Agile Embedded Software Development

James Grenning

San Jose, April 2007, ESC Class# 349

Software is hard. Embedded software is doubly so. All the problems associated with software development, such as delivery-time and defect-rate are amplified by the constrained and indirect environment faced by embedded developers. As a result, embedded developers tend to be more disciplined and skilled. They are more closely aligned with engineering than with programming. It seems likely that this exclusive talent pool is more effective at addressing the problems of schedule and quality than normal programmers; and yet the problems remain. Embedded projects are often late, over budget, and have high defect rates.

Agile Development has been effective at improving the performance of many software development teams. Their productivity has increased and become more predictable; and their defect rates have fallen; sometimes by as much as an order of magnitude. This has caused something of a "fad" in traditional software circles, with all the attendant non-critical thinking that that implies.

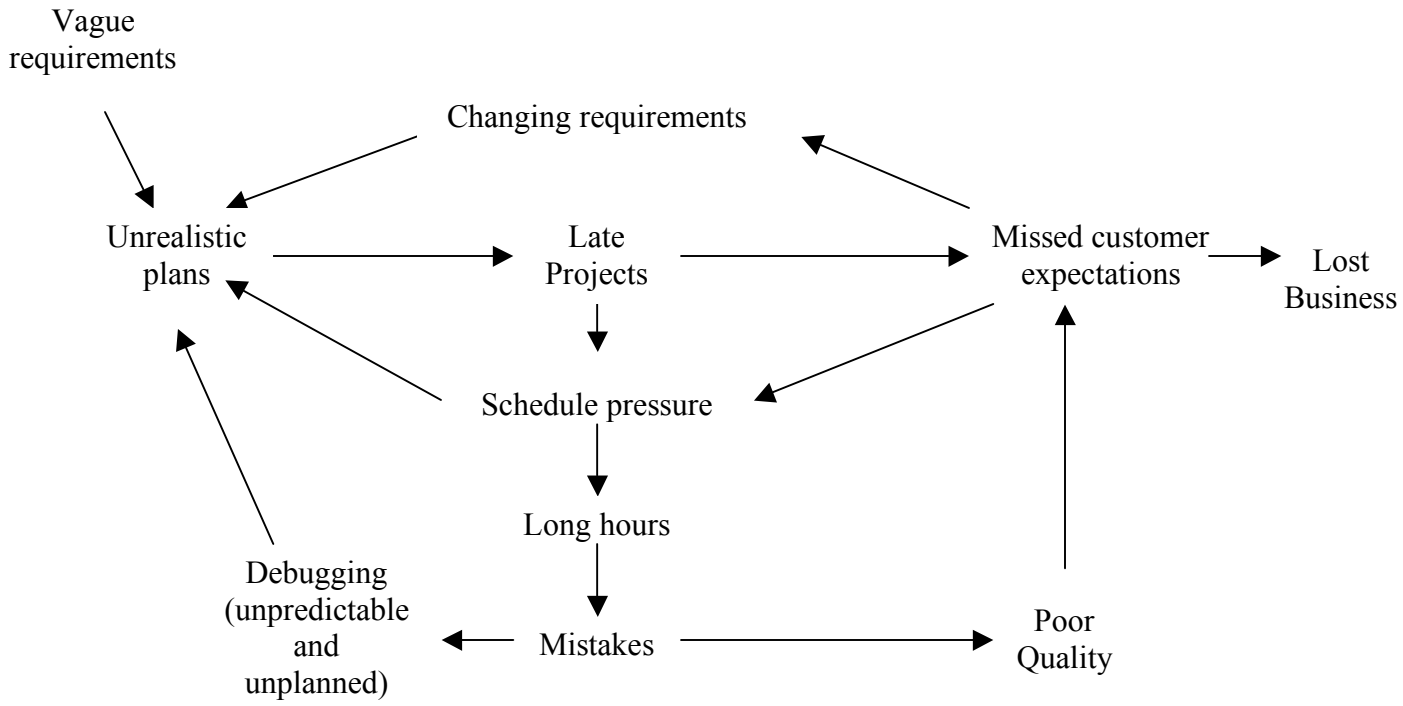
Embedded software developers don't react well to "fads" and for good reason. Embedded software is hard enough to get right without being blown about by every new process that comes along. In embedded circles, being conservative is survival.

Having said that, adaptability is also a survival trait. The problems that have faced traditional software development, and that have been improved by the adoption of Agile methods are the same problems that have faced embedded developers. It is time for the embedded community to start the process of adapting their ways to demonstrably valuable methods presented by the agile community.

