Embedded Agile Project by the Numbers With Newbies

Nancy Van Schooenderwoert
Agile Rules, http://www.agilerules.com
nancyv@agilerules.com

Abstract

There is a class of projects that can only be accomplished via agile practices due to their complexity and risks. It is rare for such a project to be staffed with newbies but it happened, and this paper tells the story. It is also rare for an agile project to have substantial hard numbers to tell its story but this one does. This paper presents detailed metrics from a three year long agile project where a newly formed development team produced a new embedded software product from scratch. The team experimented with various agile practices while recording data on bug rates, bug root causes, code size, schedule compliance, and labor expended. Although most of the team members lacked important technical skills for this work, they outperformed far more experienced teams. The natural learning environment and safety net that Agile provides allowed them to learn fast while keeping them from serious mistakes.

1. Introduction

The project in question was a joint venture between a multinational corporation and a major agriculture equipment manufacturer to produce a mobile spectrometer that could analyze grain. The Grain Monitoring System (GMS) would tell a farmer exactly how much protein, oil, or other constituent is present in their corn (or other grain) while it's being harvested.

This project was compelling enough to justify the risk of using developers who were less experienced than the company would have preferred. Staffing took place at the height of the dot com bubble, making it very difficult to recruit people with the preferred skills.

The team built the embedded software for the grain monitor and also built seven other associated Windows-based utility applications. These ranged in size between about 1,000 and 3,000 lines of code. The bulk of the work by far was the embedded application to control the grain monitor. That code base is the subject of this paper. The utility software was not built using agile team practices.

2. Project Setting and Challenges

The multinational corporation was made up of several commercial business lines but the division near Boston had mainly done defense work. The grain monitor project was an entry into commercial technology products for them. The project was ideally suited to agile methods due to these risks:

- Newly designed complex algorithms to compute the grain results
- Extremely low noise electronics necessary for the sensor input signals to be detectable
- A new prototype sensor being used to collect the grain's light spectrum
- Software and hardware had to be integrated with vehicle systems designed by the partner
- No existing microprocessor powerful enough to handle the initial algorithm
- The unit had to work accurately in extremes of temperature, vibration, and electrical noise

This is clearly the type of project that makes waterfall methods stall out completely. Even a seasoned embedded team could not have built this product using a traditional plan-driven approach.

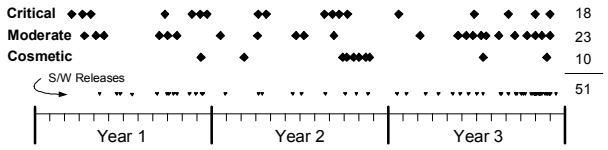

Figure 1. Project timeline showing actual releases and defects

The GMS team delivered this product after three years of development, having encountered a total of 51 defects during that time. The open bug list never had

more than two items at a time. Productivity was measured at almost 3 times the level for comparable embedded software teams. The first field test units were delivered approximately six months into development. After that point, the software team supported the other engineering disciplines while continuing to do software enhancements.

3. The Team

For several reasons it was difficult to get software developers with the desired experience levels. The main reason is that the labor market was tight at that point. Also our manager was obliged to use available internal people and train them where necessary.

The skills most needed were:

- C programming
- Multitasking experience
- Electronics background
- Firmware programming

The below figure gives an idea of the level of these skills necessary for the work, and how the team members' qualifications compared with that need.

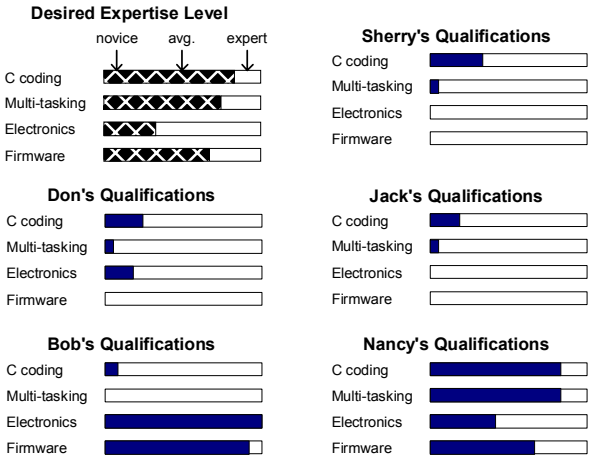

Figure 2. Team's experience levels in key skill areas

Windows programming was the main skill for three of the developers – Don, Sherry, and Jack (not their real names.) The project did call for creation of several Windows-based utility applications for use with the grain monitor. These allowed the creation and downloading of calibration tables, or helped in the testing of the units. The GMS embedded software was the only deliverable; the other code bases were custom tools.

3.1 Team's Context in the Organization

The GMS software team was one among several teams associated with the project at our company. There were small teams to build the hardware, the optics, develop the mathematical algorithms, and the mechanical design. This totaled approximately 35 people at the height of the project.

The software team also needed to coordinate their design with a small software group at the partner company, where they had designed a framework for all the software systems mounted on farm equipment. The GMS software had to behave as one node on the vehicle's CAN bus network. The arrangement between the two companies was structured as a joint venture.

3.2 The Technical Lead/ Coach

For better or worse, I was sure of one thing: I wanted to have hard numbers, not mere anecdotes, once all was said and done. My role was that of Software Technical Lead. I was expected to formulate and analyze the requirements, and assemble a team to carry out the work.

Throughout the project I was able to spend about half my time as a technical contributor on the project. The rest of the time was taken up by coordination and management duties. I did not have any formal authority over the team members; we all reported to the same manager. I also had no budget authority.

The Extreme Programming community has come to recognize several roles within agile teams. One is 'coach', someone who watches over the process and helps others play their parts within it. That role is a good description of what I did in addition to the technical lead duties.

