CSE565

Unit 1

Testers vs Pollsters

- Both paid to make prediction

Classic Testing Mistakes

- Starting too Late
- Testing only through the user-visible interface
- Poor bug reporting
- Adding only regression tests when bugs are found
- failing to take notes for next testing effort
- Test Automation
  - Attempting to automate all tests
  - Expecting to rerun manual tests
  - Using GUI Capture/replay tools to reduce test creation cost
  - Expecting regression tests to find a high proportion of new bugs
- Code Coverage
- overreliance on beta testing
- sticking stubborlnly to the test pln
- recognition of the importance of testing
- believing that programmers cant test code
- physcial separtion between devs and testers
-
- 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.

Errors: Mistakes made by human
Defect/Fault: Result of error manifested in the code

failure: SOftware doesnt do what is is supposed to do

Lifecycle Methods

- Waterfall
- Agile
  - Manifesto
    - individuals and interactions are more important than processes and tools
    - Working software over comprehensive documenatation
    - Customer Collaboration over contract negotiation
    - Responding to change over following a plan
  - methodology
    - Scrum
      - Sprint
  - agile Testing
    - Continuous Integration
      - Static Code analysis
      - Compile
      - Unit Tests
      - Deploy into test environment
      - Integration/Regression Test
- TDD
- Red Phase
  - Write a minimal test on the behaviour needed
- Green Phase
  - Write only enough code to make the failing test pass
- Refractor Phase
  - Improve code while keeping tests green
- order:
  - Red -> Green -> Refractor

Software development process vs test development process
SW de

- Requirements.
- Desgin
- Code
- Test
- Maintenance

Test

- Test Objectives
- Test design (Approach that we followed) [Sampling Strategy]
- Writing Test Cases
- Executing the test
- Update tests

sw we spend a lot of time in planning

Testing Levels

- Unit/Component
  - Done by individual Developer, typically at their workstation, their unit does what its supposed to do
  - High degree of automation
  - Huge component of verification
    - code coverage
- Integration Testing
  - Integrate them
- System
  - Test all functional and non functional requirements
  - To find important problems and predict reliability
  - Kinds of testing where al of the software is integrated together, but bring in other things
  - software hardware integration
  - meet all functuonal capabuilitues and objectives
  - safety testing
- Acceptance testing

    - Kind that the customer would be doing
    - installing it at the customer side
- Beta
    - Idnetify custoimer and let them do some testing for us

Test Types

- Functional Testing
    - Verification, validation Activities, that the software does what it is supposed to be doing
- Non-Functional Testing
- Structural Testing
    - White box testing
    - Applicable in unit level
- Regression Testing

Testing principles and attitudes

- Principle 1
    - Resting 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
        - Some portions of the code is more complex than the others, hence more erros in that section
        - Programmer capabilities
- Principle 5
    - Testing is context dependent
- Principle 6
    - Absence of errors fallacy

Testing attitude

- Independence
- Customer perspective
- Demonstrate that the system works (testing intended functionality) - "break it testing" (verification)
- Demonstrate that the system is bulletproof (testing unintended functionality) - (validation)
- professionalism

Classic Testing Mistakes

- Beileiving the primary objective of system testing is finding bugs
  - Test must focus on finding important problems
  - Test must provide an estimate of system quality
- Not focussing on usability issues
- Starting too late
  - Test must help development avoid problems
- Delaying stress and performance testing until the end
- Not testing the documentation
- Not staffing the test team with domain experts
- Not communicating well with the developers
- Failing to adequetly document and review test designs
- Failing to learn from previous testing activities

Best Testing Practices

- Assess software reliability via stistical testing
- Develop an agile test design
  - accomomdate late chanhs
  - emphasis on regression testing
- Utilize model based testing technique
  - State diagram,
  - Create a model and generate the test automatically once changes take place
- Develop cross-functional development and test teams

When to stop testing

- Out of time or money
- no more defects found
- Demonstrated all requirmeents are met
- Demonstrated code coverage
- Meets reliability objects
- Customer is satisfied

ISTQB code of ethics

Public:

- Certified testers shall act consistently with the public interest

Client and Employer:

- Certified software testers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest

Product:

- ENsure that deliverables they provide meet highest professional standards as possible

Judgement:

- Shall maintain integrity and independence in their judgement
- when testers are urged to stop testing

Mangemement

- Test managers and leaders shall subscribe to promote an ethical approach to management of software testing

Profession

- Shall advance the integrity and reputation of the profession consistent with the public interest

Colleagues

- SHall be fair to and supportive of their colleagues and promote cooperation with devs

Self:

- Shall participate in lifelong learning regarding the practice of their profession and shall promote on ethical approach to the practice of the profession

---

Quiz 1:

1. What kind of activity is “break it” testing?:

- [x] Validation activity
- [ ] Verification activity
- [ ] Neither a validation nor verification activity
- [ ] Validation and verification activity

2. Agile development does testing through:

- [ ] Segmented integration
- [x] Continuous integration and testing
- [ ] A red, green, and refactor cycle
- [ ] A test session at the end of the project

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

- [x] Once the coding phase is complete
- [ ] After the requirements phase is completed and before the design phase starts
- [ ] Before the requirements phase is complete
- [ ] After the design phase is completed and before the coding phase starts

4. A project is due in two days, and the team of testers needs at least three days to finish testing. Based on the ISTQB Code of Ethics, what should the team of testers do?

- [ ] Only test until the deadline
- [ ] Stop testing immediately

- [x] Ask for an extension on the deadline to be able to test the software fully

5. Testing only shows the ---- ?

- [ ] Presence of defects and correctness
- [ ] Correctness
- [x] Presence of defects

6. Testing is the examination of the behavior of the program by executing it on all of the possible data sets.

- [ ] True
- [x] False

7. True or False? Model based testing requires modeling the behavior of the system.

- [ ] False
- [x] True

8. True or False? Does an absence of errors in a program show that the right product was built?

- [x] False
- [ ] True

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

- [x] True
- [ ] False

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

- [ ] False
- [x] True

Specification Based Testing: Black Box testing

- Applicable at every level of testing
- Requirements based testing
- Scenario Based testing
- Contrasting w traditional req testing
  - Analogous with TDD
  - Functional testing
  - Testing the system as a black box
  - Literally a black box
  - test the system w/o any knowledge of code
  - all levels of testing
  - integration testing, service testing, component, feature
  - trying to test as a customer
  - devloping cases from documentation (user lvel)
  - req specification docu
  - Two startegy
  - static req driven
  - Traditional
  - primarily a verification activity
  - Behavioral / scenario based etsting
  - helps address both verification and validation activities
  - Req driven testing
  - Test cases are developed around a list of requirements to verify that each requirement is met in requirements based testing. This is a verification activity. There is no check as to whether there are the right requirements, which is a validation activity.
  - looking at the req of the system and developing test cases form them
  - single test will verify several req
  - traceability of req to test cases (coverage)
  - every requirement will map at least to one test case
  - Lots of tools for traceablitly bw req and test cases'
  - Doors
  - Req traceablitly example
  - ATM Banking system
  - traceablility matirx