Software development projects often suffer from long development cycles, late delivery, unpredictable schedules, poor quality, missed customer expectations and developer burnout. These problems often interact to become a positive feedback loop. Unpredictable delivery leads to schedule pressure, and unrealistic plans. Schedule pressure leads to long hours and short cuts. Long hours lead to burn out. Short cuts lead to defects, defects lead to more long hours debugging. Bug removal is an inherently unpredictable activity leading to even more schedule pressure, short cuts, defects, etc.. Figure 1 shows some of the interactions software development problems.

james@grenning.net

Figure 1 – Super-Vicious Cycle



You can see a few positive feedback loops in this super-vicious cycle. There are scheduling, defects, and requirements vicious cycles. These are important problems to solve for which the iterative approach of Agile Development has been shown to be effective. The specify-design-test-build-test-deploy approach used by waterfall sounds appealing but has proven time and again to give less than adequate results.

Iterative development, one of the core practices of Agile development has been around for decades. As far back as 1987 Fred Brooks' as chairman of the Defense Science Board Task Force on Military Software recommended that the waterfall process be replaced with iterative development due to waterfall's history of failure on large DoD contracts. Itertaive development has been used successfully on some very high profile projects from the 1950's to present day including: the X-15 rocket plane, project mercury, trident missile submarine control systems, the space shuttle avionics, and the Canadian Automated Air Traffic Control System, to name a few.[LARMAN]

james@grenning.net

What is special about Embedded Software Development?

The differences that make embedded software development so much harder than regular development include, but are not limited to:

1. Hardware and software are developed concurrently.
2. The development machine architecture is usually different from the target machine.
3. The hardware for the target machine is usually not available until late in the project.
4. It is often unclear whether a bug is hardware or software. You can't trust either one.
5. Hardware bugs are often "fixed" with software workarounds.
6. Spending $50K in software development to save $0.50 in hardware is often the right trade-off, so the software is often asked to do things that "ought" to be done by hardware.
7. There may be real-time constraints, concurrent processing, and safety issues.
8. Typical human-computer interfaces are not traditional, so the computer operating the machine is hidden from the user.
9. Resource constraints such as limited memory space or processing power are the norm.

These differences are significant. They make embedded software development an austere and challenging discipline. It has also been suggested that these differences might invalidate the use of Agile Development; but we not found that to be the case.

What is Agile Development?

A good place to start in describing agile development is to see what the group of people that coined the term had to say about it. The Agile Manifesto says:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.[AGILEMAN]

The first point stresses the importance of human interaction and teamwork. Many development processes try to take the human element out of software development, but the agile manifesto's leading statement is about leveraging the people and their interactions. Tools are needed, but it is good people, working in teams who build successful software products. This point is often misconstrued to say that processes do not matter. Processes and discipline do matter, but people matter more.

james@grenning.net

This statement could be misconstrued to suggest that Agile developers get things done by sitting in a circle and singing Kumbaya. However, there is another interpretation. An increase in teamwork does not automatically lead to a decrease in discipline. Consider, for example, the teams at the Skunk works, or Burt Rutan's team who built SpaceShipOne to win the X-Prize. These teams capture the intent of the Agile Manifesto well. While they deeply value discipline and process, they value teamwork even more.

The second point stresses the importance of having working software as a measure of progress. Documents may be valuable, but working software is a more meaningful gauge of software development progress. I have heard this misinterpreted as, "We're doing agile, so we aren't doing documentation". Balderdash! Documents are often invaluable. Those that are, must be produced. However, documentation is expensive to create and maintain so it is important to create only those documents you truly need. In document-centered development, I've heard more than once that the reason for the document is "our process requires it". This is wasteful. Agile developers articulate and validate the reasons for documents. Then the produce them if, *and when*, they are needed.

Remember also that most documents don't execute; so they cannot be used as effective measures of project completeness. Agile developers believe they are 50% complete when 50% of the features of the system have been demonstrated.

The third point addresses the need to work closely with the customer. By customer we meant the person or persons that are specifying the product. (In this article I'll use this meaning of customer.) Ideally it is the person using the product, but in mass marketed products the customer role is internal and indirect at best. Customer interaction is favored because software is very difficult to completely specify up front. Requirements and market needs change over time. The customer has to be part of the team to help make trade-offs and to *see* what they have asked for.

The forth and final point deals with the reality of any complex endeavor. Plans are important, but situations change that require constant adaptation. Plans cannot be viewed as static, even though specific delivery dates are. This point is often misunderstood to mean there are no dates or commitments in Agile development. On the contrary, dates and commitments are taken very seriously in Agile development. Agile developers create working software in very short cycles in order to *measure* compliance to the plan.