4. What Was Measured

The team followed the IEEE guidelines on basic metrics. Those guidelines said that at a minimum, you should capture four items:

- Effort expended
- Size of the software built
- Schedule compliance
- Quality of the deliverable

Effort and schedule were already captured in project tracking documents. We could use tools to record code

size. It took extra effort to track defects thoroughly and create a root cause analysis for each one. Every incorrect or unexpected software behavior was considered a bug once it got past unit test. There was no attempt to catalog all the bugs caught in unit testing or earlier activities.

4.1. Parametric Estimating

Parametric estimating is a technique that relies on the existence of data from prior projects that are similar to the one being estimated. Several software estimation tools exist that use this technique, and we made use of one of them early in the project before any staffers were aboard.

Our managers made a detailed evaluation of the "Seer SEM" software estimation tool from Galorath Inc. in September of 1998. As part of that evaluation, the consultants at Galorath collected data on the GMS project in order to do an estimate of the embedded code to be built.

I took Galorath's breakdowns for hours and lines of code and found that their industry productivity data could be expressed as 1.2 ESLOC/hr. (ESLOC is Effective Source Lines Of Code, i.e. omitting comments and blank lines.) Their product uses a database of thousands of real-world projects, taking the "actuals" from real-time embedded projects comparable to ours. Therefore the figure of 1.2 ESLOC/hr had validity across the embedded software industry.

Project data was captured with a view to comparing the team's productivity with this, and any other software industry productivity figures that could be found.

5. What Happened

As staffers came onto the project in the first month, it was evident to all that they were not well equipped to deliver this project in the aggressive time span that management wanted. As team lead, I held a meeting with them and said I was prepared to teach them all they needed to know about real-time software, and operating systems, etc. provided they agreed to work as a real team and help each other. By "work as a real team" I meant supporting the team based practices I wanted to institute right at the start. These included:

- Collective code ownership
- Writing strong unit tests, and maintaining them
- A bug tracing system that could be always enabled within the code

The foundation necessary for all these was a coding standard. So I had them work that out *prior* to writing any production code. There was pressure to produce code immediately but I resisted that. It would undermine our ability to deliver consistently over time, because we didn't have the foundations in place yet.

The team also had to configure the tool set so that we'd have a fast code-compile-debug loop, with all compiler warnings turned up to the max. This was part of the safety net we constructed.

5.1 The First Iteration

The team delivered iteration 1 after a few months, and only two bugs were found in that software. Some of the utility software had also been built as part of that effort.

I had done the estimating using a figure of 2.5 ESLOC/hr based on my experience with similar projects where some of the team-based practices were in use. In order to see if we had achieved that, I did an analysis of the labor that went into iteration 1 to separate out the time that went only into the embedded software work. I included all the duration of that iteration, even the part when we were just doing the tool setup and working out a coding standard.

The result was that the team had produced 3.5 ESLOC/hr of tested, working code – almost 3 times the industry average according to the Galorath numbers. To be sure I had used their breakdowns correctly in my analysis, I spoke to one of the consultants from Galorath and got agreement that my analysis was valid.

5.2 Learning via Mistakes and Safety Net

5.2.1. Globals. We had to use a small number of globals to interface with programmed logic and other hardware. At one point early on the team started adding more global variables, and I tried to convince them that it was a bad practice. They believed the convenience of globals outweighed the seemingly abstract concerns I raised about reentrancy. I decided to let them use the globals awhile so they would see first hand the problems caused.

Our early releases were just single-threaded code because the commercial operating system we planned to use hadn't been purchased yet. When we got the OS and converted the code base (4,000 lines of C at that time) to using it, trouble started to appear. During integration test the system was halting unless interrupts were disabled. This gave me a chance to explain how that could happen due to a task switch in the middle of updating a global structure or variable.

The idea that certain kinds of problem simply could not be traced was new for all except Bob (who had plenty of hardware and firmware experience). This kind of problem had to be prevented by using semaphores. Once they had that idea down, we eliminated most globals but needed to keep some others. We divided those into atomic ones and non-atomic. For all the non-atomic globals we assigned semaphores, and changed the code to utilize them. Now they had an appreciation of the re-entrancy issue, and how to handle it. Better yet, it wasn't textbook learning.

Some people will say it would have been simpler for me to just outlaw globals instead of going through all these adventures. I think it was worthwhile because there is no substitute for the level of understanding they developed. If I had just made a decree then I'd have to police it eternally. This way the developers enforced the practices with far more energy than they ever would have if I'd simply ordered them to get rid of the globals.

5.2.2. Semaphores. We again had a bug that disappeared when interrupts were disabled!

The root cause of this bug was two-fold. We were releasing one semaphore twice, but the operating system had a bug that let the semaphore count go from 1 to 2 on the second release, when it should have given us an error return. (These were binary semaphores, not counting semaphores). We shouldn't have had a double release in our code, but the operating system also had a bug for allowing it.

The team fixed the task sequencing situation to prevent the second semaphore release from happening, but we needed to be sure we'd know if our code ever again tried to double-release a semaphore. We added to our safety net: we built a wrapper around all of our semaphore calls to the operating system to make sure we'd know promptly if the situation ever came up again.

5.2.3. Hardware Individuality. About a year into the project we had several GMS units ready to be tested out in farm fields. We prepared a release and thoroughly tested it on the GMS unit we had in our lab.

The problem was that we finally did a non-debug release (as we should have been doing long before this!). Though the non-debug build was completely tested, the test was done with just one unit. That unit's lamp current counts happened to be within the valid range. The range check was skipped for 'debug' since we never had a spec on what it should be. Certain other units' lamp current counts value did not pass this test. (Each unit had a special lamp to illuminate the grain for collecting spectra readings).

