Overcoming the NAH Syndrome for Inspection Deployment

Pankaj Jalote
M. Haragopal
Infosys Technologies Limited
Electronics City
Bangalore - 561 229; India
91-80-852 0187
jalote@inf.com

ABSTRACT: Despite considerable evidence to show that inspections can help reduce costs and improve quality, inspections are not widely deployed in the software industry. One of the likely reasons for this is the "not applicable here (NAH)" syndrome - developers and managers believe that in their environment, inspections will not provide the benefits seen by other organizations. One of the big challenges for deploying inspections is to overcome this syndrome. In this report, we describe two experiments that can be conducted, with little effort, in an organization to obtain data from the organization to build a case for inspections. By conducting one of these experiments, we were able to effectively overcome the NAH syndrome in our organization - many developers and managers are now ready to try inspections in their projects. Though the purpose of the experiment was to overcome the syndrome, the data from the experiment also shows how code inspections compare with unit testing in terms of defect detection capability, and the effect of inspections on the overall cost of development.

1 INTRODUCTION

Inspections were introduced over two decades ago by M. Fagan [Fag76, Fag86]. Since then, data has been collected showing the benefits of inspections, both in quality and cost [Rus91, Gil94, Gra94, Wel93]. Many experiments have been conducted to study the effect of various factors on the effectiveness of inspections (e.g. [Joh97, Por95, Por95a, Por97, Sea97, Ste97]). Different variations of the inspection process have been proposed to make inspection more effective (e.g. [Kni93, Mas93, Ste97, Vot93]). Experiments have also been conducted to study the use of the Web technology for inspections [Per97]. The software engineering and the process community is so convinced of the benefits of inspections, that inspections are a part of the Software Engineering Institute's (SEI) capability maturity model, CMM [Hum89, Pau93], in which they form a separate Key Process Area (KPA). It is now widely believed that inspection is one of the best technologies for improving quality and reducing costs. Despite all this, and the presence of many consultants trying to preach and spread inspections among software developing organizations, inspections are still not deployed in most of the organizations. A report by the SEI indicates that only 22% of the software organizations deploy some form of inspections [Kit93].

So, the situation is that while software development organizations are always in need for

methods to improve quality and productivity, and while much of the published work about inspections claim that inspections improve both quality and productivity, still inspections are not widely deployed in software organizations. Why does this seemingly paradoxical situation exist? Though there can be many reasons for it, one likely reason is the "not applicable here (NAH)" syndrome. That is, most organizations believe that inspections are all right for IBM FSD, HP, or other places, but it is not applicable in their context as their business is different. In other words, people generally believe the published data, but do not believe that in their organization inspections will show results similar to the ones found by others. The lurking suspicion is that (in their context) inspections will only add to cost and will not show sufficient reduction in downstream testing costs to have an overall decreasing effect on cost.

If inspections are to be deployed in an organization, then the NAH syndrome has to be overcome - both the managers and the developers have to be shown that even in their context, inspections can provide benefits. By definition, data from other organizations cannot be used to overcome the NAH syndrome. The only way to overcome the NAH syndrome is to get data from within the organization itself to build a case for inspections. As the organization is not deploying inspections, and people are not fully in favor of inspection deployment, this data will have to be obtained by conducting some limited experiments. For this, an experimental setup is needed that can be quickly deployed in real-life scenarios to evaluate the suitability of inspections in the organization. In fairness, such an experiment cannot be conducted to "prove that inspections are useful" but to actually evaluate the suitability of inspections as a technique to improve quality and/or productivity.

In this report we describe two simple experiments that can be used for this purpose. These experiments can be performed in a short duration in an organization and the data from the experiments can be used to evaluate the suitability of inspections. If the data from the experiments supports inspections, then the data from the experiments can be used to evangelize inspections throughout the organization. As the data is from within the organization, and from actual projects, building a case using this data, along with published data from other organizations across the world, becomes a considerably simpler task. And for the "doubting Toms", such a case is much easier to accept.

