Lecture 15: Introduction to Testing

Defects vs. Failures

Effectiveness of defect detection strategies

Role of testing

Testing strategies

© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.

Defects and 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

© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.

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. divde 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

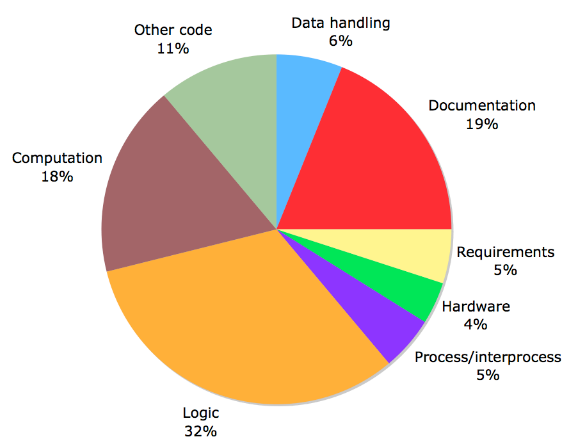
Recovery faults
- incorrect recovery after another failure
- e.g. incorrect restore from backups

Hardware faults
- hardware doesn’t perform as expected

© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.

Defect Profiles

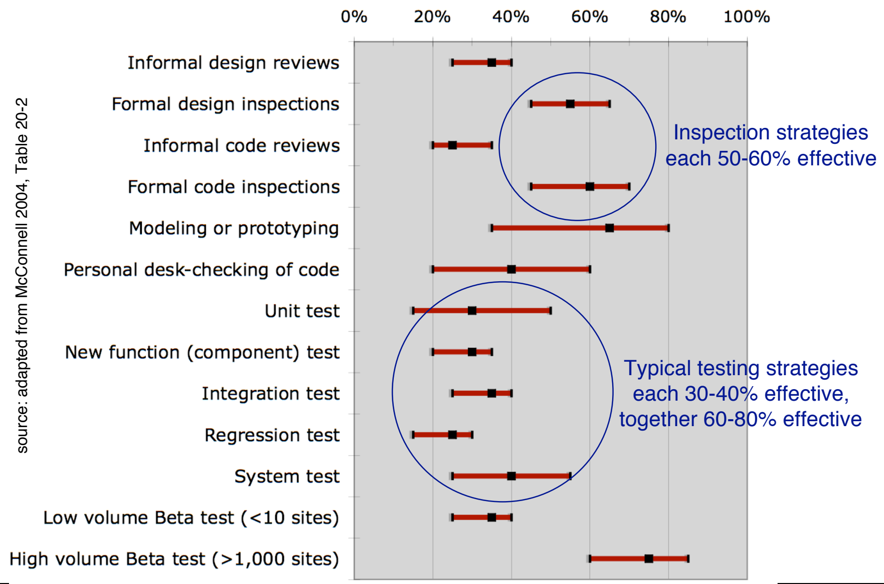
E.g. Data from Hewlett-Packard:



source: adapted from Pfleeger & Atlee 2006, Figure 8.2

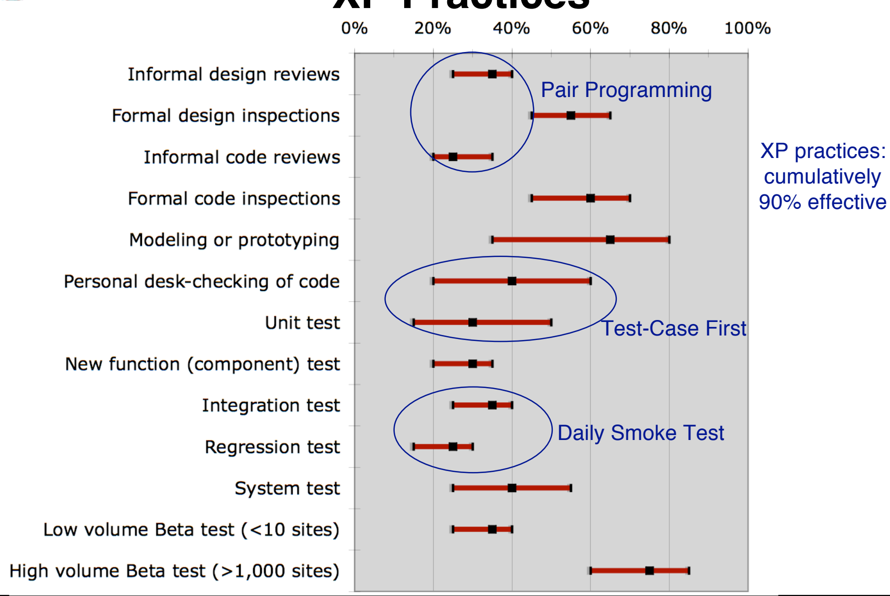
© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.

