

Embedded Extreme Programming: An Experience Report

Nancy Van Schoonderwoert
Agile Rules Consulting
nancyv@agilerules.com

Class 220 and 230 at ESC Boston, September 2004

Overview	2
The Challenge	2
Analysis Paralysis	3
Breaking the Logjam	3
Staffing Issues	4
XP's Safety Net for Team Learning	4
Our Software Process – Generic Agile to XP	5
Mistakes and Recovery	6
Processes We Added to XP	7
Embedded Testing Techniques	8
What about Design?	9
The View in Numbers – Our Metrics	9
Bug metrics	14
Summary	14
Results	15
Implications for Software in a Regulated Environment	16
Conclusion	17

Overview

Extreme Programming (XP) is often viewed as risky – something you wouldn't want to use in a situation where there is great schedule pressure or where a software malfunction could threaten safety. This paper describes a project that began waterfall-style and had to become more agile in order to make progress at all.

My role was as the software technical lead, and I believe the experience of this project shows that agile software methods such as XP can help in situations where there are significant unknowns. Some would say that if there are significant unknowns then perhaps you shouldn't start the project at all. The question really hinges on whether the risk is worth the potential reward, and that is ultimately a business decision. This paper describes a situation where that judgment had been made, and it describes how XP worked in that context.

The Challenge

Our goal was to build a mobile spectrometer ruggedized for use on farm equipment. It would use light to measure the protein level of wheat as it's being harvested. The 4-member software team worked with a larger team that included mechanical, optical, and electrical engineers, chemists, and mathematicians to design the measurement algorithms. The target microprocessor on the spectrometer board was to interface with a control and display subsystem on farm combines via CAN bus, an RS232 client primarily for lab work, and a sensor array front end. The team built the code for the target microprocessor in C using a dual target RTOS (Nucleus PLUS operating system).

There were a number of new technology risks in this effort. The microprocessor being used was at the start of its development. Final silicon wasn't yet available, and we were among the first commercial users of it. The science behind the spectroscopy concept was new though it had been prototyped using a different sensor. The electronics had to have very low-noise signals. Mechanically, the whole works had to handle vibration, heat, cold, and humidity. We were the first users of the operating system port to this CPU. Our partner company had a network architecture that our code had to fit into. This meant we had to design the boot-up code together. They were using a version of CAN protocol that was not standard, and our spectrometer had to communicate with it.

The team also developed auxiliary Windows-based software to test the embedded application and to prepare calibration tables and load them to flash memory. The calibration tables would let various grains be measured, and allow other constituents besides protein to be measured.

Analysis Paralysis

Our company planned to partner with a farm equipment manufacturer for this project because we had the technology expertise but they understood farming economics and had relationships with the people who would ultimately buy and use the grain monitors we intended to build. We attempted to catalog the product's requirements but found this extraordinarily difficult. The moment we'd try to go beyond the basics of what had been demonstrated in research, there were so many new questions that our partner company had real difficulty telling us what they wanted.

Even the mathematics, which had been demonstrated in the lab to give an accurate percent protein reading for spectra input data, was problematical. Our analysis to find out how much computing power would be necessary to run this algorithm at say, a 2-second interval, showed that it would need close to all of the CPU's power. The mathematicians told us that they could come up with many other ways to do the same with less matrix math. They started to work on that idea while I and the electrical lead looked at an idea to use a PLD to pre-process incoming data to free up some CPU resources.

The effort to create a complete set of requirements and a traceability matrix stalled. The partner company had no engineers dedicated to the project this early on, so we had no counterparts to talk with. The business concept made sense, but translating this into sequential technical steps was like a chicken-or-egg problem; there seemed to be no real starting point. We could see our way to exploring the PLD idea and new algorithm versions, but in pushing to do complete requirements we were spinning our wheels.

Breaking the Logjam

New sensor hardware was being built by our company to bring in spectra to the instrument. It was important to see what that data would look like and how the algorithm would perform using it. I had intended all along to use some type of spiral or iterative software development approach, and this situation showed me what the first iteration should be – just build enough of the architecture to pull data in through the new sensor and upload it to a PC where the mathematicians could crank it through all the versions of the algorithm they had created.

We built two of the domains that the software design called for: “Acquire Spectra” and “Utility Access”, the serial communications part. The full design called for the data to pass from Acquire Spectra to “Algorithm” which would send just the % protein number out to the “CAN Comms” domain, which would be an interface to the external display unit. The “Utility Access” domain was just a backdoor serial comms utility to help us develop the software.

In order to get things moving, we kept the design concept intact and simply rearranged two of the domains so that they were the first and only ones implemented in our initial release. Care was taken to keep “Acquire Spectra” and “Utility Access” very decoupled

because we knew that we intended to rearrange them when it was time to implement the other domains in the design.

Once this start had been made, it was natural to implement other domains of the design incrementally. This initial release was very useful to all of the related engineerings and allowed them to take the next step in their work. For the mathematicians, the ability to upload raw spectra data was so useful that we retained that feature in the final product. In this way, we used the design that we had prepared early on but we also let useful new ideas evolve in the iterations.

Staffing Issues

Early in the project, the staff was made up of just the leads for each engineering discipline. The effort to create detailed requirements stalled and that made it difficult for the other leads to do estimates, especially staffing estimates.