We describe the experiments and our experience in using one of them in our organization. Though the main purpose of these experiments is to build a case for inspections in an organization, the data from deploying the experiment in our organization also gives a somewhat different and interesting view of effectiveness of code inspections.

The paper is organized as follows. In the next section we first give some general requirements for experiments that are to be used for overcoming the NAH syndrome, and propose two simple experiments for this purpose. Then we describe how we conducted the first experiment in our context, and present the data obtained during the experiment. Then we briefly describe the effect on the NAH syndrome of the experiment.

2 EXPERIMENT DESIGN

We decided to focus on code inspections as the coding activity always has a formal output (i.e. code), coding is something that developers relate to more, and coding is usually the source of most number of errors. Historically also, inspections started with code, and were later extended to design, requirements, test plans, etc. Once a strong case can be built for code inspections, and they can be deployed, then the advantages from inspections will themselves build a case for other inspections later on.

There are a few key requirements for any experiment that is to be used to overcome the NAH syndrome. First, as building a case is the main purpose of the experiment, it is essential that the experiments be performed on real projects from within the organization whose NAH syndrome is being tackled, rather than on practice exercises. Second, it is extremely important that experiments are easy to execute

(i.e. they do not consume too much effort), otherwise finding volunteers will be hard. Third, as the goal is to counter the psychology of the NAH syndrome, a few data points may be enough to convince the developers and managers, that inspections are a useful and cost effective technique even in their context and that they should at least be tried. That is, an elaborate multi-team, multi-inspection experiment is not necessary to convince people to start trying inspections in earnest. If this level of conviction is reached, the war is almost won. Once people try to use inspections in earnest, then they can determine whether they are useful or not. Getting people to try in earnest is the hard part when NAH syndrome is at play. Finally, the data from the experiment should clearly quantify the effect on both quality and cost (i.e. effort), as both are important in deciding the usefulness of a technique.

In a typical software development process which does not deploy inspections, before the coding activity starts, the project is generally broken into "units", which are scheduled for coding. These units typically undergo some unit testing, before they are put together to form a system or a sub system. The system or the sub system then undergoes testing of its own (integration testing, system testing, etc.). If code inspections are deployed, generally a piece of code is inspected before unit testing. That is, with inspections, a unit will undergo inspections before unit testing of that unit is done. Due to this, generally, another step is added in the process. (Though there has been some situations where inspections, when fully mature, replace unit testing, in the start, the most likely scenario is that inspection will be an additional step before unit testing).

Two major factors driving any project (and a software organization) are cost (or effort) and quality. An organization, or a team of people doing a project, will only accept changes in processes if the changes can bring about a reduction in total effort or can catch more defects (i.e. fewer defects are present in the released software). And the case is much stronger, if the change is beneficial for both cost and quality. It is important to understand that frequently effort is a much more powerful driving force in a commercial software development setup and techniques that increase quality at a substantial increase in cost are not likely to be acceptable. Hence, if a case has to be built for code inspections through experiments, the experiments have to demonstrate a reduction in cost (without sacrificing quality), or improvement in quality with minor increase in cost, or that there is reduction in cost as well as improvement in quality. Clearly, the last scenario is the one that will build the strongest case.

2.1 Experiment 1

The purpose of the first experiment is to experimentally demonstrate how inspection compares with unit testing, as one of the main hindrances in accepting code inspection is that "we have unit testing, so why do we need code inspections; unit testing will catch all the defects inspection can hope to find". There are two objectives of this experiment. First, to see how the defect detection capabilities of unit testing and inspection compare with each other in the context of the organization in which the experiment is to be conducted. Second, to study the impact of inspections on the overall cost of development. The surest way to compare the defect detection capability of the two approaches is to independently apply the two techniques on the same code and then compare the defects found by the two. This is what the experiment does. With some data about downstream testing effort, this type of experiment can also be used to study the effect on overall effort. The basic experiment steps are shown in the flow diagram shown in Figure 1.

