

Project Icebergs: - Where is the Missing 70%?

Do you ever get the feeling that too much work goes into your projects for what comes out? Have you ever tried to put your finger on just “where all the work goes”? Is it a case, like the iceberg, that we need this submerged and invisible activity because without it we would just sink?

In this short article I want to discuss my belief that we do more work than is necessary to deliver a successful project and that large amounts of work never contribute to the final deliverables and that we can avoid this.

Our experiences over a range of system and application projects have enabled us to implement some tools and techniques which can be applied quite generally. It is only a matter of wanting it to happen.

The first thing to say is that I am not claiming that any staff deliberately waste time on useless tasks. What I am referring to in the Iceberg analogy is large amounts of wasted activity, large enough to account for a discrepancy of perhaps 60% to 70% in terms of work performed against work required for delivery. These project “Black Holes” will provide opportunities for vastly improved performance if they can be identified and closed.

In the many projects in which I have been involved I have always practised “management by walkabout”, as one way of keeping a finger on the project pulse. This allowed me to develop an awareness of project hotspots, those areas which were forever causing problems, which continued to surface and resurface over the project lifetime. Often this would lead to a local redesign and reimplementation. Was this the End of Problem? Not really. It is always useful to ask just how the problem got there in the first place and how ultimately it was fixed. We came up with several points.

- 1) The *original* source of the problem, directly or indirectly, was lack of clarity in specification and consequent lack of understanding.
- 2) The visible hotspots were just a small part of the total. On reflection it seemed that the whole thing should have been glowing! It was covered in hotspots.
- 3) The fixes to unpublicised hotspots (found and not escalated to management) were usually local and caused repeated redesign-code-test cycles. The publicised hotspots were usually fixed via global redesign-code-test cycles. These cycles were most of my missing Project!

The consequence of each one of the hotspots was a loss of productivity. Together they contributed to a background of confusion and ultimately poor motivation.

Amongst the key effects we identified were:

1) Procrastination

Essential work was postponed until the actual requirements had “settled down”. This would lead to a kind of blockage in process as individuals postponed work needed by others who then attempted to get around the problem in ad hoc ways (including postponing their own pieces of work).

2) Errors in Understanding

These errors were frequently uncovered during Integration testing, or later. The cost of the rework to fix such lately discovered errors is very high and has a considerable effect on the rest of the project.

3) Treading Water

Many project artefacts needed substantial rework to keep up with the latest understanding. Documents were going through substantial revisions, or even not being maintained, due to effort and schedule pressure. Much heat, but not much light was being generated.

4) Low Morale

Confusion and solution-avoidance lead to a despondent team and further poor performance.

Clearly these are all topics for a Project Manager. However, we also uncovered some other issues.

More of the Iceberg- The Unit Test Process

Another topic that became visible during the walkabouts was that programming staff seemed to be coding a great deal more than they were actually delivering. Given that I knew that our staff were pretty good programmers, there had to be something going on. Separating out the consequences of the hotspots mentioned above still left a lot of unaccounted work.

By digging around what we uncovered were enormous activities at unit code level to test the actual code. Frequently the activity of testing was at least equivalent to the effort of coding, and although we had considerable investments in coding standards and code-deliverable specifications, we had none at all on unit testing! It also turned out that most tests were failing due to errors in the test code! We had no test-the-test process at all. What we had found was a thinly identified area of work performed at considerable cost almost in private within the project.

We saw:

- 1) Significant work going into unit test implementation
- 2) Essentially non-reusable unit tests (lack of documentation, unclear dependencies, manual-labour intensive, etc.)
- 3) High failure rate of the unit –tests themselves
- 4) Lack of visibility of unit-tests (where were they, were they maintained, etc.).
- 5) Abandonment of the unit-test code following delivery of product.
- 6) Test code only able to be run by the individual developer of the code under test.

This seemed more like a cottage industry than the work of leading IT practitioners, which is what it should have been.

Even more worrying for current Project Managers, is that programmers who follow the modern paradigm of “code a bit, test a bit”, are likely to produce even larger amounts of ad-hoc low level test code. Further, the modern staged approach to software delivery is very likely to cause recompile/retest/re-release cycles, which will accumulate the inefficiencies which we uncovered. Clearly something has to change here.

Yet More

Sadly there is not time to discuss the many other areas of “lost work”. The issue is pervasive throughout a project and in general it can best be resolved by a similar scale of solution, involving all concerned with the project. For now however, I concentrate on the two topics, Hotspots and Unit Tests, which seem particularly relevant in current methodologies.