Testing on one set of hardware was not good enough. To be valid, the tests had to be done on all the field units. When it came time for full production, it had to be done at least on a statistically valid range of units.

Through this bug the team learned that debug builds, though convenient, can hide problems. They also learned that every piece of hardware is an individual. It's voltage or current readings can vary widely from other hardware but the software has to defend itself from hardware that is out of spec. Our software monitored many "test points" on the circuit board that were meant to tell us whether the hardware was working adequately. But we needed to run it on *all* the hardware, i.e. on each of the field units.

5.2.4. Memory Corruption. After we were regularly supplying units for field tests, an interesting bug was reported. A GMS unit would seem to be running but the data readings would start to remain the same at every sampling interval. The software log told us that the timing interval value was being over-written by a very large number, so instead of having a couple seconds between data samples it was set to hours.

The detective work that the team members had to go through to track down this problem was quite an education for them. As soon as I saw the incorrect value in the timing interval variable I knew memory was being corrupted, and I explained to them several ways that can happen. They systematically combed through the code to eliminate each possibility I outlined.

Their effort paid off. That time interval number was in a table right after the software log in memory. The software log was overwriting its bounds by one message length. Therefore, if a message was long, it corrupted the timing interval variable, but only after the software log message buffer filled up. The bug appeared only if the software log circular buffer was full and only if the last message before the "seam" was longer than so many bytes. This was a case where our unit test wasn't thorough enough, so we added a test for this problem.

5.2.5. Safety Net. Our safety net consisted of sets of unit tests for each module in the system, and also a run-time trouble logging system that was always enabled. It simply stored log messages in a circular buffer in RAM. Because it used very little memory and executed very fast, we could afford to have it always enabled. This avoided the problem of bugs that appear only when the log system is disabled.

This safety net was extremely effective. In three years of development, we were never stuck for more than a few hours troubleshooting a bug. Anyone coming onto the team could just take the unit tests and step through them in the debugger to learn the behavior of the code.

5.2.5. Code Reviews. Another practice that contributed to learning for everyone was holding code reviews by stepping through the code with a projector screen connected to our integration PC in our team room. It was an opportunity to discuss defensive coding practices, and raise questions and new ideas. I found that just talking about the code while looking at it together was quite valuable. It seemed that we caught the superficial bugs this way but missed the tougher ones. I'd say the code reviews were more of an educational tool than an effective bug prevention tool. More about this later.

5.2.6. Kludging the Code. Once upon a time I believed that I'd never stoop to putting a kludge in my code just to get something out the door. Reality intervened. When we had units out for field tests, the company was spending a lot to have people and equipment out in other countries. If a problem developed in the field, we had to do something about it immediately, even if the "something" wasn't graceful for the architecture.

I used these occasions to have a brainstorming session with the team. The rule was that the best idea wins, no matter who it comes from. "Best" meant the change we could be most certain would fix the problem on the first try. Usually my idea would win but not always. I deferred to others when their idea was clearly simpler, even when it was uglier for the architecture.

Whenever we kludged the code, we'd make it our top priority to get a proper fix into place promptly so that we could rip the kludge out.

I believe this practice of "let the best idea win" helped the less experienced folks see that they could be peers with the more experienced ones. Especially the fact that I lived by the rule too, seemed to make an impact. They became quite well seasoned embedded developers by participating in this project.

5.2.6. Leadership Lessons. I found that allowing developers to temporarily damage the code was a good teaching method. It was definitely better than banning certain practices to protect people from themselves. That doesn't encourage investigation or self-reliance. I made it a rule (for myself) never to tell anyone to do something just because I have more experience and I say so. I knew I'd never listen to such advice so I couldn't expect them to. Rather, I'd try to convince the others on the merits of my idea, and failing that, I had to let reality convince them.

Working within this constraint was often frustrating, and it was well over a year before I got comfortable with it. I had no authority to compel anyone to use a particular practice and in hindsight, that is a good thing.

5.3 Subsequent Iterations

The team used iterations varying from about a week to two months, but after the first year shorter iterations predominated.

5.3.1. Code Reviews Dropped. We eventually gave up code reviews in favor of discussions about what design approach a developer (or pair) would use for implementing a story. We could trust each other to follow the coding standard, and our unit tests were more effective for hunting bugs than our code reviews had been.

Throughout the project, as I did a root cause analysis of each bug, we'd modify our practices if we saw a pattern of problems occurring. One pattern that did occur was failure to do enough refactoring. That was the root cause of many bugs. Another recurring root cause was code review not being thorough enough, or bugs slipping past our unit tests. Overall, our defenses were quite good – we were averaging only 17 bugs a year.

5.3.2. Product Owner AWOL. At one point about mid way through the project, our business player (in XP's Planning Game) became very busy and because he was confident in our ability to deliver, he refused to participate in three consecutive releases, leaving us to just work from the product backlog list. I knew that this would lead to problems, and kept talking to him about it until he agreed to come back to the iteration planning meetings.

We didn't have any serious problems develop due to his absence for those iterations, but my concern was that he'd be quite upset if we made wrong assumptions without his inputs. He was relying heavily on us because we had been delivering reliably, and he wanted to instead pay more attention to other parts of the project that were not doing as well.

5.4 End Game

During the project our partner company went through merger with another farm equipment business. The new owners placed lower importance on our project.

Meanwhile our own company's situation had changed and they were more interested in pursuing military projects again. Both agreed to complete the GMS work to the point where it was ready for production and then license the technology to some other organization to pursue further. The product had undergone field trials all over the world, and was a big success with its intended customers.

All the known defects were systematically addressed in the final couple months, and then the development team was let go.

