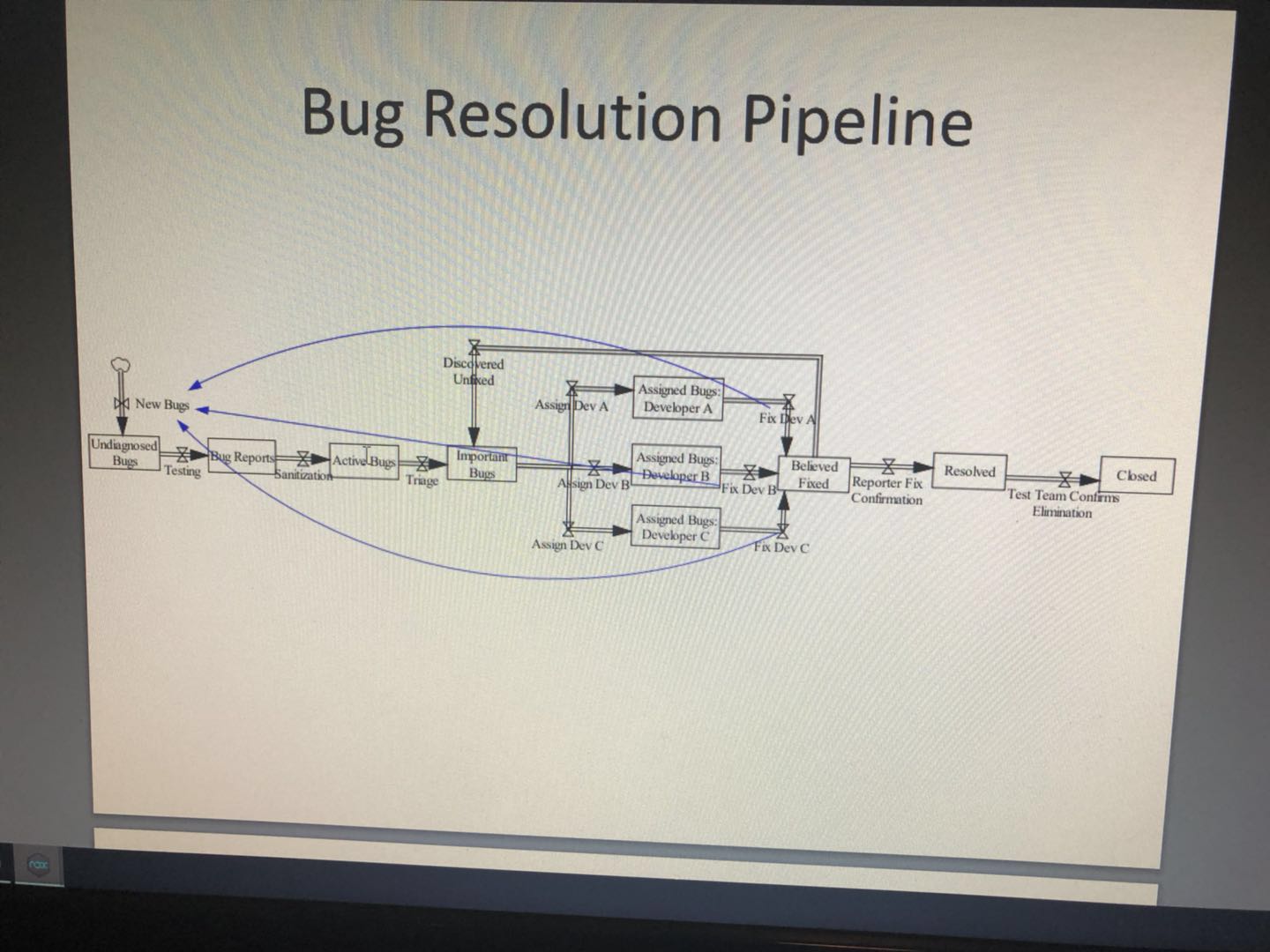
Elements of Test Design