The Agile Principles back up the manifesto.[AGILEP] The principles are:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

james@grenning.net

- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

- Business people and developers must work together daily throughout the project.

- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

- Working software is the primary measure of progress.

- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

- Continuous attention to technical excellence and good design enhances agility.

- Simplicity--the art of maximizing the amount of work not done--is essential.

- The best architectures, requirements, and designs emerge from self-organizing teams.

- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Iterations

Agile development is based on iterative and incremental development (IID). IID provides regular feedback by breaking the project into iterations that are generally one to four weeks in length, with two weeks nominal. The output of each iteration is working software. Each iteration is like a stand-alone project ending after fixed amount of time, and delivering some executable version of the product. There are two main roles in agile development, the customer role, and the developer role. Each role is usually represented by a team of people. In a few words, the customer defines the product and the developer builds the product. I will cover these roles more completely later.

Embedded software engineers should understand feedback. The control systems we design always have feedback mechanisms to keep the systems under control. An agile project is based on iterations that provide feedback to the critical variables of the project: schedule, requirements, and design. Think of agile as a control system for software development.

james@grenning.net

The team estimates, plans and organizes work into iteration-sized chunks. The schedule feedback loop provides data that the team is on or off track by monitoring its ability to complete each iteration as planned. The team's estimations and on-going completed work are used as feedback in calibrating the development plan. If there are ten two-week iterations, and 200 effort points of work to do, the team needs to complete about 20 points per iteration. This feedback loop can provide a fact based early warning system of schedule problems. The project stakeholders use this information to adjust the project plan, maybe by adding resources, moving the date in or out, or adding or removing functionality.

Requirements are the broken into smaller demonstrable units called stories. They are the estimate-able, testable, and deliverable units. They are small enough so that many can be completed within a single iteration. When the developers complete a story the customers get feedback on the requirements by seeing and touching what has been developed. In the early iterations, prior to hardware availability some of these stories are demonstrated though tests and simulations; but the demos are based on real working code.

By building the software incrementally the developers get feedback on the design as it evolves. The stories cut across elements of the design provide early integration of simplified versions of the subsystems giving the developers valuable experience with the architecture. Something that looks great in UML does not always look so great once it is coded, and it is better to figure that out sooner than later.

The incremental or evolutionary design approach is central to Agile development. Software requirements are constantly evolving. Software is expected to be used year and after year and evolve along with the market and changing hardware and technology. So it is essential that code has to be built to last, and that means being built to change. Designs and code will continue to be changed throughout the life of the product and consequently there is the ever-present risk of side-effect defects. I'll describe alter how agile developers use automated test that help to lock in the existing behavior as new features are added.

Developing iteratively gives the business great power. The approach can be used to either manage to a specific delivery date, or to manage to specific feature content. The team's track record is input used to adjust the plan based on facts rather than wishful thinking

In a single-pass waterfall situation, considerable time is spent up front on items that may never make the final product. In Agile development effort is first expended on the highest priority features and capabilities, requirements are elaborated in parallel with development and consequently little time is wasted. Scope is managed at the detail level. When you have 10 must have features, they are broken into smaller stories, and the product content is managed at the detail level, deferring less important parts of the critical features.

james@grenning.net

Concurrent engineering

Imagine two trains running on parallel tracks, one train represents requirements definition and the other development. The requirements train leaves the station a little before the development train. Requirements are discovered, refined and handed off while both trains are moving forward toward the final destination. In phased development only one track is needed and the development train would not leave the station until the requirements train arrived and wired back the requirements. Maybe, just maybe, the development train will go faster knowing all the requirements, but it will never make up for all the time spent waiting for departure.

Concurrent engineering is a business strategy designed to shorten development time-to-market by doing development activities in parallel that have traditionally been done serially. Concurrent engineering is core to Agile, and Agile teams have to be skilled at working incrementally, without detailed knowledge of the whole picture, to be successful. This takes some getting used to after having waterfall based project management in the limelight for last twenty or so years. The business environment makes time to market critical so we need to find ways to finish development sooner. A way to finish development sooner is to start building the product sooner. We need to begin development before all the requirements are known or we will lose unnecessarily delay the product. If you think about it, the requirements are never all known up front so what are you giving up! Have you ever been on a project where there were no surprise requirements changes? The world of changing requirements is the world we have to live in, so lets master it.