I had experience with software projects of a similar complexity to this one and I was able to use that and the high-level software design I had produced to generate a staffing estimate. True, I didn't have detailed requirements, but I based my estimate on the present set and assumptions I was confident in from past similar projects. I did a breakdown of the present software design to smaller units. Using metrics from previous projects I then summed up the tasks.

My manager was investigating software tools that do parametric estimation and my estimate was in line with that data. Nevertheless, there was great pressure on all the technical leads to back down on their estimates of staff needed. I knew the software effort would be doomed if I did that. So I offered to adjust my estimate if anyone else could show me a better basis for it than what I had used. I published the detailed breakdown I had used, and all the figures. No one had anything concrete to offer, and eventually the requested staffing went through.

The next hurdle came when I could only get staffers who lacked important background. I had one very good electrical engineer with a year's experience coding in C. The other three had software experience but not in embedded, and none of them had multitasking OS experience. But among us all the necessary skills were present in at least one person. This situation screamed out for some mechanism to efficiently spread knowledge around the team.

XP's Safety Net for Team Learning

If you create the kind of unit tests XP calls for, and you run them frequently, you get a very effective tool to help team members learn areas of the code that they are unfamiliar with. They can step through the code with a debugger to see the detailed workings. This is far better than having to pull another developer away from their work to answer

questions, and it answers questions with perfect clarity. Better still, if someone makes a mistake and cannot get the code working again, they can back off to the point where all tests ran before their changes.

Working together on the coding tasks – whether by pair programming or a looser collaboration – is a great way to pick up tool tips, and coding techniques from each other. It also provides some further confidence in the code since another person has had a look at it.

Working out the coding standard they will use helps a team to form, as well as to learn. The chance to discuss techniques and show each other new ideas is important for getting to know each other. There is the possibility that people will dig in on opposite sides of an issue that is purely a matter of style. To avoid this, a team has to understand that it's more important for the group to make progress. In our case, the odds were so much against us that no one wanted to quibble over minor things. Everyone knew that the only way forward was by finding ways to cooperate.

Our Software Process – Generic Agile to XP

The ideas within XP didn't just spring up. They've all been around before. That's not meant to take any credit away from Kent Beck for articulating them so well and for combining them so effectively. Years ago I had the opportunity to work with a team that practiced strong unit testing and collective code ownership. From my work in hardware and embedded systems I learned the value of planning to build a system incrementally. When I had this opportunity to do a new product with a new team, I put together the best practices I had seen used so effectively before. I had not heard of XP – we were getting underway around the same time Kent Beck was completing his book “Extreme Programming Explained” in 1999.

The development phase of this project went on for three years. During the first half of that period we used what I'll call “generic agile” - the set of best practices that I accumulated from past projects. These amounted to all of the XP practices except for Pair Programming and the Planning Game. The most important ones were

- writing unit tests for every module
- collective ownership of the code, based on the use of tests and a coding standard that the team created.
- iterative releases of a partial but fully-tested system

We were using code reviews as our way of spreading knowledge around the team and to catch bugs early. Another difference from XP is that we didn't automate our unit tests. Each module's tests could be run with one command, but we hadn't created a mechanism to chain them all together so one command could execute all existing tests. Also, our iterations tended to be long – two to eight weeks.

By the summer of 2000, I had read Kent Beck's book "Extreme Programming Explained" and the Planning Game seemed to be just what we needed. Several constituencies were tossing new requirements at us, and I knew we shouldn't be the ones to prioritize them. I was also interested in trying pair programming. We were already doing the other practices in some form, but we started to bring them more into line with the XP view. For instance, we had been integrating often but not really as soon as a feature was completed. We tightened this up. Now that Refactoring had a name and some respect, although we had been doing it all along, we saw room for improvement. We started writing scripts to automate those tests.

Mistakes and Recovery

The Planning Game worked well for us... after we finished making our mistakes in using it! Initially, I discussed it with my manager and it seemed like a good idea to have the project manager at the partner company play the "Business" role in the Planning Game. They were by far the main source of requirements and we talked with them frequently. Before proposing it to them, we estimated one release then completed it our usual way. This was in order to "calibrate" our estimating ability. We needed to know how many "points" worth of work we could complete in a week.¹

I explained the Planning Game to the project manager at the partner company and he was willing to try working that way with us. It turned out that they wanted more work than we could do and the date was immovable. They wouldn't budge, no matter what I said. So finally I had to tell them that we'd work the features in the priority order they gave us, and we'll try to get those last couple in but the odds are against it. "If the date arrives before all the features are done we'll be ready to release what we have if you want it." I said.

We did make an effort to get the extra work done, and our manager let them know that. Unfortunately, our estimate was exactly right and we could only complete the amount we originally tried to hold them to. That triggered an odd thing: our "Business" player came to respect our estimating ability, and they were impressed that we had produced a high quality release. After that they stuck to the established speed limit.

A second mistake was made in connection with the Planning Game. A few more releases were done with the partner company playing the "Business" role. But some needed features weren't getting prioritized. If our electrical group or mechanical folks needed something, they couldn't get the partner company to prioritize that over other things. It became clear that we had picked the wrong player for the Planning Game. A change was made so that our project manager would become the "Business" player, and everyone else would go to him to ask for the features they needed. This worked very well.

A third mistake we made was connected to refactoring. Schedule pressure made us put

off refactoring more than we should. The sign of it was that after awhile our velocity (the number of points per week we could complete) went way down. It was clear why: changes were getting much harder to make than they should be. The code was becoming brittle and we needed to clean it up.