5.5 The Story in Numbers

Early in the project I persuaded the team members to voluntarily track their time so that we'd be able to tell when we should hire other developers or support people. Our time was tracked by the company (via weekly time sheets) but not broken out into categories. The labor categories we used are in the table below.

Labor Type	Description
Detailed Design	Design work at the feature level.
Code and CSU test	Unit testing and coding of functions.
CSC Integration and Test	Integration testing of GMS only.
Team Process Development	Team coordination, discussion of work procedures, etc.
S/W Management	Management activities that do not require technical skills.
Tool support	Setup, maintenance, troubleshooting of development tools.
Technical Lead	Management activities that require technical skills.
Administrative support	Clerical work that anyone can do.
CSCI (GMS-DPC) test	Test involving the grain spectrometer and the diagnostic PC.
System Integration	Integration between software and other engineerings.

Table 1. Labor tracking categories used for GMS software

The labor distribution changed in interesting ways throughout the project. For the first iteration it was very different because we were a newly assembled team and had to set up our tools and decide how we were going to work together. The "Process devel" bar reflects time spent talking about processes we'd use and settling on a coding standard. Note also the large amount of "Tool support" work.

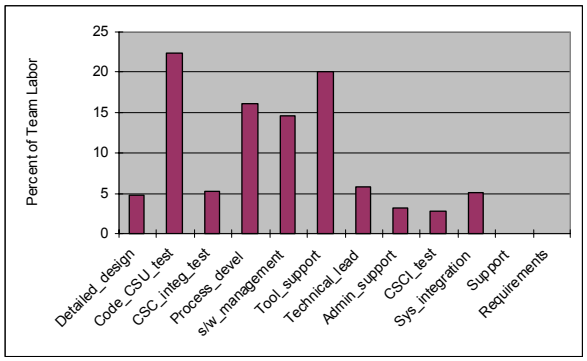

Figure 3. Team's labor distribution for first iteration

The below two figures show full-year labor distributions. The one for the first year overlaps the data shown for the first release.

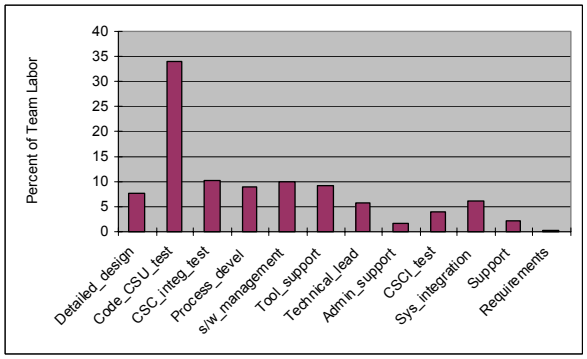

Figure 4. Team's labor distribution for first year of project

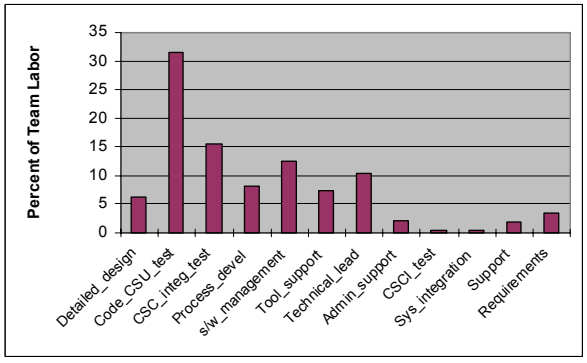

Figure 5. Team's labor distribution for second year of project

Labor data was not collected for the third year of the project.

Each full time developer averaged between 36 – 40 hours per week for 1999 – 2000, including vacations and all other absences. As technical lead, my average was 46 hours per week. These numbers are simply the gross hours worked divided by the number of weeks.

It is worth noting that there isn't a practical way to determine just what labor went into the embedded GMS software. The team also built custom utilities to go with the unit but that work was not done using agile practices, and that labor was not tracked separately from work for the embedded GMS deliverable.

The staffing changed mid way through the project. Bob and Don left the team. Jack had been part time, and left during the first year. Sherry and I continued and two new developers were added. By this point the software development practices were well established, and the low bug rate was also well established by the original "newbie" team.

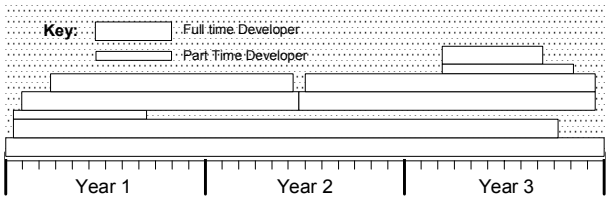

Figure 6. Staffing profile for the GMS project

A typical set of scheduled releases is shown in the figure below, to give an idea of how often we delivered on time. The third release down from the top was seriously late due to a staffing change that wasn't known when we planned that release. During this period we were trying out the "Planning Game" practice from Extreme Programming, and wanted to track how well we did at delivering on time.

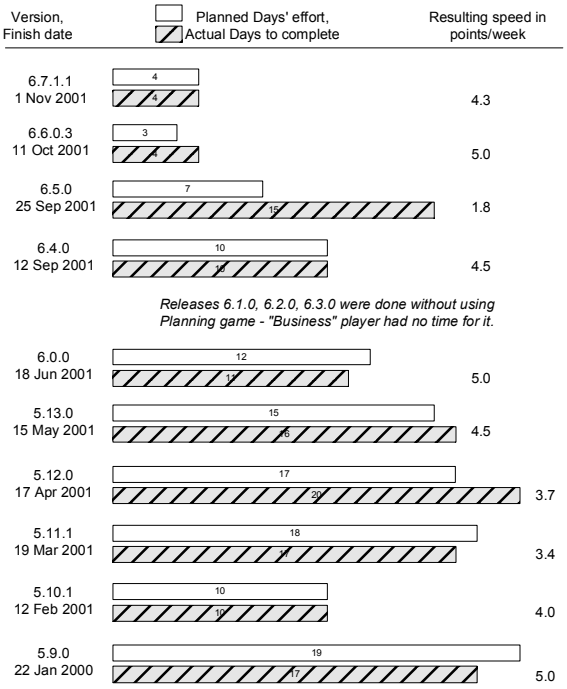

Figure 7. Schedule performance while using extreme programming planning game

About mid way through the project we were learning about Extreme Programming and bringing our practices more in line with that. The figure below shows that we were already doing most of those practices to some extent, and it tries to express the degree to which we used each practice.

