# Regression Testing

## Reading Assignment

- R. K. Doong and P. G. Frankl, "The ASTOOT Approach to Testing Object-Oriented Programs," ACM Transactions on Software Engineering and Methodology, 3 (2), April 1994, pp. 101-130.

# Clarification on JUnit

- Unit testing approach
- Can use any unit test data selection technique
  - Structural
  - Functional
- Encode the test cases so they are easy to rerun
  - Framework for rerunning test cases and evaluating the results
- Unit based regresson testing approach
  - Select all test cases

# Why is regression testing a problem?

- Large systems can take a long time to retest
  - e.g., 6 months of regression testing before every release
- Sometimes it is difficult and time consuming to create the tests
- Sometimes it is difficult and time consuming to evaluate the tests
  - e.g., sometimes requires a person in the loop (GUI and simulation examples) to create and evaluate the results
- Cost of testing can prevent software improvements

# Regression Testing

- Primarily selecting from existing test data

- Plus, adding some new test cases
- Perhaps, deleting or updating some old test cases

# Regression Testing

- retest software after it has been modified
- trying to instill confidence that changes are correct
  - new functionality and corrected functionality behave as they should
  - unchanged functionality is indeed unchanged

# Automated support for Regression testing

- Test environment or infrastructure support
  - Capture and replay
- Test data selection support
  - Select a subset of the existing test cases
  - Select test data to exercise new functionality

# Test environment or infrastructure support

- Automatic results comparison
  - Set of {test drivers, test stubs, {test cases, previous results}}
  - A□□utomatically collects and saves test cases and previous results
  - Reruns previous test cases and reports any discrepancies
- Often very effective
  - Saves considerable amount of time and human resources
  - Doesn't work very well for some applications

# Test environment or infrastructure support

- Doesn't work very well for GUI software
  - Specialized capture and replay tools
  - Capture all the keystrokes, window layout, buttons, etc.
  - Small changes in GUI tend to affect many outputs
- Doesn't work well for complex simulation software/hardware
  - Many displays, coordination among displays
  - Analog results within acceptable ranges
  - Use scenario books and "person in the loop"

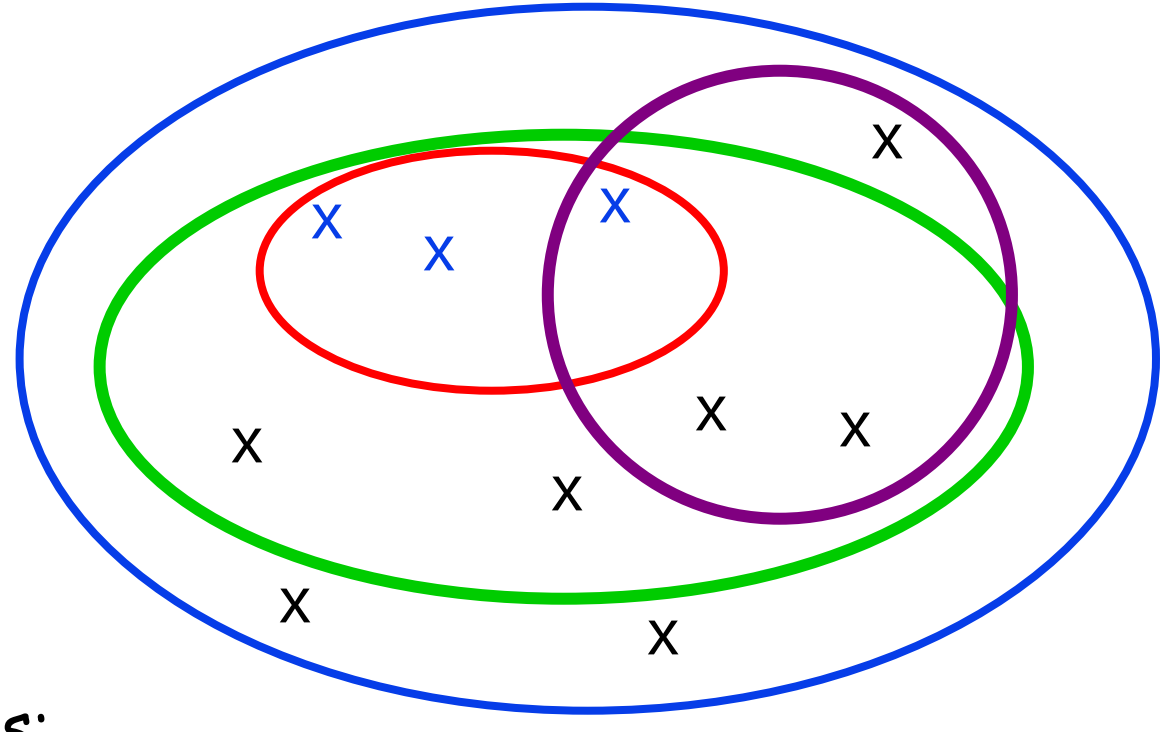
# Test data selection support

- Approaches
  - retest all- chooses all existing test cases
  - selective retest - chooses a subset of the old test cases
    - But, what criterion should we use for selecting test cases?