After several months of steady on-time releases, our manager began to drop out of attending the Planning Game meetings. There was a higher level of schedule pressure at that point and we'd been so steady that he chose to devote his time to areas that were in trouble. Two more releases were done without management participation, and I was concerned that it would just be a matter of time before we started to have problems as a result. The schedule crunch began to ease and I managed to persuade our manager to resume his participation in the Planning Game.

Processes We Added to XP

Other “best practices” were in use by my team, though they are not explicitly called for in Extreme Programming. We used checklists, which we created as needed. We also used any code checking tools available to us, e.g. turning up the compiler warnings setting all the way, using lint and similar code checking tools.

Short developer-level documents were created as needed, and maintained by the team. We called them “Mini-docs”. If a document proved its worth by being brief and telling you something that wasn't easy to get from looking at source code, then updating it was usually something we did as part of thinking about how we'd change the feature it described. In this way, updating documentation became a design activity – a thinking tool – rather than an extra chore to do after the coding was finished.

We held a team meeting for 30 minutes twice per week so that questions and issues that involved the team, the code, or the project could be discussed. This let us all be aware of what areas of the code were being worked on by the others, and any unexpected problems they might be having. Longer topics sometimes came up and we'd set another meeting to discuss those. The half hour team meetings were working sessions for us to coordinate with each other and ask questions that were of interest to everyone. This practice continued all through the project. No one viewed it as a necessary evil – on the contrary, it became a good place to park questions that didn't need immediate handling.

We continued using code reviews for code that wasn't pair programmed. Some of us tried pair programming and liked it. Two others didn't want to try it. Needless to say that limited the variety of pairings available. For those who didn't want to try pair programming, they did sign up for joint responsibility for features with other team members, and they made an effort to work more closely on code together. Often this meant working individually for a couple hours, then testing together, then doing another stretch of individual work.

Special test techniques for embedded code had to be developed. The automated tests described so far were used only on the desktop PC.

Embedded Testing Techniques

The information I read about XP described a payroll project, but the testing techniques for that kind of system only go part of the way for embedded work. Here are the ways we covered the gap.

Like many embedded projects, hardware was not available early on, so we used dual-targeting. The operating system we were using came with a version that could be run on top of Windows NT, and we could link to that or to the version for our target microprocessor. Three levels of testing were available to us on the two platforms (X86 and target CPU). Those three levels were unit test, domain test, and full system test. When running on the X86, a `#define` in the preprocessor let us skip over hardware-dependent code. If that would bring in data, then dummy data was substituted, giving us still more flexibility in testing.

For those modules that touched hardware, we used a second tester function intended for execution on the target CPU. Those tests were run manually while we'd watch the hardware to check its response. In this way motors and LEDs could be checked easily. Since they were being driven by the same code that the full system would invoke, it gave us confidence in our hardware-specific code. For some time I believed that I should find a way to automate these hardware-specific tests. That has proved unnecessary because once you establish that the driver software is correct, that software together with the hardware it drives becomes a component that is not re-opened. At least that was true for simple drivers – they were very seldom modified. If it's decoupled from the rest of the system and you haven't changed it, the need for re-testing it goes way down.

Very early we created a software trouble logging module. This allowed us to trace execution of the code and upload a log file to a laptop via serial port. Many embedded systems use something like this, and we found it a very valuable adjunct to our other testing techniques. Our trouble log stored entries to a circular buffer in RAM, and because it used so few resources it was always enabled. With logging systems that are normally disabled, you risk distorting the behavior of the code by simply enabling the logging – that can make it impossible to find certain kinds of bug.

The developers didn't have the math background to write complete unit tests for the code doing the grain algorithm. So we wrote what we could for unit tests and relied on system level tests for the math as a whole. We created a test mode in the software that would conditionally link in files containing test input data and the correct results for each of the input data sets. These files were easy for the mathematicians to generate from their models, so they gave us these each time they needed us to update the algorithm domain. The algorithm had several stages, so we got those intermediate results too, and then our

code could test itself and pinpoint what stage of the math had a problem.

What about Design?

Our design was created “up front” and was analyzed using several use case scenarios. But implementing it piecemeal was a good approach because it allowed feedback, and it helped us keep each domain very decoupled from the rest of the system. As we built each domain, we made test stubs to stand in for the rest of the system. In this way the test stubs were a design & development tool rather than being extra work later on just for testing.

As we began each iteration toward a new software release, we revisited the design of the features we were adding. Just before meeting with management to schedule features for the next iteration, we'd estimate them in detail to avoid unpleasant surprises after we started coding. But this was more than just estimating – we'd look at the code and make sure that the new feature would go in as we expected. So we were really doing detailed design in order to have a valid estimate. This might take a few minutes to a couple hours. Then we'd discuss these estimates. It was a chance for less experienced developers to do some designing, and get feedback on their ideas from the more experienced team members.

The View in Numbers – Our Metrics

The first software iteration for the Grain project began with the team working on getting their tools set up and configured the same for each person. A good portion of time went into defining how we'd work together (i.e. Team process development). The below figures illustrate how the distribution of labor shifted as the initial software was constructed and tested.

The labor categories are defined as follows:

Detailed Design – design work at the feature level.

Code and CSU test – unit testing and coding of functions.