- Alternative
- Scenario testing
- Early OO type
- inputs and stimuli
- ultimately think of output and responses
- All functional req in use-case
- Map req to test cases
- Non functional req (perf req) can be viewed as attributes for use cases
- use case acan provide a way to abstract and organize the req
- Use case testing approach
- develop from specifications if they do not already exist
- verify and validate the use cases
- eg: Packing for a vacation
- some thrwo things
- some organized
- they come up checklist
- they check things on the list
- this is similar to req based approach
- theres still a question on how complete the checklist is
- same applies with testing
- scenario based would be diff
- how you're gonna be using the things
- you'll think of use cases
- things you'll do on your trip
- combination of work and vacation
- presentation materials
- vacation, golfing materials
- input stimili output helps with identifyoing missing items
- Scenariop testing
- think of user needs in terms of use cases
- use case broken down inot scneraios
- normal and alt scenarios
- every scenario map to one test, multiple test for each scenario
- use cases must be carefully examplkined
- completer nad perfect
- all cust needs addressed

- identify actors for your test cases//use cases
- all interactions are addressed
- all req are covered
- each actor- role int he systesm, and use cases for that actor and then detail it
- eg: actors - customer
- withdraw, check balance, etc.
- each becomes a use case
- we have normal and alt scenarios for this
- Creating test cases that validates these use cases
- In scenario based testing, requirements are developed by creating use cases, and then tests are developed around the use cases. Having use cases makes it easier to validate whether the right requirements are there for an activity and then verify that test cases exist for each requirement, similar to the packing example from the lecture.
    - Intro to scenario testing paper
        - Five key factors for scenario:
            -

- Equivalence Partitioning testing
    - At all levels of testing
    - def: equivalence partioninig
        - technique for dividing the input domain for a program, function, feature into a finite number of eq testing
            - valid and invalid partitions
            - everyy level of testing
        - Functional coverage is demonstrasted by mapping equivalence partition to test cases
        - phone application example
            - absolute value of x
                - testing this?
                - word size -32 bit
                - we wont try all inputs
                - try a +ve -ve 0 max min numbers
                - this is eq partitioning
            - input domain into finite number of partititons

        - char / numbers, char error, number greater than the mx etc
        - skill:- divide the input domain into partitions and then selcet one or more from these partititons
        - 3 steps:
            - For each input identify a set of equivalence partitions and label them
                - Invalid ones are that lie outside the valid ones
                -
            - Write test cases covering as many of uncovered valid equivalence partitions as possible
                - identifying test cases:
                    -
            - for each invalid equivalence partition write a test case that covers one of the uncovered equivalence partitions
        - validating a password
    - huge assumption
        - inputs are independent of each other
            - Does not worry about combinations
- Eq paritioning
    - Simple technique
    - look at our input domain, select values, once selected, valid tested togethertr, invalids tested one at a time
    - huge assumption:- we make an assumption that the inputs are INDEPENDENT of each other
    - does not worry about testing combinations
- Boundary value testing
    - Selecting test cases to test boundary conditions, corr to situations on, above, below the edges of input and output equivalence partitions
    - password length example
        - <6, 6-10, >10
        - 5,6,7 9,10,11
    - EP x BP :
        - select test cases that match both, that satisfy both at once

        - password length
            - {5} for less than 6
            - {6} for at 6-10 any valid
            - {10}
            - {11} for any value above 10
            - {5,6,10,11}
        - output of boundary values also matter
        - boundary value correct: code inspection
        - Summary:
            - edges of eq partitioning
            - combine testing,
                - emphasis on inputs and outputs
    - Eq and Bv testing paper
        - eq uni dimensional, var independent
        - multi-dimensional eq partitioning
        - extended
        - weak robust eq testing
            - invalid partitions
        - strong robust eq testing
            - test cases for all valid and invalud elemnts
        - boundary Value testing
- Cause Effect analysis
    - Cause effect analysis can be used when the inputs are dependent on each other.
    - Multiple inputs
    - imp to test the combinations of the inputs
    - necessary where cobination of decision needs to be addressed
    - makes use of decision trees and decision tables
    - exmaple of customer and order
        - they are not independent
        - they dependent
        - make a decision table, all diff combinations (3x3 here)
        - each column becomes a test case, each combination

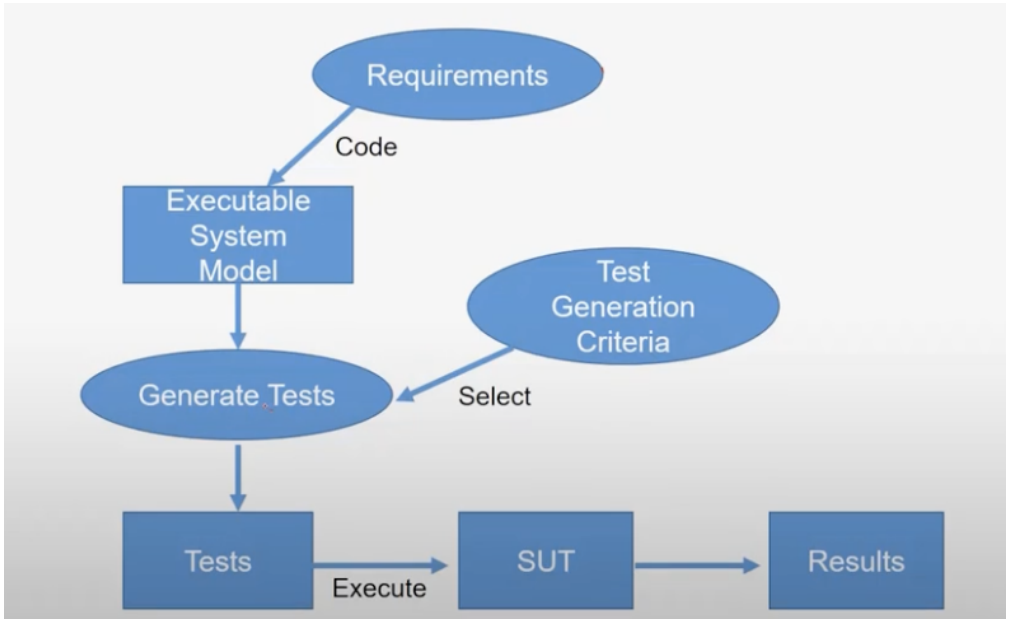
- Can also be done using decision tree
  - drawn from left to right
  - each of the paths become a test case
  - Practical Problems:
    - combinatorial explosion
      - all the variations of the varibales multiply each other
      - product of terms, a hige number
      - to reduce this, we can reduce the size of the terms
      - slightly move from black box to white box testing
      - another approach:
        - look carefully to reduce dependency
    - being able to reduce test case is imp
    - cause effect to deal with combinations
    - decision tree
    - assumption:
      - error handling bw diff types of customer aere independent
    - Ep x CE x Bv
- Cause and effect
  - through decisin trees and rtables
  - all inputs are dependent of each other - decision tables
    - decision tables we only look at valid test cases
    - test case - 8
    - fully exhaustive testing
  - Decision tree
    - Starting with first state - device
    - states - inputs
    - transitions - posssible input choices
    - fewer number of test cases - 6
    - assuming some cases
  - choice depends on situation and time for testing
  - Eq indepenedent
  - cause adn effect - dependent
  - both independent and dependent inputs -EP x CE