# Selective retest goals

- Use a selection criterion that selects effective test cases
  - Select test cases that will reveal faults
  - Select test cases that could reveal faults
  - Remove test cases that could not reveal faults
  - Do not remove test cases that will reveal faults
- Use a selection criterion that is not too costly to compute

# Retest selection




Some Alternatives:

Only reveal faults (Conservative and Precise)
Will reveal faults (Conservative, but not precise)

Retest all:
Will reveal faults (Conservative, but not precise )

## Steps in regression testing

- Given: a program P originally tested with test set T producing results R and a modified version of the program P'
- Identify the changes to P
- Select T' a subset of T to re-execute P'
- Test P' with T' and reestablish correctness of P' with respect to T'
- Create new tests T'' as necessary

# Steps in regression testing (continued)

- Create a new test history consisting of the results from executing
  - T' the revised test history
    - May need to update expected results
  - T'' the newly expanded test history

  - If the selection criterion is conservative, then (T - T') is part of the old test history that is still valid
    - Some test cases may become obsolete if the domain changes

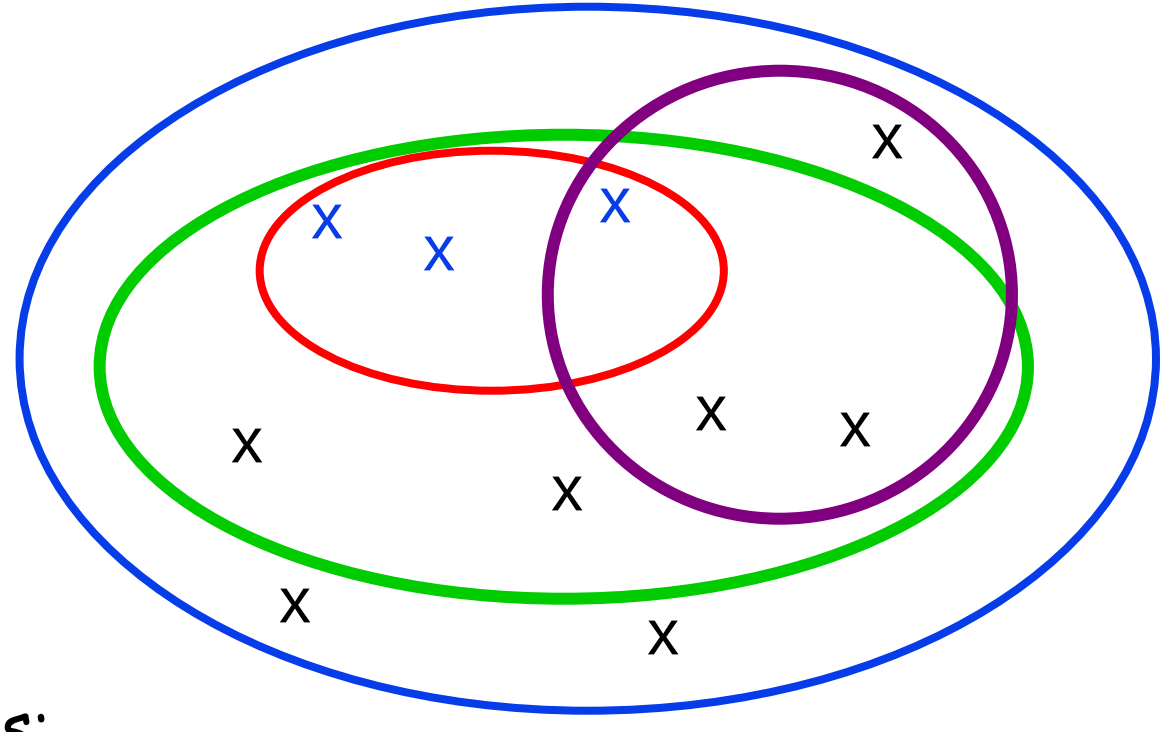
# Analyzing Regression Testing Selection Techniques

