EECS 4313

Software Engineering Testing

Topic 14:

Empirical Studies in Software Testing

Zhen Ming (Jack) Jiang

Empirical Studies

- The word “**empirical”** means information gained by experience, observation, or experiment. The central theme in scientific method is that all evidence must be **empirical** which means it is based on evidence. In scientific method the word "empirical" refers to the use of working hypothesis that can be tested using observation and experiment.
  - Empirical research can be defined as "research based on experimentation or observation (evidence)". Such research is conducted to test a hypothesis.
  - Empirical studies (use of experience, observation) have become important for software engineering research.

Empirical Software Engineering

- **Empirical software engineering** is a field of research that emphasize the use of empirical studies of all kinds to accumulate knowledge.
  - Test theories
  - Evaluate new process and tools
- Approaches
  - **Survey**: interviews or questionnaires
  - **Controlled Experiment**: in the laboratory, involves manipulation of variables
  - **Case Study**: observational, often in-situ

Empirical Study Approaches - Surveys

- Pose questions via interviews or questionnaires
- Process: select variables and choose sample, frame questions that relate to variables, collect data, analyze and generalize from data
- Uses: descriptive (assert characteristics), explanatory (assess why), exploratory (pre-study)

Resource: E. Babbie, *Survey Research Methods*, **Wadsworth, 1990**

Empirical Study Approaches - Controlled Experiments

- Manipulate independent variables and measure effects on dependent variables
- Requires randomization over subjects and objects (partial exception: quasi-experiments)
- Relies on controlled environment (fix or sample over factors not being manipulated)
- Often involves a baseline (control group)
- Supports use of statistical analyses

Resource: Wohlin et al., *Experimentation in Software Engineering*, Kluwer, 2000

Empirical Study Approaches - Case Studies

- Study a phenomenon (process, technique, device) in a specific setting
- Can involve comparisons between projects
- Less control, randomization, and replicability
- Easier to plan than controlled experiments
- Uses include larger investigations such as longitudinal or industrial

Resource: R. K. Yin, *Case Study Research Design and Methods*, Sage Publications, 1994

Empirical Approaches: Comparison

Factor	Survey	Experiment	Case Study
Execution Control	Low	High	Low
Measurement Control	Low	High	High
Investigation Cost	Low	High	Medium
Ease of Replication	High	High	Low

Problems for Empiricism

- Threats to validity: factors that limit our ability to draw valid conclusions
- Three types of threats
  - External Validity: ability to generalize the results
  - Internal Validity: concerns the impact of confounding factors on the results of study.
  - Construct Validity: concerns about the impact of measurement to the results of the study.

Examples of External Validity

- Subjects (participants) aren’t representative
- Programs (objects) aren’t representative
- Environments aren’t representative

Examples of Internal Validity

- Non-homogeneity among groups (different in experience, training, motivation)
  - E.g., most of the highly experienced developers also received lots of training

Examples of Construct Validity

- Lines of code may not adequately represent amount of work done [measurement subject]
- Devices or measurement tools faulty
- The act of observing can change behavior (of users, certainly, but also of artifacts)

Coverage is not strongly correlated with test suite effectiveness (ICSE 2014)

The Limits of Software Testing

- Dijkstra: “Program Testing can be used to show the presence of defects, but never their absence”.
- It is impossible to fully test a software system in a reasonable amount of time or money

Test Suites and Code Coverage

- Software testing uses test suites to expose faults
- Code coverage (recommended by many textbooks) as one of the metrics for measuring the fault detection effectiveness of test suites
  - [Intuitively appealing] a test suite cannot find bugs in code where it never executes
- But what is the strength of code coverage and fault detection effectiveness?