CSC Integration and Test – integration testing

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 engineering

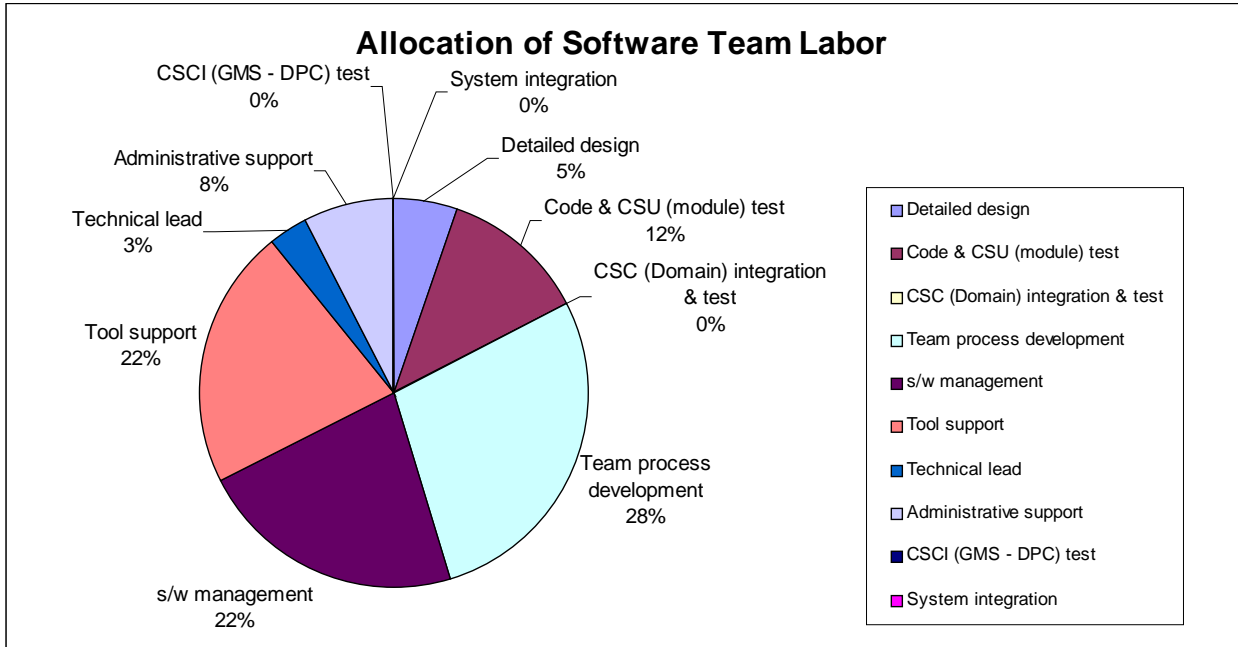


Figure 1. February 99 software labor allocation

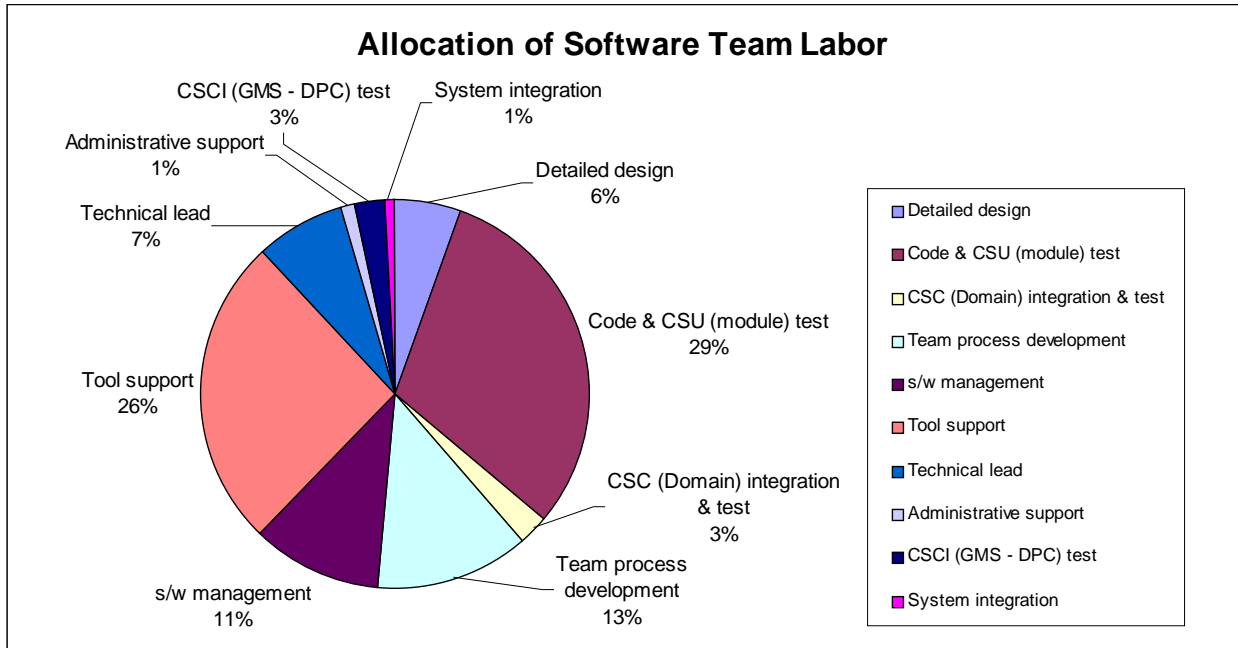


Figure 2. March 99 software labor allocation