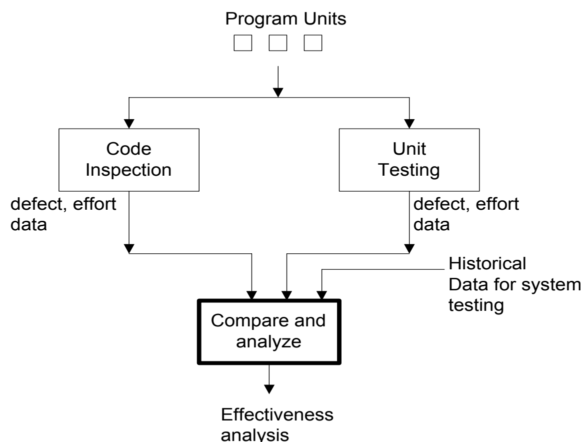

Figure 1: Steps in Experiment 1

For the experiment, first select a project that is reaching its coding phase and whose members are willing to try the experiment. Of course, a project will try an experiment only if it does not add substantially to its (usually already overloaded) work schedule. In the project, select some units at random. It is desirable to select a few groups of 3-5 units - then authors of code units in a group can form the inspection team (this helps in motivation as an inspector is not just inspecting someone else's code - in return, his own code also gets inspected).

For each unit, during the experiment, two independent paths are followed. In one, the unit is inspected, and in the other it is unit tested. Clearly, the people inspecting the code, and the people doing the unit testing should be different and should have no communication with each other. One way to organize the inspections is to form an inspection team of the authors of the code units in a group. This team inspects all the units in the group. We have to make sure that in each inspection, the author is not the moderator or the paraphraser. If this approach is followed, unit testing of the module will have to be done by someone other than the author. In other words, for the experiment, we need to have "independent unit testing". For both the paths, the effort spent, and the defects detected are recorded. Defects can also be classified to understand the impact of the nature of defects on the detectability of the two techniques. As inspections are not being regularly conducted in the organization, it is important to make sure that people have been properly trained in inspection and have done some exercises in inspection before they do the actual experiment. There is no similar requirement for unit testing as it is likely to be something people have experience with.

If the sets of defects found by the two approaches are not the same and one set is not a subset of the other, then we can claim that inspections do indeed find a different set of defects than unit testing. The nature and volume of these defects are then used to determine if a sufficiently strong case can be built for inspection as far as defect detection is concerned. Defects detected per person-day, defects detected per KLOC are other measures that can be used to compare the two techniques in their defect detection capability. In general, it should not be too hard to show that strictly in terms of defect detection, adding inspections will be beneficial and more defects will be caught by introducing inspections.

Understanding the impact on cost is harder (and where most doubts exist). For cost, we have to evaluate the effect on overall cost of the project if inspections are introduced as an extra step. This has to be estimated based on past data for system testing. Suppose, for a unit, inspection finds m defects, out of which n are ones that unit testing did not find. The actual inspection cost has been recorded. Now, we have to see how much cost saving will result by having detected these n extra defects in inspection. One way to estimate this is to assume that the defects that unit testing did not detect but inspection did, will be detected and fixed later during system and acceptance testing. By using the average effort for defect fixing for the organization, or for similar projects, we can estimate the saving that will be achieved later during system testing. If inspection were done before unit testing, then during unit testing, fewer defects will be detected and fixed (with the same set of test cases). Effect on reduction in unit testing cost can be estimated by using the defect detection rate in unit testing and the number of common defects that were found both by

inspections and unit testing. The sum of these two savings is the estimated savings later in the process, if inspection is used. This can be compared with the actual inspection effort to see how much overall saving (or additional cost) accrues by adding code inspections. This approach can be generalized by assuming a distribution for detection of defects found in inspection, but not in unit testing, among later stages, and using the average cost of fixing a defect at each stage.

