Combining Inspections, Static Analysis, and Testing to Achieve Defect Removal Efficiency Above 95%

Draft 1.0 January 5, 2012

Abstract

For more than 50 years the software industry has depended primarily upon testing to remove defects. Unfortunately testing by itself has not achieved average levels of defect removal efficiency higher than about 85%. When pre-test inspections and static analysis are used prior to testing, defect removal efficiency average more than 95% and sometimes approach 99%. Development schedules and costs are reduced at the same time. Even better, pre-test inspections and static analysis combined with formal testing can reduce technical debt or the costs of post-release defect repairs by more than 80%.

Capers Jones, President
Capers Jones & Associates LLC
Web: www.Namcook.com
Email: Capers.Jones3@Gmail.com

INTRODUCTION

The software industry has achieved a bad reputation for poor software quality. In part this reputation is deserved, and is due to reliance upon testing without using pre-test inspections or static analysis.

Most forms of testing are only about 35% efficient or find about one bug out of three. This is why six to eight different kinds of testing are needed. Formal inspections of requirements, design, and code have been measured for more than 50 years and these methods achieve defect removal efficiency levels of about 85%.

A new form of defect removal called "static analysis" has been added to the set of defect removal methods. There are many static analysis tools and they vary in effectiveness from about 40% to more than 80% of code bugs. However static analysis tools only work for about 25 programming languages out of a global total of more than 2,500 programming languages. At least one static analysis tool also operates on natural language text, and therefore can be used to find defect in requirements, design, and other text documents.

A synergistic combination of formal inspections, static analysis, and formal testing can achieve combined defect removal efficiency levels of 99%. Better, this synergistic combination will lower development costs and schedules and reduce technical debt by more than 80% compared to testing alone.

Software Defect Potentials and Application Size

Studies of software quality indicate a strong correlation between application size and the total number of bugs or defects that have to be eliminated. A simple rule of thumb can provide an approximate but useful estimate of potential defects related to software application size:

Raise the size of the application in function points to the 1.22 power and the result will yield the approximate total number of defects that must be found and eliminated.

This rule of thumb includes all major sources of defects: requirements defects, design defects, coding defects, documentation defects, and "bad fixes" or secondary defects that are accidentally included in attempts to repair other defects.

This rule of thumb works well between about 10 and 10,000 function points in size. As applications size grows above 10,000 function points, the number of bugs predicted will be somewhat higher than will probably occur. Examples of this rule of thumb in Table 1 show the following values:

Table 1: Approximate Numbers of Software Defects by Application Size

Function Points	Potential Defects	Defects per Function Point
10	17	1.70
100	275	2.75
1,000	4,570	4.57
10,000	75,858	7.59

Like all rules of thumb these results are only approximate and should not be used without additional validation and more careful estimation. Strong software development methods such as the Rational Unified Process (RUP) or Team Software Process (TSP) will have fewer defects. Careless methods such as "cowboy programming" or waterfall may have more defects than those indicated in table 1.

Example of the Defect Prediction Rule of Thumb

Here is an example that illustrates the use of the defect prediction rule of thumb for a medium application of 1,000 function points:

1,000 function points raised to the 1.22 power = 4,570 potential defects in all categories.

Because this rule of thumb includes defects in requirements, design, source code, user documents, and bad fixes or secondary defects, it is useful to show an approximate distribution of defects across these sources:

Table 2: Defect Distribution for a Software Application of 1000 Function Points

Defect Origin	Percent of Total	Number of Defects	Defects per Function Point
Requirements	15%	685	0.69
Design	30%	1,371	1.37
Code defects	40%	1,828	1.83
Document defects	10%	457	0.46
Bad fixes	5%	229	0.23
TOTAL	100%	4,570	4.57

While this may seem like a large number of defects, it should be recalled that a significant percentage, possibly more than of 97%, will be found prior to delivery of the application to users.