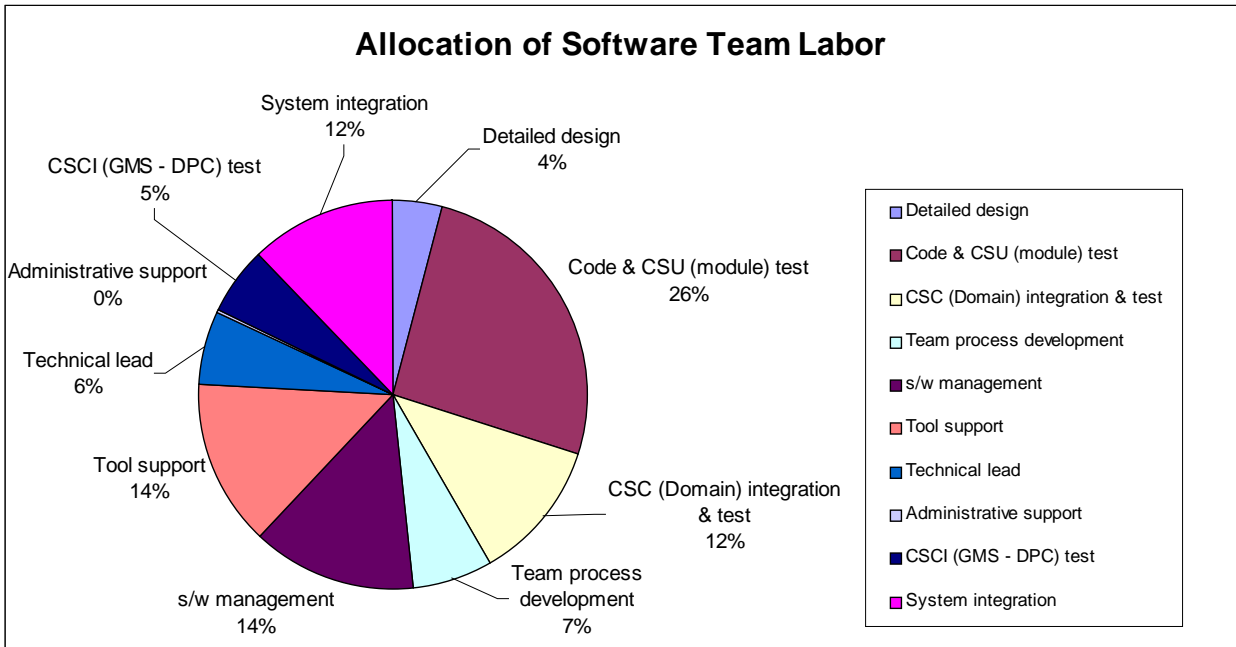


Figure 3. April 99 software labor allocation

The team voluntarily tracked their labor, using categories they defined. The following figures show labor allocation for full years. Team members averaged 40-45 hour work weeks. The technical lead averaged 55-60 hour work weeks.