- Rothermel and Harrold
- Developed a framework for analytically comparing regression testing techniques
- Later used this framework to experimentally compare regression testing techniques

# Criteria for evaluating selective retesting

- **Inclusiveness** - measures the extent to which a method chooses tests that will cause the modified program to produce different results
  - if it can be demonstrated that a change will not impact the results of a test case then there is no reason to retest with that test case
  - e.g., no syntactic dependence
- **Precision** - measures the ability of a method to avoid choosing test cases that will not cause the modified program to produce different results
  - e.g., including all test cases is inclusive but not precise

# Retest selection




Some Alternatives:

Only reveal faults (Conservative and Precise)
Will reveal faults (Conservative, but not precise)

Retest all:
Will reveal faults (Conservative, but not precise )

# Criteria for evaluating selective retesting

- **Efficiency** - measures the computational cost of a method
- **Generality** - measures the ability of a method to handle realistic and diverse language constructs and modifications
- **Accountability** - measures the ability to support some testing criteria (e.g., coverage)

# Program dependence must consider

- modified statements
- new statements
- deleted statements

- All of the above can change program dependences
- What program dependence information do we need?

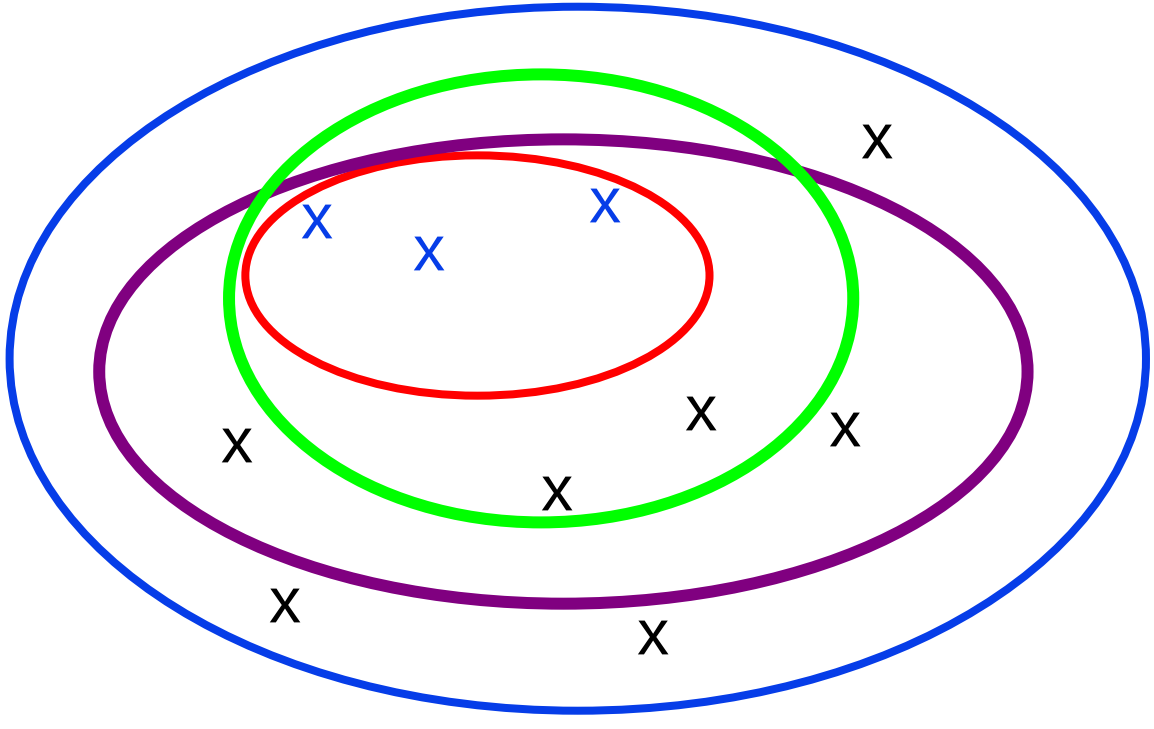
# Measuring Inclusiveness

- Suppose T contains n modification revealing test cases and a method selects m of them, then the inclusiveness of the method for P, P', and T is (m/n*100)%
- Not a good metric Why?
  - Depends on the program and the test set
  - What if n= 0?
- Better to have a metric that is independent of a particular program or test suite

# Modification-traversing

- a test case t ϵT is fault-revealing if it produces incorrect outputs for P'
  - In general, can not determine which elements of T are fault revealing
- a test case t ϵT is modification-revealing if it produces different outputs for P than for P'
  - Modification-revealing test cases over-approximates the fault revealing test cases
  - In general, can not determine which elements of T are modification revealing
- a test case t ϵT is modification-traversing if it executes a statement in P' that has changed
  - Modification-traversing over-approximates modification revealing
  - Can be computed