Definitions and Examples of Software Defects

A software defect can be defined as an error in a software deliverable that would either cause the software to stop working or to produce incorrect results. Software defects can range in severity from critical severity 1 defects which stop the application from running through minor severity 4 defects such as a spelling error in a text message.

Software defects can be found in five origin points, and all origins can cause serious problems.

Requirement defects can be errors of commission or errors of omission. An example of an error of commission was the insistence of using 2-digit date fields, which caused the famous Y2K problem. An example of an error of omission is the lack of any hard-copy backup or method of validating the results in electronic voting machines. This omission means that such machines can possibly be hacked or modified without detection. Requirements defects average about 1.0 per function point with perhaps a plus or minus 75% range based on requirements methods.

Design defects can also be errors of commission or errors of omission. An example of a design error of commission was noted in an application where the format of an output data field was different from the format expected by the input reader. As a result, the messages could not be read until both formats were made identical. The two fields were coded by different teams, each of whom followed the design specifications provided to them. A serious example of a design omission was leaving out February 29th in a mainframe security calendar function. This caused all computers using the security package to lock up at midnight when February 28th changed to February 29th. Design defects average about 1.25 per function point with about a 35% range based on design methods.

Code defects are usually errors of commission but sometimes are errors of omission. An example of an error of commission is a branch to the wrong location. An example of an error of omission is to omit validation of inputs or failure to initialize variables. Code defects average about 1.75 per function point, but have a variance of around 80% based on programming languages, reuse of certified code, and coding practices.

Documentation defects can be found in user manuals, HELP screens, or other text materials provided with software applications. An annoying but common error of commission is providing incorrect instructions for installing an application. An annoying but common error of omission is to forget to tell application users how to shut down the application when they are finished and wants to quit. Document defects average about 0.6 per function point with a range of about 40% based in part on the accuracy of the requirements and design documents, and in part on documentation methods and the use of text analysis tools.

Bad fix defects are new errors accidentally included in attempts to repair previous errors. These bad fix defects average about 7% of attempted defect repairs, but have been observed to approach 60% for error-prone modules with high levels of cyclomatic complexity. An example of a bad-fix error was noted in a financial application where an attempt to fix a format error introduced a mathematical error. Bad fixes can usually be found by static analysis or regression testing so their presence indicates somewhat careless quality controls. Bad fix injections average about 0.4 per function point. The range is very wide and is proportional to development methods, cyclomatic complexity, programming languages, and several other variables.

There are other kinds of defects besides these five. For example data errors in data bases probably outnumber software defects, but are outside the control of software groups. Other sources of defects include architecture, reusable materials, purchased materials, and somewhat surprisingly, test cases. An IBM study found almost as many errors in test cases as in the software being tested!

Definitions and Examples of Software Defect Removal Methods

Since not all readers know about all forms of defect removal, it is appropriate to define and illustrate the methods covered by this article.

Formal Inspections: Inspections were developed by IBM in the early 1970's. The impetus for the development of inspections was the rapid increase in application size combined with IBM's measures of defect removal efficiency. These measures indicated that for applications large than about 1,000 function points in size, testing defect removal efficiency began to decline.

Inspections are team activities in which between four and eight personnel examine software materials using a formal process and following formal roles. The major participants in inspections include a moderator, whose job is to keep the inspection on track; the recorder, whose job is to record defects, effort, and take notes; the developer, whose work is being inspected; and one or more additional inspectors who may come from quality assurance, testing, or be other software engineers.

Inspections can be applied successfully to any deliverable including plans, architecture, requirements, design documents, source code, and even test cases. Inspections have the highest measured efficiency of any form of defect removal, and typically find more than 85% of the bugs or defects in any item inspected.

Static Analysis: This method is relatively new and originated in about 1987. Static analysis tools do not execute the code, but rather go through the code line by line and apply rules to find errors or certify correctness. There may be several thousand such rules. Static analysis is very effective in finding common problems such as branching errors and syntax errors. Unlike testing which can only find defects, static analysis can validate applications by not finding defects. One common issue with automated static analysis tools are "false positives" or identifying a correct structure as a probable error.

