

CS 350, slide set 6

M. Overstreet
Old Dominion University
Spring 2005

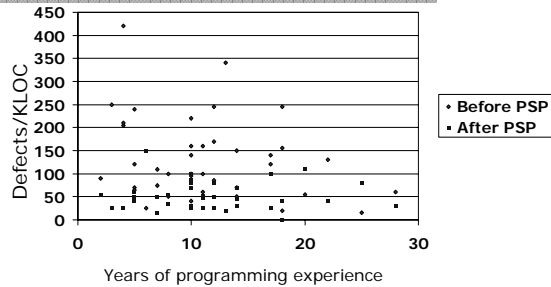
Reading

- PSP text, ch. 15, 16, 17, 18, 19, 20

Topics

- Projecting defects
- Economics of defect removal
- Design defects
- Quality

Defect rates (from 38 developers taking PSP course)



Lessons from table

- Very high defect rates for practicing software developers
 - 100 defects/KLOC (or more) before PSP training
 - Still 50 defects/KLOC after PSP training
- Training in tracking defects much more effective than experience in reducing defect levels

Defect estimation

- Defect density (Dd) is defects per 1000 lines of code
- If you have data on *i* previous program which include size and number of defects, then
 - $Dd_{plan} = 1000(D_1 + \dots + D_i) / (N_1 + \dots + N_i)$
- New entries on the Project Plan form:
 - Plan Defects/KLOC
 - Plan Defects Injected (for each phase)
 - Plan Defects Removed (for each phase)
- Text has completely worked out example

Comments

- Filling out tedious forms and counting defects won't improve code quality.
- If **you** want to reduce your own defect levels, then these techniques can likely help. If you don't care, the forms won't help.

Defect problems

- New systems are bigger and more complex
 - Text example: early laser printer SW had 20 KLOC; new has 1,000 KLOC
 - 10 years ago cars had 0 KLOC; now several KLOC

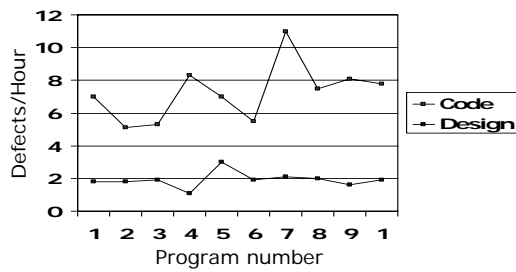
Defect metrics

- We've already talked about Defects/KLOC as a measure of code quality
- **Yield** measures the percentage of defects found by a removal method (such as testing or code review)
- As such, yield is an indication of the effectiveness of an defect removal activity.

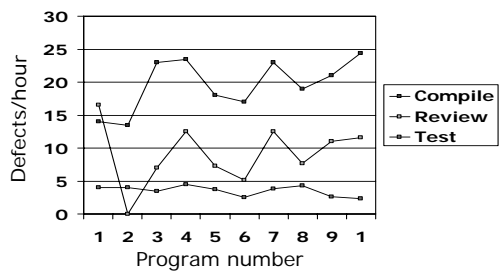
Computing yield

- Can be computed for both reviews and testing.
- For code review,
 - Yield = $100(\# \text{ defects found before compile}) / (\text{defects injected before compile})$
- For testing:
 - Yield = $100(\# \text{ defects found during testing}) / (\text{defects present before testing})$
- Problem: how do we know the number of defects present before compile or testing?

Defect-injection rates



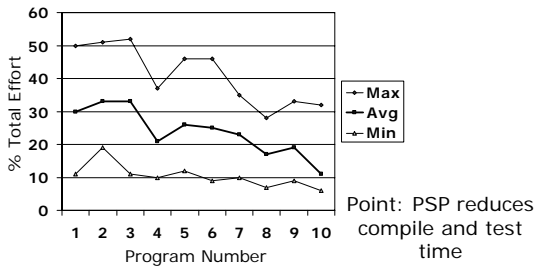
Defect removal rates



Typical numbers for small, class-size programs

- Not using PSP
 - Inject about 100/hr
 - Find 50/hr in compile
 - Find 40/hr in testing
- Using PSP
 - Inject about 50/hr
 - With code reviews, remove 60% to 70% before first compile
- For a 500 loc program, PSP saves about 7 hours overall (more for review, less for debugging).
- Think scaling: for a 10,000 loc program, same numbers mean a 140 hr savings.