# Retest selection



Some Alternatives:

Fault revealing (Conservative and Precise)

-impossible to compute

Modification revealing (Conservative, but not precise)

-hard to compute

Traversal revealing -easier to compute

Retest all (Conservative, but not precise )- trivial to compute

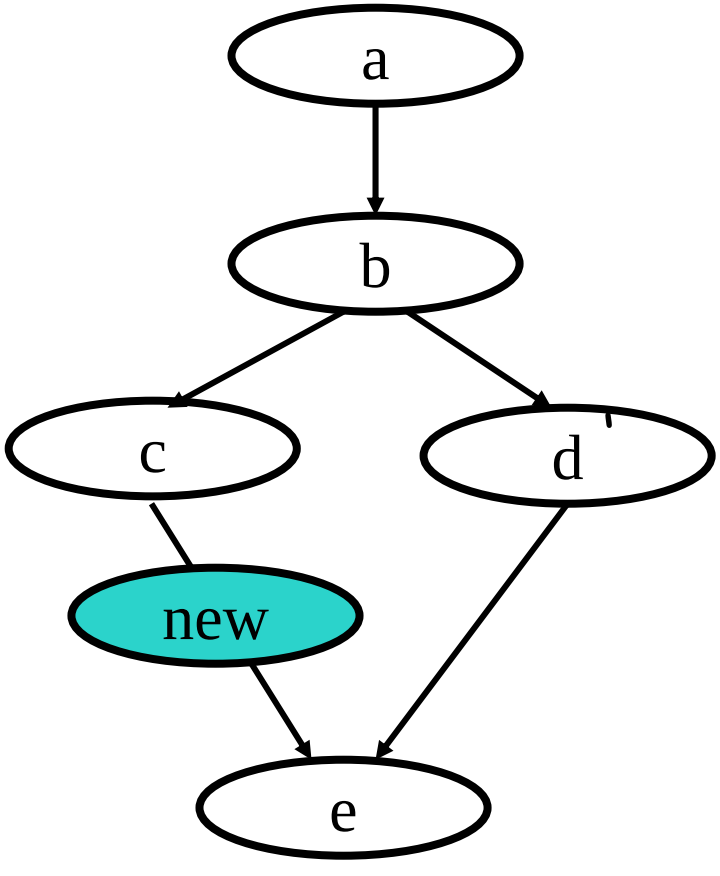
# Four Different Major Approaches

- Minimization techniques
  - Select at least one test case that exercises the components that have changed
- Coverage techniques
  - Select test cases that exercise components that have changed and that satisfy some coverage criteria
- Safe techniques
  - Select every test case that could expose one or more faults
  - Can only eliminate a test case if it can be proven that it cannot expose a fault

# Four Different Major Approaches (continued)

- Data Flow based
  - Modification-Need to select all the test cases that exercised a statement that was modified
  - Deletion-Need to exercise all the test cases that exercised a deleted statement
  - New-need to exercise all the test cases that
    - exercised a statement that is now directly data or control dependent on a new statement, or
    - exercised a statements that the new statement is now directly data or control dependent on

# Modifications, deletions, additions




- Modification-Need to select all the test cases that exercised a statement that was modified
- Deletion-Need to exercise all the test cases that exercised a deleted statement
- New-need to exercise all the test cases that exercised a statement that is directly data or control dependent on the new statement (e.g., e) or the new statement is directly data or control dependent on it (e.g., c)

# An Empirical Study

- Graves, Harrold, Kim, Porter, and Rothermel, TOSEM April 2001
- Experiment to evaluate
  - Fault detection effectiveness
    - Regression testing is usually not more effective than the original test set
    - Retest-all has good fault detection effectiveness, but may not be cost effective
  - Cost effectiveness
    - Are there techniques that have the same fault detection effectiveness but the cost of the analysis is significantly less than the test cases it eliminates
    - Cost to compute T' versus the cost of executing (T-T')

# Program studied

- 7 C++ programs from Siemens
  - 138- 516 LOCs
  - Many versions of each
    - 9-41 versions
    - Each version had one seeded fault
- 2 larger programs
  - 6 Klocs/ 33 versions/multiple faults
  - 49 Klocs/ 5 versions/ multiple faults

# Test pools, test suites, test cases

- Test pools
  - Test cases with known edge coverage
- 1000 edge-coverage test suites selected from the pool randomly
    - Selected test cases to achieve edge-coverage
      - Assume $n_k$ test cases needed for the kth suite
- 1000 non-edge coverage test suites
  - Selected randomly from the pool
  - Kth test suite has $n_k$ test cases, so non-edge coverage has a "buddy" edge-coverage test suite

# Regression testing techniques studied

- Minimization - test cases from a suite that covers the edges associated with changes or nodes that have changed
  - often resulted in a single test case

- Safe - every test case in a suite that exercises a statement that has been deleted or was modified or is new
  - How do we know if a test case exercises a new statement?