It should be clear that this experiment is easy to conduct in almost any situation. Furthermore, the cost is quite low - the additional effort is the effort for conducting the inspections on the chosen units (this effort also is not strictly "additional" as it saves later testing costs and detects extra defects), plus the cost of analysis (which does not affect the project). The data from this experiment, if it is favorable for inspections, should be sufficient to convince developers and managers to at least start trying inspections.

2.2 Experiment 2

The second experiment is also one that can be done on a live project to study the impact of inspections on cost and quality. Unlike the previous experiment, this has no redundant activities and should actually reduce the overall development cost of the project on which the experiment is being executed. However, this experiment spans the entire life cycle of the project and analysis can be done only after the project is finished, unlike the previous experiment where analysis can be done after the units have been inspected and unit tested.

A project with multiple programming units is selected for this experiment. After the design is done, and the programming units defined, some units are chosen randomly to undergo formal code inspection, followed by unit testing. Other units follow the regular approach of going through unit testing. The sizes of the units are also recorded. During later testing stages (integration, system, acceptance, etc.), the defects found are attributed to the programming units in which they are found, and the total effort in testing is recorded. Once the testing effort and defect data is available, we can analyze the effect of inspections on quality and cost.

The impact of inspection on quality can be easily understood by looking at the defect rate (say, per KLOC) during later testing phases for the units that were inspected and the defect rate for the units that were not inspected. In general, the data is likely to show that defect rates during system testing and acceptance testing are lower in the modules that have undergone inspection before unit testing.

The cost benefit analysis can be done as follows. We allocate the effort in later testing stages (i.e. integration and system testing) among units in the ratio of number of defects attributed to the units. That is, we consider testing (and debugging) as the activity that is done to identify and remove defects, and assume that its effort increases in proportion to the increase in the number of defects. Hence, we attribute the later testing effort to the units in ratio of the defects they contributed. This gives us the cost incurred in later testing stages for a unit. More refined and elaborate cost distribution models can be built, if needed. However, this simple and "fair" cost allocation approach should serve the purpose in most cases. The cost of unit testing of a unit is already known. Hence, once the cost of system and integration testing is distributed, we know the "total" cost of downstream testing for each unit. The inspection cost for units that underwent inspections is already known.

The basic case for inspection is that it catches defects early, thereby reducing the costly testing and rework effort later. And it is generally believed and said that the longer a defect stays in the system, the more it costs to remove it. That is why identifying and removing defects early is considered advantageous. If this is true in this project, then we should find that the cost of inspection is lesser than the cost saved in defect removal in later testing stages. To check the validity of this hypothesis and build a case for the cost effectiveness of inspections, we find out the cost per KLOC of all testing stages for units

that were not inspected and the cost per KLOC for the units that were inspected using the approach mentioned above. As fewer defects are likely to be found in later testing stages in units that were inspected, the difference between the two testing costs (per KLOC) will give us the cost savings achieved due to inspections. This difference should be larger than the inspection cost per KLOC, if inspections are indeed cost-effective in this project. That is, if inspections are cost effective, then the cost per KLOC for conducting inspection and testing is lesser than cost per KLOC of performing just testing without inspections. The actual data about savings can then be used to build a case for inspections and overcome the NAH syndrome.

3 DATA FROM DEPLOYMENT OF EXPERIMENT

The banking unit of our organization has over 150 software engineers and one major banking product that is continuously upgraded to include new features. A typical release cycle is of about 4 months duration. During preparation of a release, two type of changes are done to the software. One is to fix the defects found (in the field or otherwise), that is, to fix the software trouble reports (STRs). The other is to implement enhancements to the product, called the software enhancement requests (SERs), which are decided by the steering group giving direction to the product.

It was noticed that during a development cycle, about 40% of the effort was spent in implementing the STRs. That is, the developers in the banking unit were spending 40% of their time fixing defects that were introduced in previous versions and were not removed. The need to improve development of SERs was very clear - if the implementation of SERs was of high quality, there will be fewer defects to fix in later releases.