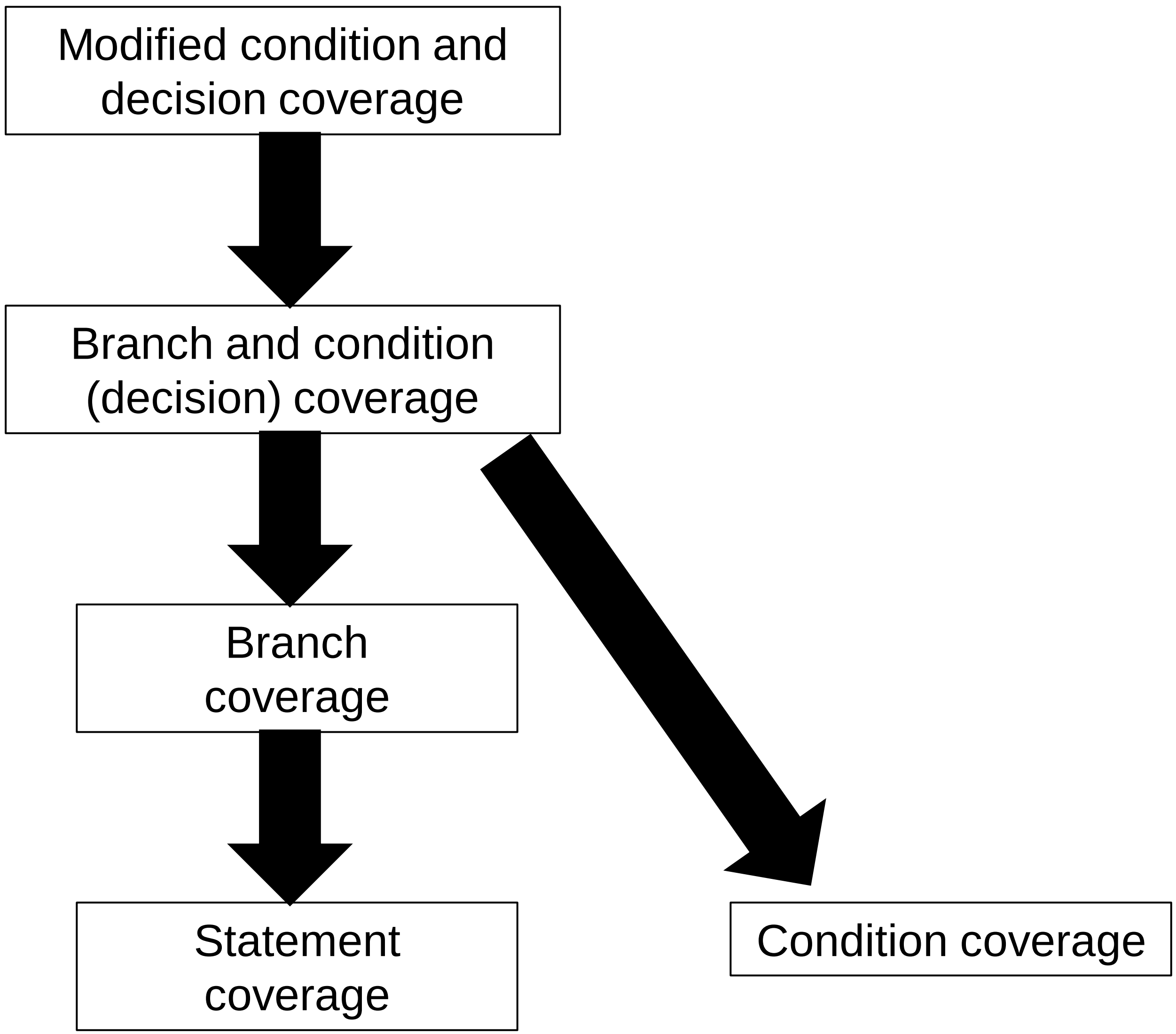
Goal of this empirical study

- An empirical study on the relationship between test suite size, code coverage and effectiveness in **Java** programs
- Test suite size
  - SLOC
  - # of test methods
- Code coverage metrics studied
  - Statement coverage,
  - decision coverage,
  - and modified condition coverage
- Analysis method
  - Statistical correlation

