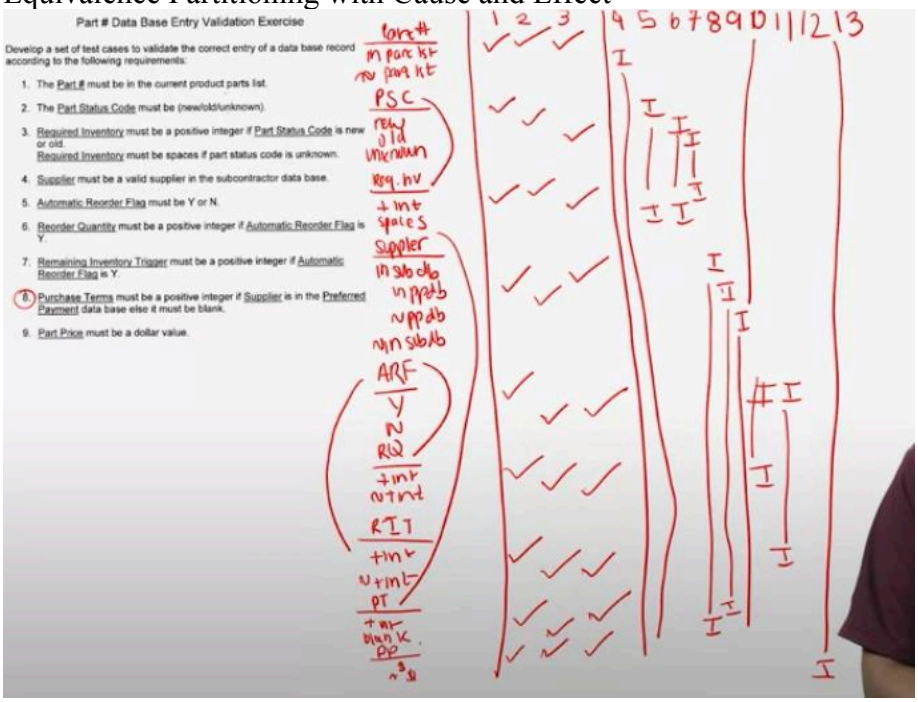
    - What are a good set of equivalence partitions for a password testing program where a password must be between 6-8 characters?
        - 6-8
        - < 6 characters
        - > 8 characters
        - These equivalence partitions test all the possible inputs there could be for a password: a valid password between 6-8 characters, an invalid password with < 6 characters, and an invalid password with > 8 characters.
- Boundary Value Testing
    - See slides
- Boundary Value Testing Knowledge check
    - A valid student ID can only contain digits between and including 2 and 7. What are the correct boundary values that need to be tested?
        - 1,2,7,8. 1 and 2 test valid and invalid values for the first boundary of a digit greater than or including 2. 7 and 8 test valid and invalid values for the second boundary.
    - True or False? Boundary value analysis requires selecting test cases that are only on or above the edges of the input and output equivalence partitions.
        - False. BV analysis requires selecting test cases that are on, above, and below the edges of the partitions.
- BV EP Testing: Review of Reading
    - Important points
        - Uni dimensional equivalence partitioning
        - multi-dimensional ep
        - EPxCE (Cause Effect)

- Case Effect Analysis
  - See slides
  - 

  - 

- Partitioning Example vs.
- 

- Cause Effect example
- 

- Equivalence Partitioning with Cause and Effect
  - 

  -

Cause Effect Analysis Knowledge Check

- Given 2 input variables, age and height, an output of vitamins, and a function that computes the number of vitamins a person should take based on age and height, which method(s) can be used to test this example?
  - Cause Effect Analysis. Cause effect analysis can be used when the inputs are dependent on each other. Equivalence partitioning is only used when inputs are independent of each other.
- What is one way to reduce the number of test cases in a cause-and-effect decision table?
  - Make assumptions about how the partitions are related. From the example in lecture, making assumptions about how error handling between a customer and order happens allows us to reduce the number of test cases needed.

Input Sampling Techniques Part 3

Testing Asynchronous Events

- See Slides

Testing Asynchronous Events Knowledge Check

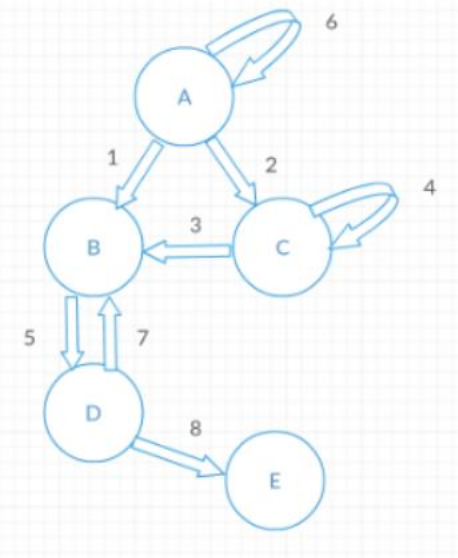
- Does creating a timeline to test synchronous events help with verification or validation?
  - Validation. Adding in events in a timeline helps identify missing requirements, which is necessary for validation.
- True or false? A timeline corresponds to a set of use cases mapped against time.
  - False. Each use case requires its own timeline.

State Based Testing

- See slides

State Based Testing Knowledge check

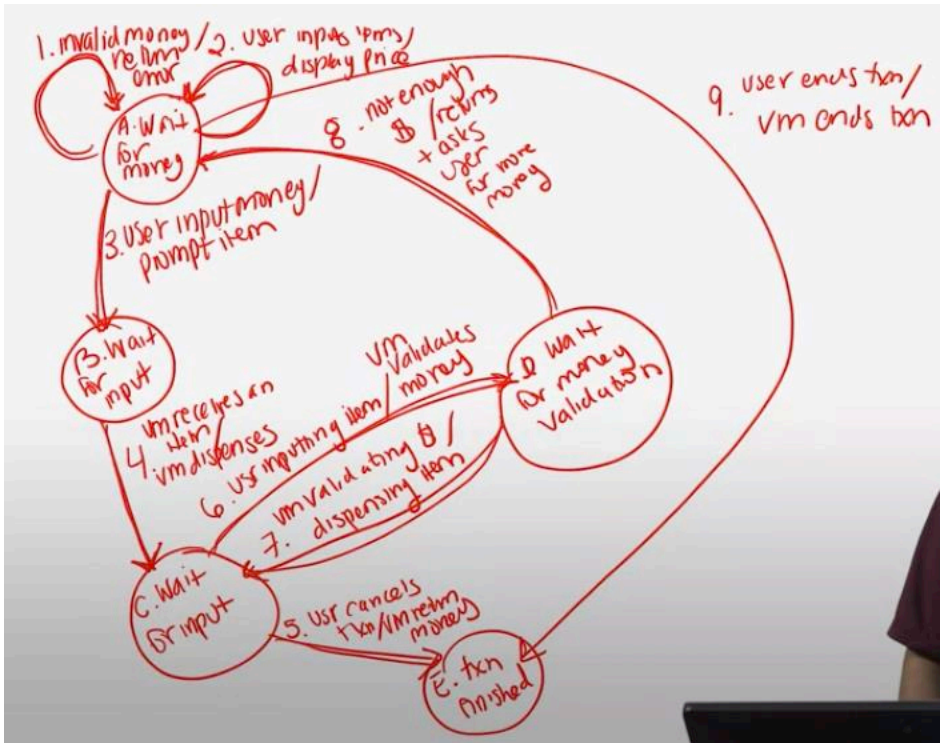
- True or False? The basic strategy of testing a state machine is to visit each state and test each transition.
    - True. Testing each state/event combination provides a minimal state machine cover.
- Given the state diagram below with start state A and end State E, which of the following is a test one would derive from the testing tree.



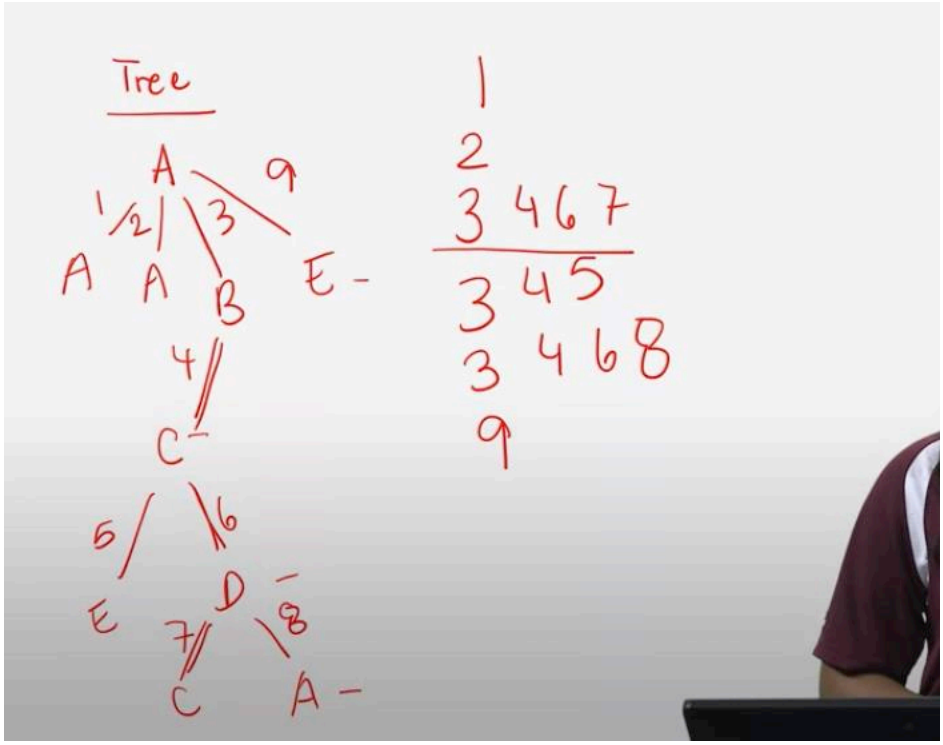
    - 1,5,8. The starting transition is 1, and then through transition 5 and 8, we reach the end state.

State Based Testing Problem example

State machine

- 

Tree derived from state machine

- 

Model Based Testing

Model Based Testing

- See Slides

Model Based Testing Knowledge Check

- What is NOT considered to be an advantage of model based testing?
  - Modeling Requires manual test case generation. Modeling gives potential for automated test generation, and does not require manual test case generation.
- What is the correct order of steps for model based testing?
  - Create a system model, select test generation criteria, generate tests, execute tests.

Unit 3 Notes

Learning Objectives

- Apply combinatorial test coverage to assess test quality
- Apply design of experiments to develop tests
- Understand mutation testing
- Understand fuzz testing
- Define metamorphic testing
- Apply defect-based testing techniques
- Describe the role of exploratory testing

Combinatorial Testing Techniques

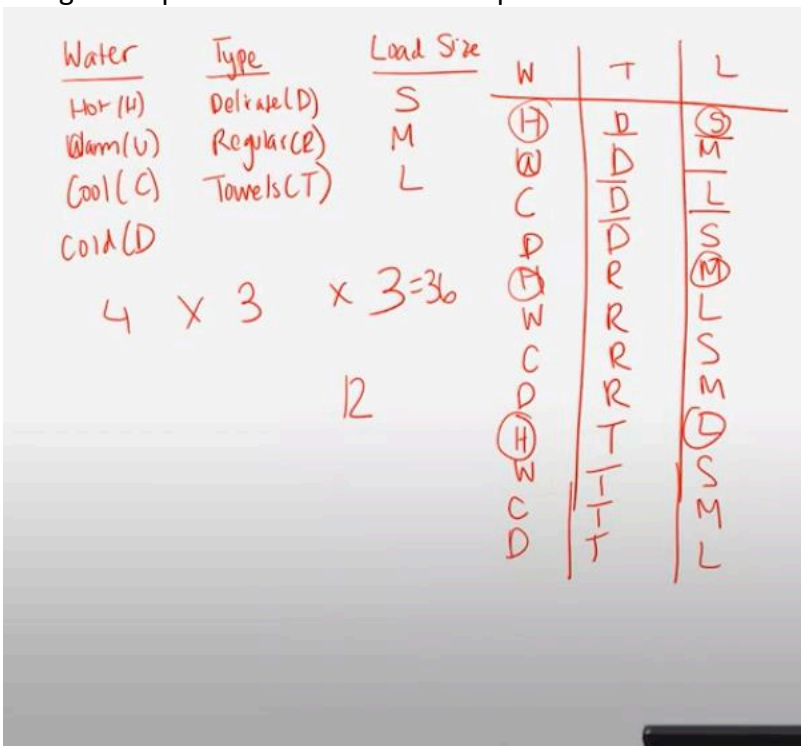
Combinatorial Coverage as an Aspect of Test Quality

- See slides

Design of Experiments

- See slides

Design of Experiments: Problem Example **



Combinatorial Testing Techniques Problem Example

- Combinatorial coverage looks at parameter values being individually tested.
    - False. Combinatorial coverage looks at how combinations of parameter values are tested together.

2. Assume we are testing a function with 3 variables:

   Variable A: has values 0 and 1

   Variable B: has values 0 and 1

   Variable C: has values 0 and 1

   What is the total 2-way variable value configuration coverage achieved by the following tests:

   A=0; B=0; C=0

   A=0; B=1; C=1

   A=1, B=1, C=0

   - (●) 9/12
   - ( ) 7/12
   - ( ) 10/12
   - ( ) 8/12

   ✓ **Correct**
   Possible combinations: A=0 B=0; A=0 C=0; B=0 C=0; A=0 B=1; A=0 C=1; B=1 C=1; A=1 B=1; A=1 C=0; B=1 C=0

-
- What is the goal of design of experiments?
  - Minimize the number of tests we need to run. We are testing pairs of values for each input to minimize the number of tests.
- True or False. Design of experiments pairwise combination involves systematically testing all combinations of inputs.
  - False. Only pairs of values for each input are tested together, not all combinations of values of inputs.
- Given 3 inputs: P1 with values V1 and V2; P2 with values V3, V4, and V5 and P3 with values V6 and V7, what are the correct tests for a pairwise combination design of experiments?

◉

V3	V1	V6
V3	V2	V7
V4	V1	V7
V4	V2	V6
V5	V1	V6
V5	V2	V7

○

V3	V1	V6
V3	V2	V7
V4	V1	V7
V4	V2	V6

✓ **Correct**
This table tests every combination of pairs of values.

○

Using Combinatorial Testing to Reduce Software Rework: Review of Reading

-

Combinatorial Coverage as an Aspect of Test Quality: Review of Reading

-

Mutation Testing

Mutation Testing

- See slides

Mutation Testing: Knowledge check

- What is NOT an example of a mutation?

2. What is NOT an example of a mutation?

○ Modifying Boolean expressions

○ Modifying variables

◉ Renaming a class and anywhere that the class is called or found

○ Modifying arithmetic operations

✓ **Correct**
If a class is renamed everywhere, that would not introduce an error (mutant) in the program.

○

Mutation Testing: Review of Reading

-

Fuzz Testing

Fuzz Testing

- See slides

Fuzz Testing Knowledge Check

- True or False? Fuzz testing consists of random, invalid or unexpected inputs that are created automatically.
    - Fuzz testing is an approach to testing where invalid, random or unexpected inputs are automatically generated.
- True or False? Fuzz testing looks only for undesirable behavior or crashes.
    - Fuzz testing is not looking at specific inputs or outputs, but is instead looking for an error or a wrong behavior.

Metamorphic Testing

Metamorphic Testing

- See slides

Metamorphic Testing Knowledge Check

- True or False? Metamorphic testing makes the assumption that if there is a program with input x that results in output y, and there is a change to input x, that same change is not reflected in output y.
    - False. Metamorphic testing makes the assumption that when changes are made to an input, it is possible to predict changes on the output.
- Without using a calculator, what would be the expected output of this example using metamorphic testing for the third test case?:
    - (◉) 7.2
    - ( ) 14.4
    - ( ) 3.6
    - ( ) 28.8

    ✓ **Correct**

    When the values are incremented by 5, the standard deviation was 7.2 In the third test, the values are incremented by 5.

Defect Based Testing

Defect Based Testing

- See slides

Defect Based Testing Knowledge Check

- True or False? Defect based testing can only be applied at the unit level.
    - False. Defect based testing can be applied at any level of testing.
- True or False? Defect based testing looks to create test cases that target specific defect categories.
    - True. Defect based testing can target any defect category from the Beizer Generic Defect Taxonomy Categories.

Exploratory Testing

Exploratory Testing

- See slides

Exploratory Testing Knowledge Check

- True or False? In exploratory testing, all test scripts are not developed in advance.
    - True. What is tested next is based on the results of the previous tests.
- True or False? Exploratory testing focuses on a tour that helps detect a specific error.
    - True. Exploratory testing can consist of requirements, features, continuous use, documentation, etc. tours that focus on different errors.

Unit 3 Quiz
All Correct

1. Assume we are testing a function with 3 variables:

   Variable A: has values 0 and 1

   Variable B: has values 0 and 1

   Variable C: has values 0 and 1

   What is the total 3-way variable value configuration coverage achieved by the following tests:

   A=0; B=0; C=0

   A=0; B=1; C=1

   A=1, B=1, C=0

   A=1, B=1, C=1

   - ○ 6/12
   - ○ 3/8
   - ◉ 4/8
   - ○ 4/12

2. Assume we are testing a function with 3 variables:

   Variable A: has values 0 and 1

   Variable B: has values 0 and 1

   Variable C: has values 0 and 1

   What is the total 2-way variable value configuration coverage achieved by the following tests:

   A=0; B=0; C=1

   A=0; B=1; C=1

   A=1, B=0, C=0

- ○ 3/8
- ○ 7/12
- ◉ 8/12
- ○ 9/12

3. True or False? Design of Experiments allows for examination of both single and combinations of inputs.

- (●) True
- ( ) False

4. True or False? Defect based testing utilized defect taxonomies to create test cases to target certain defects.

- (●) True
- ( ) False

5. Given 3 inputs: P1 with values V1, V2, V3; P2 with values V4, V5, V6 and P3 with values V7 and V8, how many tests are there for a pairwise combination design of experiments?

- (●) 9
- ( ) 18
- ( ) 6
- ( ) 8

6. True or False? Assume all test cases pass for program x. Assume a mutant of the program y is created and all test cases also pass for y. This provides confidence that the original test cases are good.

- ○ True
- ◉ False

7. True or False? Automation helps run large numbers of tests against the original program and mutants of a program

- ◉ True
- ○ False

8. True or False? An assumption of mutation testing is the belief that detecting small errors can also help detect complex errors.