The basic development process is very heavily coding and testing oriented. SERs are assigned to developers, who implement them and then do some self testing. Then they are unit tested. Once all the SERs that had to be implemented in a release are done, system testing is done by the test group. After that the product is released to some Beta sites.

It was clear to us that inspections have a great potential in this context to reduce the number of defects we deliver. However, as each development cycle was on a very tight schedule, there was resistance in “adding” inspections as a process step as it was feared it will add to cost without substantially improving quality. And as perhaps in most other organizations that do not deploy inspections, published data from industry was viewed as “not applicable here” or with some doubt and skepticism. In short, the NAH syndrome was very much present. It also became clear to us that the main problem in deploying inspections was not training of people but to counter this mind-set of the NAH syndrome.

We decided to conduct experiment 1 first, as it can be completed quickly. For the experiment, we selected 6 SERs belonging to two different domains (the banking product has been divided into about 8 domains). Six developers, each with an experience of at-least 2 years, were assigned one SER each. These six developers were first trained in the inspection process, and for practice they were given the implementation of one earlier SER which had some defects seeded in it, to inspect. Once the developers were trained, they were given the specifications of the SER assigned to them. Each developer was assigned one SER to implement. The set was divided into two groups of 3 each. Each group formed an inspection team (the minimum size of an inspection team can be 3).

Each developer was asked to implement the SER, compile his code and do some self test, before submitting it. Once submitted, as described earlier, it went through two independent paths - inspections and unit testing. For inspections, two groups of 3 inspectors was formed, each consisting of authors of three SERs. During the experiment, an inspection group inspected the code for the 3 SERs developed by the members of the group. In each inspection, it was made sure that the author is an inspector only and not the moderator or the paraphraser. Standard forms were used to collect defect and effort data for

the individual inspection as well as the inspection meeting. In parallel, the SERs were unit tested independently by the module leader for the domain to which the SER belonged. This module leader was not in any inspection team and did not interact with any of the inspectors. The sizes of the different SERs, the total effort and the number of defects found in the two paths the SER goes through are given in Table 1.

		Inspections		Unit Testing	
SER	Size (in LOC)	Total Effort (in Hr.)	Total No. of Defects	Total Effort (in Hr.)	Total No. of Defects
1	968	8.0	8	2.0	4
2	432	5.0	8	1.5	3
3	85	4.0	4	1.5	1
4	667	6.5	26	1.5	7
5	50	1.5	3	1.5	0
6	408	2.5	5	2.5	5
TOTAL	2610	27.5	54	10.5	20

Table 1: Effort and Defect Data

It is clear from the table that through the inspection route, more defects were detected as compared to the unit testing route. And this was consistently true for all the SERs. Overall, inspections caught about 2.5 times as many defects as unit testing did. However, inspections also consumed more effort as compared to testing, largely because inspection is a group activity while unit testing is a one-person activity. However, if we look at the number of defects detected per person-hour, we see that inspection and unit testing are similar – both detecting about 1.9 defects per person-hour. Now let us look at the nature of the defects found by the two approaches. This is shown in Table 2.

This data shows that almost in all categories inspections caught more defects than unit testing, particularly for categories which related to quality attributes like "maintainability", "portability", etc., (this is to be expected as testing generally focuses on errors in functionality). However, the data also shows that even in logic and interface defects (which unit testing focuses on), inspections do better than unit testing. This data was an eye-opener for many developers. They did not expect more logic defects to be caught during inspections. From this data, the case for adding inspections to improve the error detection capability was abundantly clear and convincing. The data further shows that the defects that were found both by inspections and unit testing (i.e. the "common defects") are not too many - only a total of 12 defects were common to both unit testing and inspections. This can be used to strengthen the case that unit testing and inspections are complementary and both should be deployed if defects are to be caught early.