A recap on the code coverage metrics

- **Statement coverage** is achieved when all statements in a method have been executed at least once
- **Decision coverage** is computed by considering both branch and individual condition coverage measures
  - **Branch coverage** is achieved when every branch from a node is executed at least once
  - **Condition coverage** reports the true or false outcome of each condition.
- **Modified condition/decision coverage** extends branch and decision coverage with the requirement that each condition should affect the decision outcome independently

Test Coverage Criteria Subsumption



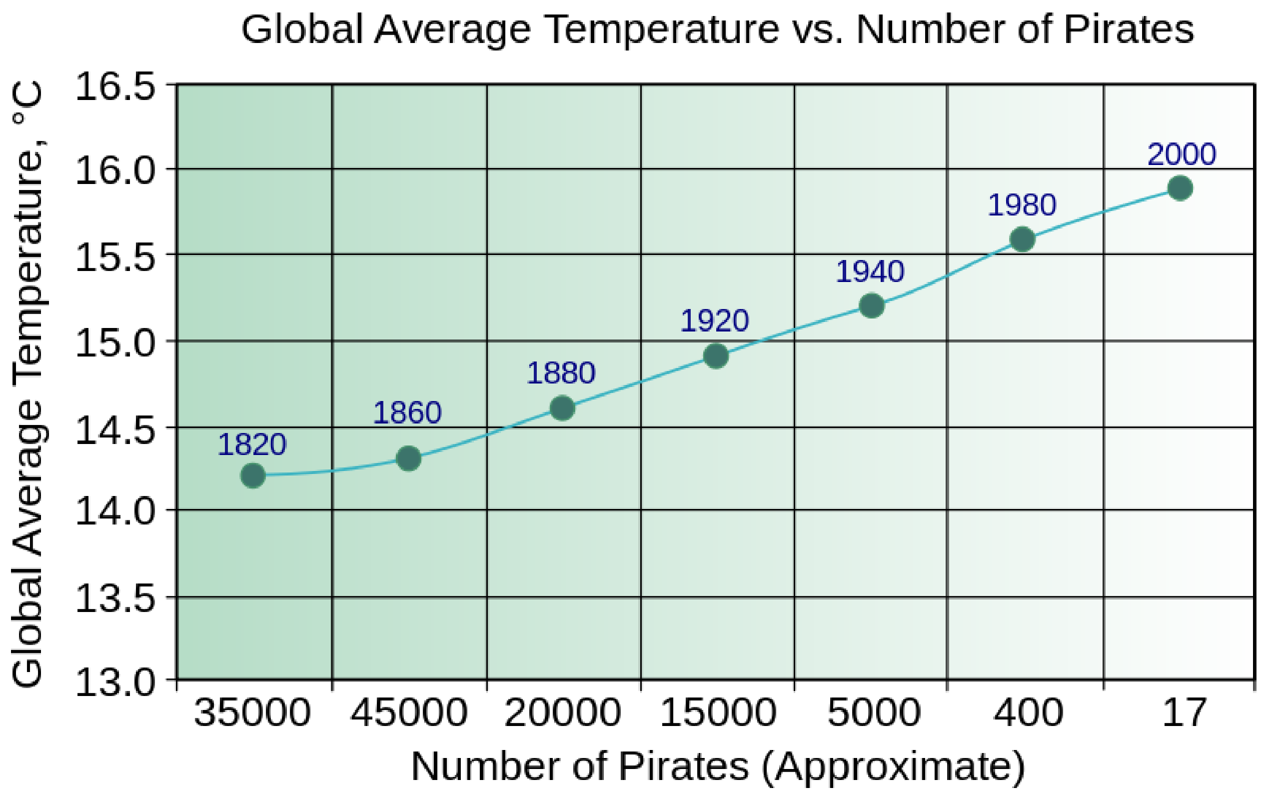
A recap on statistical correlation

- Correlation coefficients are used to describe relationships among quantitative variables.
- The sign ± indicates the direction of the relationship (positive or inverse), and the magnitude indicates the strength of the relationship (ranging from 0 for no relationship to 1 for a perfectly predictable relationship). The actual range varies from books to books:
  - No correlation
    - (-0.1, 0.1)
  - Weak correlation
    - (0.1, 0.3), (-0.3, -0.1)
  - Moderate correlation
    - (0.3, 0.5), (-0.5, -0.3)
  - Strong correlation
    - (0.5, 1), (-1, -0.5)

High correlation does not imply cause and effect

Correlation != Causation

- Does this mean pirates cause global warming or vice versa?



Study Design

1. Select a set of (Java) program to study
2. Make test suites
3. Measure test suite coverage
4. Measure suite effectiveness
   - Mutation testing
   - Representative of fault detection effectiveness

Subject programs

- Five open source Java programs
  1. Apache POI: API for manipulating Microsoft documents
  2. Closure: JavaScript optimizing compiler
  3. HSQLDB: relational database management system
  4. JFreeChart: library for producing charts
  5. Joda Time: open source replacement for Java Date and Time classes

Generating test suites

- Identify all the test methods in a program
- Generate new test suites of fixed size by randomly selecting a subset of these methods without replacement
- We run these test suites and measure the code coverage using the *CodeCover* tool

Mutation Testing

- Faults are introduced into the program by creating many versions of the program called **mutants**
- Each mutant contains a single fault
- Test cases are applied to the original program and to the mutant program
- The goal is to cause the mutant program to fail, thus demonstrating the effectiveness of the test suite
