CSCC01 – Introduction to Software Engineering

Software Testing

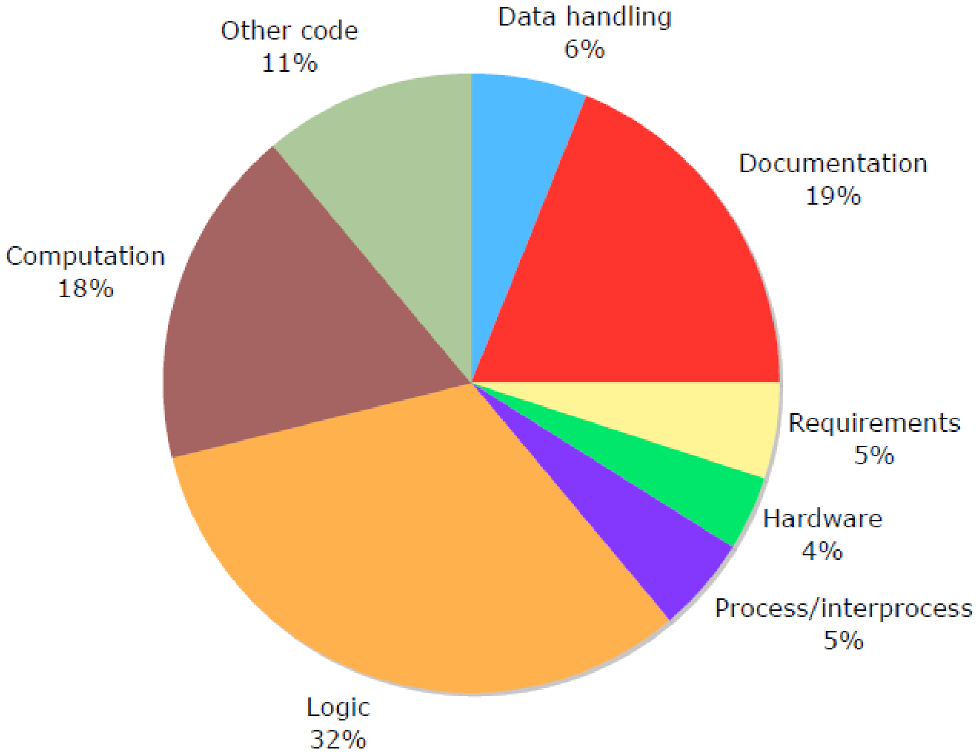
Winter 2024

What is Software Testing?