Defect Type	Inspections	Unit Testing	Common Defects
Data	3	1	0
Function	4	2	0
Interface	14	11	7
Logic	12	5	4
Maintainability	11	0	0
Portability	5	0	0
Others	5	1	1
Total	**54**	**20**	**12**

Table 2: Defect Distribution

This data, along with the data about average cost of identifying and fixing a defect in system testing, can be used to do the cost analysis also. As the number of common defects is low (which itself is a good enough reason to add inspection as a step before unit testing), we assume that the reduction in effort of unit testing due to inspection will be minimal (this is the worst case for inspections). From past experience and data we know that during system testing, it takes about 4 person hours (about 8

times the per defect cost of unit testing) to identify and remove a defect. And if a defect goes past system testing, it takes about 2 person-days (17 person hours) to identify and remove a defect (this data is for identifying and fixing the defect and does not include the fixed cost of testing).

Testing will generally not catch maintainability and portability type defects. We assume that all the logic, interface, function, and data defects that are not caught by unit testing are caught later. The number of such defects (after eliminating common defects, which are also caught by unit testing) is 3 + 4 + (14 - 7) + (12 - 4) = 22. Assuming that all the defects are caught in system testing, from our data we can say that if no inspections are done, then during system testing an additional 22 defects will have to be detected and fixed. That is, the system testing cost will increase by 22 * 4 = 88 hr, or about 11 person-days. This is the "most benign" case - the defects are caught before the software is delivered. If we assume that about 25% of these defects will slip by system testing and will be caught later, the additional cost of system testing is then 0.75 * 22 * 4 = 66 hr, or about 9.5 person days, and additional cost of fixing defects found later is 0.25 * 22 * 2 = 11 person-days. That is, an additional 9.5 + 11 = 20.5 person-days be spent in fixing the extra defects, if no inspections are done. In other words, the cost saving due to inspections is 11 person days if all defects are caught in system testing, and 20.5 person days if 25% of the defects are not caught in system testing. And the cost of inspections, due to which these savings have been obtained, is about 3.5 person days. The case is very clear - if we spend 1 additional day in code inspection, we can expect to save about 3 - 6 days in defect fixing later in the development cycle.

The computation above gives estimates only for direct savings in testing and bug fixing in the later part of the same development cycle. In addition to this, there are other savings in the future (i.e. in later cycles) as inspections catch other quality defects (e.g. maintainability, portability, etc.). These may not immediately affect the working of the software, but generally do add extra work in future development cycles when code has to be ported or changed. However, we cannot quantify these benefits. These are over-and-above the direct and immediate benefits in rework that we can estimate. Of course, there are other long-term benefits in terms of learning that comes from inspection (developers inspecting others code learn from others; developers having their code inspected learn to avoid similar mistakes in future). This also we are not able to quantify. However, just by the saving on testing effort, which we can estimate, we can build a case for introducing inspections.

4 IMPACT OF THE EXPERIMENT

We were able to conduct the experiment, whose data we have given in the previous section, within two weeks. The impact of the experiment was very substantial on the organization. For some time the Software Engineering Process Group (SEPG) has been trying to deploy formal inspections in the organization. But, the resistance was quite stiff. And in the banking unit, due to the schedule pressure, developers were just not willing to believe that examining code written by others in a structured manner can help identify more defects and help save costs.

With the results of the experiment, a sea change has come in the scene. The results of the experiment were enough to convince developers and managers alike that inspections need to be tried. The data from the experiment also indicated that the benefits of inspections are lesser if the code is simple or small (in smaller size SERs, the benefit was not much). Using this, a policy decision was taken to classify the SERs in three categories - simple, medium, and complex, and consider formal inspections for all the complex modules.

So, overall, the climate for inspections changed considerably when the data from the inspections was presented to the developers and managers. The NAH syndrome was successfully overcome!

