Lecture 15: Introduction to Testing

Defects vs. Failures

Effectiveness of defect detection strategies

Basics of Testing

- Testing and integration
- Types of test coverage

Defects vs. Failures

Many causes of defects in software:

- Missing requirement
- Specification wrong
- Requirement that was infeasible
- Faulty system design
- Wrong algorithms
- Faulty implementation

Defects (may) lead to failures

- but the failure may show up somewhere else
- tracking the failure back to a defect can be hard

Program Defects

Syntax Faults
- incorrect use of programming constructs (e.g. = for ==)

Algorithmic Faults
- Branching too soon or too late
- Testing for the wrong condition
- Failure to initialize correctly
- Failure to test for exceptions
  - e.g. divide by 0
- Type mismatch

Precision Faults
- E.g. mixed precision, floating point conversion, etc.

Documentation Faults
- design docs or user manual is wrong

Stress Faults
- E.g. overflowing buffers, lack of bounds checking

Timing Faults
- processes fail to synchronize
- events happen in the wrong order

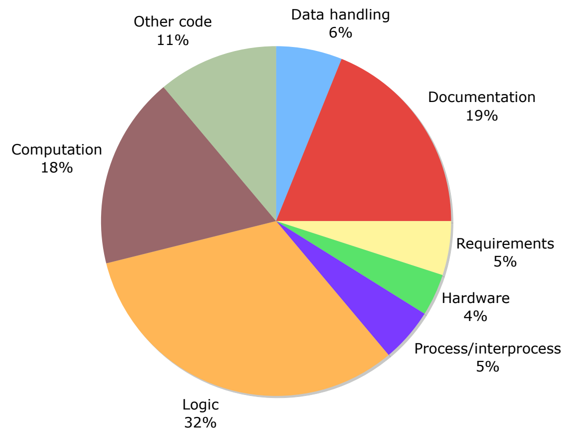
Throughput Faults
- Performance lower than required

Recovery faults
- incorrect recovery after another failure
  - e.g. incorrect restore from backups

Hardware faults
- hardware doesn't perform as expected

Defect Profiles

E.g. Data from Hewlett-Packard:

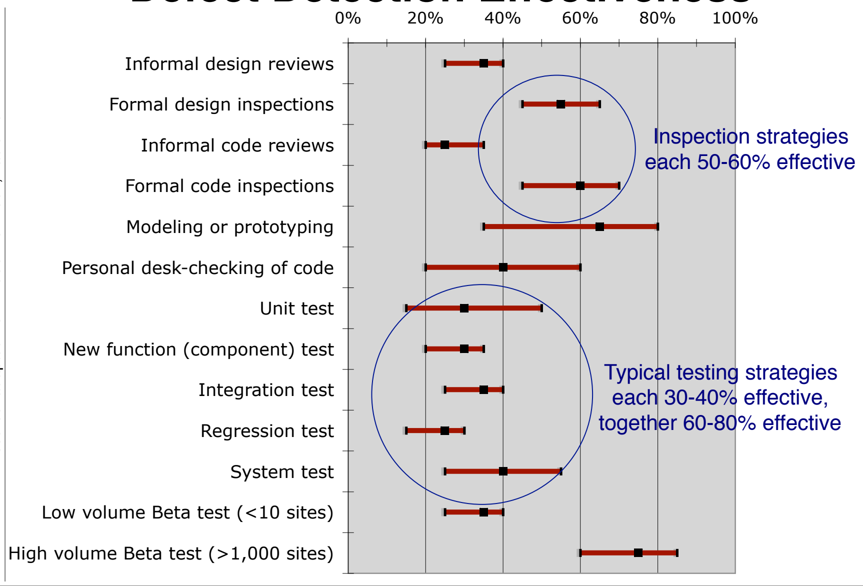


source: adapted from Pfleeger & Atlee 2006, Figure 8.2

University of Toronto Department of Computer Science

Defect Detection Effectiveness

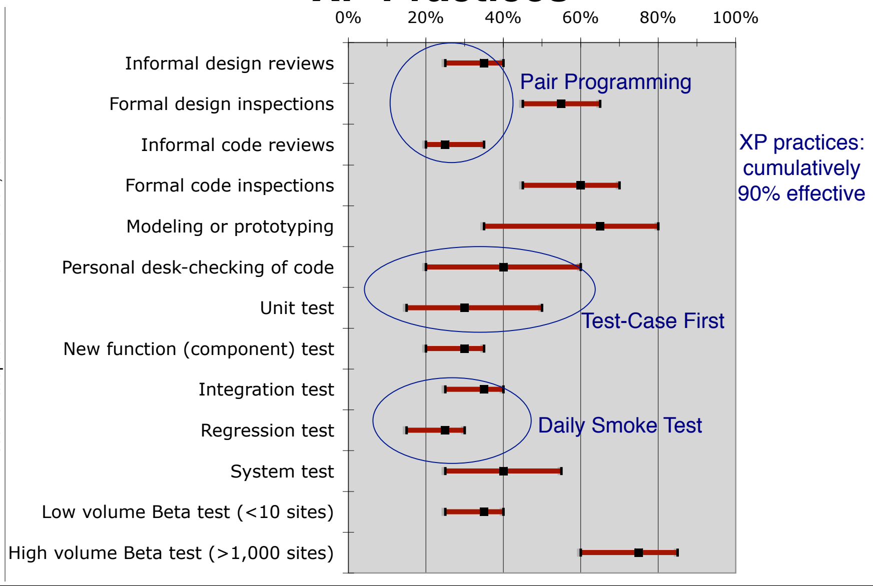
source: adapted from McConnell 2004, Table 20-2