- Mutation testing is used to generate faulty programs in this study
- The mutation testing tool is *PIT*

Mutation Testing Algorithm

- Generate program test cases
- Run each test case against the original program
  - If the output is incorrect, the program must be modified and re-tested
  - If the output is correct go to the next step ...
- Construct mutants using a mutation testing tool
- Execute each test case against each alive mutant
  - If the output of the mutant differs from the output of the original program, the mutant is considered incorrect and is **killed**
    - **“Good test cases kill the mutants"**
  - Once we find a test case that kills a mutant, we can forget the mutant and keep the test case. The mutant is **dead**
- Two kinds of mutants survive:
  - **Functionally equivalent to the original program**: Cannot be killed
  - **Killable**: Test cases are insufficient to kill the mutant. New test cases must be created.

Mutation Coverage Criteria

- Mutation Coverage (MC)
  - For each mutant m, test requirements (TR) contain a requirement to “kill m”
    - Mutation score is the percentage of mutants killed
- The **mutation score** for a set of test cases is the percentage of non-equivalent mutants killed by the test data
  - Mutation Score = 100 * D / (N - E)
    - D: Dead mutants
    - N: Number of mutants
    - E: Number of equivalent mutants
  - A set of test cases is mutation adequate if its mutation score is 100%.

Findings

- There is a *low to moderate* correlation between code coverage metrics and test suite effectiveness
- If you code coverage is slow, the likelihood of exposing faults is low
  - Hence, code coverage is useful to identify under-tested parts of a program
- However, stronger coverage do not provide greater insights into the effectiveness of the test suites
  - Hence, code coverage should not be used as a quality target because it is not a good indicator of test suite effectiveness

Potential Threats

- What about other programs not written in Java?
- What about other coverage metrics (e.g., data flow or concurrency coverage)?
- It assumes any mutants that are not killed by the master suite (original test suites) are equivalent mutants
  - Overestimates the # of equivalent mutants
  - Scale to large size programs
- Are faults seeded in mutation testing representative of real faults?

Are mutants a valid substitute for real faults in software testing? (FSE 2014)