        - validation routine
        - Video: Equivalence Partitioning with Cause and Effect Analysis
- Testing asynchronous events
    - external events , things that real owrld that ultimaltely currespond to corr to common error, diff types of messages
    - Time lines: (system egineer technqiueb)
        - use case, input stimuli,
        - used to model synchronous events
        - first step involves identifying significnat use case activities and placing them on a line in which time progresses from left to right
        - tests can be generated corr to the occurence of asynchronous event along the time-line
        - Each use case requires its own timeline
        - eg: ATM:
            - significant use-case activites can be placed on a time-line to asses the impact of asynchronous events
            - Tests can be generated corr to the occurence of the event along the time line
            - Power failure for ATM
        - summary:
            - create a time line
            - during testing walk that time line
- Stat based Testing
    - State machine
        - 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 transition and a response
        - state machine table respresentation:
        - or Uml, diagramming rep
        - Example:
            - think of validation before diagramming
            - ATM Machine

    - customer authentication
- State machine testing
    - esting each state / event combination provides a minimal state machine cover.
    - completeness (validation activity)
        - state machine complete
        - 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
    - unreachable states
        - can be complicated with nested state machines and the use fo conditional transition
    - Dead States
        - state that can be entered but not exited
- Test covers
    - make sure we tested each transition
    - ensure we tested stimuli from diff states
- Testing cover steps:
    - Deevlop a state testing tree (top-bottom)
    - Identify test sequences (paths throught the tree)
    - develop tests to contain the sequences starting with the start state and ending with observable behaviour (address each one of the sequence)
- 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 previouslt occured in the tree at a higher level
- formal sepcificatin

        - heavily annoteated
        - model based testing can be edone with them
        - basic criteria:
            - develop test covers that we have sampled froom each of the different staets and each of the different transitions
        - Eg:
            - vending machine
                -

- Model Based Testing:
    - Model based software devlopment
    - req you to develop executablke model of the behavioural system
        - UML
    - a model that can be analyzed and simulated
    - take the model and genrate the code from the model
    - software going through diff forms
    - focussing on what the system has to do
    - bypassing the code state
    - when req are changed , updating the model , new code genrtaed
    - Generate tests based on the systems model
    - MBT Steps:
        - Create a system model (hard step)
        - Select some test generation criteria
        - Generate Tests
        - Execute tests

- 

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

---

assignnent -2

user name (apply bv and reduce the number of test cases)
5-10 chars (v)
= 5
= 10
= 4 (i)
= 11 (l)
valid ones (req0)

above that was mentioned for decision tree
9.7 decison table

UNIT 3
More Black Box Testing

Specification Based Testing

- Specification Based Testing
  - OBJECTIVES:
    - Combination of inputs to be tested
    - Exploratory testing
    - defect based testing
    - MUTATION TESTING
    - Apply combinatorial test coverage to test quality
    - Fuzz testing
    - Metamorphic testing
    -Combinatorial Coverage
      - Combinatorial Test coverage strategy
        - Combinatorial testing techniques are distinguished by their ability to efficiently and exhaustively test pairs of inputs.
        - Combinatorial coverage as an aspect of test quality
          - NUmerous strategies for tests
            - Code Coverage
            - Mutation Testing
            - Combinatorial Coverage
          - Combinatorial coverage looks at how combinations of parameter values are tested together
          - Variois Studies show that most failures can be detected with combinations of a small number of parameters
          - Studies show most failures can be detected with combination of a small number of parameters
          - Total t-way coverage
            - for given n variables and values, proportion of t-way variable-value combinations that are executed
          - How effective our test cases are
          - How do we know if we are doing a good job
          - Three-way testing

        - --a-|-b--|-c--
        - ---|---|---
      - Various way to asses how completer our test cases, how good our sampling is
      - here we will have multiple inputs and we should be able to asses
        - choose the one with the highest parameters to left (in class explanation)
        -

- Design of Experiments
  - TO develop test cases
  - Goal is minimize the runs
  - DOE systematic approach for evaluating a system or process
    - originated in manufacturing and quality engineering
    - doing experiments with a very efficient investigation of the behaviour of the system
    - Traditional evaluation of the behaviour of a system involves designing an experiment in which one factor is modified and the behavior of the system is assessed.
  - The behaviour of sw system is impacted by a lot of factors
  - Factors combine to create interactions
  - Combinations of factors, how they work together to impact system performance
  - values assocuiated wiht each factors
  -
  - diff factors, diff vlaues, try to do these experiments that minize the number of runs
  - DOE CLassification
    - Create an experiment that is a full factorial design
      - For every factor value combination
      - Analogous to Decision table
    - Fractional Factorial Design
      - Factor of 2 - most offten, pariswise combination testing