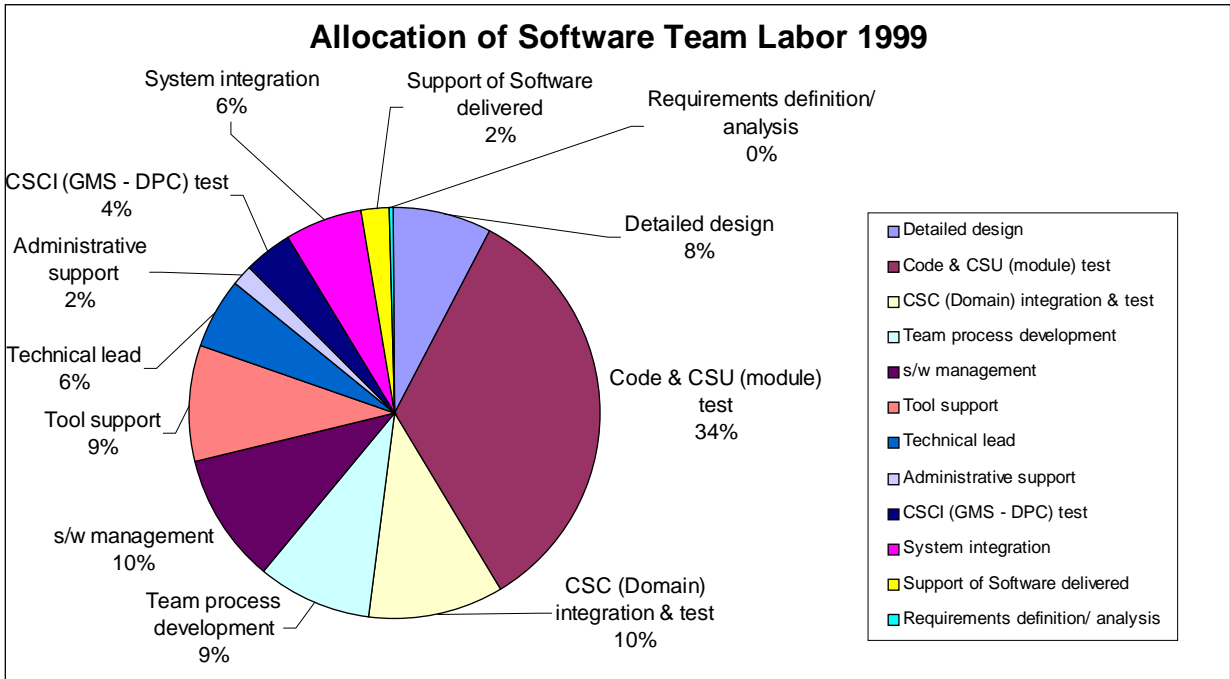


Figure 4. Software labor allocation – full year 1999

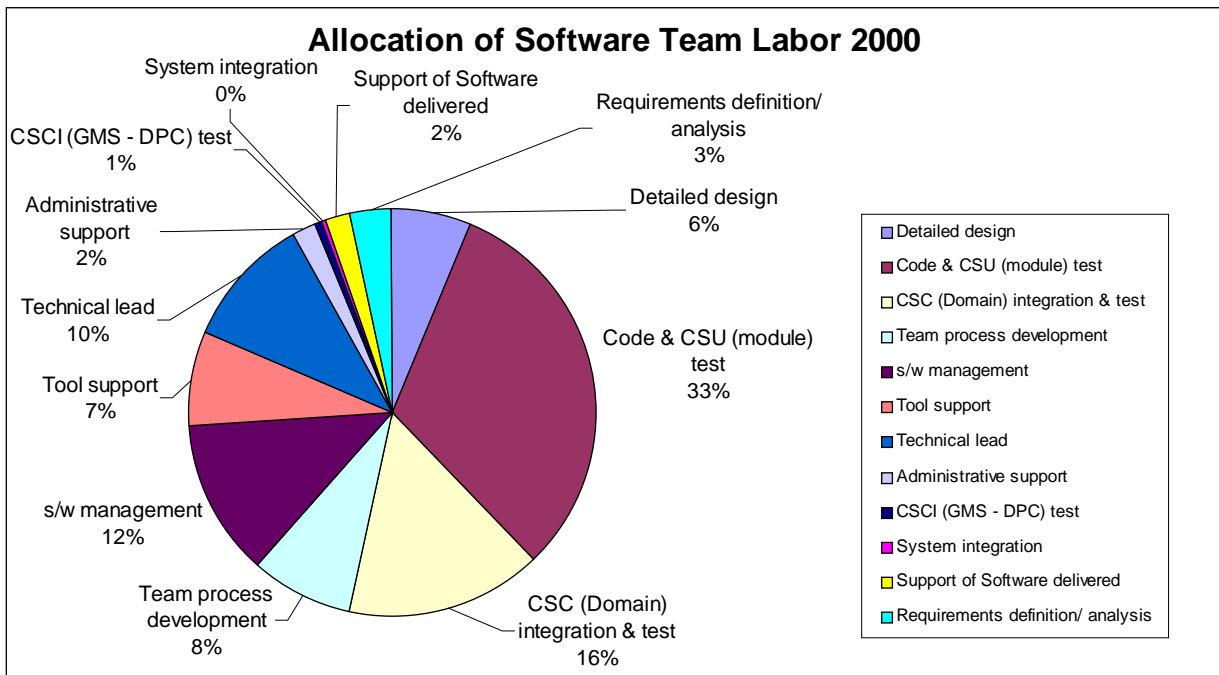


Figure 5. Software labor allocation – full year 2000

The change to using the Planning Game in August, 2000 resulted in shorter releases that were generally on time. The below figure illustrates the performance against plan for a continuous sequence of releases. Releases prior to the use of the Planning Game were often late. There isn't a good record of the planned dates to compare actuals against. Erratic schedule performance is one reason why we tried using the Planning Game.