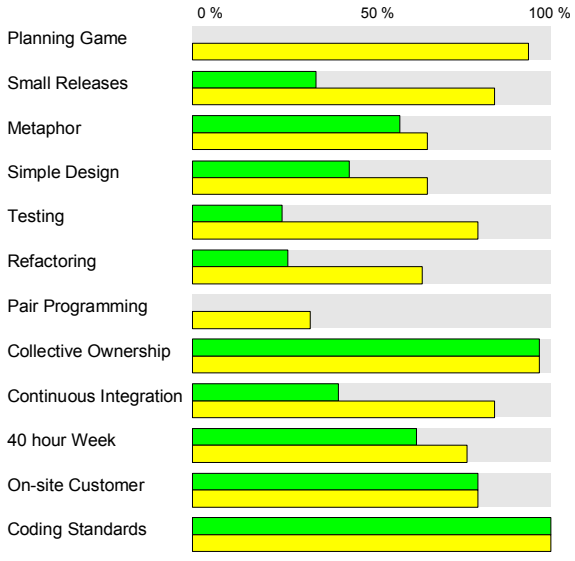

Figure 8. Practices used before (green) and after (yellow) adopting Extreme Programming

In three years the code base grew from zero to 60,638 lines of code, mainly in C with some assembler code. That's raw lines of code. Our code averaged 60% comments and blank lines. In other words about 40% of it is effective lines of code (ESLOC).

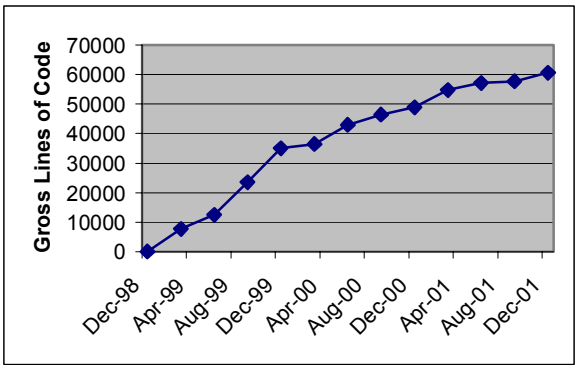

Figure 9. Growth of the code base over three years

As the code base grew the defect rate remained steady at about 1.5 defects per month. The below figure shows the cumulative defects over the life of the project. Note that it is linear.

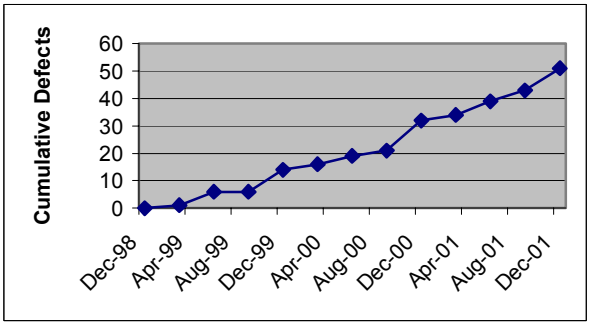

Figure 10. Cumulative defects over three years

During the entire project, there were never more than two open defects at one time.

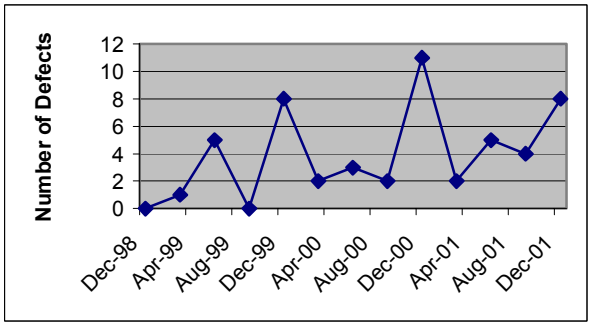

Figure 11. Absolute number of defects per quarter

Throughout most of the project our "open defects" list held one or no items. The definition of a defect (for our purpose) is any unexpected or undesirable behavior in the software after unit testing has been completed. We did not play any games redefining bugs as "features".

A root cause analysis was done for each defect to identify how it was inserted, and to decide how to best prevent more occurrences. The below figure shows when bugs were introduced.

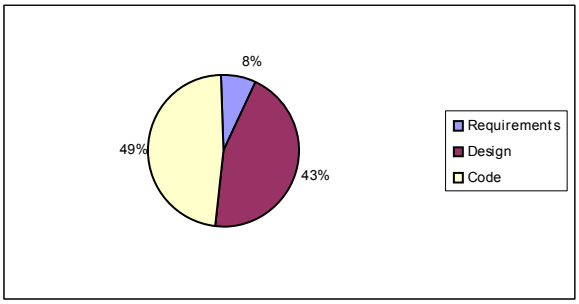

Figure 12. Phase where defects inserted

Many defects were removed during unit tests and earlier activities but we did not track those. We only tracked defects removed at integration test and later. Later phases included internal release to engineers within our company, and external release to our partner company. The below figure shows the phase where defects were discovered.