        - Only a fraction of the comibination are addressed.
        - Orthogonal arrays often used to address limited combinatioon of facotrs
        -
    - Initiaily focus on Pairwise Combination
    - Design of Experiments Pairwise Combinations
        - Identify the parameters that define each configuration
        - Partition each of the parameters (for each of the different inputs what are the partitions)
        - Specify the constraints prohibiting particular combinations of configuration partititons
        - Specify configs to test which cover all pairwise combinations of config parameter partitions satisfying the constraint
    - Several Companies have used DOE in sw testing and have reported good results
    - DOE has been shown to acheive reasonable code coverage
    - Warnings
        - DOE tends to do well for config testing,(form or system integration testing, prdt can work on many diff cocnfigs, eg: testing mobile apps)
            - Diff combinations
        - Might miss some functions and combinations untested, due to logic
- Using combinatorial testing to reduce sw rework
    - Paper
- Combinatorial coverage as an aspect of test quality
    -

Mutation Testing

- Organization executes the tests against a program
- All tests pass
- Can we conclude its a good product
    - It depends on our tests
- So mutation testing makes variarion of initial program / errros into the program
- Run the tests on the ones its incorrect

    - if it passess, we cannot really sya anything about the program
    - Doesnt mean if test passed, its good tests
- Introduce defects into program undergoing test
- Check to see if test cases can detect the mutant
- TOdays automated testing environments make mutation testing feasible
- Creating Mutants
    - typically created by Syntactic Modifications of source code
    - Mutation generation tools exists for this
    - Eg:
        - Modify Boolean Expressions
        - Delete Statement
        - Modify variable
        - Modify arithmetic operation
- Program needs to run, when you make these changes
- Mutation Testing Assumptions
    - The Competent Programmer Hypothesis
        - Programmers generally create code that is close to being correct reflecting only minor rerrors
    - The Coupling Effect
        - Belief that the test data that can detect small errors can also detect complex errors
- The mutation testing process
    - being able to asses quality of test , req covergae and code coverage
- Equivalent Mutants
    - Take the correct program, injecting defects.
    - even after injectimg defects, the result is program is still equivalent
    - Steps
        - generate mutants
        - Execute system
        - Result Analysis
- Mutation cost reduction techniques

Fuzz testing

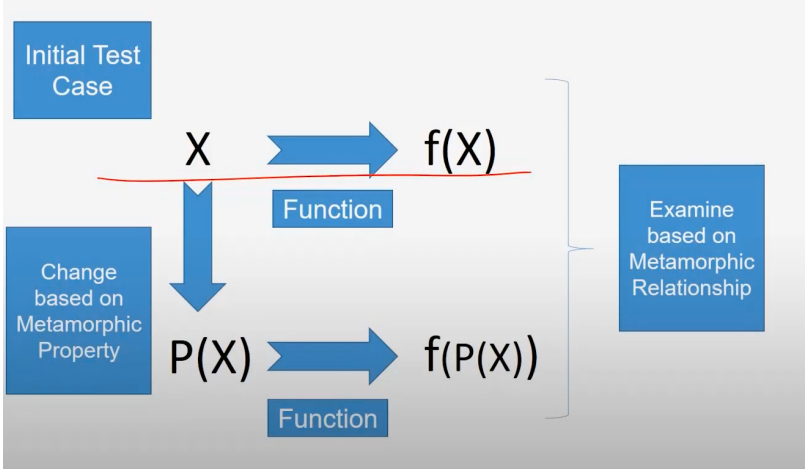
- Invalid, random or unexpected inputs are going to be generated into the system
- Fuzz testing is an approach to testing where invalid, random or unexpected inputs are automatically generated.
- Often used by hackers to find vulneratilities
- Key - Automation
- Also Addresses the test oracle problem
    - By def identify inputs and stimuli but also need to determine expected results
    - for a test case to pass, it must be equal to actual result
    - being able to detemine actual results must be tough
    - easier for it to fail, no test oracle needed
    - In fuzz testing we will be only looking at crashes
- Goal of fuzz testing
    - Monitor for crashes and determine undesirable behaviour
- Fuzzing tools (types of Test Generators)
    - Mutation Based
        - Automatic generation of test inputs by random modification of valid testdata
        - Driven by testinbg seed - valid tezt case
        - given the seed, they will take it and modify it
            - Modifications might be totally random or follow some pattern tied to frequent error types
            - Long or blank strings
            - Maximum or Minimum Values
            - SPecial Characters
            - No need of the knowledge of structure here
        - Some tools use "but flipping" - ocrrupt input by chnaging diff types of bits
    - Generation Based
        - Generate random test data based on specification of test input format
        - Need to know abt grammar and how test cases are generated
        - Being able to go in and make modification based on the structure known to us

        - Anomalies are based on what we know
        - Special Cases and then mutate for that
    - When to Stop Fuzz Testing
        - Utilize code coverage tools
            - This testing method helps in code coverage
    - Summary
        - Low Cost
        - Attempts to go in and break the system
        - No Need a test oracle
        - Fuzz testing is an approach to testing where invalid, random or unexpected inputs are automatically generated.
- Paper on Fuzz testing
    - Test Oracle - tell us if the test cases are successful or unsuccessful
        - Very difficukt to have
    - Getting rid of oracle
    - Almost genreate random tests
    - Lot s of test and checek against the data
    - if it passes, we dunno if it works, but if it fails, we know that we have to fix.
    - Mutation based
        - Mutate the test data
        - Taking existing test data and mutating it.
    - Generation Based
        - Driven from test Specification
        - Good spec for what these variables would be
    - Limitation of Fuzz Testing
        - Exhaustive testing is infeasible for a reasonable large program
        - It is hard to exercise program thoroughly without detailed understanding, fuzz testing maybe limited to shallow weaknesses
        - Finding out weaknesses in code caused the crash, maybe time consuming

Metamorphic Testing

- Test Oracle Problem

- Applications its tough to determine expected results
  - Graphics Applications
  - Complex Processing
  - Machine Learning
  - Big Data
- Compiler testing example
  - Infeasible
- Fuzz testing can be used to identify crashes (not enough)
- MetaMorphic Testing can assist
- Assumptions
  - Programs have properties such that when changes are made to inputs it is possible to predict changes on outputs
  - Some relationship that we can capitalize
  - eg:? variance of a sequence of numbers
  - Eg: Map testing example