With more than 50 static analysis tools available, it is hard to state a specific result. But they seem to range between about 40% and 85% efficient in finding defects, with perhaps a 5% false positive ratio.

One new static analysis tool works on text documents instead of code. Several older text analysis tools also work on text and compute readability indices such as the FOG index. These are also rule-based static analysis tools.

Testing: Testing of various kinds has been the most common software defect removal method since the industry began. As of 2012 there are about 40 kinds of testing, although most companies only use about six kinds: 1) unit test; 2) function test; 3) regression test; 4) component test; 5) system test; 6) Beta or acceptance test.

Some of the specialized forms of testing include performance testing, usability testing, security testing, nationalization testing for international projects, independent testing, supply chain testing, and combined hardware/software testing.

Most forms of testing are only about 35% efficient in finding bugs, which is why at least six different test stages are needed, and sometimes as many as 12 test stages are needed for mission-critical applications.

(The author's 2011 book The Economics of Software Quality contains data on 40 kinds of testing and 25 kinds of pre-test defect removal operations.)

Predicting Defect Removal Efficiency (DRE)

One of the most important quality metrics is that of defect removal efficiency (DRE). This metric can be tricky, but in principle it is very easy to measure. Keep track of all of the defects found during development. After release, keep track of all of the defects reported by customers. After 90 days of customer use, combine internal defect counts with customer defect counts. Then calculate the percentage found internally. If the development team found 90 defects and customers reported 10 defects in the first three months of use, the total count is 100 and defect removal efficiency is 90%. This simple metric is probably the single most important quality metric.

The U.S. average for defect removal efficiency as of 2012 is only about 85%. But the best companies average more than 95% and sometimes approach 99%. High levels of defect removal efficiency above 95% are only possible when using a synergistic combination of inspections, static analysis and testing. Testing alone seldom tops 85%. and is often below 80%.

All of these various topics are combined in table 3 which shows six pre-test removal activities and six test stages. Table 3 assumes 1000 function points and 55,000 Java statements.

Table 3 was created using the author's Software Risk Master tool, which predicts any combination of pre-test and test stages. Table 3 shows the combined results of pre-test inspections, pre-test static analysis, and six common forms of testing:

Table 3: Results of Six Pre-Test Defect Removal and Six Test Stages

Application size (FP)	1,000
Size (KLOC)	55
Size (LOC)	55,000
Language	Java

		Require. Defects per Function Point	Design Defects per Function Point	Code Defects per Function Point	Document Defects per Function Point	TOTALS
	Defect Potentials per Function Point	0.57	1.14	2.39	0.46	4.55
	Defect potentials	569	1,139	2,391	455	4,554
	Security flaws	1	3	8	0	12
	Pre-Test Defect Removal Methods					
		Require.	Design	Code	Document	Total
1	Text static analysis	50.00%	50.00%	0.00%	50.00%	23.75%
	Defects discovered	285	569	0	228	1,082
	Bad-fix injection	9	17	0	7	32
	Defects remaining	276	552	2,391	221	3,440
2	Require. inspection	87.00%	10.00%	1.00%	8.50%	7.42%
	Defects discovered	240	55	24	19	338
	Bad-fix injection	7	2	1	1	10
	Defects remaining	29	495	2,366	202	3,092
3	Design inspection	25.00%	87.00%	7.00%	26.00%	21.22%
	Defects discovered	7	431	166	52	656
	Bad-fix injection	0	13	5	2	33
	Defects remaining	21	51	2,196	148	2,416
4	Document Inspection	5.00%	5.00%	10.00%	90.00%	14.74%
	Defects discovered	1	3	220	133	356
	Bad-fix injection	0	0	1	0	1
	Defects remaining	20	49	1,975	14	2,059
5	Code static analysis	2.00%	10.00%	65.00%	3.00%	53.38%
	Defects discovered	0	5	1,284	0	1,290
	Bad-fix injection	0	0	39	0	39
	Defects remaining	20	44	653	14	730