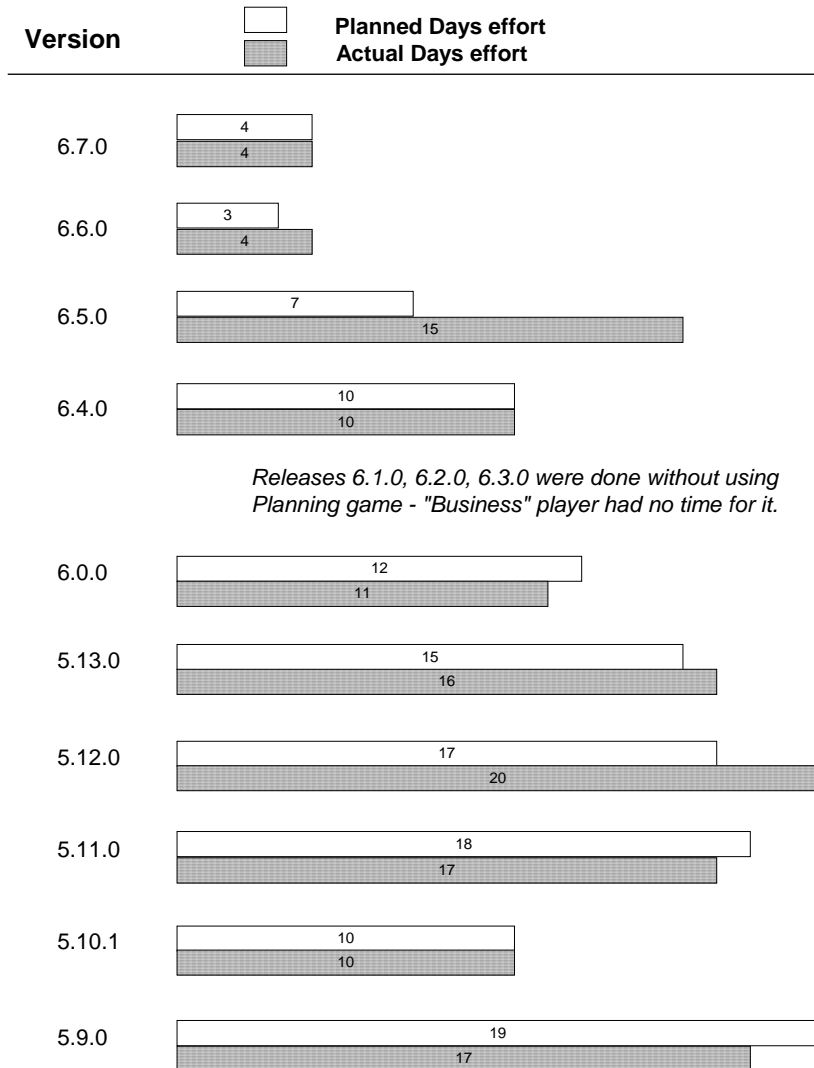


Figure 6. Planning Game releases - actual and planned

In the above figure the release for version 6.5.0 was very late due to a staffing change that was not known at the time the release was planned.

Bug metrics

Bugs were tracked at the integration level and later. Any undesired or unexpected behavior in the software after unit testing was considered complete was logged as a bug. Interestingly, our bug rate stayed the same as we adopted the full set of XP practices.

	How many	Percent of Total
Before XP		
Bugs found in:		
Continuous Integration	10	48
QA Testing	0	0
Internal Customer Usage	3	14
Customer's QA	0	0
Customer Usage	8	38
TOTAL	21	100

	How many	Percent of Total
After XP		
Bugs found in:		
Continuous Integration	14	47
QA Testing	1	3
Internal Customer Usage	4	13
Customer's QA	2	7
Customer Usage	9	30
TOTAL	30	100
Total all bugs	51	

Figure 7. Bugs by software development phase before / after XP changeover

Throughout development our open bug list never contained more than two items. In three years of development our grand total was only 51 bugs. A root cause analysis was done on every bug. Before our XP change, a frequent root cause was that the code review wasn't thorough enough. After XP, a common root cause was insufficient refactoring; unnecessary code complexity was tripping us up.

At the time of the first software release, we measured a bug rate of 0.48 bugs per thousand lines of code (comment lines excluded).

Summary

What was built:

- 29,500 lines of embedded C (and some assembler), comment lines excluded
- 5,000 lines (approx.) of Visual Basic code residing on "Diagnostic PC" Windows platform (to communicate with the embedded software for testing)
- 2,000 (approx.) lines of perl code for creating calibration tables

The team consisted of 4 developers for most of the project, with two more added for about six months near the end of the project (one of these was part time). The team’s labor produced more than just the deliverable spectroscopy application. Some coding effort went into the above-mentioned software utilities, and some into a related Windows application for “offline” spectra analysis. Detailed bug statistics were only kept for the deliverable embedded application software.

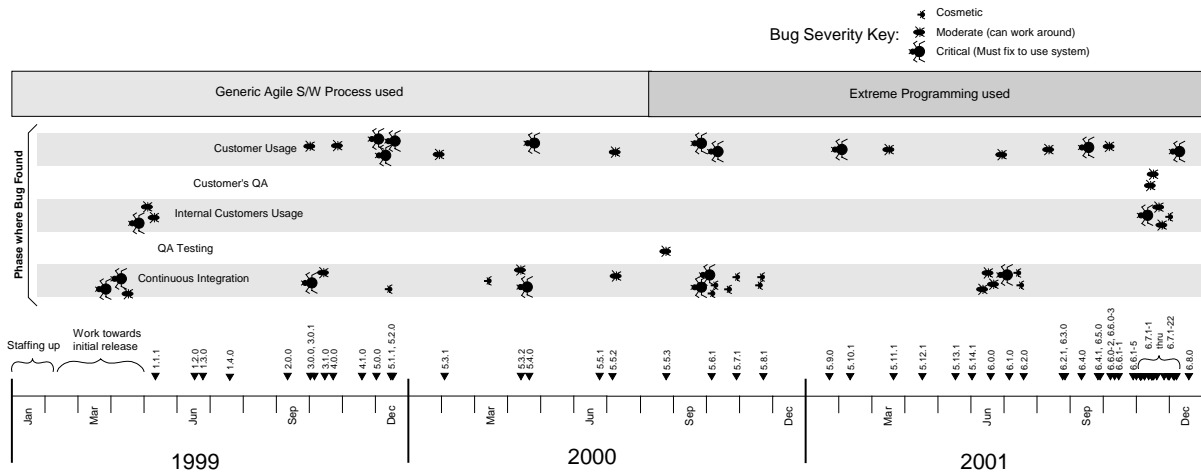


Figure 8. Timeline of releases and bugs for Grain Project

The Timeline figure gives an indication of bug severity along with the development phase where it was discovered. Note that very few bugs were found by QA. Throughout most of the project neither company had any QA personnel assigned. The “Internal Customer Usage” indicates other departments within our company. Bugs that got past our unit testing had an even chance of getting to QA or further.