HOTSPOTS

By far the largest Black Hole in our projects is caused by “re-work”. That is, revisiting work which should have been completed, to correct, revise, and improve or whatever. The result is extra primary work and consequential secondary work as the revised document again progresses through the project for (re) sign-off and acceptance. The ripples spread out downstream as the consequences are absorbed. Here is an interesting statistic; changes made to product due to rework / fixing are *three times as likely to introduce further faults as was the original work*. Another statistic shows that 60% of follow-on changes introduced to remove faults, themselves introduce further faults.

This particular Black Hole is enormous and spreads over the entire project lifecycle. In situations where rework is the accepted mode of working, staff may feel that they are doing little more than wheel spinning. For the Project Manager, progress of the team becomes hard to measure as the team-ethos encourages revisiting of “closed” topics. The costs of finding and fixing faults, by project phase are given in the table below:

Relative Cost of Finding Defects

Who (Ref.)	In reqts	In design		In Coding			In Use
IBM			Prior to coding		Prior to test	During test	In field use
(Humphrey)		1.5	1	1.5	10	60	100
TRW					Develop. test (in)	Acceptance test (in)	Operation
(Boehm '81)	1	3-6		10	15-40	30-70	40-1000
IBM			Design reviews		Code Inspections	During machine test	
(Remus)			1		20	82	

You can see that the later defects are detected and fixed in the project process then the greater the costs. This is the core of the unnecessary rework that goes on, and the strategy for correction is to put in place a *method for forcing the detection of faults as early in the process as possible*. Traditional testing is positioned at almost exactly the wrong place to achieve this!

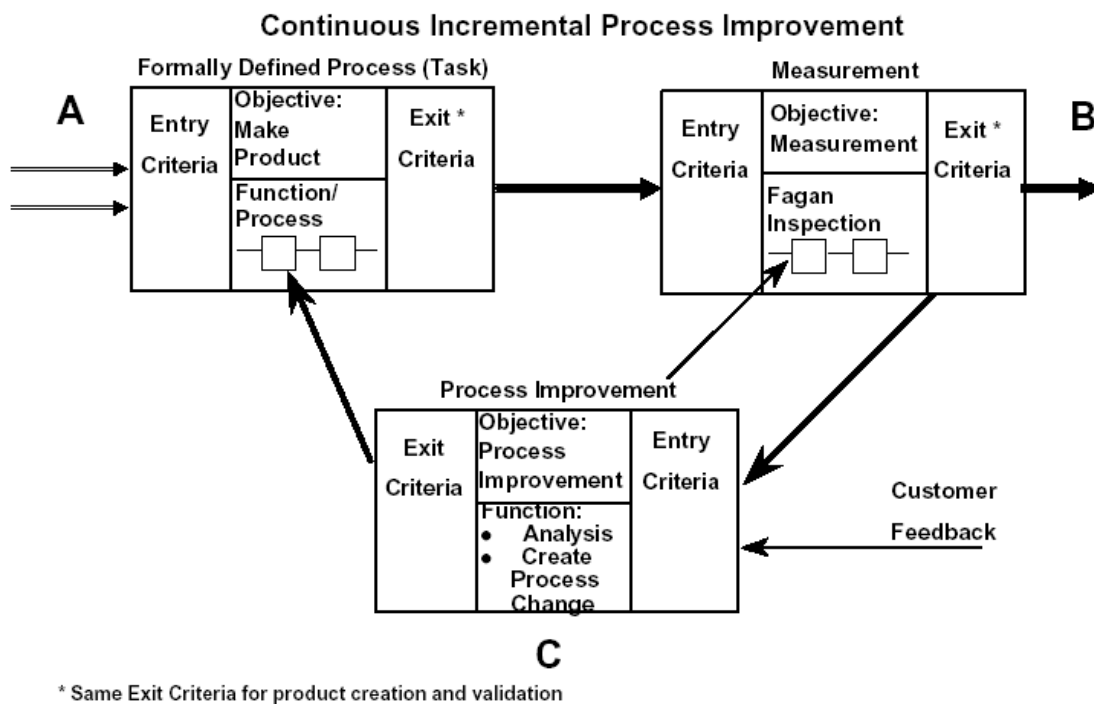
Now we should apply a caution here. These statistics, and there are many like these available in the industry, come from an era where the Waterfall lifecycle was in place and where downstream changes were not expected or wanted. In that era most of the lifecycle processes were supported by heavy investments in manual labour. To translate this to the modern day, because it is still applicable, we need to be a little more refined. We need to focus not on avoiding change per se, but on avoiding unplanned or uncontrolled change, where the associated costs risk spiralling out of control. To be successful in our projects we still need some pegs in the ground and we still need to apply our available skills in the best way possible.