Defect Detection Effectiveness



© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.

XP Practices



© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.

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)

© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.

"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"

© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.

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

© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.

Testing is Hard

Goal is counter-intuitive

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

Goal is unachievable

- Cannot ever prove absence of errors
- Finding no errors probably means your tests are ineffective

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!

© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.

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

© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.

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

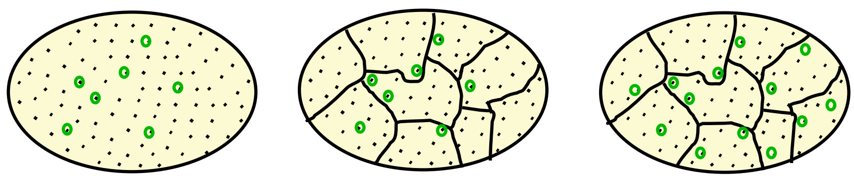
© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.

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

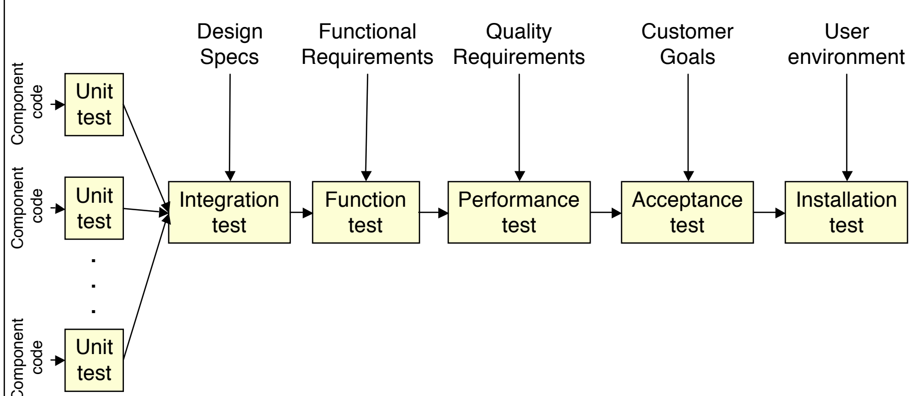


How do you identify suitable partitions?

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

© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.

Types of Testing



source: adapted from Pfleeger & Atlee 2006

© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.

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

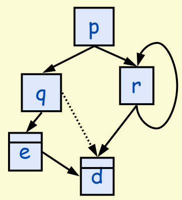
units are tested together to check they work together

two strategies:

Bottom up

for this dependency graph, test order is:

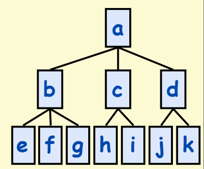
1) d
2) e and r
3) q
4) p



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

© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.

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!

© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.

Automated Testing

Source: *Adapted from Liskov & Guttag, 2000, pp239-242*

Ideally, testing should be automated

- tests can be repeated whenever the code is modified ("regression testing")
- takes the tedium out of extensive testing
- makes more extensive testing possible

Will need:

- test driver - automates the process of running a test set
  - sets up the environment
  - makes a series of calls to the unit-under-test
  - saves results and checks they were right
  - generates a summary for the developers
- test stub - simulates part of the program called by the unit-under-test
  - checks whether the UUT set up the environment correctly
  - checks whether the UUT passed sensible input parameters to the stub
  - passes back some return values to the UUT (according to the test case)
  - (stubs could be interactive - ask the user to supply return values)

© 2008 Steve Easterbrook. This presentation is available free for non-commercial use with attribution under a creative commons license.