- ◉ True
- ○ False

9. True or False? Mutation fuzz based testing is based upon a testing "seed", otherwise known as a valid test case.

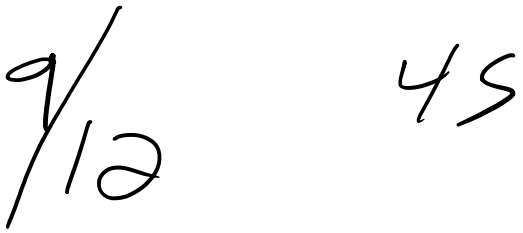
- ◉ True
- ○ False

10. True or False? Fuzz testing requires having a test oracle.

- ○ True
- ◉ False

Some Classic Testing Mistakes

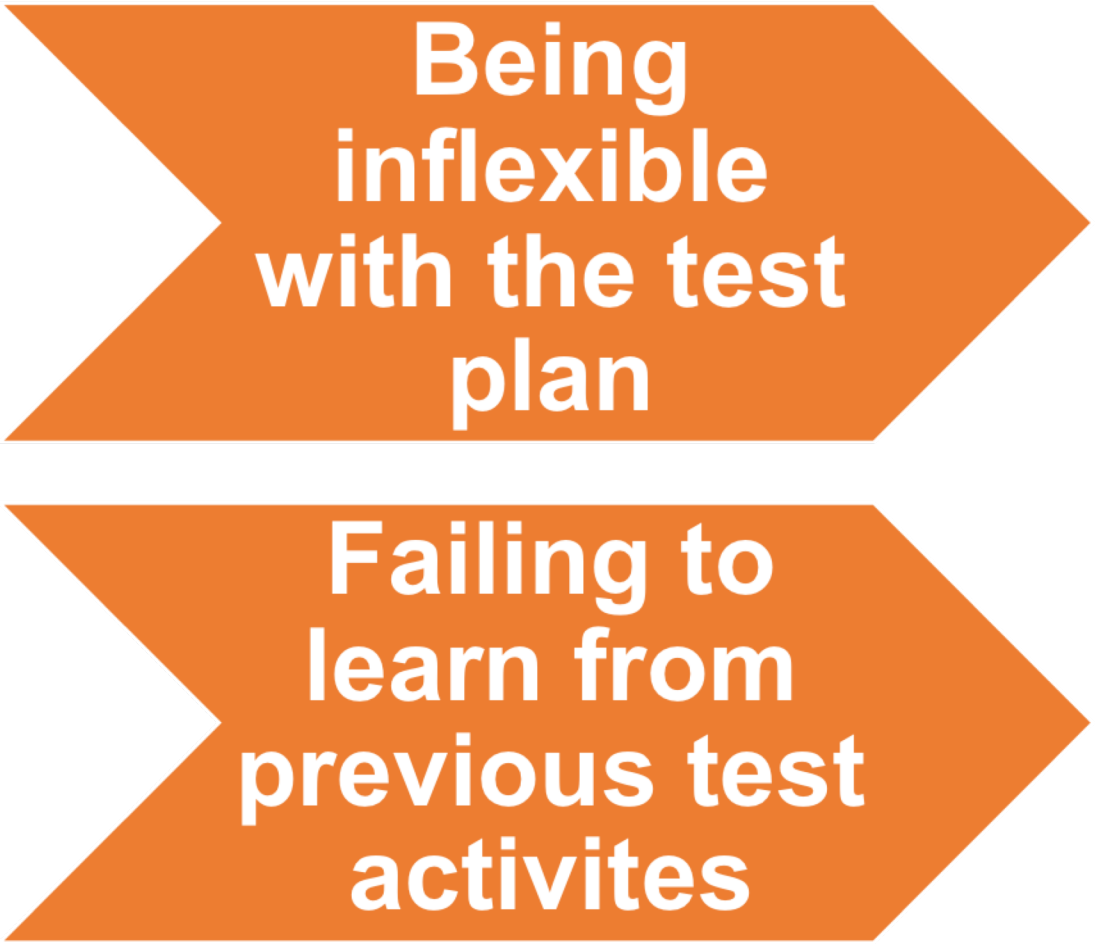
- **Believing the primary objective of system testing is to find bugs**
  - Test must concentrate on finding important problems
  - Test must provide an estimate of system quality
- **Not focusing on usability issues**
- **Starting too late**
  - Test must help development avoid problems



Some Classic Testing Mistakes

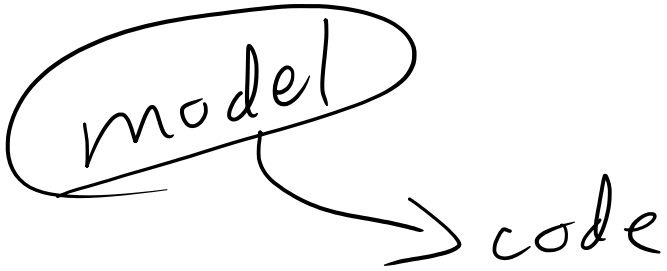
- Delaying stress and performance testing until the end
- Not testing the documentation
- Not staffing the test team with domain experts
- Not communicating well with developers
- Failing to adequately document and review test designs

Some Classic Testing Mistakes

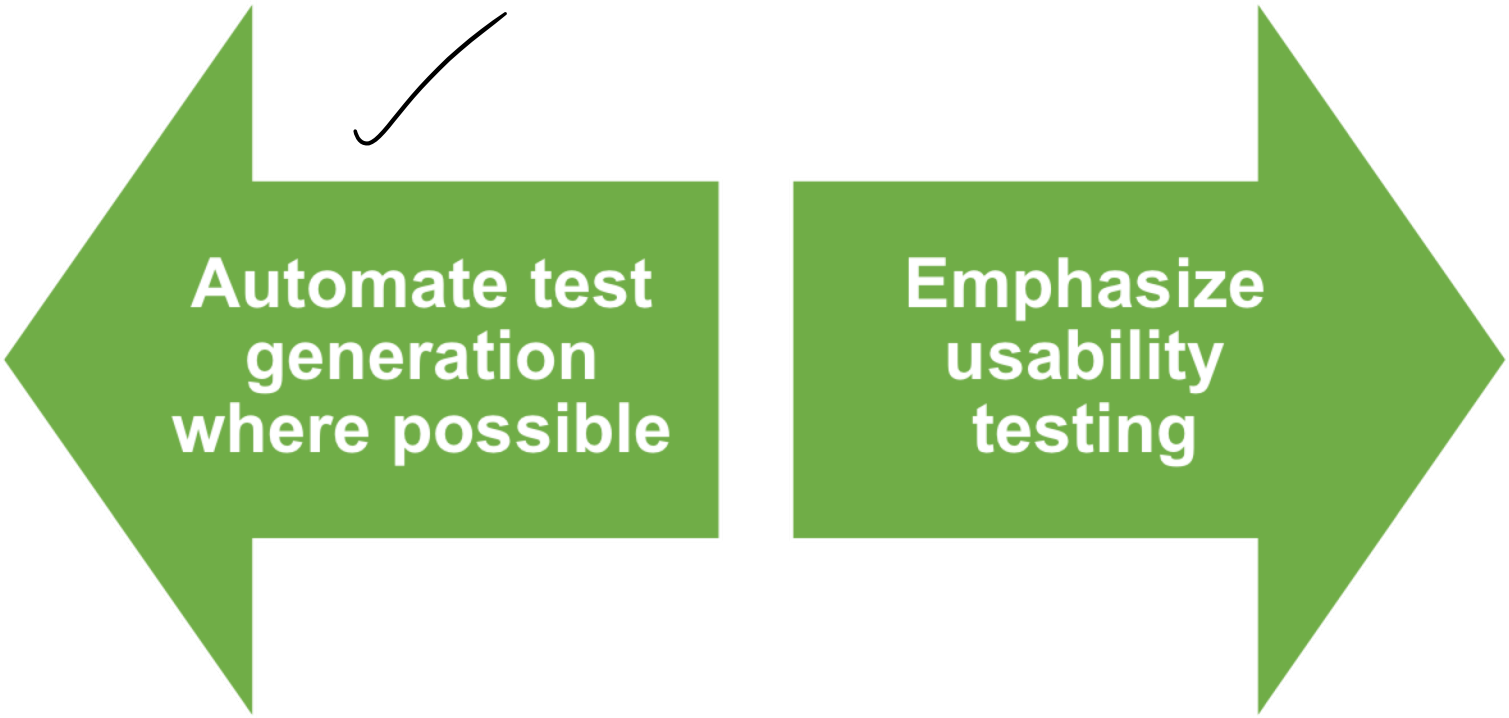


Best Testing Practices

- **Assess software reliability via statistical testing**
- **Utilize model-based testing techniques**
  - State diagrams
- **Develop an agile test design**
  - Accommodate late changes
  - Emphasis on regression testing
- **Develop cross-functional development and test teams**



Best Testing Practices



Definitions

Validation:

- Are we building the right product?

Verification:

- Are we building the product right?

Testing:

- The examination of the behavior of the program by executing it on sample data sets

Definitions

Error:

- Mistake made by a human

Defect/Fault:

- Result of error manifested in the code

Failure:

- Software doesn’t do what it is supposed to do

Poor Reliability

- Software defects rates are around 1 delivered defect per thousand lines of code
- With applications spanning millions of lines of code, customers experience many defects

Definitions

Reliability:

- The probability that a software program operates for some given time period without software error

Testers vs Pollsters Analogy

Introduction to Software Verification, Validation and Testing

Testing Background

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Define common testing terminology

History

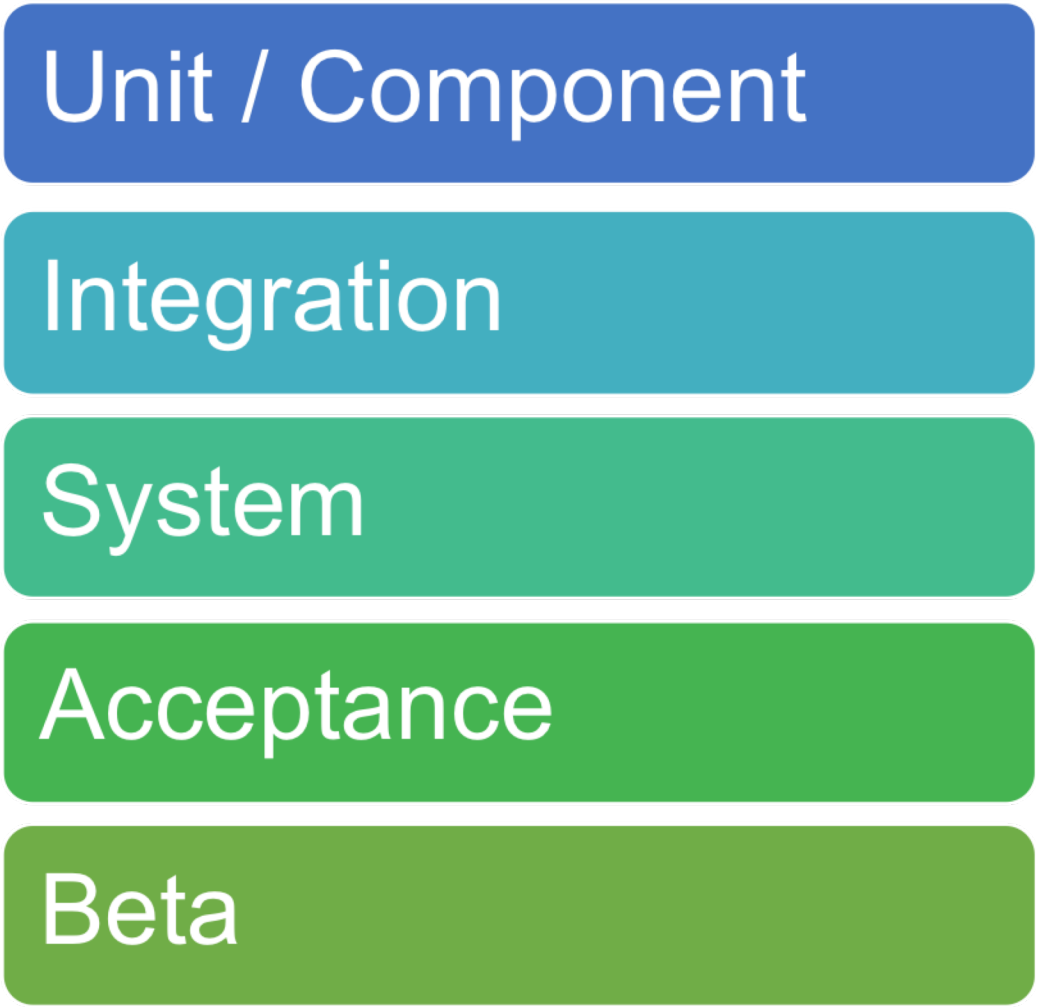
The term software engineering was first used at a workshop in West Germany in 1968 considering the growing problems of software development

- High Cost
- Difficult to Manage
- Poor Reliability
- Lack of User Acceptance
- Difficult to Maintain

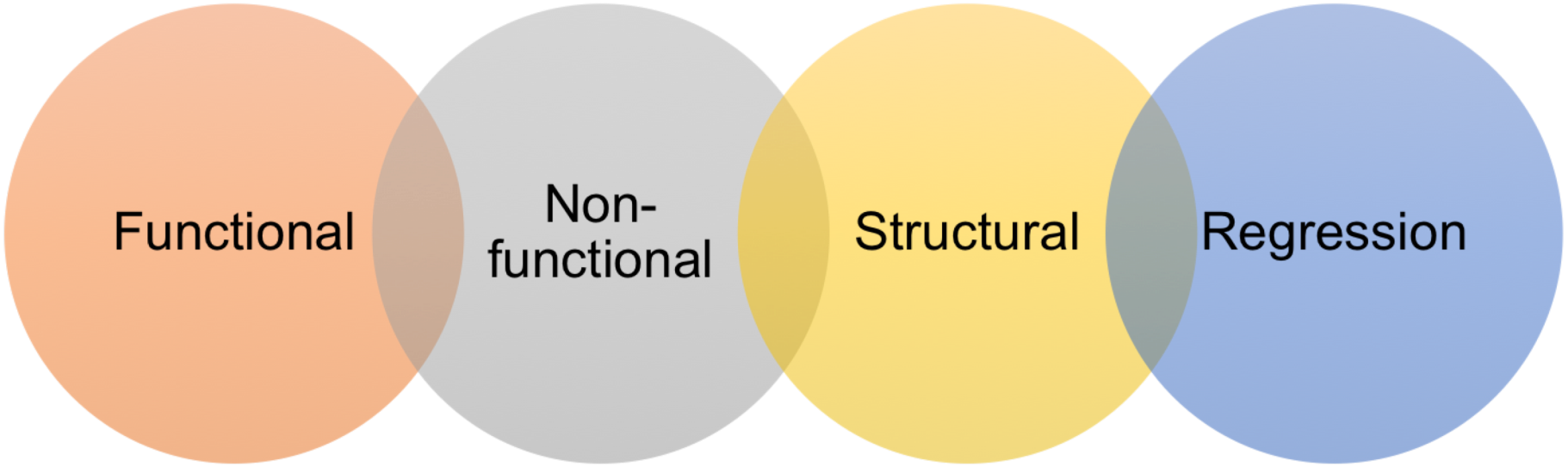
Current State of Software Development

- High Cost
- Difficult to manage
- Poor Reliability
- Lack of User Acceptance
- Difficult to Maintain

Testing Levels



Test Types



Summary

Introduction to Software Verification, Validation and Testing

Testing Principles and Best Practices

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Explain best practices for software testing.

Testing Principles

Principle 1:

- Testing only shows the presence of defects – not proof of correctness

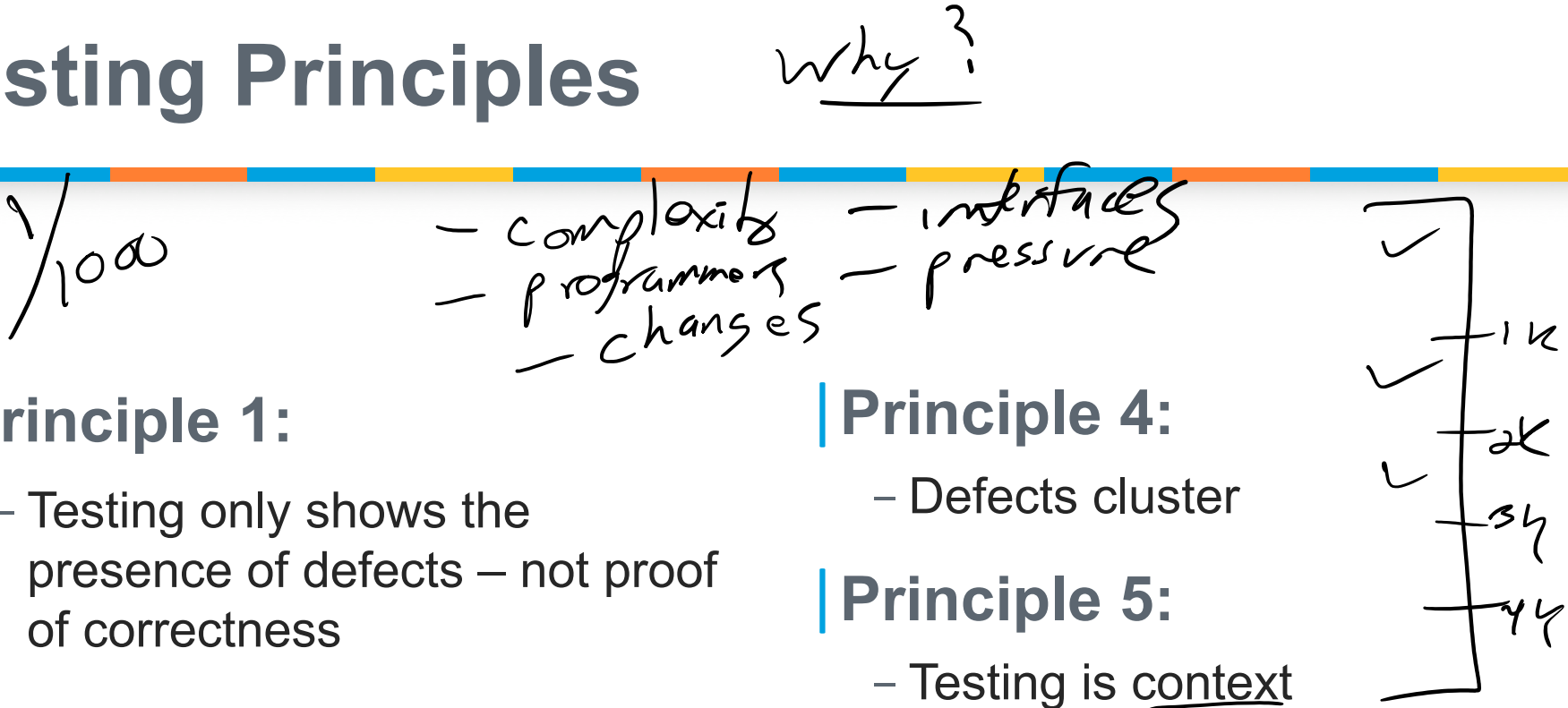
Principle 2:

- Exhaustive testing is impossible

Principle 3:

- Start testing early

Testing Principles



Principle 1:

- Testing only shows the presence of defects – not proof of correctness

Principle 2:

- Exhaustive testing is impossible

Principle 3:

- Start testing early

Principle 4:

- Defects cluster

Principle 5:

- Testing is context dependent

Principle 6:

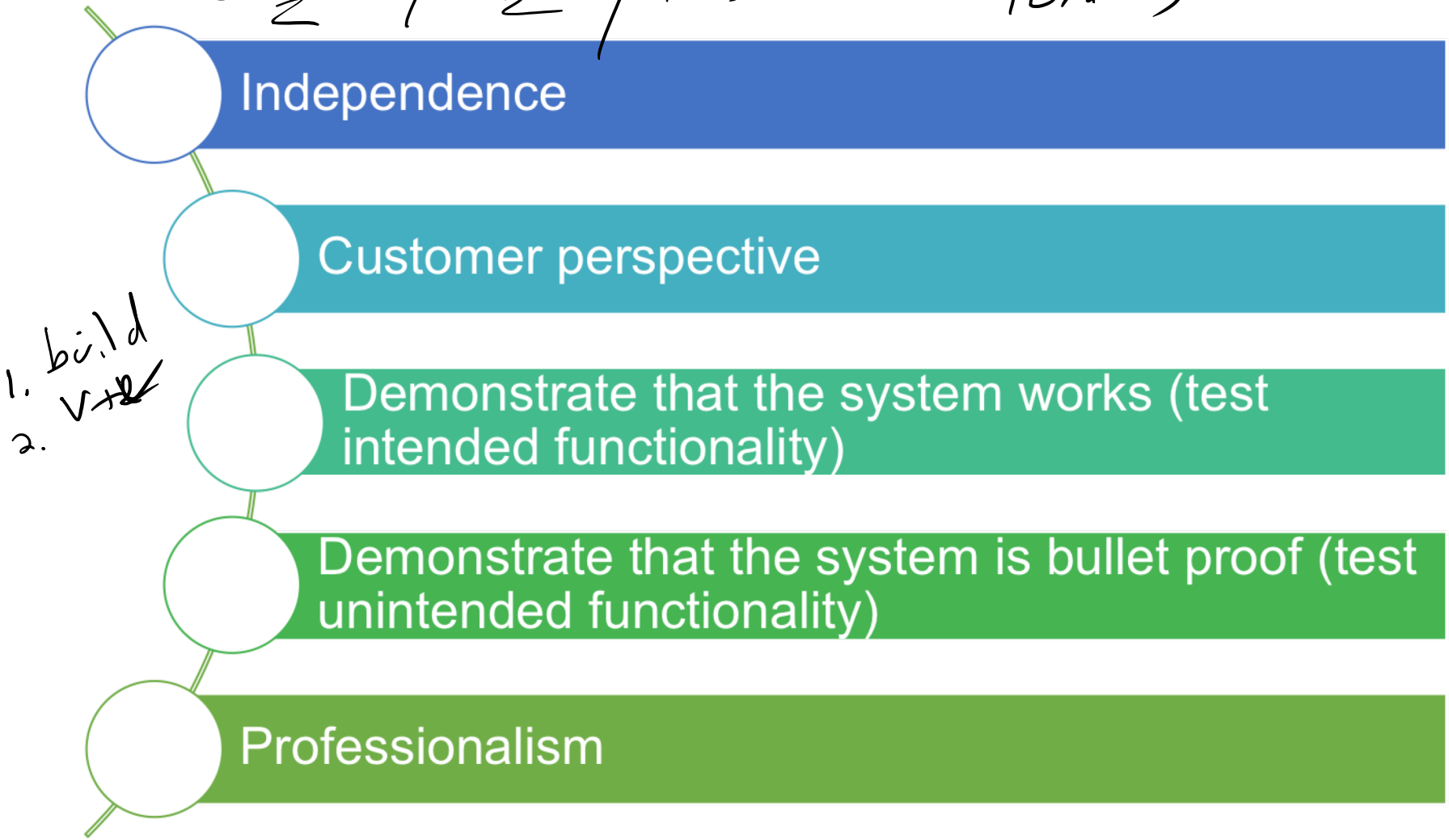
- Absence-of-errors fallacy

Testing Attitude

SE Licensing

Education / Experience / Tests

TEXAS



1. build
2. V+V

Introduction to Software Verification, Validation and Testing

Testing Throughout Life Cycle

Ira A. Fulton Schools of Engineering
Arizona State University

Objectives

Objective

Describe how testing is integrated into software development phases

Objective

Define the objectives of the different levels of testing

Testing Throughout Life Cycle

- Waterfall
- Agile
- TDD

- Agile Methodology: The Complete Guide to Understanding Agile Testing

Agile Testing

Continuous Integration (at least daily)

- Static Code Analysis
- Compile
- Unit Test
- Deploy into Test Environment
- Integration / Regression Test

Test Driven Development (Red, Green, Refactor Cycle)

Red Phase:

- Write a minimal test on the behavior needed

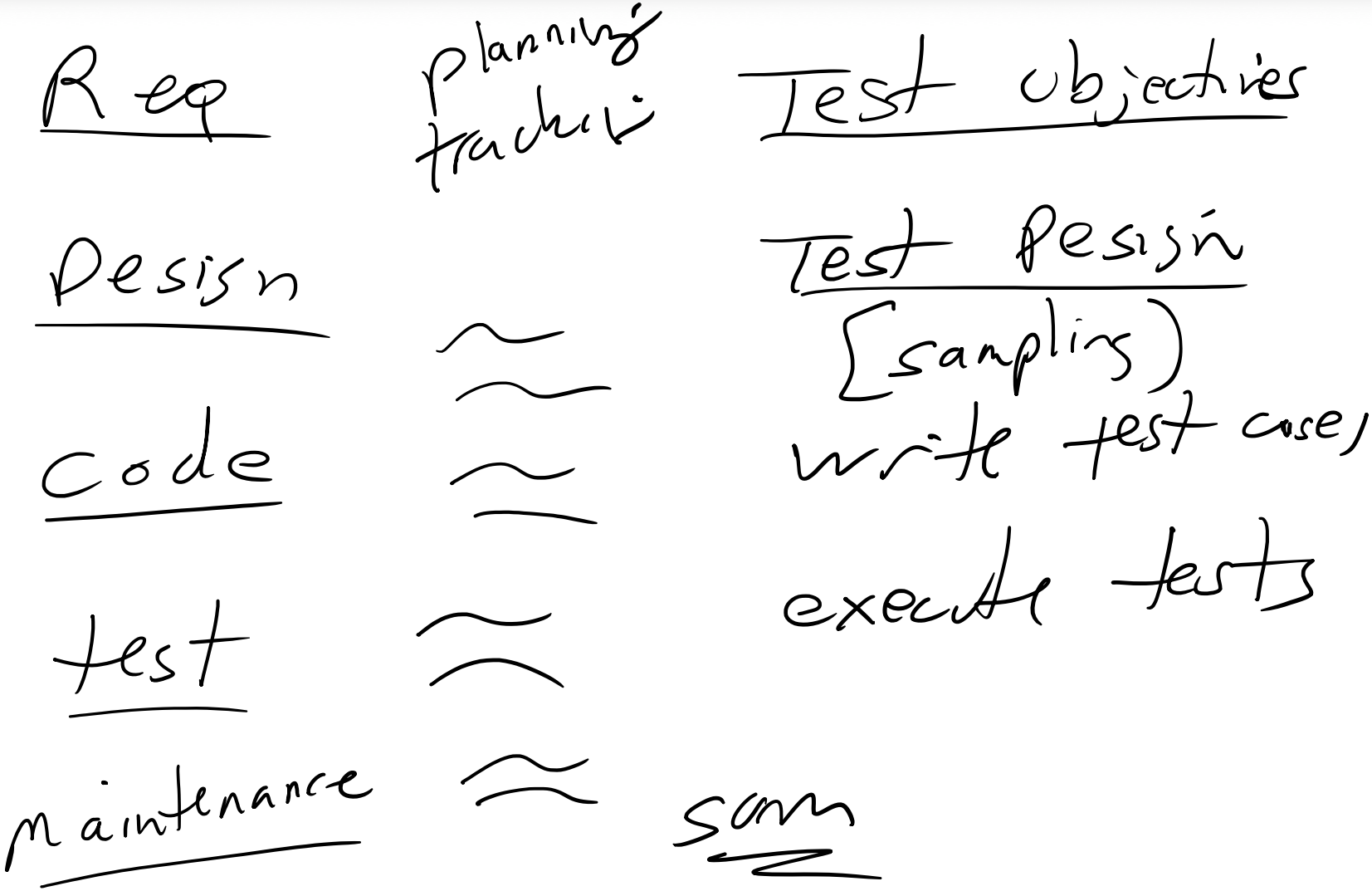
Green Phase:

- Write only enough code to make the failing test pass

Refactor Phase:

- Improve code while keeping tests green

Software Development Process vs Test Development Process



Specification Based Testing – Part 1

Boundary Value Testing

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Apply boundary value testing technique

Boundary Value Testing

Select test cases to test boundary conditions corresponding to situations directly on, above and below the edges of input and output equivalence partitions

Boundary Value Testing

Summary

Specification Based Testing - Part 1

Cause Effect Analysis

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Apply cause-effect testing technique

Cause-Effect Analysis

- Necessary for testing functions where combinations of inputs must be tested together
- Utilizes decision tables and decision trees

Example

Consider a function which has two input variables, Customer and Order, and one output variable, Discount. Customer may be of type A, B or C and Order has a range of 1 to 1000. The function computes Discount which is based on Customer type and Order. For this example, assume the following rules apply.

1. Customers of type A receive a 0% discount for less than 10 items, 5% discount for 10 to 99 items, 10% discount for 100 or more items.
2. Customers of type B receive a 5% discount for less than 10 items, 15% discount for 10 to 99 items, 25% discount for 100 or more items.
3. Customers of type C receive a 0% discount for less than 10 items, 20% discount for 10 to 99 items, 25% discount for 100 or more items.

Test Matrix

Partitions	1	2	3	4	5	6	7	8	9
A									
B									
C									
Order < 10									
10 < order <100									
100 < order < 1000									
Results									
Discount 0%									
Discount 5%									
Discount 10%									
Discount 15%									
Discount 20%									
Discount 25%									

Cause Effect Analysis / Decision Tree

Summary

Equivalence Class Partitioning and Boundary Value Analysis - A Review

Asma Bhat
Research Scholar, Department of Computer Science
University of Kashmir, Hazratbal, Srinagar, Kashmir,INDIA
Email Id: asmabhat26@yahoo.com

S. M. K. Quadri
Director, Department of computer science
University of Kashmir, Hazratbal, Srinagar, Kashmir,INDIA
Email Id : quadrismk@hotmail.com

Abstract –The purpose of this paper is to carry out a detailed review on the plethora of information available on two testing techniques which fall under functional testing methodology. A detailed analysis of equivalence class partitioning and boundary value analysis has been carried out based wholly and solely on literature survey. These techniques have been comprehensively unfolded and also the strengths and weaknesses have been highlighted. This paper can be studied before carrying out any empirical study regarding the efficiency of these two testing techniques. Then, it's only after these analytical and empirical investigations that we can come up with some sort of solid framework to effectively compare these two testing techniques with each other and also with other testing techniques.

Keywords – Black box testing, boundary value analysis, dynamic testing techniques, equivalence class partitioning, functional testing methodology.

I. INTRODUCTION

Software testing is a mechanism which helps us expose errors in a software and forms an important part of the software development process. Although it's widely accepted that it's impossible to deliver a fault free software, that does not mean that we can deliver an unsatisfactorily tested software. In fact testing should be done until the product is valid and verifiable. Verification checks whether the system in entirety works as per the specification and validation checks whether the software works according to the customers' requirements [1]. Even Pareto's principle applies to software testing wherein it states that 80% of the errors discovered during testing will likely be evident in 20% of all program components, which means that a small error can lead to a number of errors in the software. There is no doubt in the fact that the software developer must make sure that the process he/she is following must deliver a reliable software but nevertheless some errors and bugs are unraveled only after proper testing. Testing obviously takes a share in the time and resources that are available but it is only because of proper testing that a good product can be developed. The aim of testing should be to break the software and to find errors in it rather than to try and pass the software as fault free. It is for this reason that testing should be unbiased. Now there are various types of testing approaches and in order to understand which testing technique to use and where in software development lifecycle to use it we need to analyze them before empirically trying to bring them on a hierarchy. Before using any software testing technique it is of utmost importance to have a profound theoretical knowledge of that testing technique [1].

Software testing is broadly classified into two basic methodologies-Functional testing and structural testing. Functional testing methodology uses the black box approach of designing test cases wherein test cases are designed on the basis of specifications only and as the word black box signifies, the tester has no concern with the internal structure of the program under consideration., whereas structural testing methodology uses the white box approach to design test cases and the internal structure of the program i.e. the source code is used to devise test cases. Both the methodologies are imperative to software testing. Also both these methodologies have many different approaches to do the work they are meant to do. In this paper we have analyzed two of the approaches which fall under functional testing methodology and they have been studied with the help of thorough literature survey [2] [3]. We have studied equivalence class partitioning and boundary value analysis together as they are closely related and complement each other. Along with decision table testing they form the basis of functional testing methodology. In the first section of the paper we have fully analyzed the functional testing methodology and then in the next segment boundary value analysis and equivalence class partitioning have been comprehensively unfolded. Moreover, the steps to generate the test cases have been mentioned as well. These techniques are crucial to testing and it can be seen why they are still widely used by the testing teams. A review of equivalence class partitioning and boundary value analysis techniques was done by studying the vast literature available on it. We are using grounded theory type of research based upon carrying out review of literature [2]. Mostly the detailed analysis was done consulting the book sources available and also the search string "functional testing, "black box testing"," equivalence class partitioning" and "boundary value analysis" were given and the results were filtered according to requirement; to the few considered in this paper and mentioned in the references.

978-9-3805-4416-8/15/$31.00

II. FUNCTIONAL TESTING METHODOLOGY

Functional testing technique comes under the arena of black box testing methods and is also known by the names of specification testing, behavioral testing, data-driven testing, input-output driven testing, opaque testing, closed box testing [3][4]. Black box testing denotes the opacity of the tester which means that he/she has no access to the source code [5]. It can be applied to every stage of software testing i.e. unit testing, integration testing, etc [6]. These various names of functional testing themselves reveal that this testing deals only with the behavioral patterns of the software and is only concerned to check the correctness of the program using the data. In black box testing the composition of the program is not considered and test case is selected on the basis of functional requirements. The figure 1 explains functional testing methodology [6].

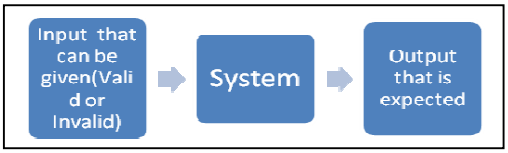


Fig. 1. Functional testing methodology in the simplest manner possible

Software is sensitive to certain input values and that helps us to check the validity of the software using the functional testing method. It is basically divided into 2 categories which are positive functional testing where valid input is given and we verify that output is correct and negative functional testing where we use a combination of invalid inputs, unexpected operating conditions and other out of bound scenarios [3]. Functional testing helps find out faulty or omitted functions, interface errors, errors in data structures or external data base access, behavioral errors, initialization and termination errors [7].

A. Advantages of functional testing methodology

It is good for a program with a large code. Code access is not needed. It separates users' perspective from developers perspective through visibly defined roles. Huge number of reasonably skilled testers can test the application having no familiarity with implementation, programming language or operating system information [8]. It has low granularity and is least exhaustive and time consuming. There is quicker test case development as well [4]. It helps find out ambiguities in specifications, if any. Test cases can be devised as soon as the specifications are completed [3].

B. Disadvantages of functional testing methodology

There is limited knowledge of tester and limited coverage. Tests are done by trial and error method and also it isn't suitable for algorithm testing. There is limited coverage and only a selected number of test cases are performed. Without clear specification test cases are not efficiently designed and thus it all comes down to unambiguous specification analysis [4]. Sometimes a tester with no information about the code might encounter a problem or a bug and not know how to correct it only due to the fact that he has no knowledge about the code. So, the advantage of having a choice to get just about any tester becomes a problem right here. Reason for a specific failure is not found out.

There are various test case design techniques as far as functional testing is concerned mentioned below, but given the scope of this paper we will be studying equivalence class partitioning and boundary value analysis [6][7][9][10][11]. Also, test cases are developed using various software testing techniques so that we can save ourselves from the nuisance of exhaustive testing which is quite impractical. Now, equivalence class partitioning and boundary value analysis are studied together because they are hugely related to each other although they do differ from one another in many ways. Functional testing techniques can primarily be divided in two types i.e. static and dynamic. Static can be further classified into performing high level or low level review of specification. Dynamic can further be classified into boundary value analysis, equivalence class partitioning, all-pairs testing, cause effect graphing, orthogonal array testing, fuzzing, error guessing, decision-table testing, combinatorial testing, model based testing and state transition table testing.

The relation will be understood as we go through the paper; however the basic differences are as follows. Equivalence class partitioning finds the number of test cases for a certain program to be tested whereas boundary value analysis determines the effectiveness of those test cases. Equivalence class partitioning checks all possible partitions for errors whereas boundary value analysis checks values at the edges because the probability of finding the error at the edges is maximum. Equivalence class method needs to be supported by boundary value analysis and not the other way round. In spite of all these differences these two testing techniques have been studied together in this paper because the relation between them is very strong and it is only fitting that they be studied together. Now, the essential point to note here is that boundary value analysis can be properly studied and understood only after fully comprehending equivalence class partitioning, as its only after equivalence class partitioning has been carried out, that boundary value analysis can be done. In really plain words the test cases that boundary value analysis creates are borne off its particular equivalence class partition only. As the names also suggest the partitions that are created in equivalence classes are further prodded in boundary value analysis and its test cases are modeled taking into consideration the edges of the equivalence class partitions. Boundary value analysis is thus like an aid to equivalence class partitioning method which helps it work smartly and reveals a lot many errors. But the irony here is that boundary value analysis supports equivalence class partitioning but itself needs its partitions in the first place to accurately identify the boundary values with which it plays.

The sections which follow unfold equivalence class partitioning method and boundary value analysis method. It would also be worthwhile to mention here that although this is

a review paper and is basically concerned only with pointing of merits and demerits but nonetheless the technical side has also been touched and we have comprehensively explained the steps to generate test cases for these two types of testing methodologies.

III. EQUIVALENCE CLASS PARTITIONING

Exhaustive testing includes all possible inputs to the system and is highly impractical. And it's because of this that the input domain is divided into a set of equivalence classes, so that if a specific program works appropriately for a certain value then it will apparently work properly for all the further values in that class [9][12]. One approach to partition the input domain is to consider one input variable at an instance. Herein, each input variable leads to a partition of the input domain. This is called uni-dimensional equivalence class partitioning. Then there is multi-dimensional partitioning where the input domain is considered the set product of the input variables and then association is defined on the input domain. This course of action leads to one partition consisting of several equivalence classes [13].

Basically, grouping of similar inputs or similar outputs or similar operations is done. And it is these that constitute the equivalence class partition [11]. Equivalence class partitioning is as much an art as it a science because it is subjective to a great extent and even if two different testers get two different sets of partitions it's okay as long as the bugs are properly revealed [10][11]. So, what this concept signifies is that if such classes are recognized to a high level of precision, even if one value from the equivalence class is tested it's like checking the whole program exhaustively within the boundaries of that values equivalence class [9]. This concept thus reduces the effort of the tester to a great degree and the results that are obtained are not of a lower level at all. Where behavior of inputs is expected to be alike; equivalence class can be formed and if a set of objects can be linked by relationships which are symmetric, transitive and reflexive an equivalence class is surely present [7] [9].The main goal of equivalence class partitioning is to define a test case that unravels classes of errors and that actually helps to reduce total number of test cases to be developed [7]. This is how equivalence class method helps us to find a way out of exhaustive testing but still delivers at par to the former. The fundamental idea of equivalence class partitioning method is that it either works correctly for all the values or none. In equivalence classes there is no near about similarity and they ought to be split if there is a minute dissimilarity [9]. Thus, it can easily be said that equivalence class partitioning is a heuristic approach to test a software whose state spaces, inputs and outputs have value ranges of a cardinality inhibiting exhaustive enumeration of every likely value within a test suite. Herein its meant that equivalence classes are partitions created within the input or the output domain to create disjoint sets of data which exhibit completeness and whose behaviour is expected to be the same based on the specification [12][14] .