6	Code Inspection	15.00%	25.00%	87.00%	30.00%	80.24%
	Defects discovered	3	11	568	4	586
	Bad-fix injection	0	0	17	0	18
	Defects remaining	17	33	68	10	127
	Pre-test removal	553	1,106	2,323	446	4,427
	Pre-test efficiency %	97.06%	97.14%	97.16%	97.89%	97.22%
	Security flaws	1	2	6	0	9

Test Defect Removal Stages

		Require.	Design	Code	Document	Total
1	Unit testing	4.00%	7.00%	40.00%	10.00%	24.49%
	Defects discovered	1	2	27	1	31
	Bad-fix injection	0	0	1	0	1
	Defects remaining	16	30	40	9	95
2	Function testing	5.00%	22.00%	47.00%	30.00%	30.37%
	Defects discovered	1	7	19	3	29
	Bad-fix injection	0	0	1	0	1
	Defects remaining	15	23	21	6	65
3	Regression testing	2.00%	5.00%	33.00%	15.00%	14.06%
	Defects discovered	0	1	7	1	9
	Bad-fix injection	0	0	0	0	0
	Defects remaining	15	22	14	5	56
4	Integration testing	20.00%	27.00%	50.00%	22.00%	30.28%
	Defects discovered	3	6	7	1	17
	Bad-fix injection	0	0	0	0	1
	Defects remaining	12	16	7	4	38
5	System testing	12.00%	18.00%	44.00%	34.00%	22.24%
	Defects discovered	1	3	3	1	9
	Bad-fix injection	0	0	0	0	0
	Defects remaining	10	13	4	3	30
6	Acceptance testing	14.00%	15.00%	20.00%	24.00%	16.03%
	Defects discovered	1	2	1	1	5
	Bad-fix injection	0	0	0	0	0
	Defects remaining	9	11	3	2	25
	Test Removal	8	22	65	8	102
	Security Flaws	0	1	2	0	3
	Testing Efficiency %	46.96%	66.16%	95.78%	80.17%	80.54%
	Total Defects	560	1,127	2,388	453	4,529

Bad-fix injection	17	34	72	14	136
Total Efficiency %	98.44%	99.03%	99.88%	99.58%	99.46%
Remaining Defects	9	11	3	2	25
High-severity Defects	2	2	1	0	4
Security flaws	0	0	0	0	1

Table 3 shows a cumulative defect removal efficiency of 99.46% which would be in the upper 1% of all U.S. software projects. Table 3 has some important lessons for the software engineering community:

1. Pre-test inspections and static analysis are more efficient in finding defects than testing.

2. Pre-test inspections and static analysis will raise testing efficiency slightly.

3. Pre-test inspections and static analysis will lead to shorter development schedules and lower development costs than testing alone.

4. Pre-test inspections and static analysis will reduce "technical debt" or the costs of fixing post-release defects by about 80% compared to testing alone.

5. Defect potentials and defect removal efficiency levels should be predicted prior to starting any major software projects. One of the advantages of estimation tools such as Software Risk Master is prediction of defects and removal efficiency levels early enough to plan effective combinations of pre-test and test removal stages.

In spite of more than 40 years of continuous success as the top-ranked defect removal method, formal inspections are not as widely used as testing. However inspections should be standard defect removal methods for all critical applications.

Static analysis is still a fairly new technology and has not yet become a standard defect removal method, although usage is rapidly increasing.

To illustrate the effectiveness of inspections and static analysis, table 4 shows the same application of 1000 function points and 55,000 Java statements using only the six testing stages, without any pre-test inspections or usage of static analysis tools. Here too the results were predicted using the author's Software Risk Master estimation tool.