From the timeline figure you can also see how much more regular our software iterations became after we started using the Planning Game.

Results

One of our testing techniques resulted in better collaboration with other engineering groups. We could build a module independently of the rest of the system, and download it to the target hardware. Then an engineer could step through the code using a debugger and exercise just one hardware driver. For example, the product had a shutter to let light through, filter it, or block it. The shutter was actuated by a motor, and the motor could be stepped through all its positions easily when that module was run “stand alone” on the target CPU.

We showed the electrical engineers and the production technicians how to use these test modules. Without this test code, a person would have to boot the full system and run it for over 20 minutes to wait for it to naturally go through each of the shutter positions. This improved our collaboration with other departments greatly, and with no extra work on our part.

Usually in an embedded project, if a problem comes up and it isn't clear whether it's in the software or the hardware, then the software is always guilty until proven innocent. But because our code was so stable throughout development, and it was so easy to test it quickly, a strange thing began to happen. I noticed that hardware engineers would double-check their hardware before deciding there was a software problem. The software was now innocent until proven guilty!

When spectrometers were out in field trials, strange readings sometimes were seen. Our trouble log could be uploaded to a laptop, and emailed to us. Then we'd talk by phone with the people doing the tests – sometimes while they were still out riding on a combine – and we could tell them what the software was doing.

The timeline figure in the previous section shows the releases speeding up at the end of the project. This was not a fit of panic coding. System-level problems became visible as more superficial problems were cleared away. The larger team needed software changes to narrow down clues, and we became their best tool for this kind of troubleshooting. The code was stable, and we were spending virtually none of our time chasing bugs. So we were able to turn around new releases for them almost daily. The problems were all solved and the product was made ready for manufacturing.

Implications for Software in a Regulated Environment

Late in the project, a potential customer in the pharmaceutical industry was considering our product and they sent an auditor to check our processes against GAMP (Good Automated Manufacturing Practice). He came out with several findings – all of them were a similar form: they said that the software team is doing X which is a good practice, but the problem is that management should be mandating that they do it and management has not mandated it. The checklists and documents that we had been keeping were valuable for the audit. Early in the project, our manager had insisted that we produce a SDP (Software Development Plan) describing the software development process we were using. We created the document he wanted, and later this was also helpful in passing the audit.

Practices we adopted to produce solid code quickly, ended up being very compatible with the changing hardware and with the GAMP audit. The key practices were:

- Unit level tests done simultaneously with coding – Prevented many bugs

- Domain level tests done simultaneously with coding – Let us fit software to the available hardware set, and keep the design decoupled
- Brief, developer-level documents and checklists created/updated at detail design time – Let us transfer knowledge around the team and gave a basis to audit against
- Making the code runnable on a PC and on the target CPU throughout development – Allowed us to quickly separate hardware and software problems, keeping a steady pace so we could deliver releases on time.

Conclusion

Ironically, Extreme Programming which is so often perceived to add risk to a project, turned out to be our best tool for controlling risk. XP isn't a cure-all, and there certainly have been cases where it did not work well. So it's worth considering why XP was successful in this instance. Here are the reasons that come to mind for me:

- First, the team did not hand responsibility for success over to XP. We used it but also used other practices that fit our needs, such as the mini-docs and the additional testing techniques. XP was our tool, not our master.
- Changes to practices were introduced over time, so the team didn't become overwhelmed.
- Team members were asked to first try out a new practice, and then we could make modifications if warranted. People need to have a say in their practices if they are to take ownership of them.
- In our technical sessions – whether on design, a bug fix, or a better way to test – the rule was always “the best idea wins” no matter who it comes from.
- All of the technical skills necessary for the job existed in one or more team members. Good process is a necessary tool but can never substitute for skills. You need both.

Extreme Programming (and other agile methodologies) is excellent at dealing with risk and uncertainty. So should it be used for everything? I've heard it said that there is no need to do any up-front planning; that you can evolve the design you need completely from scratch. Possibly, for non-embedded software. But this is debating in a vacuum. What I see in practice is a gradation of projects. At one end, the job is well understood and the technology is stable. At the other end, the job is hazy (like our spectrometer requirements) and the technology is unproven (like our new CPU). When you're dealing with well defined concepts in a stable environment, you can plan and expect to stay on

your plan. Things get interesting when a project cannot be fully specified, and/ or when the technology is shifting.

When a product is so well understood that it can be fully specified, it's likely to be a commodity – and there isn't significant financial return in doing more of what's been done before. Leading industry thinkers like Tim Lister and Tom DeMarco raise this questionⁱⁱ and say that the most worthwhile projects to do are the ones that have risk. They have risk precisely because whatever-it-is hasn't been done before. It follows that there's a financial payback for those who can handle risk without losing control.

Many embedded software projects are somewhere on that line between well-defined and hazy. In hindsight, we did the right thing. We used plan-driven methods at first but they took us only so far. We stalled out and then started using agile methods to get us the rest of the way there. The “lesson learned” for me is to get all you can from up-front planning but recognize when you've gotten all it can do for you, and have another tool in your bag. Ignore the polemics that pit planning against agile. That's a false dichotomy. Agile practitioners believe in planning – they just don't try to do it all in one go.

ⁱ Beck, Kent, *Extreme Programming Explained*, Addison Wesley, 1999, p. 85

ⁱⁱ DeMarco, Tom and Lister, Tim, *Waltzing With Bears*, Dorset House, 2003