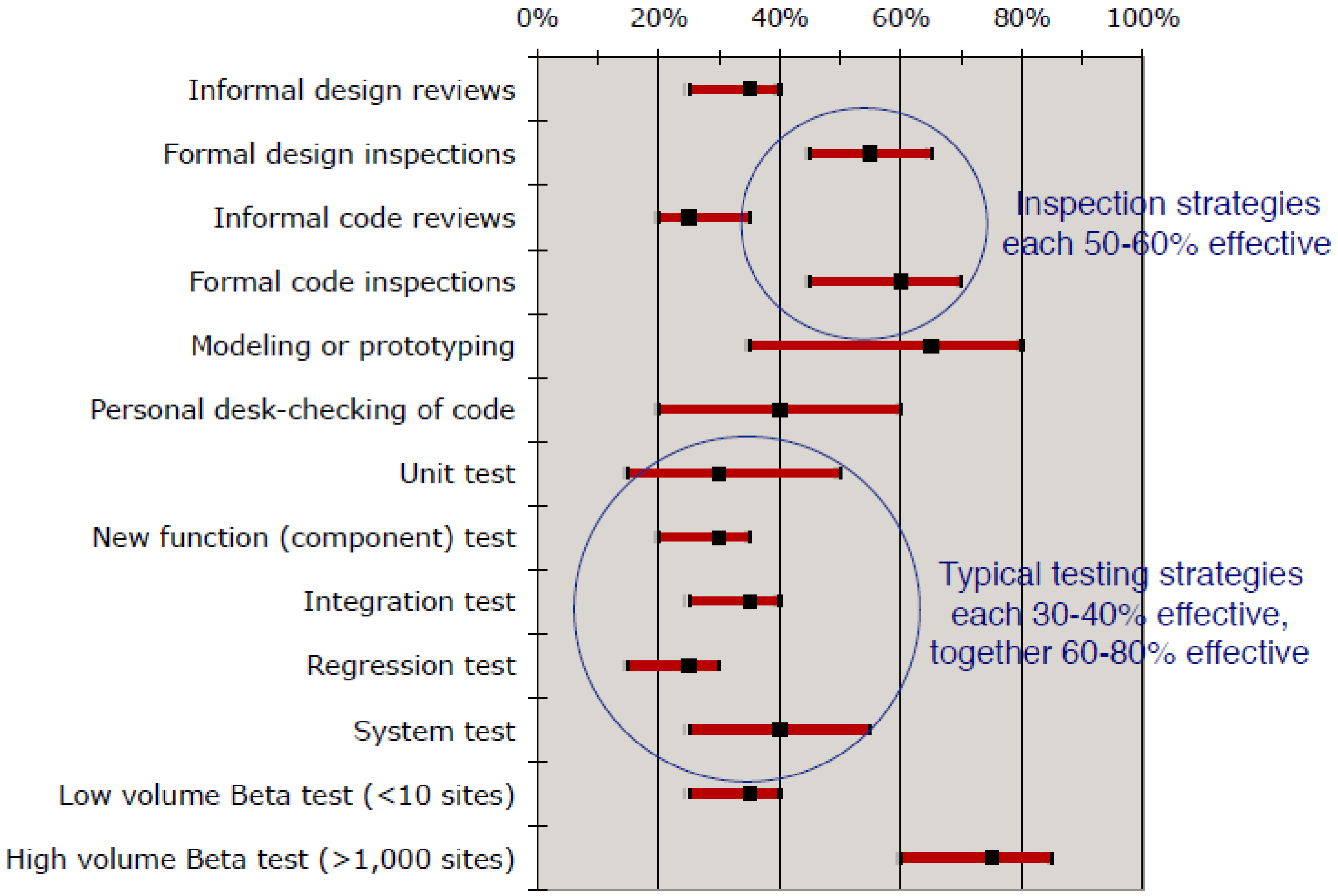
- Software testing involves running a program to detect defects
  - Cannot prove the absence of defects
- While not the only defect detection technique, software testing has the following advantages:
  - Can be automated to a significant extent
  - Supports objective evaluation of software quality
  - Helps detect defects that are related to complex runtime interactions

Defect Profiles

- E.g. Data from *Hewlett-Packard*



Defect Detection Effectiveness



McConnell, S. (2004), Code Complete, Second Edition , Microsoft Press.

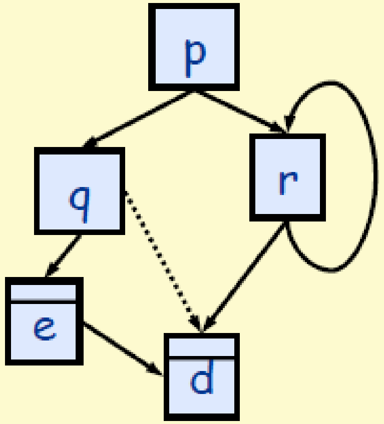
Testing Levels

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing

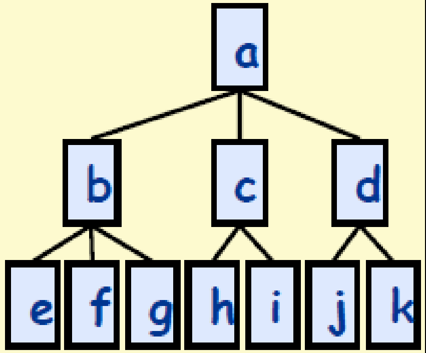
Integration Testing

- Units are tested together
- Two main approaches

Bottom up
for this dependency graph, test order is:
1) d
2) e and r
3) q
4) p



Top down
for this structure chart the order is:
1) test a with stubs for b, c, and d
2) test a+b+c+d with stubs for e…k
3) test whole system



Testing Strategies

- Structural Coverage Strategies (White box testing)
  - Statement Coverage
  - Branch Coverage
  - Definition-use Coverage
  - Logic Coverage
- Function Coverage Strategies (Black box testing)
  - Use Cases as Test Cases
  - Testing with good and bad data
- Stress Testing
  - Quick Testing
  - Interference Testing

White box Testing

- Structural coverage can be used to quantify the thoroughness of testing
- Complex coverage types may be more effective at detecting defects than simpler ones, but they are often associated with a tradeoff
  - More costly to collect
  - Achieving full coverage might not be practical

Definition-use Coverage

- A *definition-use* (or *def-use*) is defined with respect to some variable *v*
  - ***Def***: location where *v* is defined
  - ***Use***: location where *v* is used
- Can be used to define different testing criteria such as:
  - Use every *def*
  - Execute every *use*