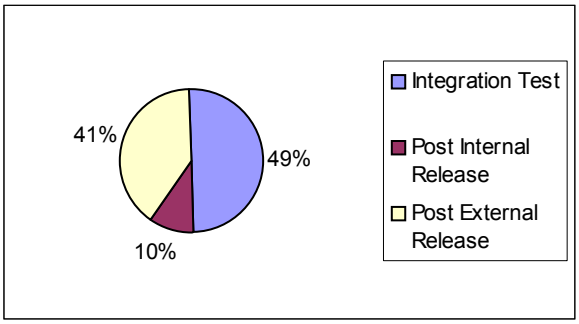

Figure 13. Phase where defects found

It has been observed that the longer a defect stays in the code the harder it is to remove. That was our experience. In order to get a sense of "a bug's life" I made the below charts showing how long each defect stayed in the code.

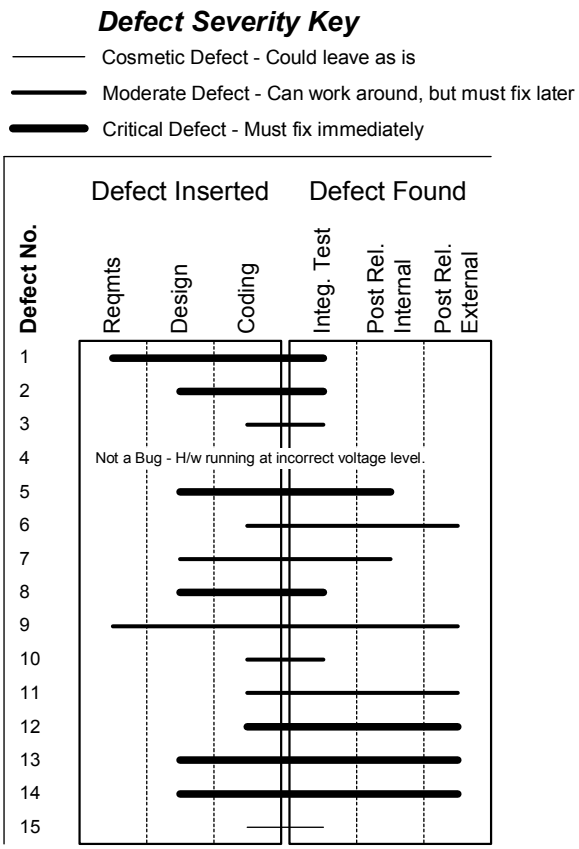

Figure 14. Defect life span, year 1 of project

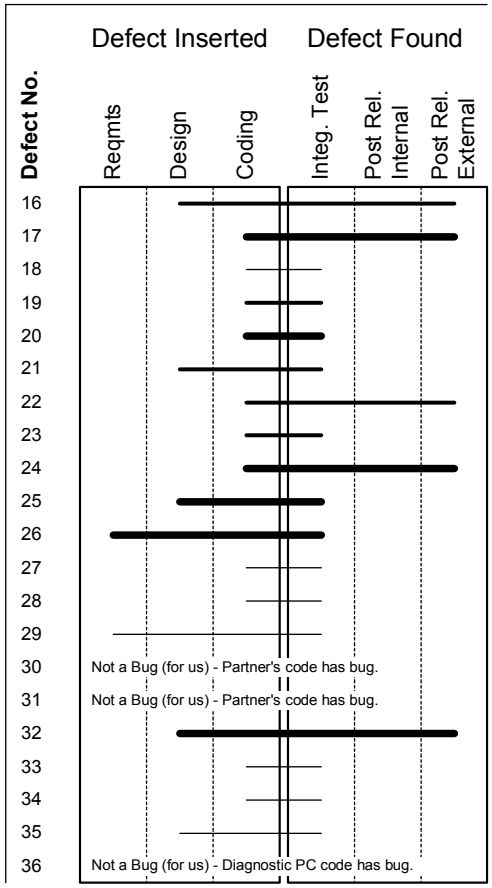

Figure 15. Defect lifespan, year 2 of project

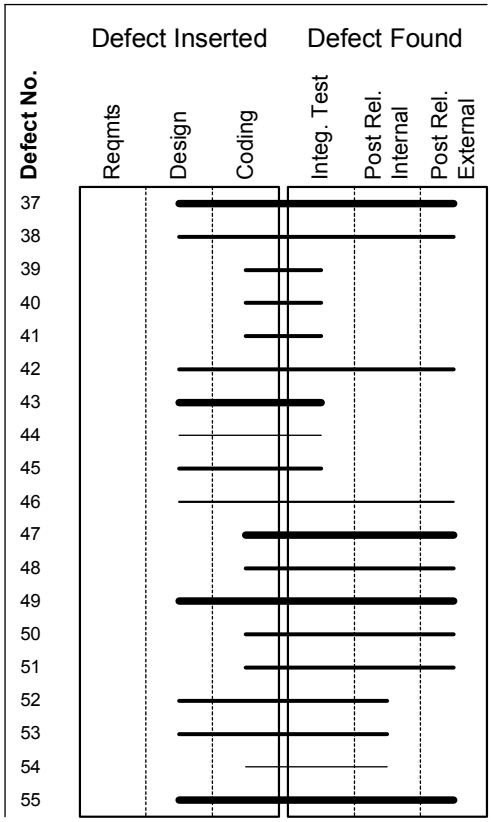

Figure 16. Defect lifespan, year 3 of project

In order to get a sense of the type of defects that made it past our defenses, here is a table of the root causes for just the critical defects – those requiring an immediate fix.