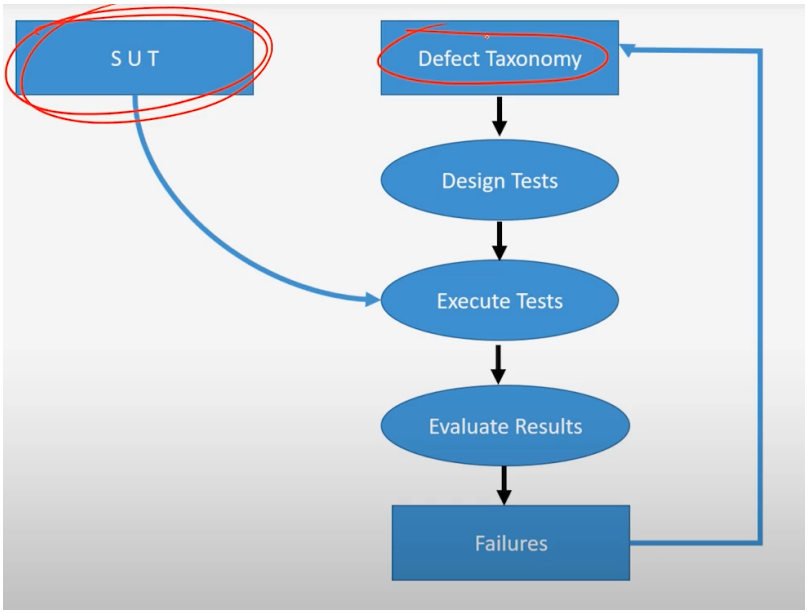
- Goal
  - Finding a relationship that we can use
  - Initial Test Case
  - Change based on Metamorphic Property
    - Look for properties, changes that we might be able to analyze
    - gives us the relationship we are looking for
  - Examine the metamorphic relatiojnship
- Summary
  - Addressing the test oracle problem
  - Recognizing relationships

    - Give confidence in creating tests
- Paper on Metamorphic Testing
    -

Defect Based Testing

- Primary -
- Defect based testing can be applied at any level of testing.
- Create test cases particularly targeted for specific types of defects that may exist in the system
- Utlize various defect taxonomies that exist
    - Organizing various types of defects and classsifying them
- Many Taxonmies exists
    - They come from analysis of defects
- Beizer Generic Defect Taxonomy categories
    - Requirement Defects
    - Feature Defects
    - Structure Defects
    - Data Defects
    - Implementing and coding defects
    - Integration Defects
    - System and Software Architecture defects
    - Tests definition and execution defects
    - unclassified defects
- Website: different categories of defects
- Goal
    - Derive test cases to target speciic defect categories
    - Can be applied at any level of testing
- Defect based testing
    - eg: divide by zero errors test cases
- Process looks like
  - SUT (system under test)
  - Defect taxonomy
  - Design Tests

- Execute tests
- Evaluate results
- Failures
- Defect Taxonomy



Summary

- Defect based
- we have made these types of defects before, we are not going to make the same mistake again
- Devlop test cases specifivally for these test cases and involve them in our test suite
- Defect based testing can target any defect category from the Beizer Generic Defect Taxonomy Categories.

Exploratory Testing

- Comparing with Planned forms of testing
- This is not considered ad hoc testing
- Analogy to exploratory testing
  - Learn as much as possible prior to exploration (do your hw)
  - Develop systematic testing strategy for exploration (a strategy)
  - Keep track of where you have been
  - Be observant of possible side effects
  - Document findings carefully

- Unlike scripted testing, Testers explore teh product and write test casess on the fly
    - Not gonna write them in advance
- Tests are driven from both requirements and previous test results (continuous learning)
    -
- There is no potential to detect errors missed by scripted and automated tests
    - Exploratory is like added on advantage
- Variation of exploratory testing (many ways)
    - Pair of testers work together for 90 mins sessions - Session Based testing
    - Testing is focused on a charter/tour - is not ad hoc. we got a purpose for what we are trying to achieve. gives a structure.
    - Session report is generated
        - What was tested
        - results
        - bugs
- Sample Tour
    - Requirement Tour
        - Find all the info about what the product does
        - does it explain adequately?
        - do results reflect the claims made?
    - Complexity Tour
        - Look for most complex features and data
        - inextricable bugs
    - Continuous Use tour
        - Leave the system for prolonged duration and observe what happens as disk and memory usage increases
    - Documentation tour
        - Go in and review the help section etc
    - Feature Tour
        - Try as many controls and features as possible
    - Inter-operability tour

        - Checkif the system, focus on that parts of the system that has interaction with 3rd party apps
    - Scenario Tour
        - Create a scenario that mimics real life interaction
    - Variability Tour
        - Look for all the elements that can be changed or customized in the system and different combination of settings
- Summary
    - Scripted/Automated - exploratory adds on to that
    - Structure this in some sort of tour
    - Report what you looked at
    - Exploratory testing can consist of requirements, features, continuous use, documentation, etc. tours that focus on different errors.

Unit 4 - Structural Based Techniques

- Looking at code, code coverage
- Primarily applicable in the unit level, service level, lill bit in integration level
- Not worried about this testing in overal pdt or system level testing
- Objectives
    - Generating test cases to achieve various control flow coverage
        - Statement, decision, multiple decision /condition
    - Devloping test cases for data flow coverage
    - Dynamic Analysis
    - Structurral technique
    - static analysis
        - Code anomalies

Dynamic Analysis

- Robustly test our code
- Look at code coverage

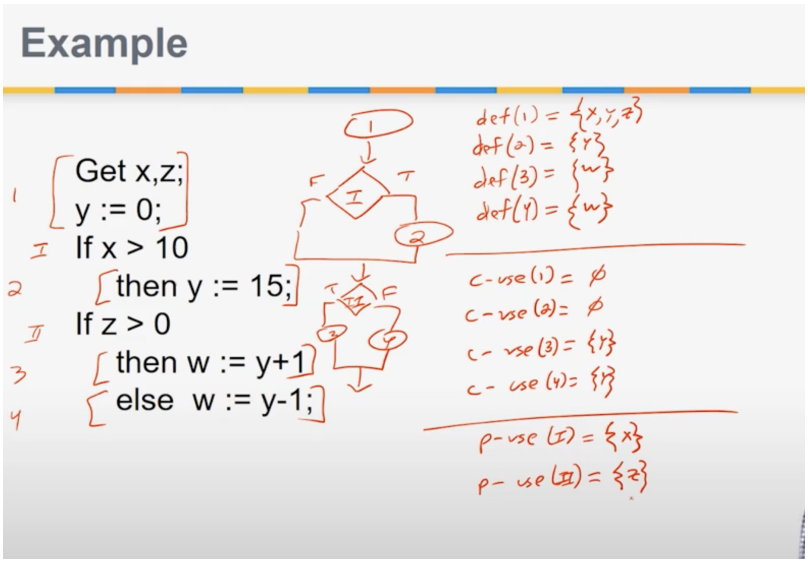
- We begin by looking at testing code coverage through control flow, which describes how well code is evaluated on different paths taken
- Control Flow
  - how well code is evaluated on different path taken
  - Typically done by individual developers
  - Lower level testing
  - High degree of tool support
  - Code coverage
    - How can you analyze and see how effective it is
    - Imp to analyze code coverage obtained by executing requirments based test cases
    - Assesed in terms of how they can execute :
      - Control flow
      - data flow
    - Failure to obtain coverage maybe due to:
      - Undocumented requirements contained in the code
      - Dead Code (code no way of being executing)
      - Incomplete Test case for a requirement
  - Coverage levels in Control Flow
    - Statement Coverage
    - Decision Coverage
    - Decision/Condition Coverage
      - branch condition coverage
    - Multiple Condition Coverage
    - Some are better than the others
  - Statement Coverage
    - Goal
      - Create testcase such that every statement is executed at least once
      - DO not stop until you;ve executed every statement at least once
      - Two statements
      - We address it through a control flow diagram
      - Control Flow Diagram