Traditionally the most effective method to reduce the amount of uncontrolled change (IMHO) is the introduction of Fagan Inspections and the associated Fagan approach to projects. The latter is

important because Inspections introduced without the concomitant underlying prerequisites will not work. The Inspection process is covered in some detail in my course, but broadly speaking it allows the concentration of the best skills on particular topics in such a way that the result is highest quality product. The Inspection process is formal, but not costly, in the sense that it saves money if implemented properly. The rule should be that *any* project artefact can be Inspected. Whether it *needs* Inspection is a decision based on risk and importance of the artefact and indeed the methodology in place.

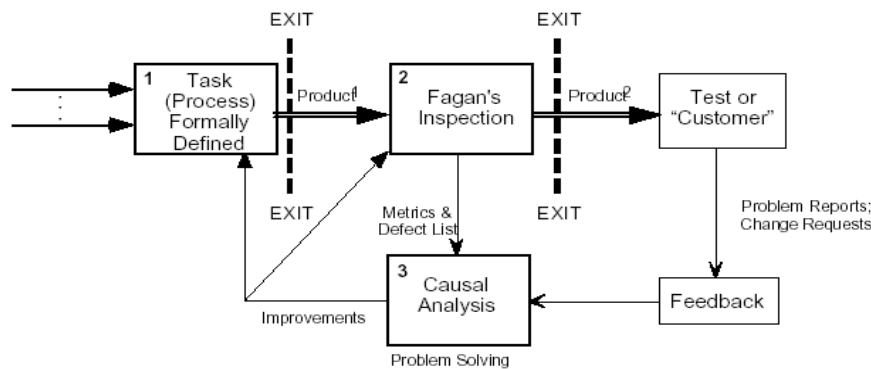
In modern project methodology Inspections are still present, but in a variety of ways, ranging from full-blown execution of Inspections (see below), to reviews, desk-checks and finally to the lowest level, pair-wise working (formerly known in my neck of the woods as the “Four Eyes Principle”). The common principle of applying multiple expert minds to a problem provides benefits throughout the lifecycle.

Introducing Fagan Inspection can have as big an impact as the execution of the Inspections themselves, because it makes clear management’s desire for quality product, superior performance and higher productivity. The diagram below shows the Fagan Inspection Cycle. It supports Continuous Incremental Process Improvement, is rather heavy, but can be watered down according to need.



There is no real conflict between current methodologies and the Fagan Inspection. It needs to be applied judiciously at key points in the lifecycle. For example, architectural decisions, Master Test Plan, the Project Plan, are all reasonable candidates for Inspections. In the various organisations that I have seen Inspections in place, the authors of documents to be Inspected welcomed the opportunity to be able to demonstrate the quality of their work as well as benefit from the opinions of their peers. Of course, if the execution of an Inspection is akin to an inquisition, it will never work properly. The fact that the Inspection technology dates back to the ‘70’s should be seen as reassuring. It is one of IT’s oldest success stories and should not be dismissed lightly.

Fagan Processes



* Same Exit Criteria for product creation and validation
Product¹: Product produced by process
Product²: Product with identified defects removed

This diagram shows Fagan Inspections applied to the entire Product Development Process. In this case feedback from the Customer is involved, and the result of this Inspection will probably identify several sub-processes to be further Inspected.

UNIT-TEST

One of my core rules in software technology is that everything is testable. Happily this is a common concept in modern methodologies too. The alternative to such a rule is unthinkable but I have certainly come across programmers who regard testing their code as a challenge for someone else to take up. I will make the rule (1) that all code must be testable, and extend the rule to state that (2) it must be testable within some generally accepted test framework. As a final extension, (3) I will say that code must be delivered together with its test framework, for easy go / no-go testing by other than the original developers.

I could go further, but this is already quite a chunk to bite off for some programmers. The question arises as to what to do with those (very few) pathological cases which are so embedded as to make testing extremely difficult or complex. For these cases, I will add a fourth rule, that such cases must embed their own validation code or specific hooks which can be activated to enable validation. This code then will ultimately embed its own self-check facilities. This idea can usefully be duplicated throughout a suite of programs, but that is another discussion.