Equivalence classes are typically formed by taking into consideration each condition specified on an input as specifying a valid equivalence class and one or more invalid equivalence class. At times we take into consideration equivalence classes in the output. Here, the idea is to have inputs such that the output for that test case lies in the output equivalence class [9]. Both input as well as output classes have been found to be effective in finding errors and both should be used in tandem with one another. Also we must always design separate equivalence class partitioning for default, empty, blank, null, zero or none condition as they need to be handled differently [11]. The efficacy of tests generated by means of equivalence partitioning for testing any application, is judged by the ratio of the number of faults these tests are able to expose to the total faults lurking in that particular application [13]. Now there are some general guidelines which are generally followed while designing equivalence class partitions as mentioned.

Input data values to a system is specified by range of value where one valid equivalence class is formed and two invalid equivalence classes are formed or it is specified by values from members of a set made of discrete elements where one equivalence class for valid input value and one equivalence class for invalid input values is formed. Another way to specify the input values is by some specific value where one valid equivalence class is formed and two invalid equivalence classes are formed .As also input values can be specified by boolean value where one valid and one invalid equivalence class is formed.

A. *Steps to generate test case for equivalence class partitioning*

Firstly, identify the input domain of the program, after properly studying the requirements, which helps identify the input and output variables, their types, and any conditions associated with their use. Secondly, divide the input domain into various categories where the participants of every category have some sort of relation with each other .Thirdly, we can expect that every test case from a category displays the same behavior. Every category is thus, a separate equivalence class and we partition the set of values of each variable into disjoint subsets which cover the entire domain, based on the expected behaviour. Fourthly, sometimes equivalence classes are combined using the multi-dimensional partitioning approach but usually this step is omitted. Nonetheless, if we do combine the equivalence classes its helps create useful tests. Lastly, infeasible equivalence classes, which contain a combination of input data that cannot be generated during test need to be identified. Constraints in the requirements render certain equivalence classes infeasible [9] [10] [13].

B. Types of equivalence class testing:

1) Weak normal equivalence class testing: It is based on single fault assumption which means that an error is rarely caused due to two or more faults occurring simultaneously meaning

that the variables are independent. Only one variable is taken from each equivalence class. In the figure below, one variable is taken from each equivalence class [16].

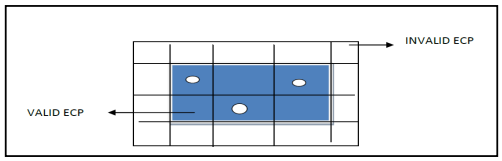


Fig. 2. One variable is taken and that too only from valid equivalence classes.

2) Strong normal equivalence class testing: Here it is assumed that errors result in a combination of faults i.e. multiple fault assumption, where the variables are not independent of each other. So, we test every combination of elements formed as a result of the Cartesian product of the equivalence relation.

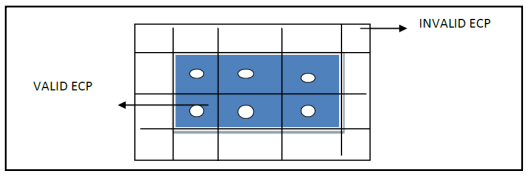


Fig.3. Test every combination of elements formed as a result of Cartesian product of the equivalence relation

3) Weak robust equivalence class testing: Here we test for one variable from each equivalence class but we authenticate and test for invalid values as well. Since it is based on the single fault assumption a test case will have one invalid value and the remaining values will all be valid [16].

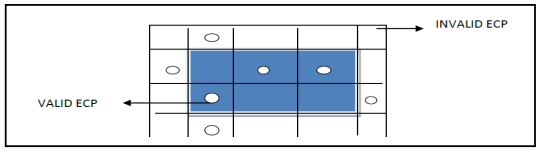


Fig. 4. Check valid as well as invalid values

4) Strong robust equivalence class testing: Here, test cases are produced for all valid and invalid elements of the Cartesian product of all the equivalence classes [16].

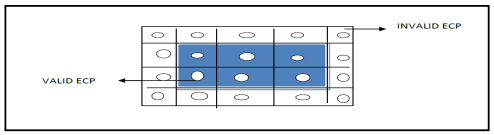


Fig. 5. Checks for all out of range combinations

C. *Advantages of equivalence class partitioning*:

Equivalence class partitioning reduces redundancy and ensures completeness of testing. It reduces effort as compared to exhaustive testing and gives if at better results in lesser time.

D. Disadvantages of equivalence class partitioning:

Equivalence class partitioning makes an assumption that the data in the identical equivalence class is processed in the similar way be the system. It needs to be supplemented by boundary value analysis [17] [18]. It is more concerned with data dependencies and testing similar inputs and outputs the same way by grouping them in classes. This reduces the test cases but at the cost of an increased effort to group them [19].It is a .sophisticated method as it is concerned with the values inside the domain.

IV. BOUNDARY VALUE ANALYSIS

Programs that work tolerably for a set of values in an equivalence class fall short on some special value which are often seen to be on the boundary of the equivalence class [9][13][20]. Programming by character is prone to bugs at boundaries. When working with boundary value analysis we play with the values on the boundary or just outside the boundary [11]. Some boundaries are quite internal to the software and might not be apparent to the end user but need to be checked nonetheless. These are called sub boundary or internal boundary conditions. There are two ways to generate boundary value analysis. One is by the number of variables and second by the kind of range [21]. A boundary value for each equivalence class includes the equivalence classes of the output as well. Figure 6 explain the phenomenon of boundary value analysis.

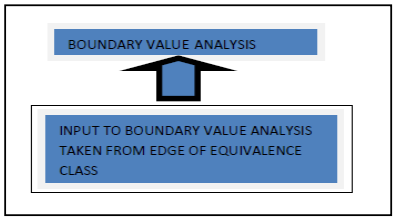


Fig. 6. Boundary value analysis

A. Steps to generate test case for boundary value analysis

Firstly, find out values for each of the variables that should be exercised during testing from the specification meaning we divide the input domain using uni-dimensional partitioning. This leads to as many partitions as there are input variables. A different method is creating a single partition of an input sphere using multi-dimensional partitioning. Also several sub-domains are also formed during this step. Secondly, identify the boundaries and select values on or close to the boundary i.e. minimum value, just above minimum value, maximum value, just below maximum value nominal (average) value. Thirdly, select the test data such that every boundary value occurs in at least one test input [7] [9] [10][13].
So, in order to generate test cases here the values will be taken within the valid input domain of any program which is under

consideration. As the following figure depicts, values on the x axis are taken on the y axis as well.

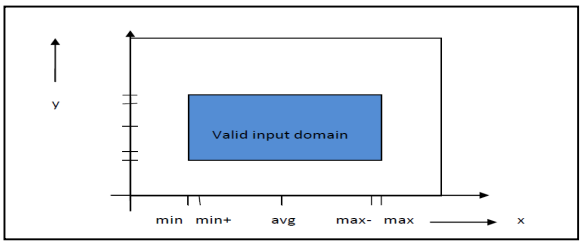


Fig. 7. Values that boundary value analysis checks

B. Types of boundary value analysis

1) Robustness testing: Here invalid values are selected and the response of the program is checked. We check the values- minimum, just above minimum, just below minimum, maximum, just above maximum, just below maximum and nominal (average) value. Robustness testing focuses on exception handling and is highly advocated when it needs to be checked [20].

2) Worst case testing: Here single fault assumption theory is not considered. Single fault assumption theory of reliability is that failures are rarely a result of simultaneous occurrence of two or more faults [20]. But here we consider that failure is due to the occurrence of more than one fault simultaneously. Input values are minimum, just above minimum, average, just below maximum and maximum. . But the most important fact here is that the restriction of one input value at any of the afore mentioned values and other input values must be at nominal, is not valid in worst case testing. It is generally used for situations where a high degree of testing is required [20].

3) Robust- worst case testing: It is also one of the special cases of boundary value analysis and here the states are the same as in robustness test. The difference lies in the number of test cases generated. It produces the largest set of test cases so far and requires a lot of effort [20]. The difference between these various extensions of boundary value analysis is actually in the number of test cases that they generate [10].

C. Divide and rule approach

A fresh algorithm is devised by [22] and based on it a testing means is developed. In here a divide and rule approach is used to sever the dependencies amongst the variables to generate self-sufficient sets of variables. When there is no dependency among the variables visibly there is no influence on each other. Sets of independent variables are thus fashioned from a set of inter dependent variables. This modus operandi is used to form sets of independent variables from sets of interdependent variables plus afterward carry out conservative boundary value analysis on every independent set fashioned. In a set of inter-dependent variables, boundaries of a variable are prejudiced by the values assumed by other variables. This style of dependency amongst variables can be called boundary dependency [22].

These are the steps to create test case when looking at boundary value analysis w.r.t divide and rule approach [22].

- Foremost divide the variables into 3 categories. First category is called D i.e. set of dependent variables whose boundary values are influenced by the values of some other variable. Second category is called B i.e. set of boundary determining variables i.e. the values of these variables manipulate boundary values of other variables. Third category is I, i.e. independent variables which are neither dependent nor influential.
- For each and every dependent variable shape a set of determining variables. This means for the dependent variable whose boundary value is influenced, find the influential boundary determining variable which causes the influence here and has an effect on the dependent variable.
- Identify and outline the boundary value range of dependent variables and structure their separate sets.
- Now, outline sets of values (boundary formative variables) which influence the boundary values of dependent variables already recognized in step 3.
- For every dependent variable, produce all likely combinations of its boundary determining sets. E.g. x and y are elements of boundary determining variables. x has 2 determining sets namely x_1 and x_2. Y has 3 determining sets namely y_1, y_2, y_3. Combinations will be x_1y_1, x_1y_2, x_1y_3, x_2y_1, x_2y_2, x_2y_3.
- For every likely combination of boundary determining sets, allot a boundary value dependent set. Association between combination of boundary determining sets and the dependent boundary value set is that if any determining variable specific combination assumes a particular value, then dependent variable can assume a value from corresponding dependent variable only i.e. the dependent variable which matches the determining set combination.
- Now, produce all probable combinations of boundary determining sets of all boundary determining variables. E.g. x, y, z are elements of boundary determining variables and have 2, 3 and 2 determining sets respectively we have 12 combinations, going about similarly on the lines of step 5.
  $x_1y_1g_1$, $x_1y_1g_2$, $x_1y_2g_1$, $x_1y_2g_2$, $x_1y_3g_1$, $x_1y_3g_2$, $x_2y_1g_1$, $x_2y_1g_2$, $x_2y_2g_1$, $x_2y_2g_2$, $x_2y_3g_1$, $x_2y_3g_2$
- For every grouping in step 7 assign boundary value sets for each dependent variable via associations in step 6.
- For every grouping assign all independent variables, giving all possible independent variable sets.

- For every independent variable set produced, carry out typical boundary value analysis to fabricate test cases [22].

D. Advantages of boundary value analysis

Boundary value analysis can be used at unit, integration, system and acceptance test levels. Experience has shown that such test cases have a high probability of detecting a fault in the software, where test cases have input values close on the boundary [10]. It is not concerned with the data or logical dependencies as it is a domain based testing technique [19]. It is computationally less costly as far as the creation of test cases are concerned [20].

E. Disadvantages of boundary value analysis

In boundary value analysis inputs should be independent, this is what restricts it's the applicability making no sense for boolean variables and no choice for nominal value, just above and below boundary values. So, it's suitable only for ranges, sets and not for logical variables. The number of test cases generated is higher as compared to other functional methods [19]. It is more proper to more "free-form" languages which are weakly typed than to strongly typed languages which necessitate that all constants or variables defined must have an associated data type. These languages were in reality created to make sure that the faults boundary value analysis discovers wouldn't arise in the first place. Even though boundary value analysis can be used with languages of this type and is constructive to a certain extent, but overall it can be seen as unsuitable for systems created using them [20]. It doesn't consider the nature of the function or the dependencies amid the variables. [20].

V. CONCLUSION AND FUTURE SCOPE

In this paper functional testing methodology was studied in a detailed manner as it falls under the arena of black box testing and is widely used in testing software. It is of immense importance to check which testing technique to use in which stage of software development life cycle apart from studying in detail which functional testing techniques stands where on the testing spectrum and here we are starting with equivalence class partitioning and boundary value analysis and how they are related to each other. These techniques are known to form the crux of functional testing methodology and a detailed review has been carried out in this paper and it becomes a starting point to carry out empirical study of the various testing techniques, to compare them and somehow understand the pros and cons of each of them. Now since the analysis is done in this paper further work would be required to gauge them with the help of studying the experiments which evaluate the effectiveness of these techniques. It would rightfully take the grounded research of this paper to a more concrete and viable conclusion.

REFERENCES

[1] S.U. Farooq, *"Evaluating effectiveness of software testing techniques with emphasis on enhancing software reliability"* P.hDthesis,from:*ojs.uok.edu.in/jspui/bit stream/1/1041/3/**Umar**%20**PhD**%20**Thesis**.pdf- 2012*

[2] Research Methods/Types of Research. *Wiki books, The Free Textbook Project*. Retrieved from http://en.wikibooks.org/w/index.php?title=Research_Methods/Types_of_Research&oldid=2616361 - 2014

[3] B. B. Agarwal, S. P. Tayal & M. Gupta - *Software Engineering and testing*. Jones & Bartlett Learning, 2010.

[4] M.E. Khan, & F. Khan, "A Comparative Study of White Box, Black Box and Grey Box Testing Techniques". *International Journal of Advanced Computer Sciences and Applications*, *3*(6), pp. 12-1 ,2012.

[5] L. Williams," Testing overview and black-box testing techniques." S*ource: http://www. agile. csc. ncsu. Edu/SEMaterials/Black Box. Pdf* - 2004.

[6] Black-box testing. In *Wikipedia, The Free Encyclopedia*, from http://en.wikipedia.org/w/index.php?title=Black-box_testing&oldid=607624827 – May, 2014.

[7] R. S. Pressman*, Software Engineering: A Practitioner's Approach*, ISBN*: 0-07-052182-4*, 1996.

[8] *"Software testing method (Black-box. White-Box. Rey-Box)*."In Tutorial Point from www.**tutorialspoint**.com/**software_testing/testing_**methods.htm - 2014

[9] P. Jalote - An *integrated approach to software engineering*. Springer, 1997.

[10] Y. Singh - *Software Testing*. Cambridge University Press, 2012

[11] R. Patton - Software *testing*. Sams Publication, 2006.

[12] D. Galin - *Software quality assurance: from theory to implementation*. Pearson education, 2004.

[13] A. P. Mathur, *Foundations of software testing*. Source: https://www.cs.purdue.edu/homes/apm/FoundationsBook Second Edition/Slides/ConsolidatedSlides.pdf- 2008,Updated 2013.

[14] W. L.Huang & J. Peleska, " Equivalence Class Partitions for Exhaustive Model-Based Testing" - 2012

[15] R. Mall, *Fundamentals of software engineering*. PHI Learning Pvt. Ltd - 2009

[16] G. Davies, "Equivalence Class Testing*"* Swansea University - 2007

[17] T. Murnane , K. Reed & R. Hall, *"*On the learnability of two representations of equivalence partitioning and boundary value analysis*."* In Proceedings of the 2007 *Australian Software Engineering Conference* (pp. 274-283). IEEE Computer Society- April- 2007.

[18] T. Murnane, & K. Reed. *"*On the effectiveness of mutation analysis as a black box testing technique*." In Software Engineering Conference,* 2001. Proceedings. 2001 Australian (pp. 12-20). IEEE - 2001

[19] C. Ferriday." A Review Paper on Decision Table Based Testing*."* Swansea University-CS339 - 2007

[20] B. Neate." Boundary value analysis." University of Wales Swansea – 2006.

[21] M. Khan. "Different Approaches To Black Box Testing Technique For Finding Errors*."* *International Journal of Software Engineering & Applications*, 2(4)- 2011

[22] K. Vij & W. Feng," Boundary value analysis using divide-and-rule approach*"*, in *Information Technology: New Generations. ITNG 2008*. Fifth International Conference, pp. 70-75, IEEE- April - 2008.

Specification Based Testing - Part 1

Equivalence Partitioning

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Apply equivalence partitioning testing technique

Equivalence Partitioning

- **Technique for dividing input domain of a program/function into a finite number of equivalence partitions**
  - Both valid and invalid partitions are considered
- **Select one or more inputs from each equivalence partition**
- **Functional coverage is demonstrated by mapping equivalence partitions to test cases**
- **Equivalence partitioning is applicable for testing functions whose inputs are independent of each other**

Equivalence Partitioning Steps

1. For each input, identify a set of equivalence partitions and label them
2. Write test cases covering as many of uncovered valid equivalence partitions as possible
3. For each invalid equivalence partition write a test case that covers one of the uncovered equivalence partitions

Equivalence Partitioning Example

Testing a procedure called:

- Validate_New_Password which accepts a password from a user and validates that the new password conforms to the following rules:

1. **A password must be between 6-10 characters**
2. **First and last character must be alphabetic, numeric or “?”**
3. **Remaining characters may be any character except control characters**
4. **Password must not be in a dictionary**

Equivalence Partitions

Length	First ch.	Last ch.	Remaining	Status of Password
(1) 6-10	(2) alpha	(5) alpha	(8) normal	(17) In dict
(10) <6	(3) numeric	(6) numeric	(16) control	(9) not in dict
(11) > 10	(4) ?	(7) ?		
	(12) control	(14) control		
	(13) other	(15) other		

Tests

1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.

Test Matrix

EP	1	2	3	4	5	6	7	8	9	10	11
1.											
2.											
3.											
4.											
5.											
6.											
7.											
8.											
9.											
10.											
11.											
12.											
13.											
14.											
15.											
16.											
17.											

Summary

Editor: **Christof Ebert**
Vector Consulting Services
christof.ebert@vector.com

Model-Based Testing

Ina Schieferdecker

Test consumes a large share of development efforts. If you don't continuously trim validation efforts and improve test efficiency while maintaining quality, your overall costs won't remain competitive. Model-based testing (MBT) strives to automatically and systematically generate test cases. In this column, Ina Schieferdecker introduces MBT technologies and methods. I look forward to hearing from both readers and prospective authors about this column and the technologies you want to know more about. —*Christof Ebert*

IN MODEL-BASED TESTING (MBT), manually selected algorithms automatically and systematically generate test cases from a set of models of the system under test or its environment. Whereas test automation replaces manual test execution with automated test scripts, MBT replaces manual test designs with automated test designs and test generation. However, a recent survey about the "Practice on Software Testing" shows that MBT hasn't yet arrived in industry, although its potential gains could be enormous.[1] Its prospective benefits include

- early and explicit specification, modeling, and review of system behavior, and early discovery of specification errors;
- better documentation of test cases, increased transparency, and enhanced communication between developers and testers;
- the ability to automatically generate useful tests and measure and optimize test coverage;
- the ability to evaluate and select regression test suites;
- easier updates of test models and suites for changed requirements and designs and for new product versions, and improved maintenance of test cases;
- higher test quality through model-based quality analysis; and
- shorter schedules and lower costs.

This column provides an overview of the state of the practice and MBT methods and tools.

MBT Synopsis

First-generation MBT consisted of test generation from system models. This research developed the main concepts and algorithms for test behavior and test-data generation. However, it showed various shortcomings such as using a single model for code and test generation. This meant that errors in the model got propagated to the code and tests and were thus impossible to detect.

Second-generation MBT uses separate test models, which are sometimes called scenario models, usage models, environmental models, and so on. Test models support specifica-

0740-7459/12/$31.00 © 2012 IEEE

tion of testing concerns in a dedicated model. This approach propagates the duality of the system and test system to the model level. This extends to the requirements level, which distinguishes between system requirements and test requirements and the models thereof.