        - Sophisticated diagram
        - basically a flow chart
        - doing every path is not ideal
        - ovals rep statements
        - diamonds are ref to as test predicates
        - goal was to do code/statement coveregae
        - What if we are attempting to modify code
            - Statement cov changes a bit
            - dev test cases new statments have beene xecuted once
            - statement covergae minimal level of coverage
            - come back and you get the report on how many statements have been executed
        - Minimum Level of coverage
- Decision Coverage
    - Develop test cases such that each branch is traveresed at least once
    - we move to test predicates now
    - For each branch we need to do true and false
    - examples of branch:
        - You can have a case statements/ select statement
        - while loop or looping type constraints
    - street map example
        - intersection and house
    - Does decision coverage satisfy statement coverage?
        - YES
    - Does statement coverage satisfy decision coverage?
        - NO
    - Decision coverage is a stronger testing criteria
- 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 posssible outcomes at least once
    - we identify conditions within a test predicate, corr to expressions
        - Each of these is seen a switch, that can be switched on and off

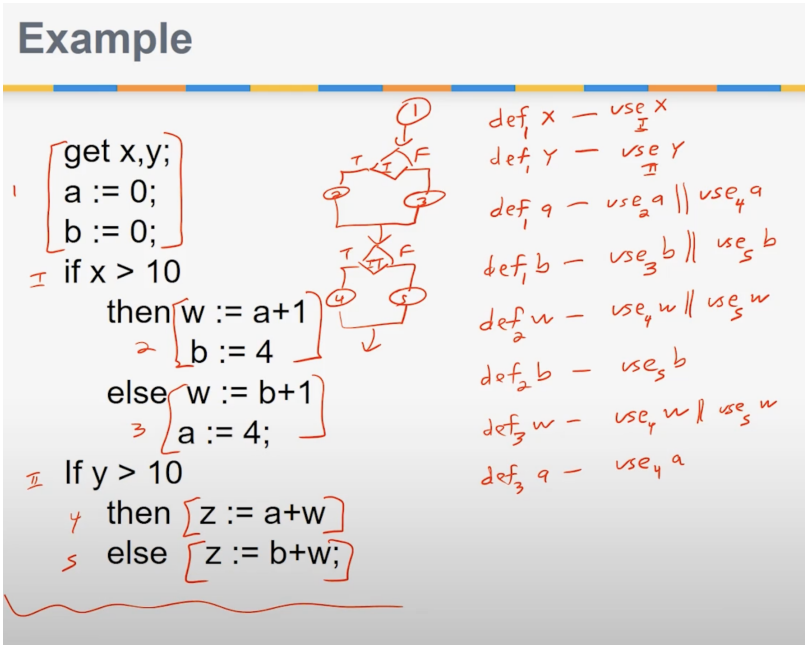
        - true and false, we have to try which satisfy all
    - Implies decision coverage
- Multiple condition coverage
    - All combination of coverage in a decision are executed at least once
    - 100% multiple condition coverage
- order
    - Multiple Condition (top)
    - Dec/condition
    - decision
    - statement (bottom)
- most of the app that we execute are last two in
- safety critical app will have top two
- Binary Search
    -

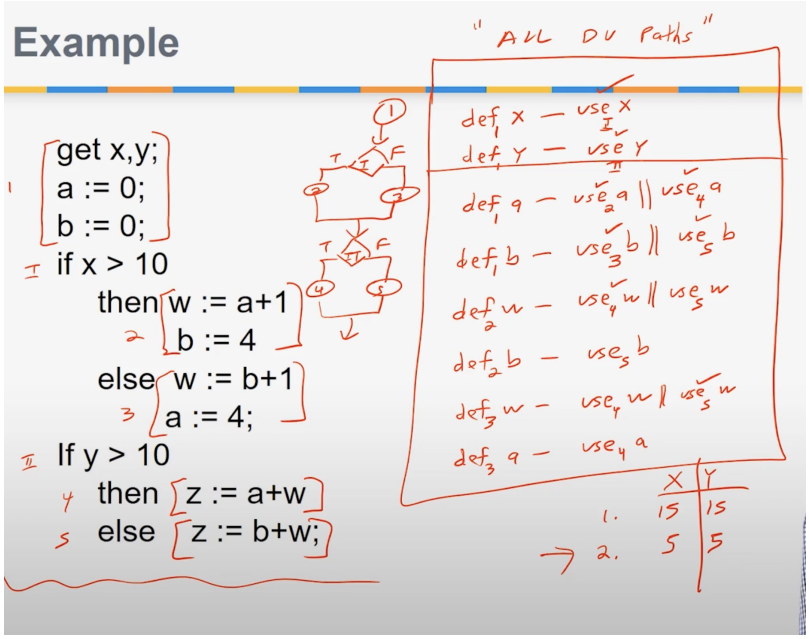
- Structured Testing
    - McCable Cyclomatic Complexity v(g)
        - e-n+2p (p connected edges)
        - v(g) = # testpredicate + 1;
        - basis paths : The number of basis paths is the minimum number of paths needed to build test cases and linear combinations for every other possible path.
        - identify subset of paths - Basis paths = no of basis paths is testpredicate + 1;
        - Testing basis path will guarantee
        - as complexity inc defects increase
        - time for understanding also went up
    - Application for Testing
        - Impossible to test all paths thorugh 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
        - Gurantees branch coverage
    - Identify basis paths