I've seen too many teams paralyzed by not knowing all the requirements. There is a fear of making mistakes and a belief that we can figure it all out, and then design the perfect architecture with no false steps or rework. This sounds great but is not practical in the complex world of software development. So at the beginning of a project we have to identify some of the core features, ones that are well enough understood, and begin development work immediately. Beginning development will lead us to confront our requirements misconceptions and design flaws. In addition, and maybe more importantly, the feedback will help us find our blind spots, the unknown risks and weaknesses we cannot anticipate. Getting development started buys time for the requirements team to work in parallel on the remaining requirements. Requirements details are delivered just in time. What functionality do we develop first?

Start with core functionality, functionality that must be in the product. If you are building a telephone switch, simulate making a phone call. If you are building a security system, simulate a secure door being opened. Because the core of a new product usually encompasses significant value and risk, starting with the core improves our ability for product success because by starting with the high-value core functionality we are choosing what will be in the product when it is delivered. We are choosing to get more experience and more put miles on the parts of the system that bring the most value.

james@grenning.net

Initial implementations explore the requirements and the design thus improving the team knowledge, clarifying the requirements and solidifying the software design alternatives. The early development of these high-value/high-risk features provides feedback to the requirements and design teams. The feedback helps to mitigate the risks of building the wrong product by providing executable feedback on requirements.

The uncertainty and risk the team is trying to manage is not limited to software, but also to hardware and the hardware/software boundary. One way to deal with hardware uncertainty is for the software developers to wait until the hardware design is complete and then start the software design. WAIT! Just kidding! I am not recommending waiting! So, please don't quote me out of context! We really need a third train track with hardware development sometimes a little ahead and sometimes a little behind the software train.

Embedded developers don't have to wait for hardware because the trains do not have to be coupled. Hardware abstractions are created in the software that decouples software from hardware. The abstractions define an interface for interacting with the hardware by defining the service the hardware will provide, without getting bogged down in volatile implementation details. Hardware abstractions enable concurrent hardware/software engineering by allowing software development and testing to start prior to hardware availability. This important practice can also provide input into the hardware requirements and help define and refine the hardware/software boundary.

The iterative approach to requirements gathering, risk reduction and development practices means that changes in customer needs, project goals and hardware architecture are more naturally accommodated. Time to market can be improved by eliminating wasteful serialization in the development process by engineering the requirements, software and hardware trains on three parallel tracks.

Automated Test

The complexity and evolutionary nature of software development means there are many opportunities to break existing working software. A simple one line change, carfully thought out, could bring the system crashing down months in the future and leave no evidence to help find the crash. Side effect defects are common and often do not get discovered until long after the defect is injected.

Most product teams rely on manual testing in the real hardware to prove out their system. Manual tests have a few problems. First and foremost manual tests take a lot of clock time and are labor intensive. They might require special lab equipment. These realities mean that the tests will not be run often enough. What we would really like is a way to verify each change that is made to the software. If you had a magic button that you could press that would tell you if your software was operating to specification, how often would you press that button? I would press it after every change, and I do.

james@grenning.net

Agile teams have a button like that because we automate unit and acceptance tests making rerunning the tests very cheap. So cheap that we can run all the unit tests whenever any change is made to the source code. This is hard to do, especially when you are first learning, but the time saved will outweigh the time you normally would waste manual testing and debugging. Automated tests support the concept that new features should not break existing features. What a concept! I can just hear marketing now, "please add basement sump pump monitoring to the home security system, and its OK if you break any other random feature, no problem." Marketing does not want that, but they might get the idea that software developers think its no problem.

If you think of the tests as part of the system, even though they are not shipped with the system, when you run the tests the system basically reports when it is broken. It is kind of like double entry accounting, if the sum of the debits does not equal the sum of the credits, we have a problem. Accountants don't just sum the credits to save time, they make sure the books balance with their built in tests. They would prefer not to go to jail for messing up the books.

Where do these tests come from? Who writes them? When are they written? Automated unit tests are written incrementally by the software developers in a tight feedback loop with the production code using a technique called Test Driven Development.[BECK] The unit tests tell developers if their code does what they intended.

Automated acceptance tests provide evidence that the system meets its requirements. The developers and test engineers working with the customer team write automated acceptance tests. The acceptance tests are written in advance of the iteration where the development is to be done. This practice changes the role of test engineers in a very profound way. Rather than drowning at the bottom of the waterfall at the end of the project by a deluge of untested software pounding down on them, the test engineers adopt a proactive role by specifying the behavior of the system in the form of automated tests. These automated tests provide an unambiguous definition of done.