Table 4: Results of Testing without Pre-test Inspections or Static Analysis

Application size (FP)	1,000
Size (KLOC)	55
Size (LOC)	55,000
Language	Java

		Require. Defects per Function Point	Design Defects per Function Point	Code Defects per Function Point	Document Defects per Function Point	TOTALS
	Defect Potentials per FP	0.57	1.14	2.39	0.46	4.55
	Defect potentials	569	1,139	2,391	455	4,554
	Security flaws	1	3	8	0	12
		Require.	Design	Code	Document	Total
1	Unit testing	4.00%	7.00%	37.00%	10.00%	22.68%
	Defects discovered	23	80	885	46	1,033
	Bad-fix injection	2	6	62	3	72
	Defects remaining	545	1,053	1,444	407	3,449
2	Function testing	5.00%	22.00%	44.00%	30.00%	29.47%
	Defects discovered	27	232	636	122	1,016
	Bad-fix injection	2	16	44	9	71
	Defects remaining	516	805	764	276	2,362
3	Regression testing	2.00%	5.00%	33.00%	15.00%	14.58%
	Defects discovered	10	40	252	41	344
	Bad-fix injection	1	3	18	3	24
	Defects remaining	505	762.2	494	232	1,993
4	Integration testing	20.00%	27.00%	47.00%	22.00%	29.61%
	Defects discovered	101	206	232	51	590
	Bad-fix injection	7	14	16	4	41
	Defects remaining	397	542	246	177	1,362

5	System testing	12.00%	18.00%	42.00%	34.00%	22.67%
	Defects discovered	48	98	103	60	309
	Bad-fix injection	3	7	7	4	22
	Defects remaining	346	438	135	113	1,031
6	Acceptance testing	14.00%	15.00%	20.00%	24.00%	16.30%
	Defects discovered	48	66	27	27	168
	Bad-fix injection	3	5	2	2	12
	Defects remaining	294	367	106	84	852
	Test Defects Removed	275	771	2,285	372	3,703
	Security Flaws	0	2	8	0	10
	Testing Efficiency %	48.36%	67.73%	95.55%	81.60%	81.30%
	Defects Removed	275	771	2,285	372	3,703
	Bad-fix injection	19	54	160	26	259
	Total Removal %	48.36%	67.73%	96.44%	81.60%	81.77%
	Remaining Defects	294	367	106	84	852
	High-severity Defects	50	62	18	14	145
	Security flaws	0	2	8	0	10

Both tables 3 and 4 had the same number of defects when defect removal started. Both tables 3 and 4 use exactly the same sequence of test stages. But when pre-test inspections and static analysis were omitted, total defect removal efficiency declined from 99.46% down to only 81.77%.

Note also that because inspections were not used on requirements and design, testing defect removal efficiency is reduced since test cases were built on incorrect assumptions in the requirements and design documents. Many requirements and design defects cannot be found by testing, but only by means of inspections and static analysis. (This is why the famous Y2K problem was not found by testing prior to about 1995. Because 2-digit dates were in both requirements and design documents, testers were prevented from testing for this problem.)

Defect removal efficiency against code defects in table 4 was 96.44%, but requirements defects had only a 48.35% defect removal efficiency. This illustrates the point that most requirements defects cannot be found by testing, but only by inspections and text static analysis.

Summary and Conclusions

As of 2012 software defect removal efficiency only averages about 85% in the United States and in most other countries. Usage of formal inspections is found in only about

10% of U.S. companies, while usage of static analysis tools is only found in about 35%, based on samples from among the author's clients.

A combination of formal inspections, pre-test static analysis, and formal testing by certified professional testers using mathematically derived test cases can consistently achieve defect removal efficiency levels that top 95% and often top 99%. This should be the norm, and hopefully it will become the norm when the effectiveness of pre-test removal methods becomes more widely known.

References and Readings

Charette, Bob; Software Engineering Risk Analysis and Management; McGraw Hill, New York, NY; 1989.

Charette, Bob; Application Strategies for Risk Management; McGraw Hill, New York, NY; 1990.

DeMarco, Tom; Controlling Software Projects; Yourdon Press, New York; 1982; ISBN 0-917072-32-4; 284 pages.