Second-generation MBT makes explicit the tester's expertise in denoting testing's essential concerns. Tests are designed by methods and tools established as industrial practice in model-driven software engineering and are documented during high-level review and discussion. The tests' quality—including correctness and coverage—can be predetermined at the model level.[2]

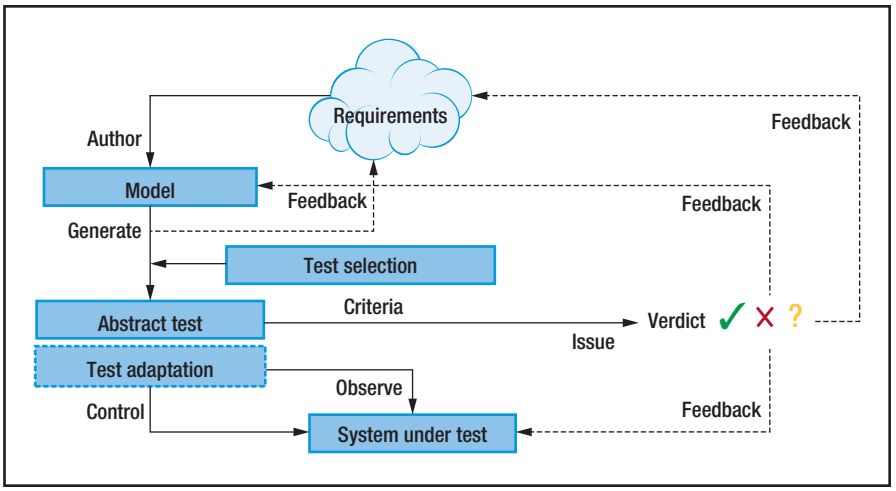


FIGURE 1. Model-based testing (MBT) elements.[3] This process consists of three main steps: designing a functional test model, determining test generation criteria, and generating the tests.

MBT's Three Main Tasks

The generated tests can be executed manually or automatically, but combining automated test generation with automated test execution achieves the biggest gain. Key aspects of MBT encompass modeling notation and principles for test models, strategies, and algorithms to guide test generation and, optionally, on-the-fly generation or offline realization of executable tests. Figure 1 shows a typical MBT process in industry, which consists of the following three tasks.

Design a Functional Test Model

The test model represents the expected operational behavior of the system under test (SUT) or its environment. Test designers can use standard modeling languages such as UML to formalize the points of control and observation of the SUT, the system's expected dynamic behavior, the entities associated with the test, and test data for various test configurations. They link model elements such as states, transitions, and decisions to the requirements to ensure bidirectional traceability between the requirements and the model and later to the generated test cases and test results. The test models must be precise and complete enough to allow automated derivation of tests from them.

Determine Test Generation Criteria

Models can usually generate an infinite number of tests, so test designers select criteria to limit the number of generated tests (for example, by selecting the highest-priority tests or ensuring specific coverage of system behaviors). A common approach for test selection is based on structural-model coverage (for example, determining the tests' coverage of model elements). Two examples of this approach are equivalence partitioning on the basis of the model's decisions or constraints and path coverage on the basis of *k*-pair transitions or selected execution sequences in the model.

Another useful type of criteria ensures that the generated test cases cover all the requirements, possibly with more tests generated for requirements with higher risk. In this way, testers can use model-based testing to implement requirements-oriented or a risk-driven test.

Generate the Tests

Typically, test generation in MBT is fully automated. The generated test cases are sequences of high-level events or actions on or by the SUT, with input parameters, expected output parameters, and return values for each event or action. These sequences are similar to high-level test sequences that testers would design manually in keyword-driven testing (see Figure 1). Typically, the test sequences are easy to understand and complete enough for manual or automated test execution. If necessary, testers refine the tests to a more concrete level or adapt them to the SUT to support their automated execution.

MBT Tools

In the past, three commercial tools have led the MBT scene: Smartesting's CertifyIT, the Conformiq Tool Suite, and Microsoft's Spec Explorer. The next generation of more sophisticated MBT tools include Tedeso by Siemens/Imbus, Elvior's TestCast Generator, and All4Tec's MaTeLo. Many tools use TTCN-3 (Testing and Test Control Notation) as the language and execution technology for generated tests.[4] Table 1 compares these tools.

Companies are also deploying MBT tools for dedicated application domains.

TABLE 1

General-purpose model-based-testing tools.

Tool	URL	Target domains	Test model	Test generation criteria	Test scripting
CertifyIT v5.1	www.smartesting.com	Software	Business Process Model and Notation or Unified Modeling Language (UML)	Test data and verification points	Textual test plans and executable test scripts in Quickset Professional and so on.
Conformiq Designer v4.4	www.conformiq.com	Data communications and telecommunications	State charts	Requirements-driven test generation, black-box test design heuristics	Textual test plans and executable test cases in Java, and so on
Spec Explorer 2010	research.microsoft.com/en-us/projects/specexplorer	Software	Spec#	Transition coverage	Executable test cases in C# or on-the-fly testing
Tedeso 3.0	www.imbus.de/english/imbus-testbench/modules/managed-model-based-testing	Software	UML activity and sequence diagrams	Model and data coverage	Executable test cases in C++, and so on
TestCast Generator Beta	www.elvior.com/motes/generator	Telecommunications, transportation, defense	UML state machines	State, transition, and decision coverage	Executable test cases in TTCN-3 (Testing and Test Control Notation)
MaTeLo 4.7.5	www.all4tec.net	Embedded systems	Enhanced Markov chains	Probabilities for transitions and inputs	Textual test plans and executable test cases in TTCN-3, and so on

For example, embedded-systems MBT tools include Reactis by Reactive Systems (www.reactive-systems.com) and Modena by Berner & Mattner (http://www.berner-mattner.com/en/berner-mattner-home/products/modena/index.html). Research and open source MBT tools include Gotcha-TCBeans (www.research.ibm.com/haifa/projects/verification/gtcb/index.html), GraphWalker (www.graphwalker.org), AutoFocus (http://autofocus.in.tum.de/index.php/Main_Page), Fokus!MBT (www.fokus.fraunhofer.de/en/motion/ueber_motion/technologien/fokusmbt/index.html), and Uppaal-CoVer (www.hessel.nu/CoVer).

MBT in Standardization

A key issue in MBT is the plethora of concepts and methods that don't follow a common convention. Since 2005, the Object Management Group (OMG) has offered the UML Testing Profile (UTP) specification to support model-based testing that's seamlessly integrated with UML. Today, several open source and commercial solutions implement UTP. Several research papers and companies (IBM, Microsoft, and so on) also reference it. OMG completed a UTP revision in July 2011.

Other standardization bodies have initiated approaches to a common nomenclature for testing. For example, several MBT tool vendors and major industrial users have developed a European Telecommunications Standards Institute (ETSI) standard to unify terminology and define a common set of concepts that MBT tools should support.[3] The MBT special interest group from the International Software Quality Institute has begun a training and qualification initiative (www.isqi.org/en/modellbasiertes-testen-mbt.html). This technology-, tool-, and vendor-independent MBT qualification has four goals. The first is to let testers apply model-driven engineering methods for test automation that are well established in software development. The second goal is to build the corresponding engineering skills. The third goal is to establish a qualification that serves as a standardized skill set for recruitment by providing a profound MBT certification rather than a number of specific tool qualifications. The final goal is to establish a qualification for career and training planning.

MBT Rollout

MBT's successful application depends greatly on proper adaption to a company's development infrastructures and processes. One potential cause for disappointment is expecting too much from MBT. Companies often underestimate its rollout costs (see the "Model-Based-Testing Costs" sidebar). In addition, they often misclassify requirements for MBT infrastructure and personnel qualifications.

Studies on the application of MBT

recommend that companies introduce a rollout with several small pilot projects that usually cover a small field of expertise in the beginning. These projects should aim for simple, realistic, but well-attestable goals—for example, increased transparency, more efficient test specification procedures, and better test case maintenance. MBT experts and domain experts should propose, prepare, execute, and assess such projects to provide the necessary adaptation to the company and domain-specific requirements. A pilot project with low-hanging fruit should provide enough motivation to escalate MBT strategies to other projects, along with a suitable technological basis for such escalation.

Software development and quality assurance processes vary according to requirements, techniques, stakeholders, and target environments. Consequently, studies show that MBT is no commercial-of-the-shelf solution.[5,6] For successful implementation, MBT tools must be tailored to fit a company's processes and infrastructure. Such tailoring includes adaptation to the existing tool environment (test management tools, test execution environments, modeling tools, data repositories, and so on) and integration with best practices and existing processes.

A Modular Tool Chain for MBT

The Fraunhofer Institute for Open Communication Systems (FOKUS) has developed Fokus!MBT (www.fokus.fraunhofer.de/en/motion/ueber_motion/technologien/fokusmbt/index.html), a flexible, extensible tool chain for MBT (see Figure 2). Fokus!MBT lets testers combine and integrate tooling as needed for MBT-based test processes.

Fokus!MBT facilitates automation of MBT processes for heterogeneous application domains. It's based on a service-oriented communication infrastructure of loosely coupled services interoperating with each other in a distributed environment. It defines a proprietary testing metamodel to formally represent test-specific information including the test model, requirements, logs, and so on. Telecommunications,

MODEL-BASED-TESTING COSTS

When considering model-based testing (MBT), you should take into account the following costs.

MBT TOOL COSTS

The cost of acquiring tools and frameworks to allow implementation is a necessary expense when switching to MBT.

MODEL-DRIVEN ENGINEERING TOOL COSTS

Companies can couple implementation of MBT with implementation of model-driven engineering (MDE) processes. To fully exploit MBT's advantages, companies should have an MDE infrastructure (tools and methodology).

ADAPTATION COSTS

The MBT methodology and tool platform must be fine-tuned with respect to the company's development processes, best practices, and domain requirements. Moreover, particular projects or project categories often require additional fine-tuning.

QUALIFICATION COSTS

The implementation, integration, and maintenance of MBT procedures require a higher level of expertise than traditional test activities. Managers must consider the costs for qualification and training of current employees as well as for new experts.

ROLLOUT COSTS

Changing existing methods, procedures, and best practices always involves rollout costs.

Requirement ID	Test case	Verdicts		
		Test case verdict	Local req. verdict	Global req. verdict
Req. 001	1	Fail	Fail	Fail
Req. 002	1	Fail	Fail	Fail
	2	Pass	Pass	
	3	Inconclusive	Fail	
Req. 003	4	Pass	Pass	Pass
	5	Pass	Pass	
Req. 004	5	Pass	Pass	Partial pass
	6	–	Not executed	

FIGURE 2. The Fokus!MBT approach. This tool chain for MBT lets testers combine and integrate tooling as needed for MBT-based test processes.

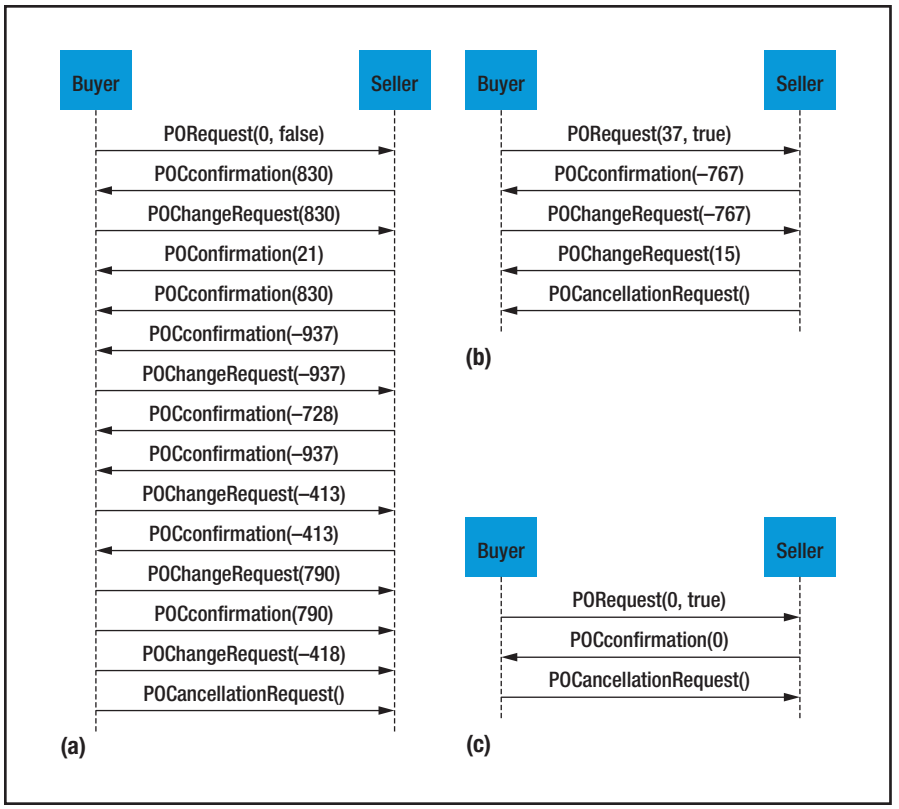


FIGURE 3. Generated test sequences and test results for Fokus!MBT. (a) Random path, 60% transition coverage. (b) Shortest path, 60% transition coverage. (c) Shortest path, all states.

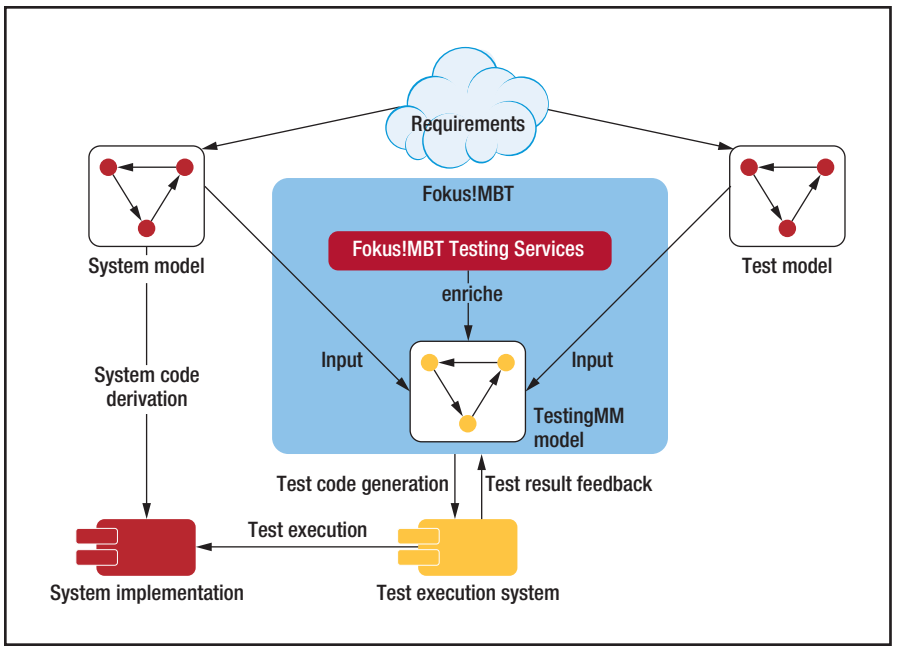


FIGURE 4. Model-based testing tool chain.

air traffic management, and safety-critical medical devices have successfully used it for model-based testing of software-based systems. Figure 3 shows some generated test sequences and test results created by using Fokus!MBT. Figure 4 shows the MBT tool chain.

Driven by technological advances and the growing need for software quality, MBT has matured from a research topic to innovative leading-edge practices. It has succeeded in a range of domains, including information systems, embedded systems, communication networks, and distributed systems. Future developments will help automate the fine-tuning of MBT technologies and tools for domain-specific use in areas such as real-time or data-intensive systems.

References

1. A. Spillner et al., "Wie wird in der Praxis getestet? Umfrage in Deutschland, Schweiz und Österreich," *ObjektSpektrum*, May 2011 (in German); www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/os/2011/Testing/spillner_vosseberg_OS_testing_11.pdf.
2. *D-MINT* (Deployment of Model-Based Technologies to Industrial Testing); www.d-mint.org.
3. *Methods for Testing & Specification (MTS); Model-Based Testing (MBT); Requirements for Modelling Notations*, ES 202 951 v 1.1.1, European Telecommunications Standards Inst., 2011.
4. *Methods for Testing & Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*, ES ETSI ES 201 873-1 V4.3.1, European Telecommunications Standards Inst., 2011.
5. I.K. El-Far and J. A. Whittaker. "Model-Based Software Testing." *Encyclopedia of Software Eng.*, J.J. Marciniak, ed., Wiley, 2001, pp. 825–837.
6. M. Shafique and Y. Labiche, *A Systematic Review of Model Based Testing Tool Support*, tech, report, SCE-10-04, Dept. of Systems and Computer Eng., Carleton Univ., 2010; http://squall.sce.carleton.ca/pubs/tech_report/TR_SCE-10-04.pdf.

INA SCHIEFERDECKER is the head of the Competence Center for Modeling and Testing at the Fraunhofer Institute for Open Communication Systems. Contact her at ina.schieferdecker@fokus.fraunhofer.de.