# Regression testing techniques studied (continued)

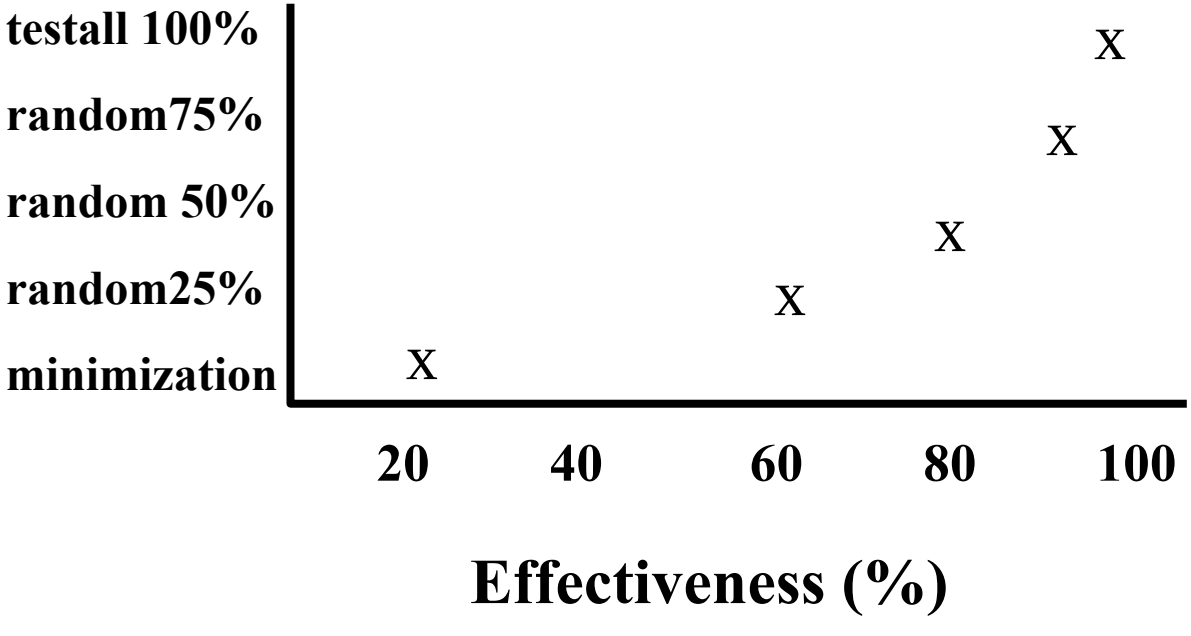
- Data flow-every test case in a test suite that exercises a def-use pair affected by a deleted or modified statement
  - Not quite safe
  - Not full dependence
- Random-select 25%/50%/75% of the test cases in a suite chosen randomly
- Retest-all

# Test case size reduction

- Random and test-all select a test suite size that is 25%, 50%, 75%, and 100%, respectively, of T by definition
- Minimization: ~1% test suite size
- Safe:60% test suite size
- Data flow: 54% test suite size

# Fault detection effectiveness

- For minimization, random, and test-all
  - The larger the test suite size the better the fault detection
  - Improvement diminishes as the % gets higher



# Fault detection effectiveness : safe versus data flow

- Safe and data flow had about the same effectiveness
  - Data flow is slightly less
  - Safe always found all the faults that test-all found (by design)
- Data flow costs more than safe to perform so there is no gain in using data flow over safe

# Fault detection effectiveness : safe versus random

- Safe test suite size averaged 60% of original, but only performed slightly better than random(75%)
- There was significant variance in the test suite reduction
  - Some programs resulted in almost no reduction in original test suite size
  - Larger programs tended to have a larger reduction in the test suite size
    - for some programs the payoff was significant
    - best case: 5% of the test cases were required

# Summary of regression testing experiment

- If fault detection is not paramount, then random (75%) might be a reasonable compromise.
  - Often as good as safe but not as expensive
  - 75% is often too many test cases