No	Root cause
1	Insufficient hardware settling time allowed due to no circuit board to test with
2	Communications hang due to change to comms protocol without sufficient h/w test.
5	Incorrect data readings due to poor synchronization design
8	OS deadlock due to incorrect task sequencing
12	Math algorithm halted – incorrect error level assigned. Insufficient team review of all s/w log error levels
13	Change to boot code comms init could not be tested by us
14	Data taken with incorrect settings due to insufficient communication of the design, and too little test prior to field use
17	Wrong settings used for data gathering due to insufficient peer review of source code
20	Overlooked the need to test with a variety of individual units – h/w tolerances vary.
24	Unit stops working due to array bounds overrun revealed by non-debug release.
25	Not all of calibration table loaded because checksum counted bytes transferred, not bytes written to table – insufficient unit test.
26	Unit couldn't access calibration data because we mis-communicated changed locations in flash memory
32	Unit hangs up when remembered grain setting not present in new cal table. Poor analysis of new requirement to recall settings.
37	Data readings off by 1, 2 or more due to incorrect task sequencing between unit and display/ controller. Design error.
43	Bad data received due to calibration table generator having error. Design mistake.
47	Controller software commands wrong grain type because shift-accumulator not cleared after use. Should have caught in unit test.
49	Incorrect settings used for data due to sharing of settings buffer. Poor design.
55	Reading wrong type of spectra due to non OO design of settings buffers.

Table 2. Root causes for critical defects

Four of the later software releases were analyzed using C-Metric v 1.0 (from Software Blacksmiths) and the average cyclomatic complexity for the whole release was 6 or 7 for those releases. Here is a more detailed look at a typical software release done near the end of the project.

GMS 6.6.1.1						
With the test code				W/out the test code		
File name	Func.	avg CC	max CC	avg CC	max CC	cc>10
VSPECTRA.C	48	1	7	1	7	0
UTIL_ACC.C	24	6	42	5	42	2
TST_CASE.C	0	0	0	0	0	0
SYS_CONT.C	11	12	44	10	44	3
SW_LOG.C	13	21	198	7	28	2
STR_UTIL.C	3	4	8	4	8	0
STR_FUNC.C	3	3	5	3	5	0
STAT_TBL.C	88	4	235	1	32	1
SETUP555.C	7	1	4	1	2	0
OS_UTILS.C	5	3	6	3	6	0
MEAS_PRD.C	12	9	42	5	13	3
LED_CONT.C	7	9	16	6	13	1
LAMP_REG.C	6	11	23	9	16	1
HW_ACCES.C	30	3	14	3	12	3
GLOBALS.C	6	3	7	3	3	0

GLOBALGO.C	0	0	0	0	0	0
GENERAL.C	7	11	68	2	5	0
FLASHDIR.C	1	1	1	0	0	0
FE_INTFC.C	30	7	85	3	38	2
EXFUNCS.C	19	2	2	2	2	0
ERR_HIST.C	0	0	0	0	0	0
DET_CONS.C	9	11	37	9	26	2
CMP_SPEC.C	13	8	18	8	18	4
CMD_SWBD.C	9	10	37	7	19	1
CFIG_TBL.C	57	4	144	1	16	1
CAN_COMM.C	45	10	155	6	29	6
CAL_UPDT.C	15	8	33	8	33	3
ASERT_TX.C	1	2	2	2	2	0
APCODEHD.C	0	0	0	1	1	0
ALG_TST.C	2	6	10	10	10	0
ACQ_SPEC.C	26	10	59	7	52	5
ABS_STEP.C	10	10	40	7	11	1

Table 3. Cyclomatic complexity for release 6.6.1.1

The above table shows file names (header files omitted), the number of functions in each file, the average and maximum cyclomatic complexity for the functions within the file. One difficulty with this measurement is that our unit tester code is kept in the same file as the production code and it's conditionally compiled. Because of this, it gives a falsely high cyclomatic complexity reading.

The same data was taken for release 6.6.1.1 after removing the unit tester code. Those results are in the right-most three columns. The column "CC > 10" indicates the number of functions within the file that have cyclomatic complexity above 10.

6. How Team's Metrics Compare with the Industry

6.1. Productivity Estimate and Actuals

The "Seer SEM" software estimation tool used 1.2 ESLOC/hr as the expected productivity rate for fully tested, working code (as discussed in the previous section "Parametric Estimating"). Since the tool is based on data from thousands of actual projects segmented by type, we can accept that figure as the industry norm for embedded real time software. My previous experience led me to believe that the team could do 2.5 ESLOC/hr if they used the team-based practices I had experienced on other teams. One example is the use of strong unit testing, together with a trouble log that's always enabled.

My staffing level plan was based on the 2.5 ESLOC/hr value. When the first iteration was completed, a detailed analysis was done to see what productivity level had been achieved for the embedded software. It was 3.5 ESLOC/hr, or 292% of the industry norm. This team of newbies to embedded programming demonstrated almost three times the productivity of a typical embedded team, and on their very first iteration!

A rule of thumb for software test says that you've found most of your bugs when you have found about 15 bugs per thousand lines of code (a 1.5% defect rate). The GMS embedded code base had 29,500 ESLOC at the end of the project. If it had a 1.5% defect rate, it would have had 443 defects. Instead it had 51 defects (that's the grand total over three years; not the number present at the end). The actual defect rate was 0.17%.

Defect Rate = (51/29,500) * 100 = 0.17%

Another useful metric is the software defect removal efficiency, or the percent of defects that is removed before the software is released. For GMS, 30 of the total 51 defects were removed before software was shipped to our partner company (our customer), so the defect removal efficiency was:

Defect Removal Efficiency = (30/51) * 100 = 58.8%

6.2. Comparison with QSM database

In an additional attempt to compare our statistics with industry standards, I submitted the statistics from Iteration 1 to QSM Associates Inc. They used to offer a free service via their website to allow you to compare your project with those in their database. (They market a software estimation tool which works using parametric estimating, as Galorath's tool does.) The "Productivity Index" they calculated for the GMS Iteration 1 ranked us in the 90th percentile! This index, as they compute it, covers code complexity (based on size), schedule, efficiency, effort, and reliability.