These tests are a very valuable investment. They make testing a repeatable process, that can be run with every change. The tests double as an executable specification. Unit tests provide examples of how a given module is used at a very detailed level, and provide feedback to the developer that the code behaves as expected. Acceptance tests demonstrate how larger groups of modules work together to deliver the product's requirements. Acceptance tests are written in a domain specific language so that non-engineers can read and possible write tests. An open source tool called FitNesse is often employed for automating acceptance tests.[FITNESSE]

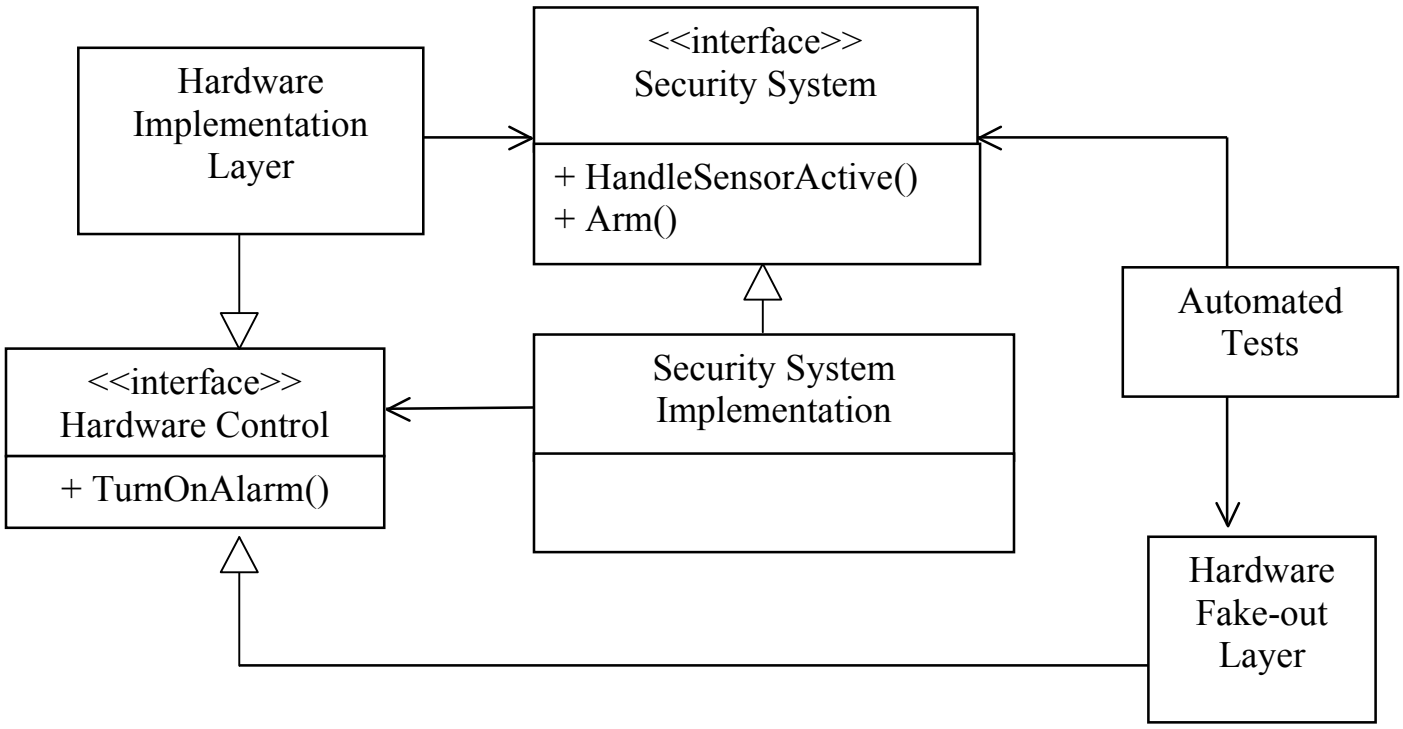
I expect that the automated test description sounds impractical to many of you. If you read this as requiring the automation of the entire environment around your embedded system, I agree, it is impractical except in cases where the cost of failure is very high (you be the judge). Rather than using expensive physical instrumentation to support

james@grenning.net

automated test, we use virtual instrumentation where possible. Virtual instrumentation uses the hardware abstractions as the interface point for event injection and detection during unit and acceptance testing, the same abstractions we talked about that facilitate concurrent engineering.