University of Toronto Department of Computer Science

XP Practices

source: adapted from McConnell 2004, Table 20-2



Observations

Use a combination of techniques

- Different techniques find different defects
- Different people find different defects
- Testing alone is only 60-80% effective
- Best organisations achieve 95% defect-removal
- Inspection, Modeling, Prototyping, system tests, are all important

Costs vary:

- e.g. IBM data:
- 3.5 hours per defect for inspection
- 15-25 hours per defect for testing

Costs of fixing defects also vary:

- 100 times more expensive to remove a defect after implementation than in design
- 1-step methods (e.g. inspection) cheaper than 2-step (e.g. test+debug)

“Quality is Free!”

Cost of Rework:

- Industry average: 10-50 lines of delivered code per day per person
- Debugging + re-testing = 50% of effort in traditional SE

Removing defects early saves money

- Testing is easier if the defects are removed first
- High quality software is delivered sooner at lower cost

How not to improve quality:

“Trying to improve quality by doing more testing is like trying to diet by weighing yourself more often”

So, why Test?

Find important defects, to get them fixed

Assess the quality of the product

Help managers make release decisions

Block premature product releases

Help predict and control product support costs

Check interoperability with other products

Find safe scenarios for use of the product

Assess conformance to specifications

Certify the product meets a particular standard

Ensure the testing process meets accountability standards

Minimize the risk of safety-related lawsuits

Measure reliability

source: adapted from Kener 2006

Testing is Hard

Goal (as commonly understood) is unachievable
- Cannot ever prove absence of errors
- Finding no errors probably means your tests are ineffective

Goal is counter-intuitive
- Aim is to find errors / break the software
- (all other development activities aim to avoid errors / breaking the software)

It does not improve software quality
- test results measure existing quality, but don't improve it
- Test-debug cycle is the least effective way to improve quality

It requires you to assume your code is buggy
- If you assume otherwise, you probably won't find them

Oh, and…

Testing is more effective if you removed the bugs first!

Appropriate Testing

Imagine:

you are testing a program that performs some calculations

Four different contexts:

1. It is used occasionally as part of a computer game
2. It is part of an early prototype of a commercial accounting package
3. It is part of a financial software package that is about to be shipped
4. It is part of a controller for a medical device

For each context:

What is your mission?

How aggressively will you hunt for bugs?

Which bugs are the most important?

How much will you worry about:

- performance?
- polish of the user interface?
- precision of calculations?
- security & data protection?

How extensively will you document your test process?

What other information will you provide to the project?

source: adapted from Kener 2006

Good tests have…

Power
when a problem exists, the test will find it

Validity
problems found are genuine problems

Value
test reveals things clients want to know

Credibility
test is a likely operational scenario

Non-redundancy
provides new information

Repeatability
easy and inexpensive to re-run

Maintainability
test can be revised as product is revised

Coverage
Exercises the product in a way not already tested for

Ease of evaluation
results are easy to interpret

Diagnostic power
helps pinpoint the cause of problems

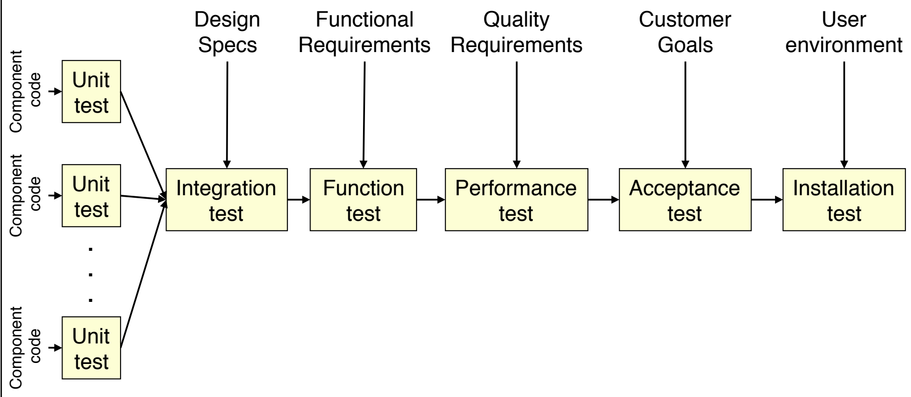
Accountability
You can explain, justify and prove you ran it