Specification Based Testing – Part 1

Model Based Testing

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Understand
model based
testing strategies

Model Based Software Development

- **Develop an executable model of the behavior of system**
  - Eg. UML
- **Model can be analyzed and simulated**
- **Utilize tools to generate code from the model**

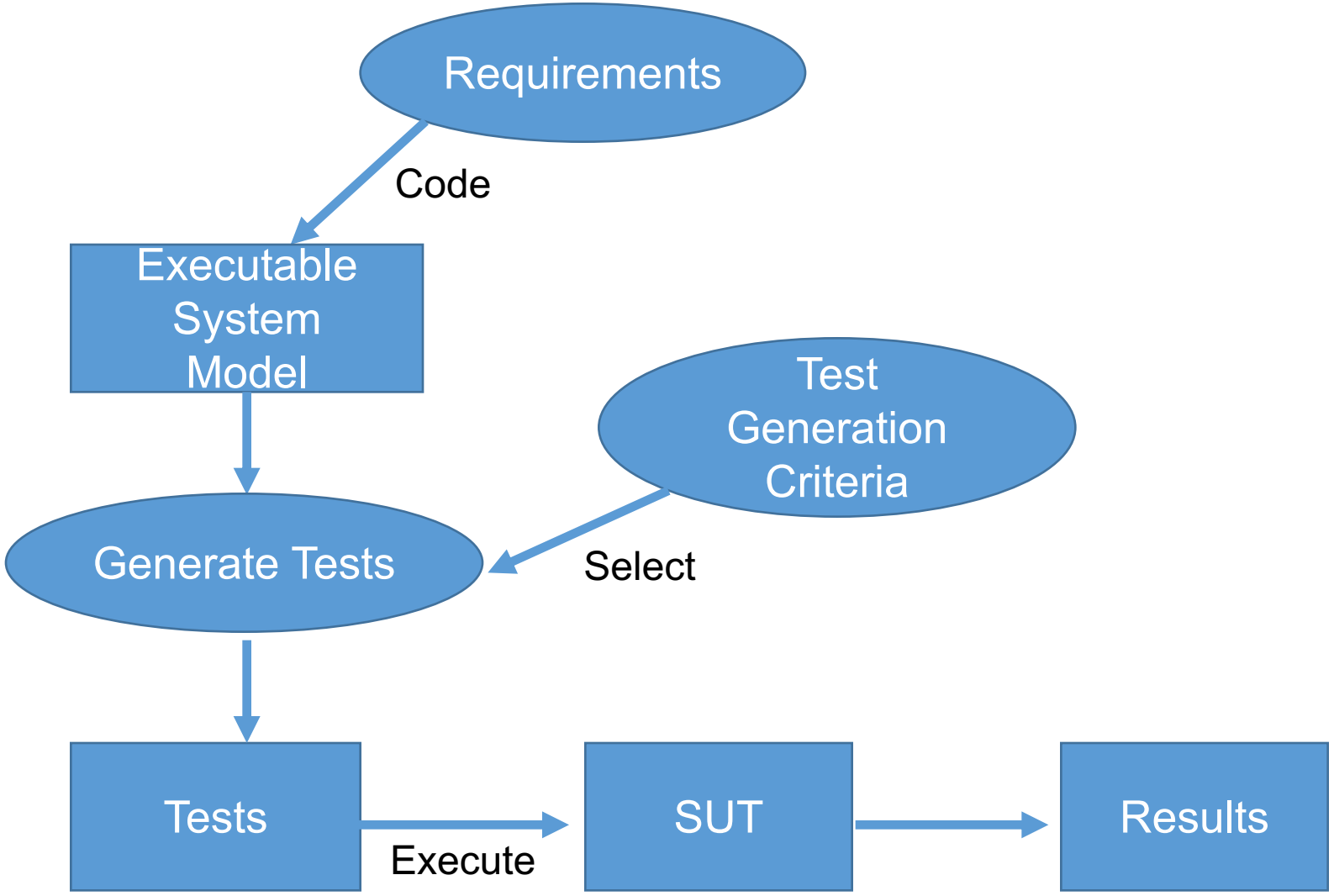
Model Based Testing (MBT)

Generate tests based on the system model

MBT steps

- Create a system model
- Select some test generation criteria (e.g. EP or BV)
- Generate tests
- Execute tests

MBT Process


Requirements
Code
Executable System Model
Test Generation Criteria
Generate Tests
Select
Tests
Execute
SUT
Results

Major Advantages of MBT

- **Modeling provides precision and reduces ambiguity**
  - Assists with program verification
- **Potential for automated test generation**
- **Changes in behavior directly translate into test changes**
  - Especially valuable for programs with volatile requirements

Summary

Specification Based Testing – Part 1

Scenario Based Testing

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Contrast requirements based vs scenario based testing

Functional Testing

- **Testing the system as a black box**
- **Testing the system the way a customer will be using the system**
- **Develop test cases from:**
  - User level documentation
  - Requirement specification documentation

Strategies for Developing Tests

Static (requirement driven)

- Traditional approach
- Primarily a verification activity

Behavioral (scenario testing)

- Supports both verification and validation activities

Requirements Driven Testing

- Carefully examine the requirements and develop test cases to ensure that each requirement is verified by one or more tests
- Normally a single test will verify several requirements
- Demonstrate traceability of requirements to test cases

Requirements Traceability Tools

- Numerous tools exist enabling software developers and engineers to “share” evolving requirements
- These tools facilitate the linking of tests to requirements supporting the necessary traceability

Requirements Traceability Example

Consider the following partial set of requirements for an ATM banking system:

1.0 System shall identify customers via a bank card and PIN

1.1 System shall prompt for card insertion

1.2 System shall validate card information

1.2.1 Stolen card shall be confiscated by system.

1.2.2 Unrecognized cards shall be returned with an error message

Requirements Traceability Example (cont'd)

Consider the following partial set of requirements for an ATM banking system:

1.3 System shall prompt for PIN entry

1.4 System shall validate PIN entry

1.4.1 System shall prompt for re-entry of invalid PINS up to 3 times.

1.4.2 System shall confiscate cards for more than 3 invalid entries.

Requirements Traceability Example (cont'd)

Consider the following partial set of requirements for an ATM banking system:

2.0 System shall present transaction options to the customer

3.0 System shall process transaction requests

3.1 System shall support a withdraw transaction

3.1.1 System shall prompt for amount and source of funds

Requirements Traceability Example (cont'd)

Consider the following partial set of requirements for an ATM banking system:

3.1.2 System shall validate sufficient funds exist

3.1.2.1 If insufficient funds, the system shall prompt for re-entry

3.1.3 System shall update the account and dispense cash

4.0 Upon end of transaction requests, the system shall generate a receipt and return the customer card

4.1 If card is not taken, system shall confiscate card

Requirements Traceability Example

(Partial test cases showing only inputs) Test #1

Customer inserts valid card, enters correct PIN, requests a withdrawal amount that is OK and takes his money and card.

Requirements covered: 1.0, 1.1, 1.2, 1.3, 1.4, 2.0, 3.0, 3.1, 3.1.1, 3.1.2, 3.1.3, 4.0

Requirements Traceability Example

Test #2

Customer inserts stolen card.

Requirements covered: 1.0, 1.1, 1.2, 1.2.1

Scenario Testing

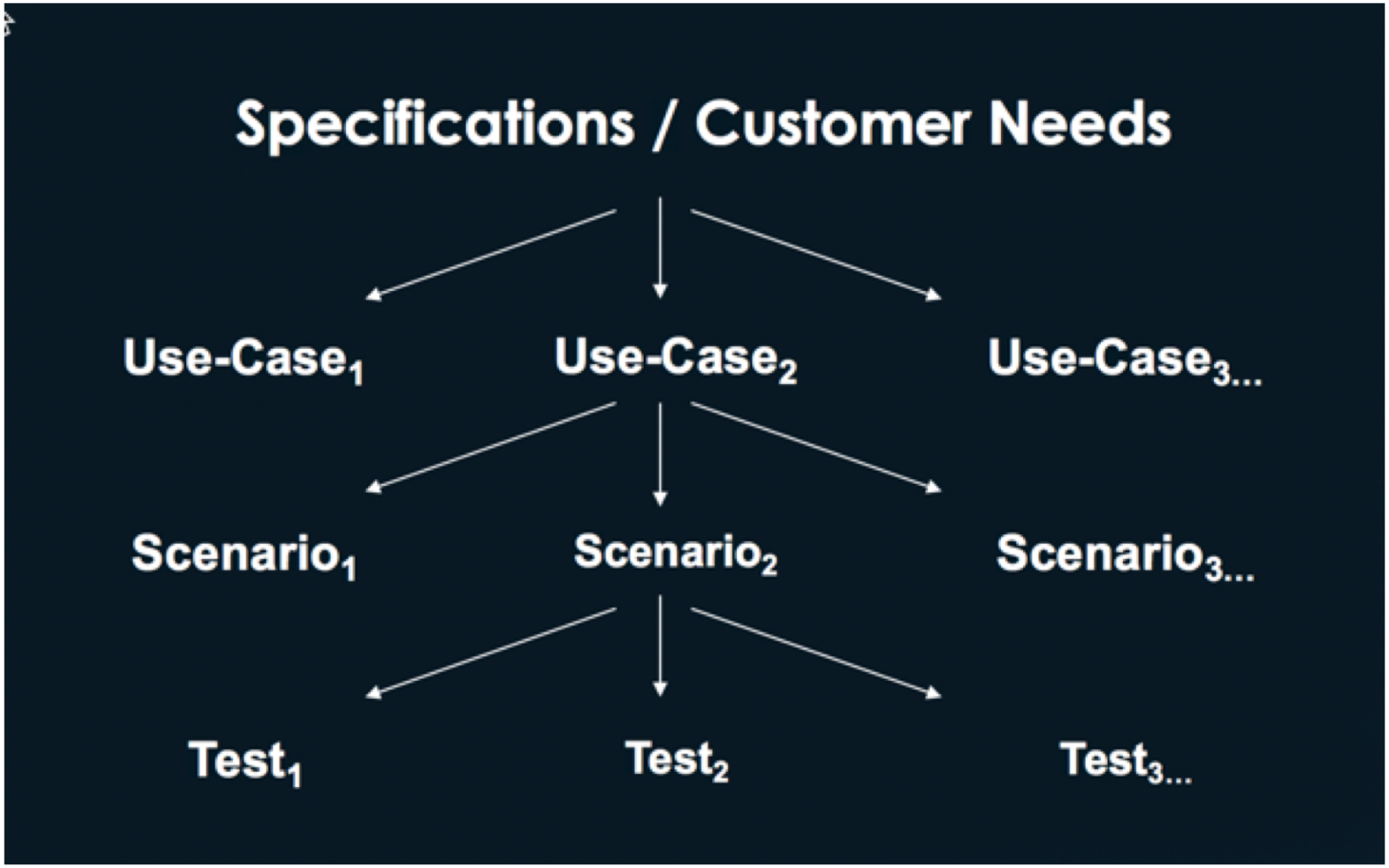
Use Cases

- All functional requirements are captured in use-cases (the requirements addressed by a use-case should be identified)
- Many non-functional requirements can be viewed as attributes of use-cases
- Use-cases provide a way of organizing and abstracting the requirements

Use-Case Testing Approach

- Develop a set of use-cases from the specifications if they do not already exist
- Verify and validate the use-cases
- Develop scenarios for the use-cases
- Test each scenario

Scenario Testing



Use-Case Verification and Validation

- Use-cases must be carefully examined to ensure that they are complete and correct
- Ensure that all of the needs of the customer are addressed
- Ensure all interactions among actors are correct
- Ensure all requirements are covered

Use-Case Construction

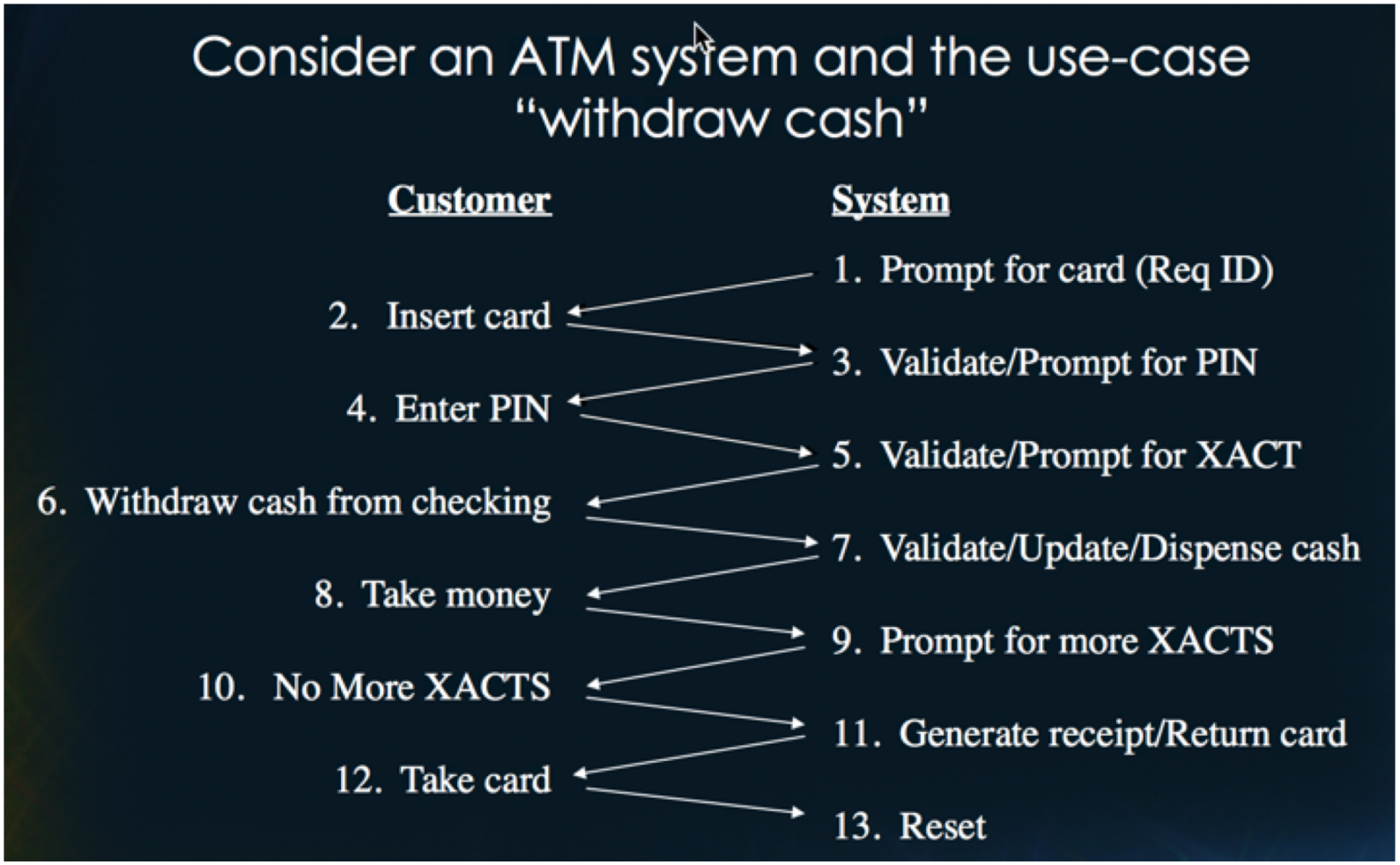
- Identify the actors (determine scope of system)
- For each actor, identify how the actor uses the system to accomplish its functions
- Detail each use-case identifying each of its processing flows as a scenario

Documenting Use-Cases

Use-cases can be documented:

1. Step-by-step description
2. State diagram

Example (Step-by-step Description)



Example Step-by-step Description (cont'd)

- Each path through the user-case represents a scenario which must be tested.

Example Step-by-step Description (cont'd)

Alternate Paths (subset of possibilities):

3a. If stolen card, then keep the card
3b. If account is not recognized, return card and generate message.
5a. If incorrect PIN, prompt for retry.
5b. If 3 invalid attempts, keep card.
7. If insufficient funds, generate message and allow re-entry.
13. If card is not taken, keep card.

Summary

Specification Based Testing – Part 1

State Based Testing

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Apply state based testing technique

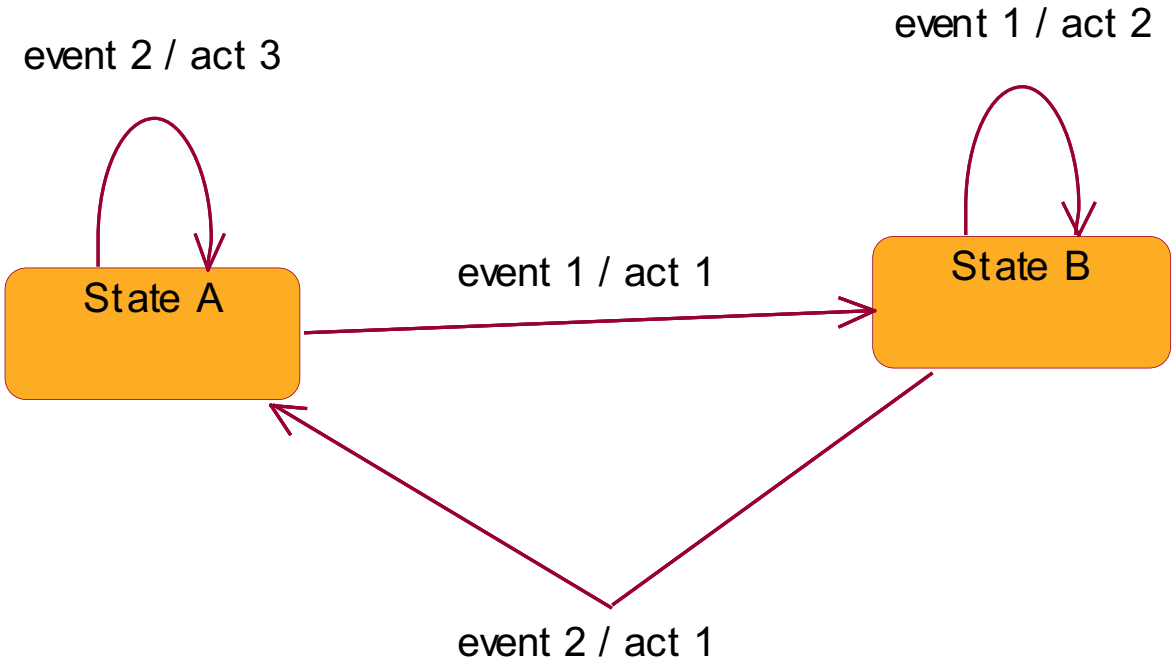
State Machines

- System behavior can sometimes best be captured in the form of a state machine
- State machine consists of a set of states and events
- When an event occurs in a particular state, it triggers a state transition and a response

State Machine Table Representation

	State A	State B
Event 1	State B / act 1	State B / act 2
Event 2	State A / act 3	State A / act 1

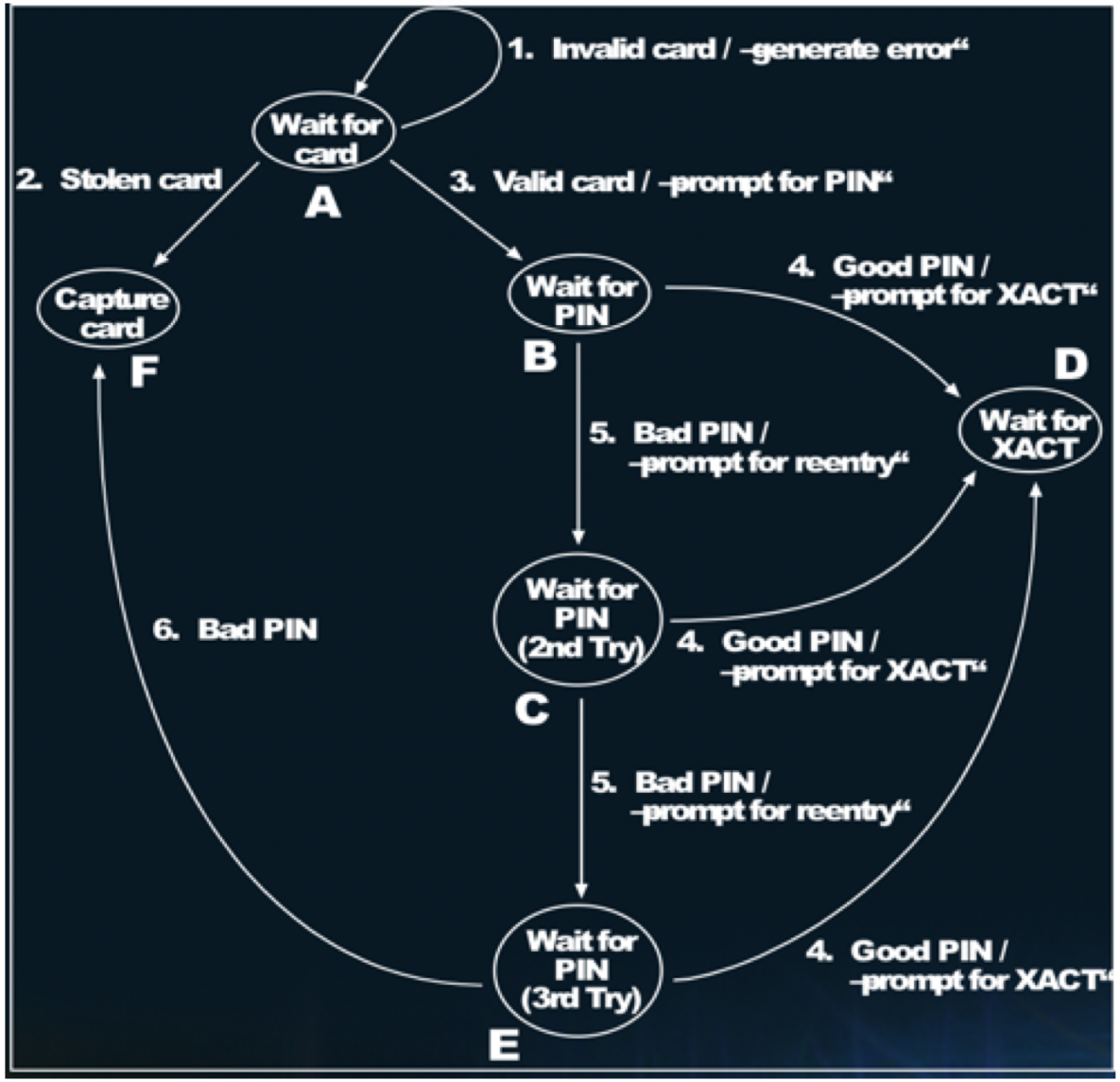
State Machine Diagram



Example State Diagram

- Consider the use-case for validating a user in an ATM system

Example State Machine



State Based Testing

State machines must be inspected for:

Completeness

- every state / event pair must be identified
- verify conditional transitions are correct

Contradiction

- 2 transitions from the same state should not contain the same event
- danger occurs with nested state charts

State Based Testing

State machines must be inspected for:

Unreachable States

- can be complicated with nested state machines and the use of conditional transitions

Dead States

- states that can be entered but not exited

Testing Finite State Diagrams

A testing cover can be developed for a Finite State Diagram by the following steps:

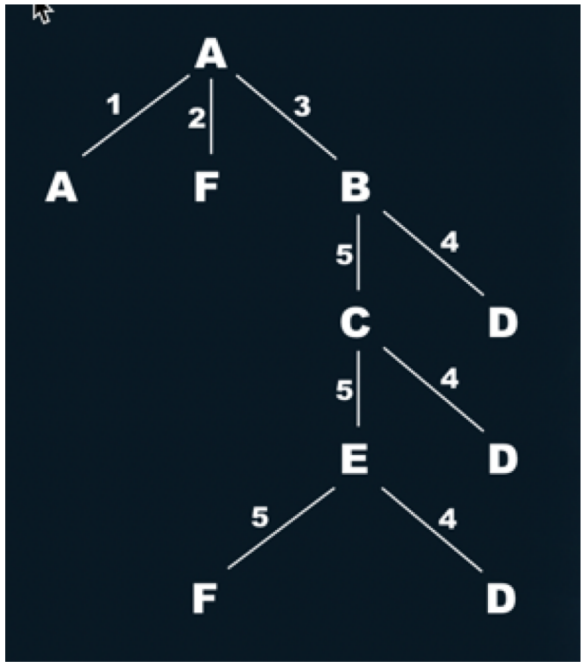
1. Develop a state testing tree
2. Identify test sequences (paths through the tree)
3. Develop tests to contain the sequences starting with the Start state and ending with observable behavior

Developing a State Testing Tree

- Begin with start state as root of tree
- At the first level, identify each of the transitions from the start state and the new states reached
- Continue expanding the tree downward for each state which has not previously occurred in the tree at a higher level

State Testing Tree

Testing Sequences	Tests
1	1
2	2
3, 4	3, 4
3, 5, 4	3, 5, 4
3, 5, 5	3, 5, 5
3, 5, 5, 4	3, 5, 5, 4



Tests map exactly to testing sequences since each sequence generates observable behavior.

Summary

Specification Based Testing – Part 1

Testing Asynchronous Events

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Test asynchronous events

External Events

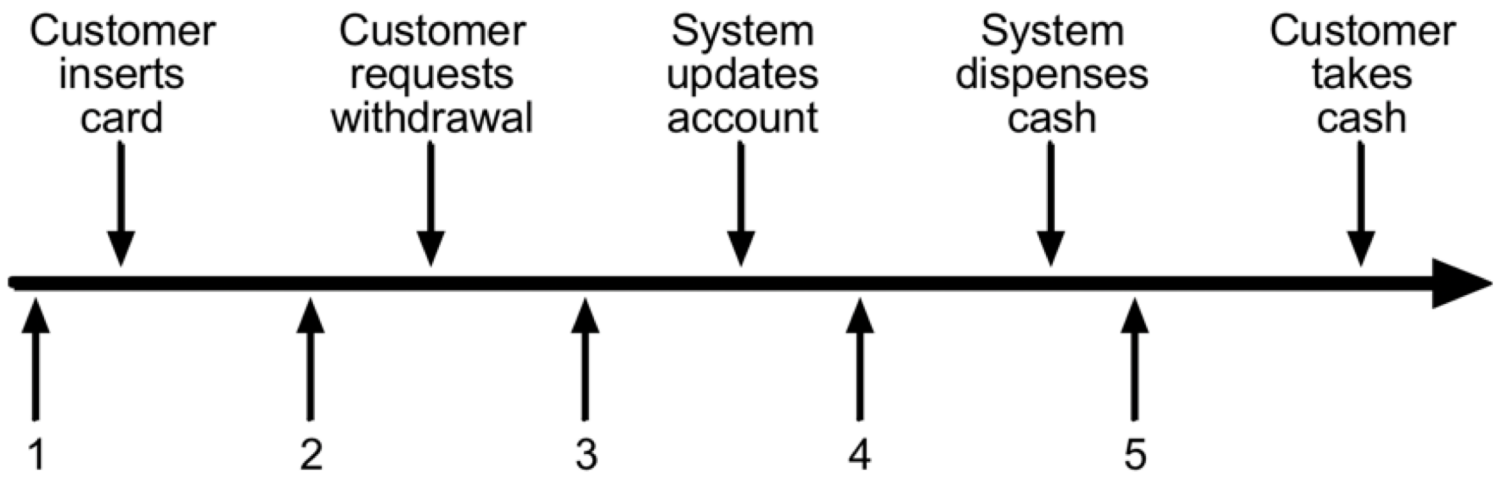
- An occurrence in the real world important to the system
- Normally characterized as messages between an actor and the system

Time Lines

- Used to model asynchronous events
- First step involves identifying significant use-case activities and placing them on a line in which time progresses from left to right
- Tests can be generated corresponding to the occurrence of asynchronous event along the time-line

Example ATM - Timeline

- Significant use-case activities can be placed on a time-line to assess the impact of asynchronous events
- Tests can be generated corresponding to the occurrence of the event along the time-line



Summary

Specification Based Testing – Part 2

Combinatorial Coverage as an Aspect of Test Quality

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Apply combinatorial test coverage to assess test quality

Assessing Test Quality

- Numerous approaches exist for assessing test quality including:
  - Combinatorial coverage
  - Mutation testing
- Combinatorial coverage looks at how combinations of parameter values are tested together

- Various studies show that most failures can be detected with combinations of a small number of parameters

https://ws680.nist.gov/publication/get_pdf.cfm?pub_id=917352

Total t-way Coverage

- For a given test set of "n" variables and values, proportion of t-way variable-value combinations that are executed
- E.g. Assume we are testing a function with 3 variables:
  - Variable a: has values 0 and 1
  - Variable b: has value 0 and 1
  - Variable c: has values 0 and 1

- What is the total 2-way variable-value configuration coverage achieved by the following tests:

  a=0; b=0; c=0

  a=1; b=1; c=1

  a=1; b=0; c=0

Summary

Specification Based Testing – Part 2

Design of Experiments

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Apply Design of Experiments to develop tests

Background

- Design of Experiments (DOE) is a systematic approach for evaluating a system or process
- DOE is heavily utilized in manufacturing and quality engineering
- DOE enables efficient investigation of the behavior of a system

Traditional Experimentation

- Traditional evaluation of the behavior of a system involves designing an experiment in which one factor is modified and the behavior on the system is assessed
- For example, consider varying oven temperature on the impact of the quality of a pizza

Weakness of Traditional Approach

- **The behavior of most systems is impacted by many factors**
- **Factors may also combine to create interactions**
- **In the pizza case, additional factors include:**
  - Rack positioning
  - Cook time

DOE Advantages

- DOE enables examination of the impact of a single factor as well as combinations of factors
- Values / ranges must be determined for each factor to investigate
- Experiments (runs) are made with combinations of the factors being considered and their impact on the system

Pizza Example Factors

Cook time

- Low / Med / High

Rack position

- 1 / 2 / 3 / 4 / 5

Temperature

- 350 / 375 / 400

Full DOE Combinations

- 45 Runs

DOE Classification

Full factorial design

- Tests for every factor value combination
- Pizza example

Fractional factorial design

- Only a fraction of all combinations are addressed
- Orthogonal arrays often used to address limited combinations of factors

Design of Experiments Pairwise Combinations

1. Identify the parameters that define each configuration
2. Partition each of the parameters
3. Specify constraints prohibiting particular combinations of configuration partitions

Design of Experiments Pairwise Combinations

4. **Specify configurations to test which cover all pairwise combinations of configuration parameter partitions satisfying the constraint**
   - “For any two parameters P1 and P2 and for any partition value V1 for P1 and V2 for P2, there is a specified configuration where P1 has the value V1 and P2 has the value V2. “

Pizza Example

1	Med	350
2	Low	350
3	High	350
4	Low	350
5	Low	350
1	Low	375
2	High	375
3	Med	375
4	Med	375
5	Med	375
1	High	400
2	Med	400
3	Low	400
4	High	400
5	High	400

Experiences with DOE in Software Testing

- Several companies have used DOE in software testing and have reported good results
- DOE has been shown to achieve reasonable code coverage

Warning

- Many software functions contain many parameters and factors
- Pairwise combination testing may leave many functions untested with normal, everyday scenarios

Summary

Specification Based Testing – Part 2

Mutation Testing

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Understand
Metamorphic
Testing

Test Oracle Problem

- A set of tests has been developed by an organization
- The organization executes the tests against a program
- All of the tests pass
- What can we conclude?

Mutation Testing

- Introduce defects (mutants) into program undergoing test
- Check to see if test cases can detect the mutant
- Work began in early 70's but was not widely adopted due to cost
- Today's automated testing environments make mutation testing feasible

Creating Mutants

- **Mutants are typically created via syntactical modifications of source code**
- **Mutation generation tools exist for this**

- **Examples of mutations**
  - Modify Boolean expressions (< vs <=)
  - Delete Statement
  - Modify Variable
  - Modify arithmetic operation

Mutation Testing Assumptions

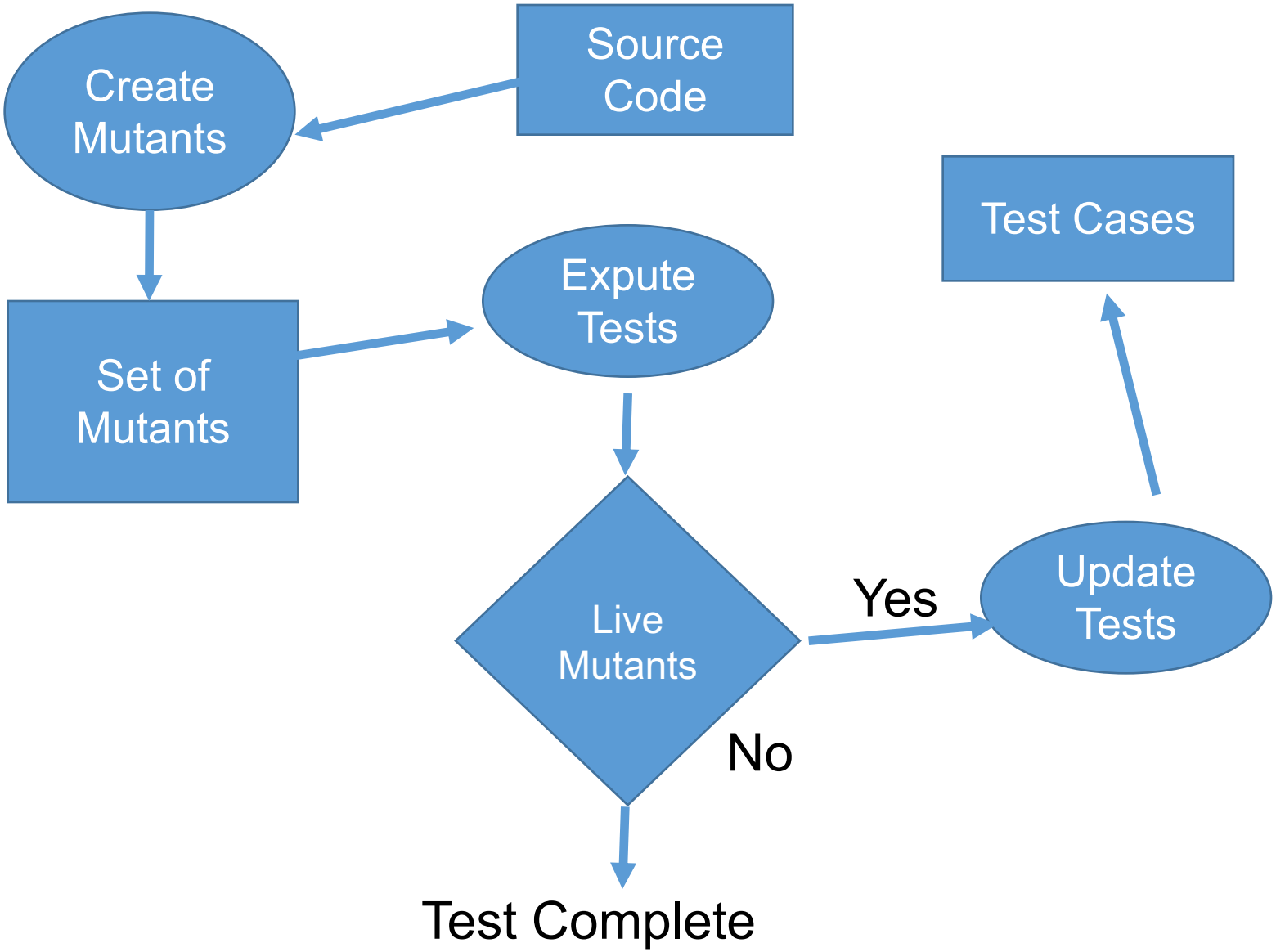
The Competent Programmer Hypothesis

- Programmers generally create code that is close to being correct reflecting only minor errors

The Coupling Effect

- Belief that test data that can detect small errors can also detect complex errors

The Mutation Testing Process



Summary

Specification Based Testing – Part 2

Fuzz Testing

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Understand Fuzz Testing

Fuzz Testing

- **Approach to testing where invalid, random or unexpected inputs are automatically generated**
- **Often used by hackers to find vulnerability of the system**
- **Test oracle is not needed**
  - Only monitor for crashes or undesirable behavior
- **Fuzzing tool used to generate inputs**

Two Types of Test Generators

Mutation Based

Generation Based

Mutation Based Fuzzing

- **Generates test inputs by random modifications of valid test data**
- **Doesn't require knowledge of the inputs**
- **Modifications may be totally random or follow some pattern tied to frequent error types such as:**
  - Long or blank strings
  - Maximum or minimum values
  - Special characters
- **Some tools use "bit flipping" - corrupt input by changing random bits in input**

Generation Based Fuzzing

- Generates random test data based on specification of test input format
- Anomalies are added to each possible spot in the inputs
- Knowledge of protocol should give better results than random fuzzing

When to Stop Fuzz Testing?

- Utilize code coverage tools

Summary

Specification Based Testing – Part 2

Metamorphic Testing

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Understand
Metamorphic
Testing

Test Oracle Problem

- **For many applications it is difficult to determine the expected results**
  - Graphics applications
  - Complex processing
  - Machine Learning
  - Big Data
- **Fuzz testing can be used to detect crashes**
- **Metamorphic testing can assist**

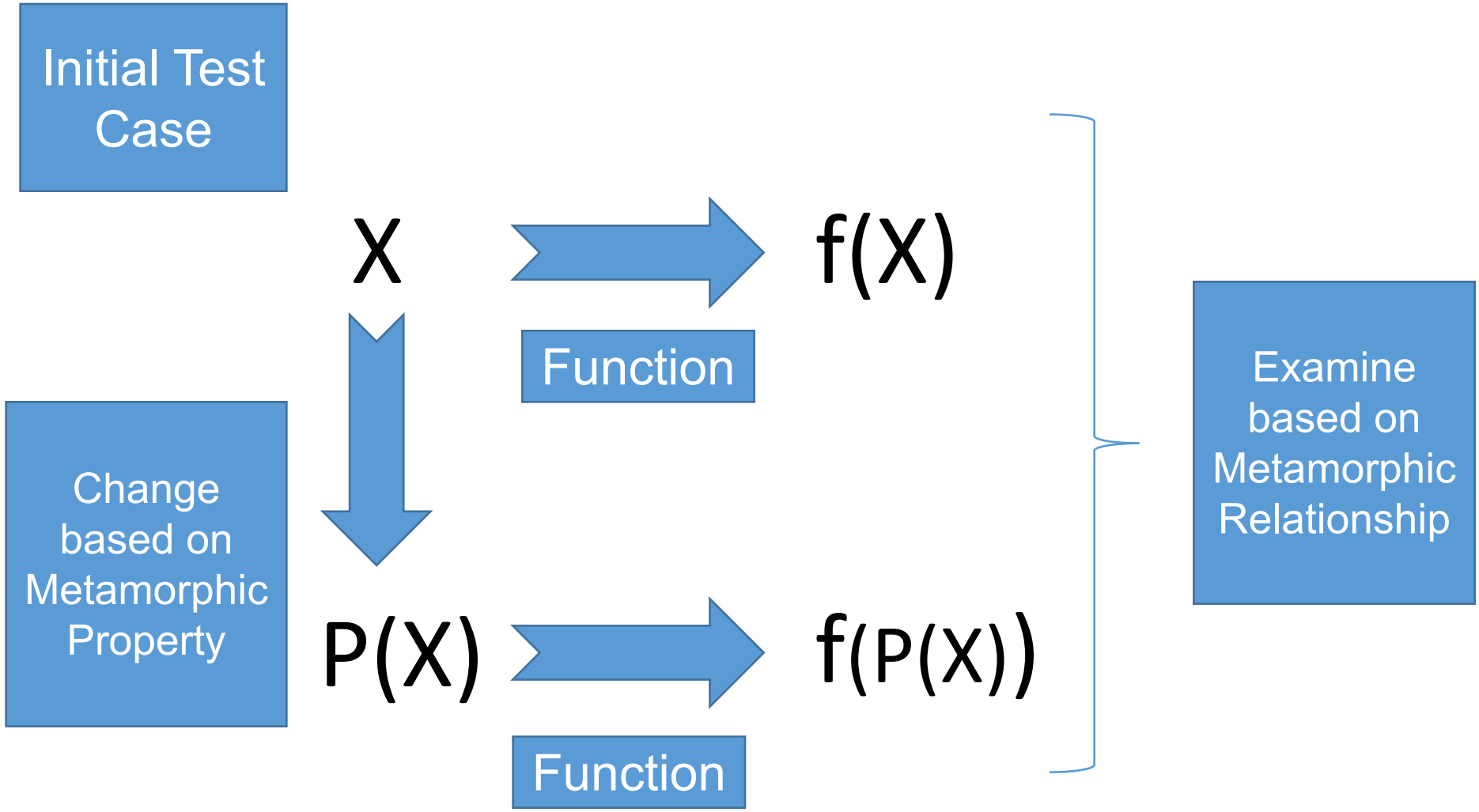
Assumptions

Some programs have properties such that when changes are made to inputs it is possible to predict changes on outputs (metamorphic properties)

Consider a service that calculates the variance of a sequence of numbers

- What is the relationship of changing the order?
- What is the relationship of adding 10 to teach number?