A hardware abstraction defines an interface to the hardware services. Figure 2 shows an example of part of a security system. The SecuritySystem shows two operations: Arm and HandleSensorActive. In the production system the arm button is sensed by some device driver and as a reaction to that event, the Arm() function is called on SecuritySystem. Likewise, HandleSensorActive is called when a specific sensor has changed state. The SecuritySystemImplementation can tell the HardwareControl interface to TurnOnAlarm as well as other operations not shown. When the automated tests are run, the incoming events don't come from the real hardware, they come from the test itself. Outgoing instructions to the hardware are intercepted by the test through the Hardware Fake-out layer. For example the test simulates that the Arm button was pressed and then generates the HandleSensorActive event just like the HardwareImplementationLayer will do in the production code. The SecutiySystemImplementation's reaction, sounding the alarm, is verified by the test by checking that the Fake-out layer was told to TurnOnAlarm.

Figure 2 Isolating and Testing Core Application Functionality



As a side note, a very exciting thing about this approach to testing is that our design has built in hardware isolation from the start. Without the automated test influence, there

james@grenning.net

would not have been an immediate need for isolating the core software form the hardware. Often good structuring like this is left out of embedded designs because of worries about performance. Please see my other conference paper on design for more discussion.[GREN-DES]

Technical Practices

Working in iterations

Each iteration is a self-contained project with a short duration. In non-embedded agile development the goal is to deliver a working increment of the product every iteration. These increments might be released to end-users. In embedded incremental development it is not practical to consider releasing these small increments especially when there is concurrent hardware development, and high deployment costs. Because value cannot be delivered each iteration some say that agile cannot be used on embedded software development. My opinion is different. The focus on what is delivered in embedded agile is demonstrable progress. Each iteration the software development progress must be demonstrated to the stakeholders. I don't mean doing show and tell on what might be build, but rather a demonstration of real working running software. If real hardware is not available, the demos are done in the simulation, or evaluation environment.

Demonstrable progress is a key feedback mechanism because unlike documents, the code developed in the iteration can be executed showing that some part of the requirements are understood and implemented. Relying on non-executable artifacts leaves considerable risk that the design ideas don't work and that we will not discover that they don't work until too late in the development cycle. These demos provide very interesting feedback to project stakeholders and developers. For stakeholders they get to see and interact with what they asked for. For developers they get to try out their design ideas and make sure they work and are what the customer wants. Both parties get feedback on the plan.

Iterations are fixed-duration time boxes. A time box ends when the time is up, it is not extended when some committed stories are not complete, and it is not ended early when work gets done ahead of schedule (yes, this should happen).

Iterations are short because people are not very good at estimating long activities. People are pretty good at a two week planning horizon. So, the long-term plans are made up of a series of smaller plans with more precision in the current and next couple iterations and more uncertainty and freedom in the further out iterations.

Because iterations are short, the work has to be broken into small pieces of functionality that can be completed within the iteration. We call them stories. The goal of the iteration is to complete all the planned stories and make the stores' automated acceptance tests pass. The stories and their estimates are used in tracking progress against the plan.

james@grenning.net

Stories

We will have to look at stories in more depth because one of the big challenges of Agile development is breaking the system into the stories that allow incremental and visible progress on the product. Embedded agile development has even more challenges in defining stories because of the added complexity of hardware/software interactions.

A story either delivers value, shows progress or reduces some risk and can be completed within one iteration. Usually a story is considered a concise description of system behavior. In that sense stories are similar to a use cases, or parts of a use case. In use case vocabulary there is the happy path and the variations. The happy path defines what happens when all is well. The variations are also paths through the system that cover special cases, differing inputs or error cases. In embedded development we have stories like these but there is other work that does not fit this mold. That work is concerned with having the software communicate with and control the hardware. These hardware centric stories can be a challenge to fit into two-week time boxes especially when starting from scratch as many embedded developers do.

Lets say that we are integrating a USB port into a security system. In the top-level requirements, there is an item called "USB port". This requirement is too high level to be actionable. High-level requirements have to be broken into stories that describe how the system uses the USB port. For example:

- Print the event log to the printer
- Backup configuration to USB memory stick
- Restore configuration from USB memory stick

These are behavioral stories that provide value to the end user of the system. I don't see any of those stories being completed in an iteration unless the underlying infrastructure is in place. Using traditional practices we might have estimated 3 months to do the USB port and let a couple engineers go away for 3 months to get the job done. Sending a couple people away for a few months makes progress less visible and on top of that people are not very good at estimating big pieces of functionality. The lack of visibility and the inaccuracy of longer term plans can cause distrust between management and development and that should be avoided. So we have to make the progress more visible and to do that we must divide and conquer.