# Summary of regression testing experiment

- If fault detection is important then need to choose a safe technique
  - Test-all is a safe technique, but expensive
  - Selective safe techniques might be worth the cost, but
    - need more experimental data
    - need better selective techniques
    - When many test cases are selected at least know they are needed (within our ability to determine this effectively)
      - Modification traversing, not modification revealing nor fault revealing

# Another experiment: TestTube

- Chen, Rosenblum, Vo
- Experimental study carried out at AT&T, 1994
- Coarse-grain regression testing for C
  - Applicable to large systems

# TestTube approach

- Monitors the entities each test case exercises
  - Dynamic slice/dependence
- Identifies changed entities
- Selects those test cases that exercised at least one of the changed entities
- Hypothesis: Works well if entities are coarse-grained
  - Entities: Functions, types, variables, macros

# Defining entities

- For a test case t can determine the set of functions $t_f$ that are exercised and the set of non-functions $t_v$ that affect those functions
  - Functions--can determine effect by monitoring
  - Non-functions--non-executing entities: declarations, macros,etc.
    - Can be computed using static analysis
    - Difficult for languages such as *C* and *C++*
      - Must deal with aliasing problems

## Proposition

- Let t be a test case for Program P. If the changes to P do not affect any of the entities in $t_f$ or $t_v$ then there is no need to test t

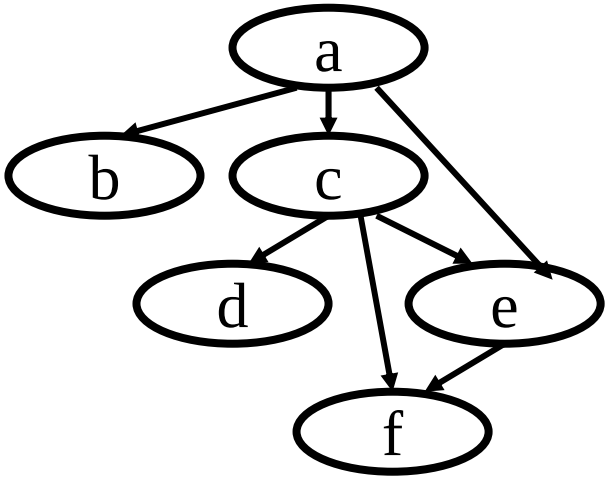
# Experimental results: SFIO

- SFIO(Safe, Fast Input/Output)
  - 11,000 LOCS
  - 706 entities: 481 macros, 175 function defs, 18 defs of global variables, 32 type defs
  - 39 test cases
- Observations:
  - Reduction in test case size depended on the change
  - Those components that were least likely to change required the most test cases
    - Core functionality versus feature functionality

# Experimental results: Incl

- Incl
  - 1700 LOCS
  - 91 entities: 37 macros, 32 function defs, 14 defs of global variables, 8 type defs
  - 8 test cases
- Observations:
  - Version 2 only required 4 test cases

# Results depended on where a change was made




- Changes to leaf components required few test cases
- Changes to root node required all test cases
- Less likely to change the root

# Summary comments on TestTube

- Demonstrated that a relatively simple technique could be effective
  - Overhead was reasonable
  - Claimed to be safe but didn't deal with some of the harder issues
    - Testing additions to the code

# Another experiment

- Where the Bugs Are

  Thomas Ostrand, Elaine Weyuker, and Robert Bell
  AT&T

- 2004
- Provides information about two "real" systems from AT&T

# Contributions

- Studied release data over several years
  - 2 systems
  - Inventory control, 17 releases over 4 years, full lifecycle info
  - Provisioning System, 9 releases, over 2 years, post unit testing info only
- Used the data to develop a model to predict where the errors would be in a new release
- Could predict which 20% of the files would have 80% of the bugs
- Used a simply prediction model (*LOC*) and still could predict where 75% of the bugs would be found

# Observations

- Could not use the error severity levels for predictions because they were unreliable
  - Value was often political, not technical
- About 3/4 of the faults were found during unit testing
- LOCS & # files increased with each release
- Fault density (faults/KLOCS) decreased with each release