6.3. Comparison with Data from Capers Jones

Capers Jones, a principal at Software Productivity Research (See http://www.spr.com/) has accumulated data from a wide variety of software projects. In order to compare the GMS team's performance with his data, we need to know how many defects per function point were in that software. We have the number of defects per ESLOC.

Function point metrics were not in use at our company, so I looked up an equivalence measure online. For C, 128 lines of code equals one function

point. (See http://www.theadvisors.com/langcomparison.htm).

GMS Function Points = 29500/128 = 230

GMS Defects per Function Point = 51/230 = 0.22

The below figure gives defects inserted along the vertical axis, and defects removed along the horizontal. The best software teams insert few defects and remove a high percentage of those. SPR (Software Productivity Research) has tagged some of this data according to country of origin for the software.

MAJOR SOFTWARE QUALITY ZONES

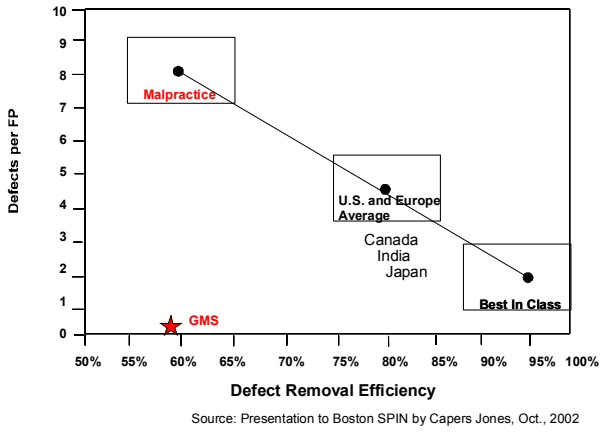

Figure 17. Software defect data from Capers Jones with GMS data point added

In the above figure GMS appears among the worst at defect removal, and with the best in terms of injecting few defects to begin with. The question is 'what does this mean?'

Each point on the above figure expresses some number of defects delivered to the customer. Consider this formula:

Defects to customer = Total FP * defects per FP * (1.0 - defect removal efficiency)

Let's look at how the "best in class" teams would perform if their code was the same size as GMS. Their defects would be 230 * 2 * (1.0 – 0.95) = 23 per the above figure. They would deliver 23 defects to the customer. The GMS newbie team delivered 21 bugs to the customer.

Function Points	Best	US Europe	Malpractice
0	0	0	0
50	5	45	160
100	10	90	320
200	20	180	640
230	23	207	736
400	40	360	1280
800	80	720	2560

Table 4. Defects delivered to customer per Capers Jones, tabular form

The above table shows the performance of the various teams in terms of how many defects get delivered to the customer. The same information is shown graphically in the figure below.

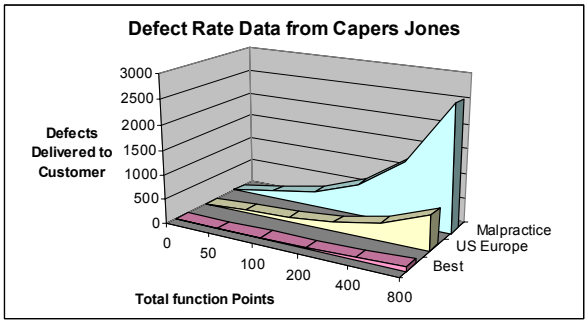

Figure 18. Defects delivered to customer per Capers Jones data, 3D diagram

Most of the projects in Capers Jones' database would be waterfall projects, and they'd be measuring the defects inserted even at unit test. That would indicate that any agile project would measure fewer defects inserted. Agile teams find bugs so fast during unit test that it's not practical to record them.

Similarly, it seems that most agile projects would record a lower defect removal efficiency than waterfall teams, if only because they didn't record all those bugs inserted during unit test.

I think most agile projects will fall toward the lower left on the software quality zones chart. Waterfall projects put considerably more bugs into a code base, and so they must use resources to get them out. All of that is waste which agile teams avoid.

7.Conclusions

There is no correlation between the defect rate and the size of the code base. That fact demonstrates that this team fully conquered the considerable complexity in this project.

The team's productivity has been compared with industry data via the Galorath database, QSM's database, and Capers Jones' database, and in every case they come out among the top performers. From the data, no one could distinguish them from the best teams in the software industry, yet they were missing a substantial degree of qualifications for this work. That

gap was overcome by agile software development techniques and the presence of senior level skills among some team members.

As Technical Lead, I wish that I had taken the time earlier and more frequently to analyze the numbers. My first attempt to analyze the project data was around 6 months into the effort. I was extra busy because I had to maintain a waterfall façade while really doing agile under-the-radar. If I had analyzed our data sooner I would have seen that we were doing far better than I thought, and that would have been a good basis for conversations with management, to get them more on-board with agile concepts.

In my view this case study is a clear demonstration of the lean principle "see the whole". Many companies sub-optimize by hiring only very experienced embedded software engineers. This is unnecessary. As long as all the needed skills are present on the team, agile practices can spread them around. I believe we might have gotten our initial few releases out faster if we had more experienced staffers (with other factors the same) but not by a very significant amount.

The team's performance despite the odds against them shows the power unleashed when technical people have full control over their work, and a clear view of what needs to be done. The lean principle of engaging the intelligence of the workers is very much in evidence here.

Managers are by necessity removed one or more degrees from the work being done. These results show that managers' best strategy is to support teams by ensuring they have the fullest control over their work, and tools, etc. and very clear goals rather than trying to control software development by decomposition and monitoring of activities. That is fundamental to lean thinking and agile software principles.