We should examine these rules to see their consequences.

1) All code must be testable

Consequences are that clear definitions of the outputs for a given set of inputs are required. Probably a good idea! Also, the interfaces should be such that the test method is clear. This is usually the natural result of good programming practice. We could also include the rule that the code documentation should be extended to include the test procedures. Why not?

2) Code must be testable within some generally accepted Framework

Specifically, this is aimed at making testing simple (reduced learning curve) and ultimately automated. We want to encourage anyone to be able to run and rerun unit tests for their own confidence level, without needing to be the code expert. Applying this rule will encourage that.

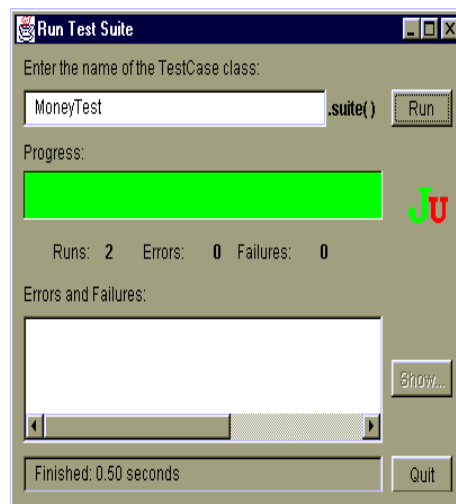
In Java, an example could be the use of JUnit as a simple harness, but a department-wide product would also be appropriate.

3) Code is delivered with its test Framework

This is a natural follow-on to (2). Once this is in place then acceptance of code for system integration can follow the successful (automated) execution of the code by its framework prior to inclusion in a build. In this case, the Framework will be running the tests in the new system environment and the execution will be by staff who were not involved in the original test execution. If it fails then the locality of the fault will be easy (or at least easier) to track.

The fourth rule goes rather further than these three, and assumes that code contains its own dynamic validation code, which can be invoked during start-up to check all is in order before full activation. I have found this to be useful (under the control of a macro-language) for embedding unit code regression testing, but it is possible that tools such as JUnit (in the Java world) probably reduce the need for this further extension.

JUnit is a particularly good example of a (simple) framework for testing at Unit Test level. Firstly it is free, but more importantly it picks up on the modern enthusiasm for the use of open, flexible solutions. JUnit is available as a free download, or integrated into IDEs such as JBuilder. As tools such as JUnit become more widely used, it should become quite normal to find my (3) rules in use. In fact, the idea of not implementing these rules in some way in a modern development shop becomes unthinkable. A sample of one of JUnit's interfaces (these are selectable) is shown below.



This shows the name of the Class under test. One test or a suite of tests can be executed by each run of JUnit.

The green progress bar shows the progression of the tests. It turns red when errors are found or assertions fail.

This window displays detailed error information

SUMMARY

I cannot claim to have found all of my missing project work, but I think a large proportion of it is now identified. The use of Fagan is important because it can be applied to any process to review its performance and improve upon it. Hence it is a generic Quality Improvement Tool and should be widely implemented and supported. Its integration into modern methodologies is possible and advantageous. The work on Unit Tests is the result of applying process inspection to identify flaws in the process and to put in place a new process to handle the defects. One point about any process

improvement is that you need metrics to measure the before and after. You might like to think about what metrics you would use for the above cases.

Our Software development processes are maturing. We are coming to a state now where delivery of working product is insufficient. We need to be able to deliver proof of working product and proof of quality processes which delivered the product. The methods described here lead towards this objective.

References:

Extreme Programming Explained, Beck, ISBN: 0-201-61641-6

The Complete Guide to Software Testing, Hetzel, ISBN: 0-00-383092-6

Fagan Inspections: Proprietary Courseware from Fagan Associates

JUnit Cookbook, Beck and Gamma, at <http://junit.sourceforge.net/doc/cookbook/cookbook.htm>

Refactoring: Improving The Design Of Existing Code, by Fowler, M. (Addison-Wesley, 1999)

About the Reviewer

George Brooke is a Technical Trainer / Consultant as well as being a PRINCE2 Approved Trainer. As Head of Software Development at Siemens Nixdorf he managed solutions development for the Financial, Retail, Hotel and Energy markets for UK/European departments and customers worldwide, from concept through to delivery and roll-out. He runs a training and consultancy company based in Cambridge UK focusing on OO design with UML, programming and implementation techniques and IT project management . Contact him at [George Brooke](#)