# Prediction Model

- LOC and the file's change status were the strongest individual predictors in the model
- Number of changes since the last release did not improve the model

| Rel | Number of Files | Lines of Code | Mean LOC | Faults Detected | Fault Density | Files Containing Any Faults |
|---|---|---|---|---|---|---|
| 1 | 584 | 145,967 | 250 | 990 | 6.78 | 233 |
| 2 | 567 | 154,381 | 272 | 201 | 1.30 | 88 |
| 3 | 706 | 190,596 | 270 | 487 | 2.56 | 140 |
| 4 | 743 | 203,233 | 274 | 328 | 1.61 | 114 |
| 5 | 804 | 231,968 | 289 | 340 | 1.47 | 131 |
| 6 | 867 | 253,870 | 293 | 339 | 1.34 | 115 |
| 7 | 993 | 291,719 | 294 | 207 | 0.71 | 106 |
| 8 | 1197 | 338,774 | 283 | 490 | 1.45 | 148 |
| 9 | 1321 | 377,198 | 286 | 436 | 1.16 | 151 |
| 10 | 1372 | 396,209 | 289 | 246 | 0.62 | 112 |
| 11 | 1607 | 426,878 | 266 | 281 | 0.66 | 114 |
| 12 | 1740 | 476,215 | 274 | 273 | 0.57 | 120 |
| 13 | 1772 | 460,437 | 260 | 127 | 0.28 | 71 |
| 14 | 1877 | 482,435 | 257 | 235 | 0.49 | 95 |
| 15 | 1728 | 479,818 | 278 | 305 | 0.64 | 120 |
| 16 | 1847 | 510,561 | 276 | 274 | 0.54 | 116 |
| 17 | 1950 | 538,487 | 276 | 253 | 0.47 | 122 |

**Table 1. Inventory System Information**

# File Prediction => 80% of the faults

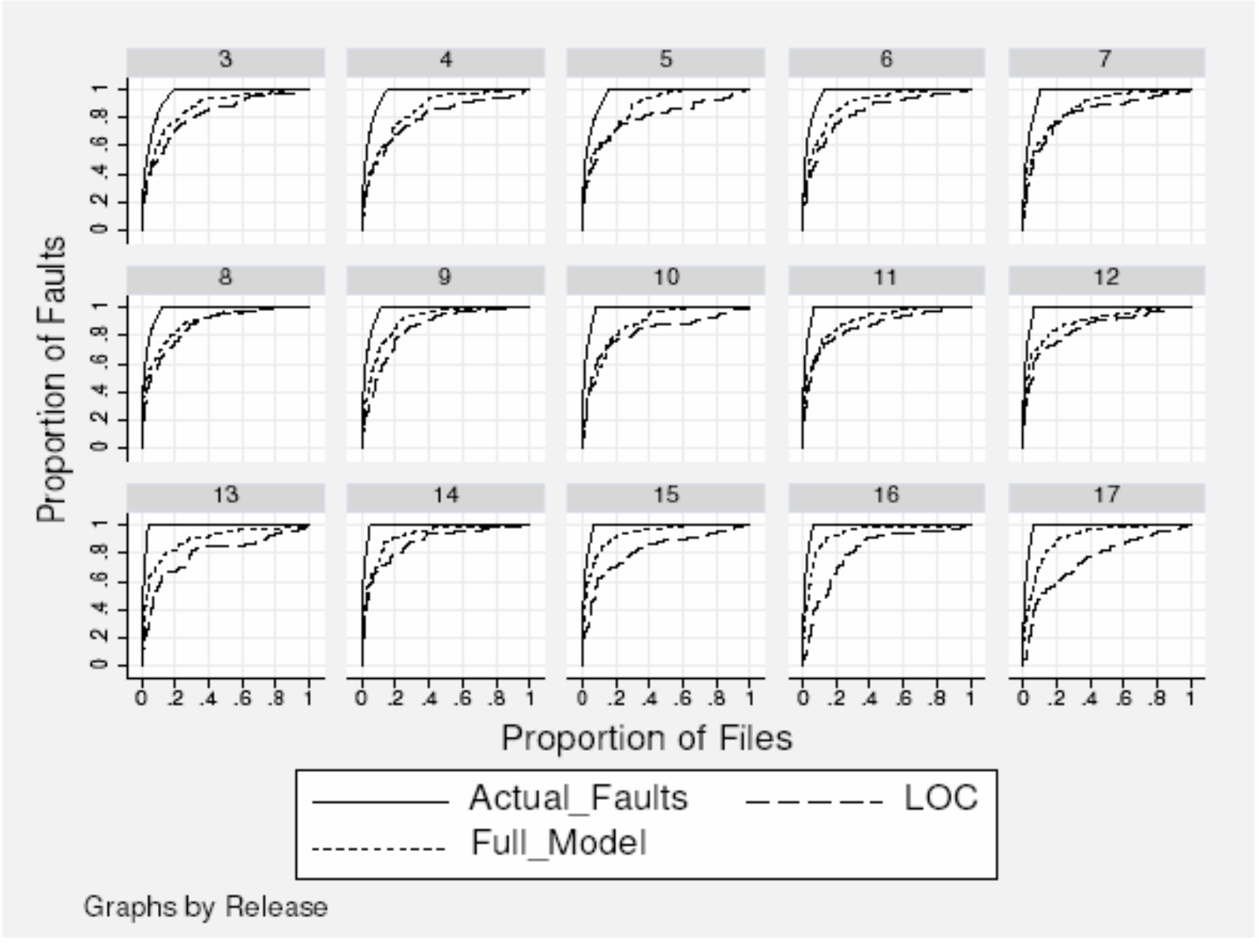
| Release | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | Avg 3-12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| % Faults Identified | 77 | 74 | 71 | 85 | 77 | 81 | 85 | 78 | 84 | 84 | 80 |