Divide and conquer is not new to engineering, although an agile team divides differently than most embedded developers are used to. Traditionally features would be broken down architecturally with integration after most of the work is done. This makes intermediate progress difficult to demonstrate because until all the pieces are complete there is no visible progress to the product stakeholders. With stories we make the process more visible.

Hardware integration and test stories are used to demonstrate progress toward the completion of the big story, or end-user story. When enough of the smaller stories are

james@grenning.net

completed, the big story is no longer so big because the underlying support is in place. I suppose what I am describing is an architectural break down, but we really only do that sort of breakdown when multiple teams are involved to partition work, or when there are hardware dependencies. Here are some example stories that demonstrate progress toward the USB feature integration and lead to completing the story the end user cares about:

- Talk to USB registers
- Program the device – verify clocks with scope
- Detect when a device is plugged into the USB port
- Detect a printer
- Eject a page
- Print a line of text
- Print the event log
- Detect a memory device
- Open/close a file on the memory stick
- Read a file from the memory stick
- Write a file

Many of those stories are about getting the hardware to do what is needed, and delivering a series of demos that make the progress visible. In parallel to the hardware dependent work, the backup and restore functionality could be developed in a hardware independent manner. Once the memory stick is working in the system, and the hardware independent backup and restore stories are also working, the big stories finally become small enough to schedule and deliver. We try to imagine that stories are independent, and many times we can make them independent by inserting test stubs and making simplifying assumptions, at least temporarily. But with hardware dependent stories there often is a specific order.

Test Driven Development

Test Driven Development (TDD) is the practice of writing automated test code concurrently with the production code. The TDD workflow consists of the following steps [BECK1]:

1. Create a new test
2. Do a Build, Run all the tests and see the new one fail
3. Write the code to make the test pass
4. Do a Build, Run all the tests and see the new one pass
5. Refactor to remove duplication
6. Repeat

Tests are written just before the code that makes the test pass. This is a tight feedback loop on the order of a few minutes in durations. Writing of the test defines exactly what the code is supposed to do. Then the code is written to pass the test. Development becomes a series of small milestones, each with specific feedback on the outcome with

james@grenning.net

the system incrementally growing in behavior. The automated tests are run every few minutes providing rewarding feedback on the developer progress. This predictable development workflow replaces the less predictable code, test debug workflow. My other conference paper goes in depth into TDD.[GREN-TDD]

Dual/multi-target development – Platform independence

Making progress through TDD means that the embedded software must be designed to be hardware independent. Object Oriented design principles are used to create loosely coupled software modules whether an OO language is being used or not. The software is designed to interact with hardware, operating system and other subsystems through interfaces. Subsystems and modules must be kept independent so they can be tested before integration in the final hardware. Early in the project, the development system or evaluation hardware may be the only test vehicles. Testing on the development system or eval boards allows testing to be done earlier, so that when hardware is ready, less application problems are found.

Design

Agile design is evolutionary. There is no attempt to completely layout the high level and detailed designs prior to starting development. This is not to say there is no upfront design, there is and the amount of upfront design needed varies per project. A project consisting of a half a dozen developers probably will not need much up front design. A project made up of several teams of a half-dozen developers will need more up front design work so that teams work synergistically.

At the beginning of an agile project is an activity called exploration. During exploration the project goals are communicated, initial stories are written, initial hardware/software division of responsibilities is identified and an initial software architectural vision is created. The architectural vision identifies the system boundary, and major subsystems. The design shows example commands, queries, and events that allow one subsystem to collaborate with another.

The architectural vision is not an exhaustive/comprehensive design document. For smaller teams it may be recorded on a white board and only take a few hours or a couple days to establish. For larger teams more artifacts may be needed. The real value of the architectural design is to partition the work so that there are responsibilities for each area, allowing multiple people and/or teams to work in parallel.

The architectural vision is not realized subsystem by subsystem with a big-bang integration at the end. Stories are implemented that cut across the different parts of the design, initially with modules and subsystems with reduced capability. As more stories are added to the system the architecture evolves, becoming more and more complete.

james@grenning.net

An architectural vision is not cast in stone, it is not rigorously reviewed, approved and signed off. The agile development team expects that some of the vision will come true, but parts of it will change and evolve as the team learns more about the requirements and the design ideas that work well and the ones that have not worked so well.

Agile development does not prescribe the documentation needed by a development team. A team may decide that intra-group communications using the team's white boards are adequate for effective communications. Multiple team projects or distributed projects will need more documentation and formality. Software in regulated business such as medical devices, may have documentation requirements along with the functional requirements.