Example

Computing the maximum of two numbers

```
s1. x = readInput();
s2. y = readInput();
s3. max = x;
s4. if(y > max)
s5.   max = y;
s6. output(max);
```

Statements	Branches	Def-uses
s1 s2 s3 s4 s6	s4→s6	(x, s1, s3) (y, s2, s4) (max, s3, s4) (max, s3, s6)

Test case (x=5, y=1)

Logic Coverage Criteria

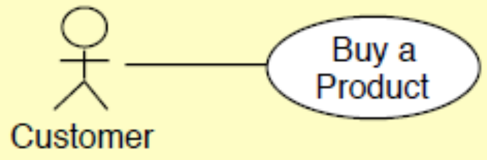
- Predicate coverage
  - The test set should make each predicate evaluate to true and false
- Clause coverage
  - The test set should make each clause evaluate to true and false
- Active clause coverage
  - Similar to clause coverage with the extra requirement that the clause should determine the predicate

Logic Coverage (Example)

- Expression: (a>b) && (c==1 || d==1)
- Tests: (a=4,b=3,c=1,d=2), (a=3,b=4,c=2,d=1)
- Which logic criteria are satisfied?

Generating Tests from Use Cases

- Test the basic flow
- Test the alternate flows
- Test the postconditions
- Break the preconditions
  - What happens if this is not met?
- Identify options for each input choice
  - Select combinations of options for each test case



Buy a Product

Precondition: Customer has successfully logged in

Main Success Scenario:

1. Customer browses catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address, 1-day or 3-day)
4. System presents full pricing information
5. Customer fills in credit card information
6. System authorizes purchase
7. System confirms sale immediately
8. System sends confirming email to customer

Postcondition: Payment was received in full, customer has received confirmation

Extensions:

3a: Customer is Regular Customer:

.1 System displays current shipping, pricing and billing information

.2 Customer may accept or override defaults, cont MSS at step 6

6a: System fails to authorize credit card

.1 Customer may reenter credit card information or may cancel

Generating Tests from Use Cases

- Use-Case Tests are good for
  - User acceptance testing
  - “Business as usual” functional testing
  - Manual black-box tests
- Limitations
  - Use cases might be out of date
  - Use cases might be ambiguous
  - Gaps and inconsistencies between use cases
- Many types of defects are unlikely to be discovered using this approach. Examples include:
  - System errors (e.g. memory leaks)
  - Performance problems

Testing with good and bad data

- Examples of bad data
  - Too little data (or no data)
  - Too much data
  - The wrong kind of data (e.g. negative salary)
  - The wrong size of data
- Examples of good data
  - “Normal” values
  - Values that are compatible with old data

Varying input values

- Values that trigger alternative flows
  - E.g. invalid credit card
- Trigger different error messages
  - E.g. text too long for field
- Inputs that cause changes in the appearance of the UI
  - E.g. a prompt for additional information
- Cases in a business rule
  - E.g. no next day delivery after 6pm.
- Border conditions
  - E.g. if password must be min 6 characters test password of 5,6,7 characters
- Enter data in different formats
  - E.g. phone numbers
- Test country-specific assumptions
  - E.g. date format

Quick Testing

- Use quick, cheap tests that aim at breaking the system
- Examples
  - Use inputs that force all the error messages to appear
  - Overflow the input buffers
  - Force a data structure to store too many or too few values
  - Find ways to violate internal data constraints
  - Force computation results to be too big or too small
  - Vary file system conditions

Interference Testing

- While one task is underway, do something to interfere with it
- Examples
  - Generate interrupts from a device or a software event
  - Change the context (e.g. change the video resolution)
  - Cancel a related task
  - Load the processor with other tasks
  - Put the machine to sleep

When to Stop Testing?

- Use a reliability estimation process (e.g. Motorola's *Zero-Failure Testing*)
  - Number of further test hours needed = $\frac{ln\left(\frac{fd}{0.5+fd}\right)\times th}{ln\left(\frac{0.5+fd}{tf+fd}\right)}$
  - fd = target failure density (e.g. 0.03 failures per 1000 LOC)
  - tf = total test failures observed so far
  - th = total testing hours up to the last failure
- Fault Seeding
  - Introduce (seed) N faults into the software
  - Start testing, and see how many seeded faults you find
  - Hypothesis: $\frac{Detected\ seeded\ faults}{Total\ seeded\ faults} = \frac{Detected\ non-seeded\ faults}{Total\ non-seeded\ faults}$