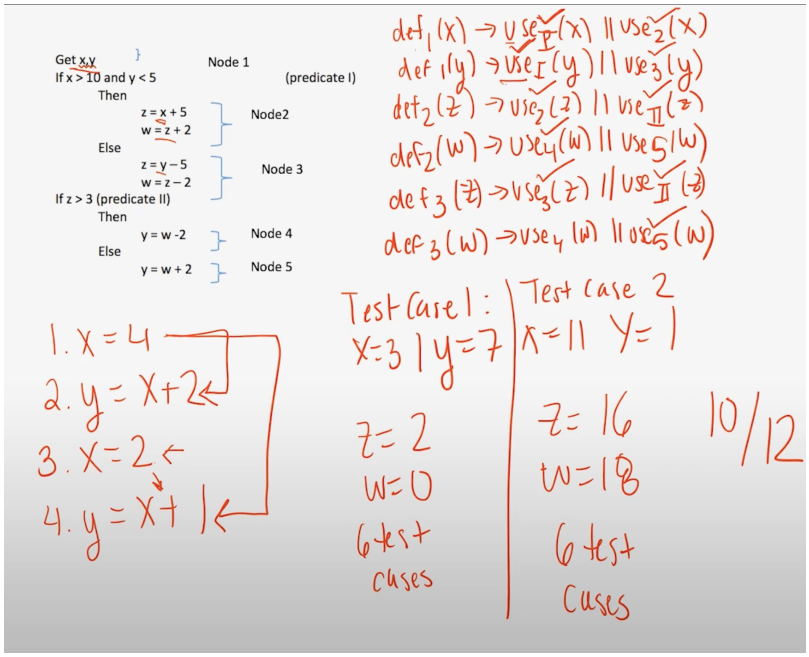
    - Select an arbitrary path through the graph as initial basis path, a longer one typically
    - Flip first decision while keeping other decision constant
    - Reset first decision and flip second decision
    - continue until all decisions have been flipped
    - Basis Path <= Total Paths
- Data Flow testing
    - Expertise in developing test cases
    - Approach
      - Annotate control flow graph with 3 sets for each node
      - Def(i) - each node i we look for a set that consosts of all the variables, set of variables that is 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



    - Clear Path: A clear path is from node "i" to node "j" for a varibale x is a path where x is defined in node i an d either used in a test predicate or conmputation in node j and there is no redefinintionof x between node i and j
    - Example

- 

- develop test cases and measure coverage
- Definition Use Coverage (DU)
  - objective is to test all DU paths
  - For each definition of a variable, develop test cases to execute all DU path
  - 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
  - 



Test automation not just for test execution paper

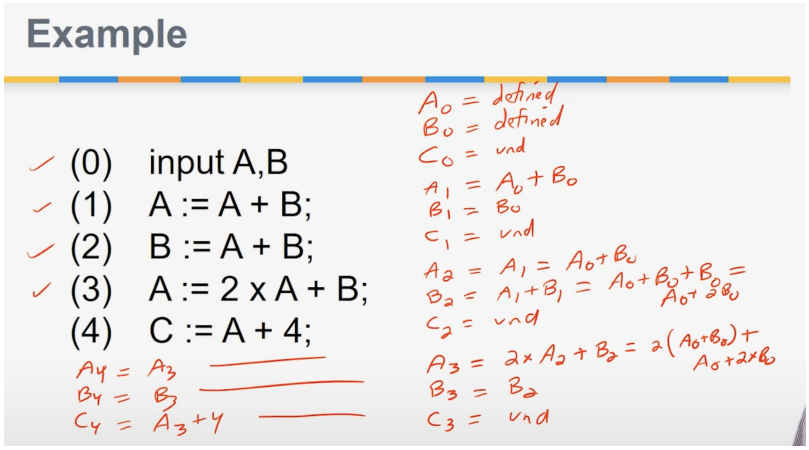
- Test Case Design
- Test Scripting
  - Detail step by step scripting
    - manual or automatic
- Test Execution
- Test evalution
- Test Results
- Test result reporting

Static Analysis

- Unlike dynamic analysis, static analysis does not require execution of a program to create and evaluate test cases.
- Static analysis attempts to model the flow of data in a program and provide insight into areas such as data flow anomalies
- Data Flow analysis
  - Model the flow of data in a program
    - Where variables are defined
    - where they are used
  - Perform analysis without executing the program
  - Look for data flow anomalies
- Examples data flow anomalies
  - Variables 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
- Huangs theorem
  - Let A, B, C be non empty 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. goign through loop twice.
  - T is gonna end being an anomaly
  - b^n indicates looping
  - This helps limit our search. Go through each one of the path twice. if the anomaly does not occur upto 2 times it wont occur 3 4 times also.
- Symbolic Execution
  - We can come up with test data that can execute a particular path in the program, or maybe to determine or prove that a particular path cannot be executed
  - Technique for formally characterizing a path domain identifying a path condition
    - Possible domain
    - path in the code
    - susbet of the path domain
  - 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: A0, A1, A2,
  - walk through the program and look at the state of the prgram after ecxecuting a statement

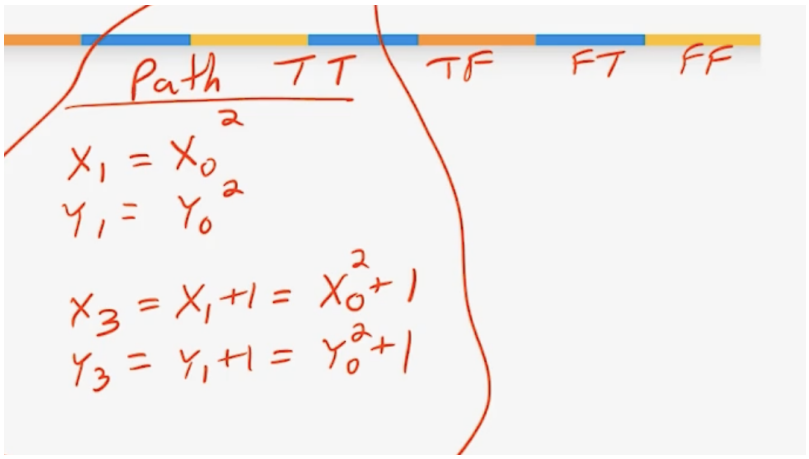
```
(0)   input A,B
(1)   A := A + B;
(2)   B := A + B;
(3)   A := 2 x A + B;
(4)   C := A + 4;
```