To push inspections further, a working group was formed to look at suitability of inspections in other service oriented projects. In the training module that is being used for the rest of the

organization, this is the main case study we present. Again, once the case study is presented by people who were part of the study, the acceptance is generally very high and the questions regarding the nature of the project, schedule pressure, capability of people, etc., which are generally used to doubt data from other organizations, are not raised.

Overall, the effect of the experiment has been very positive in countering the NAH syndrome. The experiment has fully achieved its purpose. Now, the SEPG, and the working group for peer reviews, are tackling the main problem of how to train people and how to institute inspections on a company-wide scale, when a host of logistic issues also come up. In other words, now these groups are tackling the technical and logistic issues relating to inspections and not fighting a psychological battle against closed minds. Within a three month period the large banking unit has moved from no inspections to wanting to inspect all complex modules. And this change has not come in a top-down manner. Rather there is a general acceptance by the developers to use inspections. This is a big help when deploying inspections - they don't have to be forced upon people, but the people are ready to employ them. Only the necessary changes have to be made to policies and processes, and relevant training has to be given.

5 CONCLUSIONS

Software inspections were proposed two decades ago. Since then, a wealth of information has been collected about effectiveness of inspections in improving quality and reducing cost. Despite the presence of over two decades of positive experience, inspections are not widely used in the software industry. A likely reason behind this resistance to deploy inspections is the "not applicable here (NAH)" syndrome - developers and managers of a company frequently feel that though inspections may be useful in some other organization's context, they are not suitable for their context. The basic suspicion is that in their context inspections will add to cost by adding another step in the process.

If inspections have to be deployed in an organization, then this NAH syndrome has to be overcome. The basis of the existence of the NAH syndrome is lack of data from within the organization. Hence, to overcome this, some data from within the organization has to be obtained. The best way of getting this data is to conduct some limited experiments on real-life projects in the organization and then use the data from the experiments to build a case for inspections. As people in the organization are skeptical about inspections, it is important that the experiments be such that they are easy to conduct, do not consume too much effort, and clearly show the effect of inspections on both cost and quality.

In this paper we have proposed two simple experiments, data from which can be used to build a case to fight the NAH syndrome. In the first experiment, some units of a project go through two independent paths - in one, the units are inspected and in the other they are unit tested. This experiment can be used to compare the effectiveness of unit testing and inspections. If inspections can be shown to catch different defects than unit testing, then it can be argued that having inspections will help improve quality. The effect on overall cost of the project can also be estimated through this experiment, if the average cost of removing a defect in later testing stages is known. In the second experiment, some units of a project are randomly selected. These units undergo inspections, while the rest of them don't. The defects found in the later testing stages are attributed to the units. Using effort data, cost per KLOC of testing and defect fixing for units that were unit tested and units that were also inspected can be determined. This can then be used to understand the impact of inspections on overall development cost. The first experiment can be conducted in a short duration, but may have some extra overhead. The second experiment does not have any redundant activities (i.e. which do not directly contribute to the project), but the experiment is completed only after the project finishes.

We conducted the first experiment in our Banking unit. We selected 6 program units to undergo the two paths. The data clearly showed that

inspections found more defects than unit testing for each of the unit. Overall, inspections found about 2.5 times the number of defects that unit testing did. However, inspections also consumed about 2.5 times more effort than unit testing. The nature of defects showed that inspections found more defects in all defect categories, including logic, data, and interface, and found a lot more defects in other areas like maintainability, portability, etc. The number of common defects were also small. That is, the number of defects found both by unit testing and inspections were not large. This clearly showed that the two approaches are actually complementary. Using the average cost of defect identification and removal in system testing and assuming that most of the defects that inspections found but unit testing did not will be found in system testing, we did the cost effectiveness analysis. The analysis showed that for each day spent in inspections, we saved 3-6 days of effort in defect removal after system testing.