Metamorphic Testing



Standard Deviation Example

Initial Test: 10, 20, 30, 40, 50

- Result: 14.142

Second Test: 20, 30, 40, 50, 60

- Result: 14.142

Third Test: 10, 30, 50, 70, 90

- Result: 28.284

Summary

Specification Based Testing – Part 2

Defect Based Testing

Objective

Objective

Apply Defect Based Testing Technique

Defect Based Testing

- Utilizes defect taxonomies to derive test cases
- A taxonomy is a system of hierarchical categories for classification
- Defect taxonomies provide a classification of software defects
- Numerous defect taxonomies exist
- Typically developed and evolved from defects detected in the past

Beizer Generic Defect Taxonomy Categories

- Requirements defects
- Feature defects
- Structure defects
- Data defects
- Implementation and coding defects
- Integration defects
- System and software architecture defects
- Test definition and execution defects
- Unclassified defects

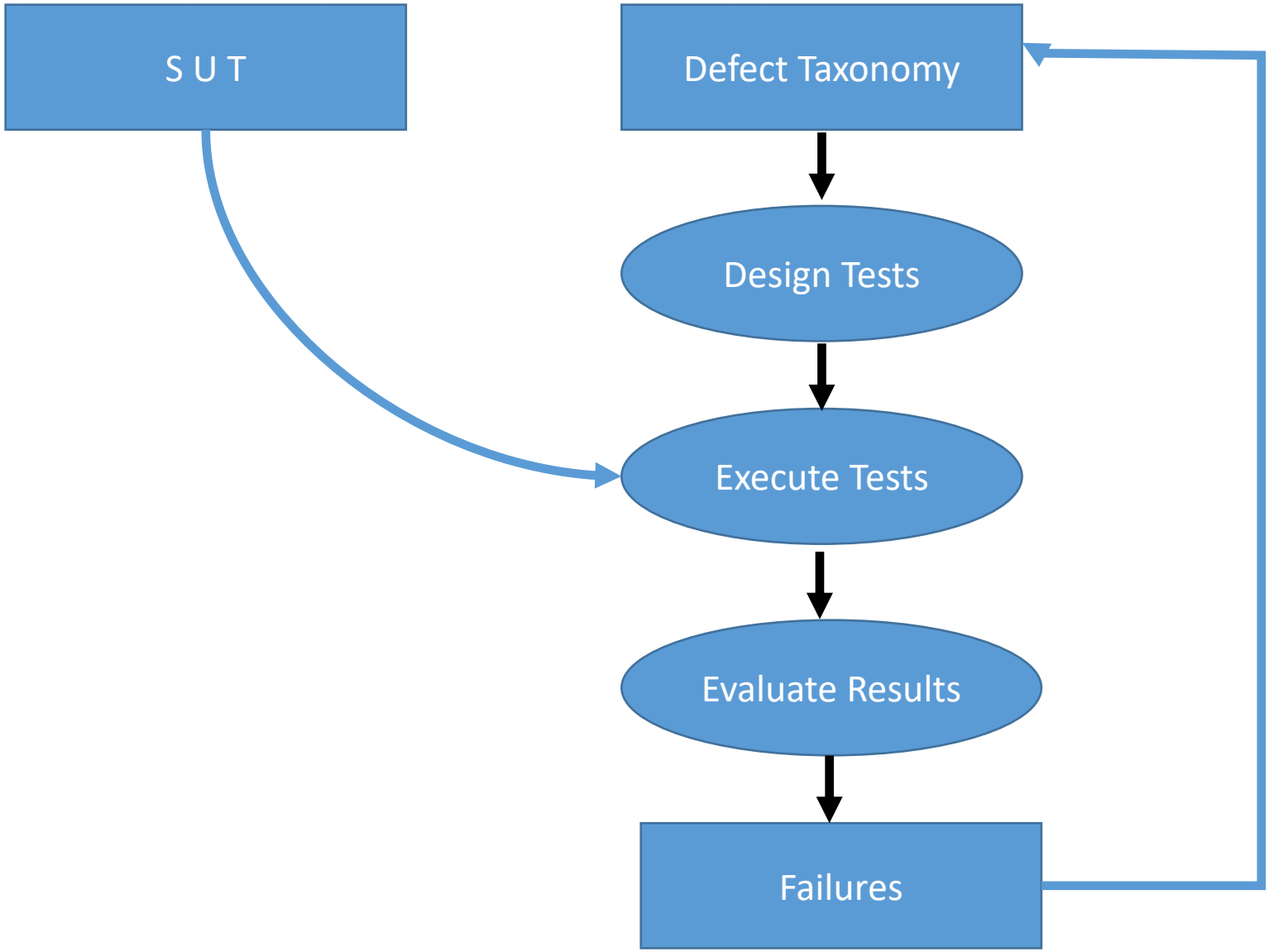
Community-Developed List of Software Weaknesses

- http://cwe.mitre.org/data/definitions/699.html

Defect Based Testing Approach

- Derive test cases to target specific defect categories
- Can be applied at any level of testing
- Example: consider developing tests to target divide by zero error in calculation

Defect Based Testing Process



Summary

Specification Based Testing – Part 2

Exploratory Testing

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Understand role of exploratory testing

Analogy to Early Explorers

- Learn as much as possible prior to the exploration
- Develop a systematic strategy for exploring
- Keep track of where you have been
- Be observant of possible side effects
- Document findings carefully

Exploratory Testing

- Unlike scripted testing, testers explore the product and write test cases on the fly
- Tests are driven from both requirements and previous test results (continuous learning)
- There is potential to detect errors missed by scripted and automated tests

Exploratory Testing (Session Based Testing)

- **Pair of testers work together for 90 minute session**
- **Testing is focused on a charter / tour (what to test)**
  - Analogous to going on a tour in a city
  - Provides structure to exploring the system (application tour, feature tour, menu tour) while focusing on different types of errors you are looking for
- **Session Report is generated**
  - What was tested
  - Results
  - Bugs

Sample Tours

Requirements tour:

- Find all the information in the software that tells the user what the product or certain feature does. Does it explain it adequately? Do results reflect the claims made?

Complexity tour:

- Look for most complex features and data, in other words, all places where most inextricable bugs could lurk

Continuous use tour:

- Leave the system on for a prolonged period of time with multiple screens and files open. Observe what happens as disk and memory usage increase

Documentation tour:

- Tour the help section of your product and follow some instructions to see if they produce the results desired

Sample Tours

Feature tour:

- Try as many of the controls and features available on the application as possible

Inter-operability tour:

- Check if the system interacts as it should with third-party apps and whether data is shared and updated as it should

Scenario tour:

- Create a scenario (user story) that mimics the real-life interaction of a user with the system and play it out

Variability tour:

- Look for all the elements that can be changed or customized in the system and test different combinations of settings

Summary

Structural Based Testing Strategies

Control Flow Testing

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Develop test cases to achieve control flow coverage

Code Coverage

- **It is important to analyze code coverage obtained by executing requirement's based test cases**
- **Code coverage can be assessed in terms of:**
  - Control flow
  - Data flow
- **Failure to obtain coverage may be due to:**
  - Undocumented requirements contained in the code
  - Dead code
  - Incomplete test cases for a requirement

Control Flow Coverage Levels

- Statement coverage
- Decision coverage
- Decision / Condition coverage
- Multiple condition coverage

Statement Coverage

Develop test cases such that every statement is executed at least once.

```
if a < 10 or b > 5
  then
        x := 50
  else
        x := 0;
if w = 5 or y > 0
  then
        z := 48
  else
        z : = 5;
```

Decision Coverage

Develop test cases such that each branch is traversed at least once.

- **What are examples of branch statements?**
- **Does decision coverage satisfy statement coverage?**
- **Does statement coverage satisfy decision coverage?**

Decision / Condition Coverage

- Develop test cases such that each condition in a decision takes on all possible outcomes at least once and each decision takes on all possible outcomes at least once

Multiple Condition Coverage

- Develop test cases such that all combinations of conditions in a decision are tested

Binary Search Example

```
inputs:   table, num, key
outputs: found, loc
start := 1;
end := num;
found := false
while start <= end and not found
    middle := (start + end) / 2
    if key > table [middle]
        then start := middle + 1
        else if key = table [middle]
                then found := true
                     loc := middle
                else end := middle -1
```

Summary

Structural Based Testing Strategies

Data Flow Testing

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Develop test cases to achieve data flow coverage

Approach

- Annotate control flow graph with 3 sets for each node
- Def(i) – set of variables defined in node I
- C-Use(i) – set of variables used in a computation in node I
- P-Use(i) – set of variables used in a test predicate

Example

```
Get x,z;
y := 0;
If x > 10
    then y := 15;
If z > 0
    then w := y+1
    else  w := y-1;
```

Definition Clear Path

A definition clear path from node “i” to node “j” for a variable x is a path where x is defined in node j and either used in a test predicate or computation in node j and there is no re-definition of x between node i and node j

Example

```
get x,y;
a := 0;
b := 0;
if x > 10
    then w := a+1
              b := 4
    else  w := b+1
             a := 4;
If y > 10
    then   z := a+w
    else    z := b+w;
```

Definition Use (DU path) Coverage

- For each definition of a variable, develop test cases to execute all DU paths
- DU path starts with the definition of the variable and ends with either a computational or predicate use of the variable along a def-clear path

Example

```
get x,y;
a := 0;
b := 0;
if x > 10
    then w := a+1
            b := 4
    else  w := b+1
          a := 4;
If y > 10
    then   z := a+w
    else    z := b+w;
```

Summary

Structural Based Testing Strategies

Static Analysis

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Identify static analysis techniques

Data Flow Analysis

- **Model the flow of data in a program**
  - Where are variables defined
  - Where are variables used
- **Perform analysis without executing the program**
- **Look for data flow anomalies**

Example Data Flow Anomalies

- Variable defined and then redefined without being referenced
- Referencing an undefined variable
- Defining a variable but never using it
- Numerous tools available to perform anomaly detection

Huang's Theorem

Let A,B,C be nonempty sets of character sequences whose smallest string is at least 1 character long. Let T be a 2-character string. Then if T is a substring of Ab^nC, then T will appear in AB^2C.

Summary

Structural Based Testing Strategies

Structured Testing

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Apply structured testing technique

McCable Cyclomatic Complexity

- **v(G) of a grap with e edges, n nodes and p connected components is e-n+p**

- **In a typical program:**
  - v(G) = #test predicates +1

Example

```
S1;
if x < 10
    then S2
    else if y > 0
            then S3
            else S4;
If z = 5
    then S6
    else S7;
```

Application for Testing

- Impossible to test all paths through code
- Structured testing provides a strategy for testing a subset of paths
- Select a set of basis paths (number is v(G)
- Linear combination of basis paths will generate any path
- Guarantees branch coverage

Identification of Basis Paths

- Select an arbitrary path through the graph as initial basis path
- Flip first decision while keeping other decisions constant
- Reset first decision and flip second decision
- Continue until all decisions have been flipped

Example

```
S1;
if x < 10
    then S2
    else if y > 0
            then S3
            else S4;
If z = 5
    then S6
    else S7;
```

Summary

Structural Based Testing Strategies

Symbolic Execution

Ira A. Fulton Schools of Engineering
Arizona State University

Objective

Objective

Utilize symbolic execution

Symbolic Execution

- Technique for formally characterizing a path domain identifying a path condition
- All paths in the program form an execution tree
- Involves executing a program with symbolic values
- Identifies test data to execute a path or determination that a path is infeasible

Notation

- A variable “x” will have a succession of symbolic values: A_0, A_1, A_2 … as a path is traversed
- Subscripts refer to the number of the previous statement executed

Example

(0) input A,B
(1) A := A + B;
(2) B := A + B;
(3) A := 2 x A + B;
(4) C := A + 4;

Multiple Paths Example

```
if (x <= 0) or (y <= 0) then
(1)          x := x2;
             y := y2;
        else
(2)          x := x +1
             y := y + 1
        endif
        if (x < 1) or (y < 1) then
(3)          x := x + 1;
             y := y +1;
        else
(4)          x := x - 1;
             y := y – 1;
        endif
```

Example

Path Conditions

- In addition to symbolically evaluating a program variables along a path, we can also symbolically represent the conditions which are required for that path to be traversed
- The symbolic path condition must be expressed in terms of the initial symbolic values of the variables

Example for T,F Path

Example for T,F Path

Summary

1. True or False?
   Statement coverage always satisfies decision coverage.

- ○ True
- ◉ False

2. Given the code below, which set of test cases will achieve 100% statement coverage?

```
If a < 5 or b > 7
        X = 50;
        c = a + b;
Else
        X = 25;
        c = a – b;
If X = 50 and c > 6
        Z = 10;
Else
        Z = 12;
```

- Test Case 1: a=2, b=10, c=12, X=25

  Test Case 2: a=3, b=4, c=4, X=25

- Test Case 1: a=2, b=10, c=12, X=50

  Test Case 2: a=5, b=1, c=4, X=25

- Test Case 1: a=3, b=10, c=13, X=50

  Test Case 2: a=5, b=1, c=4, X=25

- Test Case 1: a=3, b=10, c=13, X=50

  Test Case 2: a=1, b=2, c=3, X=50

3. Given the code below, which set of test cases will achieve 100% decision coverage?

```
If a < 5 or b > 7
        X = 50;
        c = a + b;
Else
        X = 25;
        c = a – b;
If X = 50 and c > 6
        Z = 10;
Else
        Z = 12;
```

- Test Case 1: a=3, b=10, c=13, X=50

  Test Case 2: a=1, b=2, c=3, X=50

- Test Case 1: a=3, b=10, c=13, X=50

  Test Case 2: a=3, b=4, c=4, X=25

- Test Case 1: a= 3, b=10, c=13, X=50

  Test Case 2: a=5, b=1, c=4, X=25

- Test Case 1: a=2, b=10, c=12, X=25

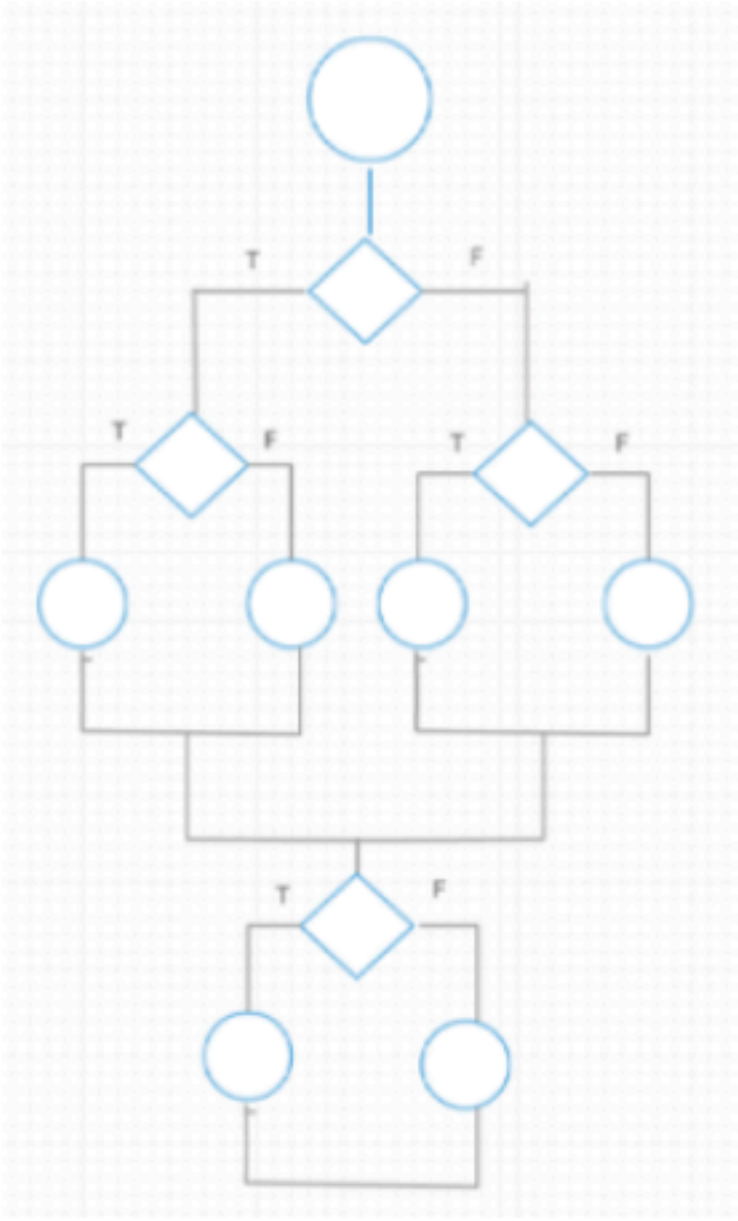
  Test Case 2: a=3, b=4, c=4, X=25

4. Given the code below, how many test cases are needed to achieve 100% multiple condition coverage?

```
If a < 10 or b < 5 or c > 15 or d > 2
        X = 10;
Else
        X = 20;
```

- ○ 4
- ○ 2
- ○ 8
- ◉ 16

5. What is the cyclomatic complexity of the given control flow diagram?



- (●) 5
- ( ) 2
- ( ) 8
- ( ) 4

6. Given the code below, what is the correct set of DU Paths?

```
Get a, b
X = 0                    } Node 1

If a >= 5 (Predicate I)
        Then c = x + 3 (Node 2)
        Else c = 0 (Node 3)

If b < 4 (Predicate II)
        Then b = c + 4 (Node 4)
        Else b = c + 2 (Node 5)
```

- Def1(a) = USEI(a)
  Def1(b) = USEII(b)
  Def1(x) = USE2(x)
  Def2(c) = USE4(c)
  Def3(c) = USE4(c)
- **Def1(a) = USEI(a)**
  Def1(b) = USEII(b)
  Def1(x) = USE2(x)
  Def2(c) = USE4(c) || USE5(c)
  Def3(c) = USE4(c) || USE5(c)
- Def1(a) = USEI(a)
  Def1(b) = USEII(b)
  Def2(c) = USE4(c) || USE5(c)
  Def3(c) = USE4(c) || USE5(c)
- Def1(a) = USEI(a)
  Def1(b) = USEII(b)
  Def1(x) = USE2(x)
  Def2(c) = USE4(c) || USE5(c)

7. Based on the code and the DU paths below, what coverage does the test case provide?
A = 2; B=2

```
Get a, b
X = 0          } Node 1

If a >= 5 (Predicate I)
        Then c = x + 3 (Node 2)
        Else c = 0 (Node 3)

If b < 4 (Predicate II)
        Then b = c + 4 (Node 4)
        Else b = c + 2 (Node 5)
```

- ○ 4/7
- ◉ 3/7
- ○ 5/7
- ○ 2/7

8. True or False? Huang's Theorem helps reduce the number of iterations over a path during anomaly testing.

- ◉ True
- ○ False

9. Static analysis techniques are applied during program execution.

- ○ True
- ◉ False

10. What is the correct path condition representation for a False False path given the code below?

```
If (x <= 0) or (y <= 0)          (0)
then
        x = x2                   (1)
        y = y2
    else
        x = x + 1                (2)
        y = y + 1
    endif
if (x < 1) or (y < 1)
        then
        x = x + 1                (3)
        y = y + 1
        else
        x = x – 1                (4)
        y = y - 1
        endif
```

- [ ] [(x_0 > 0) and (y_0 > 0)] and [(x_0 >= 0) or (x_0 >= 0)]
- [ ] [(x_0 > 0) or (y_0 > 0)] and [((x_0 >= 0) or (y_0>= 0)]
- [x] [(x_0 > 0) and (y_0 > 0)] and [((x_0 >= 0) and (y_0>= 0)]
- [ ] [(x_0 > 0) or (y_0 > 0)] and [(x_0 >= 0) and (y_0 >= 0)]