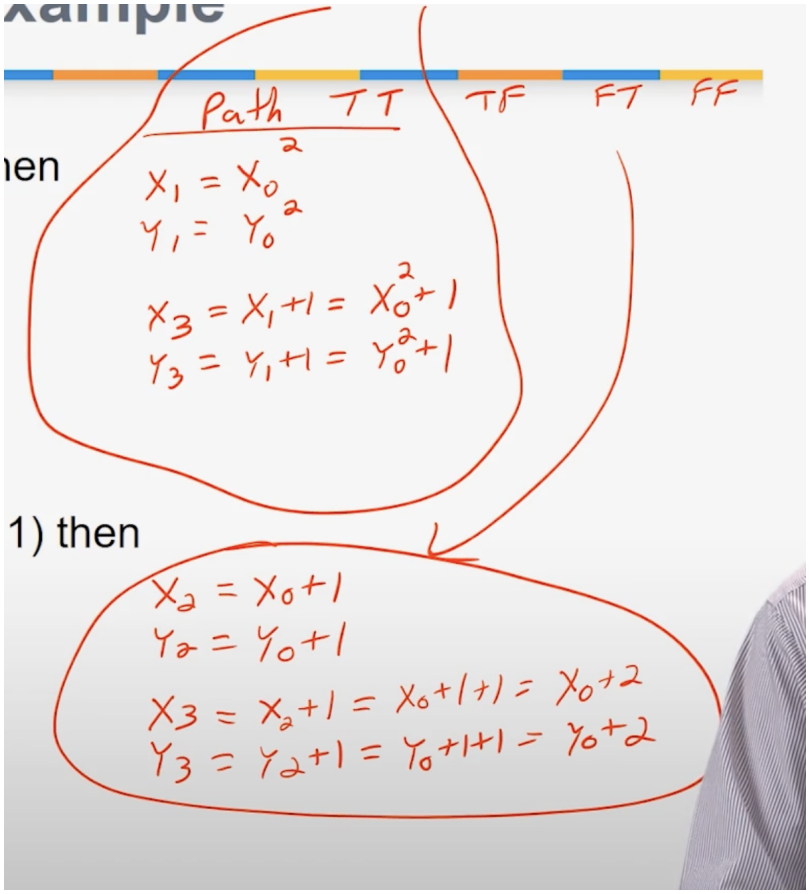


Multiple Paths Example

```
if (x <= 0) or (y <= 0) then
(1)         x := x^2;
            y := y^2;
        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
```

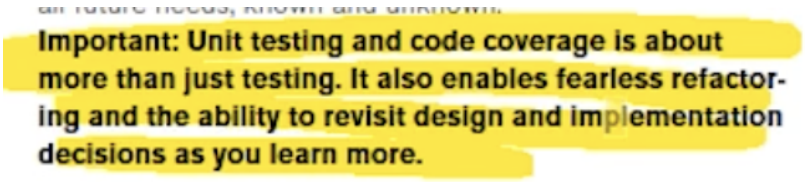
- 

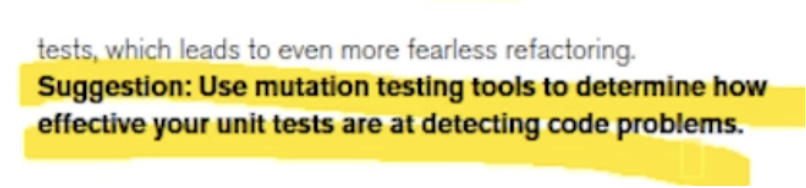
- 

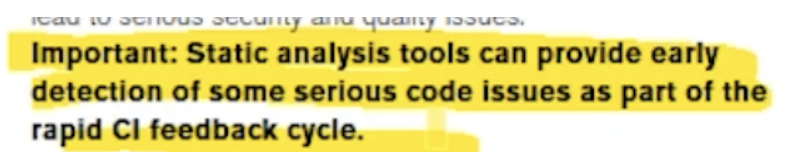
- 

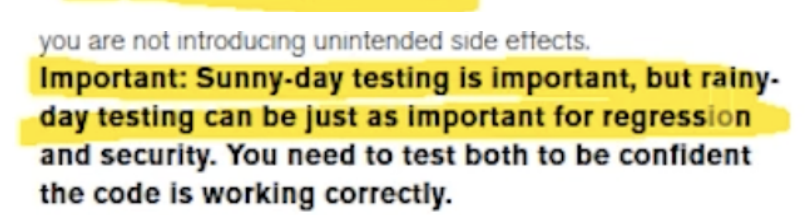
- Path Conditions
  - In addition to symbolically evaluating a program variable along a path, we can also symbolically represent the conditions which are required for that path to be traversed.
  - The symbolic path condition must be expressed in terms of the initial symbolic values of the variables
- Technique that will analyze the code in terms of symbolic values, give you the final value, final state in terms of initial values, helpful for identificationof oath condition
- make the determinations that a path is infeasible
- Issues
  - COmbinatorial Explosion, in terms of lots of paths
  - Symbolic execution with looping
  - symbolic execution in terms of dynamic variables and complex data structures
- Multiple Path Examples
-

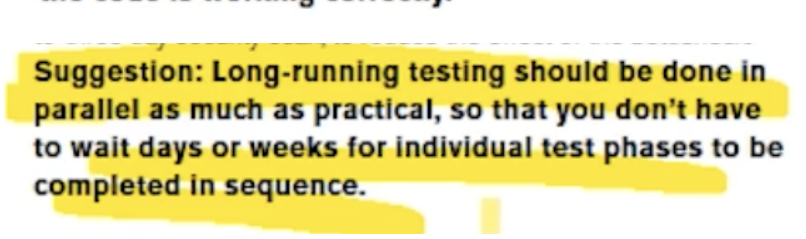
- DevOps advantages for testing

- Collaboration and integration, sw dev, operations personnel
- Agile dev
- Continuous Delivery (CD)
  - delivering sw in continuous manner
  - releases many times a day
  - a cd pipileine is inroduced
    - quality gates in there
    - all activities bw initial dev of sw and potentially its release
  - automation critical
  - start with least expensive issue
    - least expensive automated test first
    - then move to manual and exploratory testing
  - static analysis least costly technique
- 

- 

- 

- 

- 

-

Conclusion

The journey towards a continuous delivery practice relies heavily on quality tests to show if the software is (or is not) a viable candidate for production. But along with the increased reliance on testing, there are many opportunities for performing additional tests and additional types of tests to help build confidence in the software. By taking advantage of the automated tests and automated deployments, the quality of the software can be evaluated and verified more often and more completely. By arranging the least expensive tests (in terms of time, resources, and/or effort) first, a rapid feedback loop creates openings to fix issues sooner and focus more expensive testing efforts on software that you have more confidence in. By having a better understanding of the software quality, the business can make more informed decisions about releasing the software, which is ultimately one of the primary goals of DevOps.

-
-