Overall, the impact of the data on the developers and managers was tremendous. A sea change has occurred in the attitude of people. And now, in the banking group, a policy of inspecting all complex units is being considered. So, in a few months, from resistance to inspections we have been able to take the unit to a stage where they are excited about inspections and are formulating policies for inspections. The effect has been very positive on the rest of the organization also, and many groups now want to try inspections. In the training we give for inspections, the data from the experiments form the main "selling point", and it does a good job of selling inspections.

We are currently planning to execute the second experiment also to better understand the impact of inspections on quality and cost on different type of units. Experiments are also being conducted in other parts of our organization to study the effectiveness in their context and on different work products. We believe that such experiments can become an invaluable tool in the SEPG of an organization.

6 REFERENCES

[Fag76] M. E. Fagan, "Design and code inspections to reduce errors in program development", *IBM System Journal,* (3):182-211, 1976.

[Fag86] M. E. Fagan, "Advances in software inspections", *IEEE Transactions on Software Engineering,* SE-12 (7): 744-751, July 1986.

[Gil94] T. Gilb and D. Graham, *Software Inspections,* Addison-Wesley, 1994.

[Gra94] R. B. Grady and T. V. Slack, "Key lessons learned in achieving widespread inspection use", *IEEE Software,* pp. 48-57, July 1994.

[Hum89] W. E. Humphrey, *Managing the software process,* Addison-Wesley, 1989.

[Joh97] P. M. Johnson and D. Tjahjono, "Assessing software review meetings: a controlled experimental study using CSRS", *Proc. 19th Int. Conf. on Software Engg.*, pp. 118, 127, Boston, 1997.

[Kitt93] D. H. Kitson and S. M. Masters, "An analysis of SEI software process assessment results: 1987-1991", *Proc. 15th Int. Conference on Sofware Engineering,* Baltimore, Maryland, 1993, pp. 68-77.

[Kni93] J. C. Knight and E. A. Myers, "An improved inspection technique", *Communications of the ACM,* Vol 36:11, pp. 51-61, Nov. 1993.

[Mas93] V. Mashayekhi et. al., "Distributed collaborative software inspection", *IEEE Software,* pp. 66-75, September 1993.

[Pau93] M. C. Paulk et. al., Capability maturity model for software, version 1.1, Technical Report ESC-TR-93-177, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Feb 1993.

[Per97] J. M. Perpich et. al., "Anywhere, anytime code inspections: using the web to remove inspection bottlenecks in large-scale software development", *Proceedings of 19th Int. Conf. on Software Engg.*, pp. 14-21, Boston, 1997.

[Por95] A. Porter, L. Votta, and V. Basili, "Comparing detection methods for software

requirements inspections: a replicated experiment", *IEEE Tran. on Software Engg.,* 21(6), June 1995.

[Por95a] A. Porter, L. G. Votta, H. P. Siy, C. A. Toman, "An experiment to assess the cost-benefits of code inspections in large scale software development" *Third Symp. on the Foundations of Sw. Engg.,* Washington, DC, Oct. 1995.

[Por97] A. A. Porter, H. P. Siy and L. G. Votta, "Understanding the effects of developer activities on inspection interval", *Proc. 19th Int. Conf. on Software Engg.*, pp. 128-138, Boston, 1997.

[Rus91] G. W. Russell, "Experience with inspection in ultralarge scale developments", *IEEE Software,* Jan. 1991.

[Sea97] C. B. Seaman and V. R. Basili, "An empirical study of communication in code inspections", *Proc. 19th Int. Conf. on Software Engg.*, pp. 96-106, Boston, 1997.

[Ste97] M. Stein et. al., "A case study of distributed, asynchronous software inspections", *Proc. 19th Int. Conf. on Software Engg.*, pp. 107-117, Boston, 1997.

[Vot93] L. G. Votta, "Does every inspection need a meeting?", *Proc. of the ACM SIGSOFT Symp. on Foundations of Software Engg,* Dec. 1993.

[Wel93] E. F. Weller, "Lessons learned from three years of inspection data", *IEEE Software,* pp. 38-53, Sept. 1993.