PSP effects on compile and test time



Computing Defects/Hour

- Goal: measure efficiency of defect removal efforts at different points.
- Simple computation if:
 - You have accurate defect counts in each development phase
 - You have accurate measures of the amount of time spent in each phase
- A simple division for each phase

$$\text{Def./hr} = (\# \text{ defects found}) / (\text{hours in phase})$$

Suggestions for improving defect removal rates

- Focus on yield first
 - Humphrey suggests targeting a yield of 70% or better
- Do code reviews before 1st compile
 - Goal: 0 comp. errs on 1st compile
- Review error logs for reoccurring problems; if appropriate, revise checklists
- Humphrey (from Brown): "one definition of insanity is doing the same thing over and over and expecting a different result"

Reduce defect injection rates

- Record all defects
- Produce more complete designs
- Use better methods for developing designs, code, or whatever
- Use better tools
 - My experience is that it's hard to get students to use tools.
 - Can also be true of programmers

Design defects

- Sometimes tricky to separate design defects from code defects
- No real clear line between design and code
 - Lots of tools exist to convert what in the past was a design (pictures) to source code (at least in a skeletal form)
 - The amount of detail I put in a design depends, in part, on how knowledgeable I think the programmer is likely to be.
 - Want to leave out as much as possible, but some times the programmer needs guidance
 - Sometimes the programmer needs freedom instead

Let's simplify

- Humphrey suggests error types 10 through 40 as "code-like" errors, 50 through 90 as "design-like" errors

Better designs come with experience (maybe)

- Start with high-level designs then refine
- Experienced developers know when how level design components are feasible
 - Inexperienced developers can produce designs that have components that won't work at all.
- Recall NASA's design of n-version tester
 - Clearly (at least to an experienced developer) the design solved several critical problems:
 - Failed versions
 - Rounding error
 - No additional details required to see this

Testing

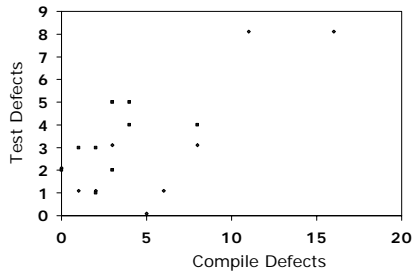
- As software gets more complex, it gets harder to test.
- IRI (an NSF-funded project for interactive distance education) is a distributed system
 - Testing is hard because much depends on the order in which code fragments run.
 - If 10 parts, then 10! possible orders
 - $10! = 3.6$ million
 - IRI may have 20 components that run in parallel.
 - $20! = ?$

Defect removal yields

Method	Approx. Yield %
Code review	70-80
Code inspection	50-70
Compile	50
Unit test	40-50
Integration test	45
Requirements test	45
Algorithm test	8

Note: This data may not be typical!

Compile & Test Data



Lessons

- (Based on limited data): Lots of compile errors \Rightarrow lots of test errors \Rightarrow lots of errors delivered to customer
- Testing less effective with large programs
 - But code reviews still work (why?)

Basic belief of CMM

- Quality of product depends on quality of process
- Use metrics to measure quality of process
 - Productivity? LOC/hour
 - Code? Defects/KLOC
 - Efficiency? Defects/hour (low for injection, high for removal)
- If you think you have a better process, then you need data comparing it with other processes

Comment on Humphrey's data

- Humphrey believes in data for making decisions on what works
- Industry will rarely release these types of data
 - Productivity
 - Error rates
 - Yield rates

Towards quality code

- Build the best possible units you can
 - Do careful reviews
 - Inspect requirements to help ensure unit does
 - All it is supposed to do
 - Does exactly what it is supposed to do
 - Do careful code inspections (we will talk about later—but it is a group activity)
 - If the unit is small enough, do exhaustive unit testing after the inspection
- Do thorough integration testing
 - Should be done by group different from the code developers
- Do comprehensive system testing

Costs of quality

- Failure costs must be balanced with the additional costs of building quality code
- Quality costs:
 - Time for code reviews
 - Time for code inspections
 - Time for developing test plans, test data, test drivers, and testing
 - Building prototypes to assist in making key design decisions
 - Can sometimes measure prototype behavior
 - Building prototypes to better understand system (or compiler or whatever) behavior

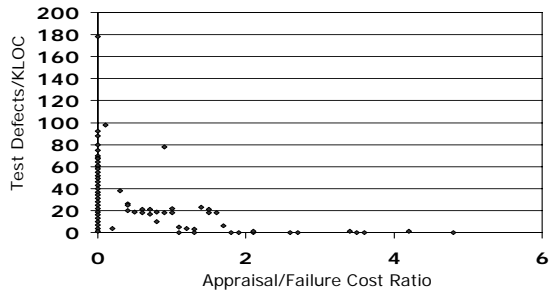
For PSP cost of quality

- (A little artificial)
- Appraisal COQ is the sum of all review times. It is a percentage of total development time.
- Failure COQ includes all time spent in compiling and testing. It is a percentage of total development time

A new process metric: A/FR

- A/FR is for "appraisal/failure ratio"
 $A/FR = (\text{appraisal COQ})/(\text{failure COQ})$
- Can also be computed as
 $(\text{code review time})/(\text{compile time} + \text{test time})$
- When A/FR is less than 1, testing is finding lots of faults
- Goal: A/FR should be larger than 2
 - How can you achieve this goal?
 - On new project, predict testing time based on your previous data
 - Then spend more time doing review!

A/FR data



Exam 1

- Available at class web site on Thursday.
- Complete by deadline and e-mail to cmo
- You must use e-mail to submit!
- Due following Tuesday by midnight!
 - Late exams not accepted.
- Designed to take about an hour to complete