1. Regression Testing
   1. Regression is when a change breaks a feature or operation that previously worked – or failed to fix a previous bug
   2. A regression test gap is the gap in testing between
      1. The current test set
      2. The entire test suite that has ever been run
   3. When we have a regression test gap, these is a serious risk of regressions
   4. Different costs if detected internally/externally
2. Varying Severity
   1. New feature
      1. Detected by testing release cand
         1. Potential cost/delay to fix
      2. Detected by Post-Release user
3. Bug resolution Pipeline 
4. High Risk of bug fixes
   1. Modification to less and equal 10 statements: 50% bug fixes work
   2. Modification to around 50 statements: 20% bug fixes work
   3. Reasons for failed fixed
      1. Problems with logical reasoning during bug fix and failed to check or incorrectly checked
      2. Misunderstanding of bug report
      3. Miscalled when claimed that fix
      4. Wrote specialized “fixes”
5. Two Notions of “regression testing”
   1. Often term “regression tests” involves both
      1. Verifying that a fix actually fixed the targeted bug
      2. Verifying that new development/bug fix didn’t break features that were working earlier
         1. Often earlier bugs reemerge. Why?
            1. Involve tricky parts of design
            2. Same misunderstanding happens twice
            3. Inadequate propagation of information about a change to the system/design between developers
            4. Merge conflicts lead to overwrite with earlier code
            5. Accidental recycling of earlier code
6. Common Regression Test Patterns
   1. Testing that all previously found and closed defects are no longer present
   2. Test similar bugs (detect overly specialized fixed)
   3. Test other feature (may have broken other features)
      1. Generally not time to test all: test rotating subset
7. Testing Perspectives
   1. Behavioral (black box, functional); how well does design match requirements?
      1. See how program reacts – from user’s perspective
      2. Requires domain knowledge
      3. Often quite specialized
         1. Security testing
         2. Performance testing
      4. Can plan once requirements complete
   2. Structural (Glass/White box, structural): how well does implementation match design?
      1. Most techniques domain independent
   3. Middle: Gray Box (partial implement, knowledge)
8. Black Box: Benefits
   1. Can plan once requirements complete
   2. Can enlist
      1. Users
      2. Less computer-trained individuals
   3. Can focus on functionality to be exercised by common user tasks
9. Creation of Test Cases for Black Box Testing: Techniques
   1. More systematic/Structured
      1. Equivalence partitioning/boundary value testing
      2. Decision tables/tree
      3. Orthogonal Arrays
      4. State transition diagrams
      5. Pareto analysis (focus where has failed in past)
   2. Less Structured
      1. Random testing
         1. Fully
         2. Semi-random
      2. Exploratory (adaptive)
         1. Adaptive planning of next steps
10. Glass-Box Tests
    1. Can help more quickly identify implementation errors than would be possible via the UI
    2. Focused tests on
       1. Problematic modules
       2. Less covered area
       3. Corner cases between actions (off-by-one errors)
    3. Algorithm specific testing
    4. Examples:
       1. Path/logic/call graph coverage
       2. Observing partially computed values
       3. Data and control-flow testing
       4. Confirming proper clean-up of resources under exceptional conditions
       5. Confirming integrity of data structure
11. Equivalence Classes and boundary value testing
    1. Characterize into equivalence classes
    2. Select representative from each equivalence class
    3. Problems: sometimes many semi-orthogonal equivalence classes across categories
       1. Consider day/moths/years
    4. Be sure to consider both valid and invalid classes
12. Examples for square root function
    1. Integers
       1. Perfect squares
       2. Others
       3. 1
       4. 0
    2. non-integers
       1. >1
       2. <1
    3. Values <0
13. Hints as to same equivalence class
    1. Test same condition
    2. If one test finds bug, others will too
    3. Commonalities by which to group things
       1. Maintain the value of predicates
       2. Involve or consider same input cause similar internal operations
       3. Yield same events
       4. Are on the same side of correct value
       5. Depend on same external environment
       6. Affect identical output
       7. Force same error conditions
14. Tips
    1. Consider looking for equivalence classes for each input field on a form
    2. Consider equivalence classes for invalid input
    3. Organize classifications in a table or outline
    4. Look for ranges of values
    5. Look for group membership
    6. Look for groups of variables calculating to the same value of range
    7. Try to identify identical output event
15. Boundary Value and Equivalence Class Testing
    1. Boundary value testing and equivalence class testing complement each other
    2. Reasoning about the equivalence classes points directly to boundary value
       1. This is particularly helpful when there are irregular equivalence classes
16. Orthogonal Arrays (“Pair-wise testing”, “All Pair testing”)
    1. Key characteristic: every pair of variable values occurs at least once (but not all combinations)
    2. Exploits fact that compatibility problems typically are pairwise
    3. Saves hugely over combinatorial test