Table 2. Percentage of Faults Included in the 20% of the Files Selected by the Model - Releases 3-12

| Release | 13 | 14 | 15 | 16 | 17 | Avg 13-17 |
|---|---|---|---|---|---|---|
| % Faults Identified | 82 | 91 | 92 | 92 | 88 | 89 |

Table 3. Percentage of Faults Included in the 20% of the Files Selected by the Model - Releases 13-17

# LOC almost as good a predictor as the full model



# Prediction only using faults found after unit testing

| Release | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | Avg 3-17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| % Faults Identified - ALL | 77 | 74 | 71 | 85 | 77 | 81 | 85 | 78 | 84 | 84 | 82 | 91 | 92 | 92 | 88 | 83 |
| % Faults Identified - PUT | 80 | 71 | 73 | 90 | 81 | 83 | 81 | 81 | 83 | 90 | 90 | 93 | 85 | 88 | 86 | 84 |

Table 4. Percentage of Post-Unit Test Faults Included in the 20% of the Files Selected by the Model

# Provisioning System

| Aggregated Rel Name | Rel No. | Number of Files | Lines of Code | Mean LOC | Faults Detected | Fault Density | Files Containing Any Faults | Pct Containing Any Faults |
|---|---|---|---|---|---|---|---|---|
| A | 1 | 2008 | 381,973 | 190 | 24 | 0.06 | 19 | 0.9 |
| B | 2 | 2085 | 397,683 | 191 | 85 | 0.21 | 63 | 3.0 |
| | 3 | 2137 | 412,621 | 193 | 52 | 0.13 | 41 | 1.9 |
| | 4 | 2104 | 406,674 | 193 | 10 | 0.02 | 9 | 0.4 |
| | 5 | 2119 | 407,724 | 192 | 6 | 0.01 | 6 | 0.3 |
| C | 6 | 2213 | 423,895 | 192 | 15 | 0.04 | 14 | 0.6 |
| | 7 | 2250 | 434,772 | 193 | 74 | 0.17 | 64 | 2.8 |
| | 8 | 2230 | 434,781 | 195 | 34 | 0.08 | 30 | 1.3 |
| | 9 | 2241 | 437,578 | 195 | 7 | 0.02 | 6 | 0.3 |

Table 5. Provisioning System Information

# Prediction by Model

| Release | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | Avg 3-17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| % Faults Identified | 71 | 68 | 71 | 75 | 76 | 76 | 77 | 77 | 80 | 75 | 68 | 80 | 71 | 67 | 58 | 73 |

Table 6. Percentage of Faults Included in the 20% of the Files Selected by LOC

# Conclusion from Study

- Applied to two programs with similar results
  - Appears to generalize
- Post-unit testing information was adequate for prediction
- LOC and change info was not as good as the more complicated prediction algorithm but it was pretty good

- Remember: Not a safe technique

# Regression testing Conclusion

- Regression testing a problem for some programs
  - Want to reduce test suite size but not fault detection
  - Need prediction models to select test cases
- Prediction models
  - Safe selection techniques might still return too many test cases
    - Need to combine with priority techniques
  - Simple selection techniques, such as LOC and change info might be sufficient

# Suggested regression testing process

- Determine the set of safe test cases
  - This can be run as a background job
  - Cost is basically irrelevant
- Select from these safe test cases
  - % selected depends on resources available
  - Rerun selected test cases
    - Cost of executing test cases
    - Cost of evaluating the results
      - Can often be automated
- Select new test cases to exercise new functionality

# How to select a subset of the safe test cases?

- Priority
  - LOCs and exposed faults previously
  - Exercised components that were faulty
  - Executes changed components that have not been re-executed yet?
    - Coverage criteria
- Recent work by Gregg Rothermel evaluates priority selection
- Regression testing-- Real and important problem