Ewusi-Mensah, Kweku; Software Development Failures; MIT Press, Cambridge, MA; 2003; ISBN 0-26205072-2276 pages.

Galorath, Dan; Software Sizing, Estimating, and Risk Management: When Performance is Measured Performance Improves; Auerbach Publishing, Philadelphia; 2006; ISBN 10: 0849335930; 576 pages.

Garmus, David and Herron, David; Function Point Analysis – Measurement Practices for Successful Software Projects; Addison Wesley Longman, Boston, MA; 2001; ISBN 0-201-69944-3;363 pages.

Gilb, Tom and Graham, Dorothy; Software Inspections; Addison Wesley, Reading, MA; 1993; ISBN 10: 0201631814.

Glass, R.L.; Software Runaways: Lessons Learned from Massive Software Project Failures; Prentice Hall, Englewood Cliffs; 1998.

International Function Point Users Group (IFPUG); IT Measurement – Practical Advice from the Experts; Addison Wesley Longman, Boston, MA; 2002; ISBN 0-201-74158-X; 759 pages.

Johnson, James et al; The Chaos Report; The Standish Group, West Yarmouth, MA; 2000.

Jones, Capers and Bonsignour, Olivier; The Economics of Software Quality; Addison Wesley Longman, Boston, MA; ISBN 10: 0-13-258220—1; 2011; 585 pages.

Jones, Capers; Software Engineering Best Practices; McGraw Hill, New York, NY; ISBN 978-0-07-162161-8; 2010; 660 pages.

Jones, Capers; Applied Software Measurement; McGraw Hill, 3rd edition 2008; ISBN 978-0-07-150244-3; 575 pages; 3rd edition due in the Spring of 2008.

Jones, Capers; Assessment and Control of Software Risks; Prentice Hall, 1994; ISBN 0-13-741406-4; 711 pages.

Jones, Capers; Patterns of Software System Failure and Success; International Thomson Computer Press, Boston, MA; December 1995; 250 pages; ISBN 1-850-32804-8; 292 pages.

Jones, Capers; Software Quality – Analysis and Guidelines for Success; International Thomson Computer Press, Boston, MA; ISBN 1-85032-876-6; 1997; 492 pages.

Jones, Capers; Estimating Software Costs; McGraw Hill, New York; 2007; ISBN 13-978-0-07-148300-1.

Jones, Capers; Software Assessments, Benchmarks, and Best Practices; Addison Wesley Longman, Boston, MA; ISBN 0-201-48542-7; 2000; 657 pages.

Jones, Capers: "Sizing Up Software;" Scientific American Magazine, Volume 279, No. 6, December 1998; pages 104-111.

Jones, Capers; Conflict and Litigation Between Software Clients and Developers; Software Productivity Research, Inc.; Narragansett, RI; 2008; 45 pages.

Jones, Capers; "Preventing Software Failure: Problems Noted in Breach of Contract Litigation"; Capers Jones & Associates, Narragansett, RI; 2008; 25 pages.

Kan, Stephen H.; Metrics and Models in Software Quality Engineering, 2nd edition; Addison Wesley Longman, Boston, MA; ISBN 0-201-72915-6; 2003; 528 pages.

Pressman, Roger; Software Engineering – A Practitioner's Approach; McGraw Hill, NY; 6th edition, 2005; ISBN 0-07-285318-2.

Radice, Ronald A.; High Qualitiy Low Cost Software Inspections; Paradoxicon Publishingl Andover, MA; ISBN 0-9645913-1-6; 2002; 479 pages.

Wiegers, Karl E.; Peer Reviews in Software – A Practical Guide; Addison Wesley Longman, Boston, MA; ISBN 0-201-73485-0; 2002; 232 pages.

Yourdon, Ed; Death March - The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects; Prentice Hall PTR, Upper Saddle River, NJ; ISBN 0-13-748310-4; 1997; 218 pages.