Low cost
time & effort to develop + time to execute

Low opportunity cost
is a better use of you time than other things you could be doing…

source: adapted from Kener 2006

Types of Testing



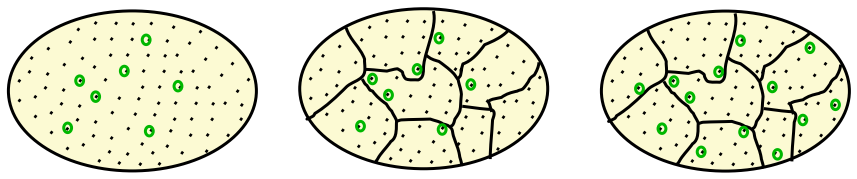
source: adapted from Pfleeger & Atlee 2006

Partitioning

Source: *Adapted from Horton, 1999*

Systematic testing depends on partitioning

- partition the set of possible behaviours of the system
- choose representative samples from each partition
- make sure we covered all partitions



How do you identify suitable partitions?

- That's what testing is all about!!!
- Methods:
  - black box, white box, ...
  - path based, state based, risk based, scenario based, …

Coverage 1: Structural

```
boolean equal (int x, y) {
/* effects: returns true if
  x=y, false otherwise
*/
if (x == y)
  return(TRUE)
else
  return(FALSE)
}
```

Naïve Test Strategy

pick random values for x and y and test ‘equals’ on them

But:

...we might never test the first branch of the ‘if’ statement

So:

Need enough test cases to cover every branch in the code

Coverage 2: Functional

```
int maximum (list a)
/* requires: a is a list of
  integers
  effects: returns the maximum
  element in the list
*/
```

Naïve Test Strategy

generate lots of lists and test maximum on them →

Input	Output	Correct?
3 16 4 32 9	32	Yes
9 32 4 16 3	32	Yes
22 32 59 17 88 1	88	Yes
1 88 17 59 32 22	88	Yes
1 3 5 7 9 1 3 5 7	9	Yes
7 5 3 1 9 7 5 3 1	9	Yes
9 6 7 11 5	1	Yes
5 11 7 6 9	1	Yes
561 13 1024 79 86 222 97	1024	Yes
97 222 86 79 1024 13 561	1024	Yes

But:

we haven’t tested off-nominal cases:

- empty lists,
- non-integers,
- negative integers, ….

So:

Need enough test cases to cover every kind of input the program might have to handle

Coverage 3: Behavioural

Naïve Test Strategy:

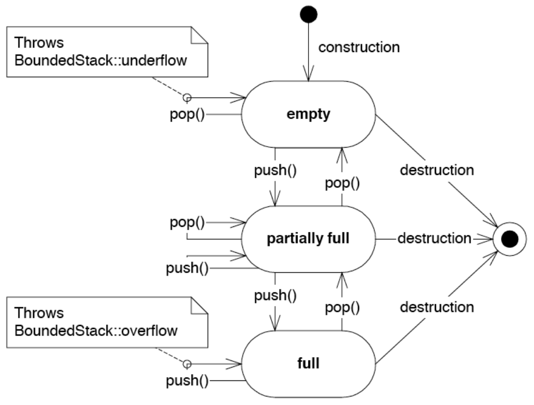
Push and pop things off the stack and check it all works

But:

Might miss full and empty stack exceptions

So:

Need enough tests to exercise every event that can occur in each state that the program can be in



Integration Testing

Source: *Adapted from van Vliet 1999, section 13.9*

Unit testing

each unit is tested separately to check it meets its specification

Integration testing

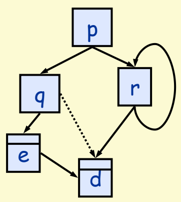
units are tested together to check they work together

two strategies:

Bottom up

for this dependency graph, test order is:

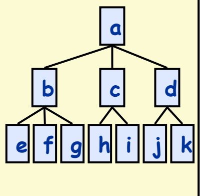
1) d
2) e and r
3) q
4) p



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



Integration testing is hard:

much harder to identify equivalence classes

problems of scale

tends to reveal specification errors rather than integration errors

Other system tests

Other things to test:

- **facility testing** - does the system provide all the functions required?
- **volume testing** - can the system cope with large data volumes?
- **stress testing** - can the system cope with heavy loads?
- **endurance testing** - will the system continue to work for long periods?
- **usability testing** - can the users use the system easily?
- **security testing** - can the system withstand attacks?
- **performance testing** - how good is the response time?
- **storage testing** - are there any unexpected data storage issues?
- **configuration testing** - does the system work on all target hardware?
- **installation testing** - can we install the system successfully?
- **reliability testing** - how reliable is the system over time?
- **recovery testing** - how well does the system recover from failure?
- **serviceability testing** - how maintainable is the system?
- **documentation testing** - is the documentation accurate, usable, etc.
- **operations testing** - are the operators' instructions right?
- **regression testing** - repeat all testing every time we modify the system!