When a document is needed because it is valuable to the team, or required by your customer, the team will try to find cost effective ways of getting the document. For example, if an architecture document is needed, the team may first get a working portion of the architecture coded and tested prior to producing the formal document. The team first works from a rough sketch of the architecture, and later after confirming the architecture the team documents it. This keeps the effort to produce and review the document down. Writing the document in anticipation leads to either an out of date document at the end of the project, or additional cost of revising the document as the architecture is revised and made to work. An implication of this approach is that the architects are also doing implementation. Some skilled designers are needed on every team.

Stories and Executable Specifications

Another aspect of Test Driven development is that much of the system specification is in the form of tests. An acceptance test provides the definition of *done* for a given story. The QA or system test function is moved from the bottom of the waterfall to the front of the process and becomes a specification role. Tests are written in an application specific test language, with many of the tests being runnable on the development system or evaluation hardware.

An acceptance test forms an executable specification that is somewhat similar to a use case. This dual function aspect of the automated acceptance tests can reduce potential waste and duplicate efforts of writing narrative requirements documents, use cases, and test cases.

Business Practices

Agile development involves more than just technical issues. The iterative nature of Agile impacts the whole development organization. Companies that view Agile as just solving technical problems will probably fail at agile adoption. The most successful adoption of agile happens when management and development are both interested in solving the problems of late projects, inaccurate estimates, and low quality. Specifications, schedule dates, and plans cannot be thrown over the wall to development. Management and

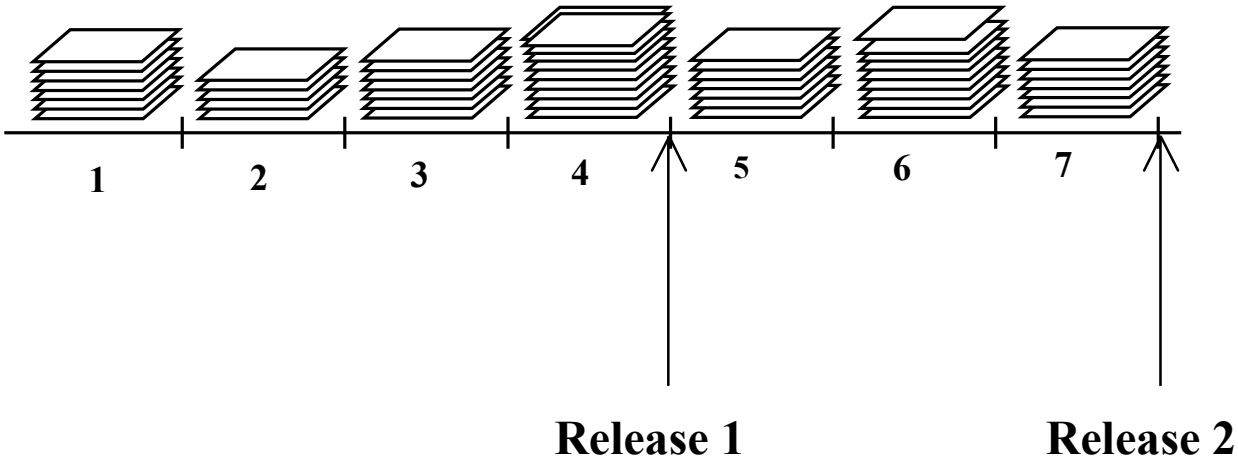
james@grenning.net

development must work together to steer the project to a successful delivery within the constraints articulated through requirements, resources, and dates.

To steer a development team, a customer team is needed. A product manager usually leads the customer team with support from test engineers, product specialists, or systems engineers. When there is hardware/software interaction, having hardware engineering represented on the team is also needed. The hardware engineers often help identify the hardware integration stories needed to realize the product.

The customer team works with the development team to produce a release plan, which is a series of iterations with critical dates identified. Each iteration in the release plan is made up of a set of stories. Stories are written on note cards to facilitate rearrangement during the planning meetings. The stories are estimated by the developers and are treated as independent. Stories are estimated in relative, but unit-less numbers. So the easiest story usually is assigned a single point and more difficult stories are estimated relative to that easiest story. So a five-point story is five times as difficult as a one-point story. The team estimates how many story points it can complete in an iteration, this estimate is known as the team's velocity. The stories grouped such that the sum of the estimates of the stories in the iteration do not exceed the teams estimated velocity. Figure 3 represents a release plan. Each iteration is made up of a stack of stories.

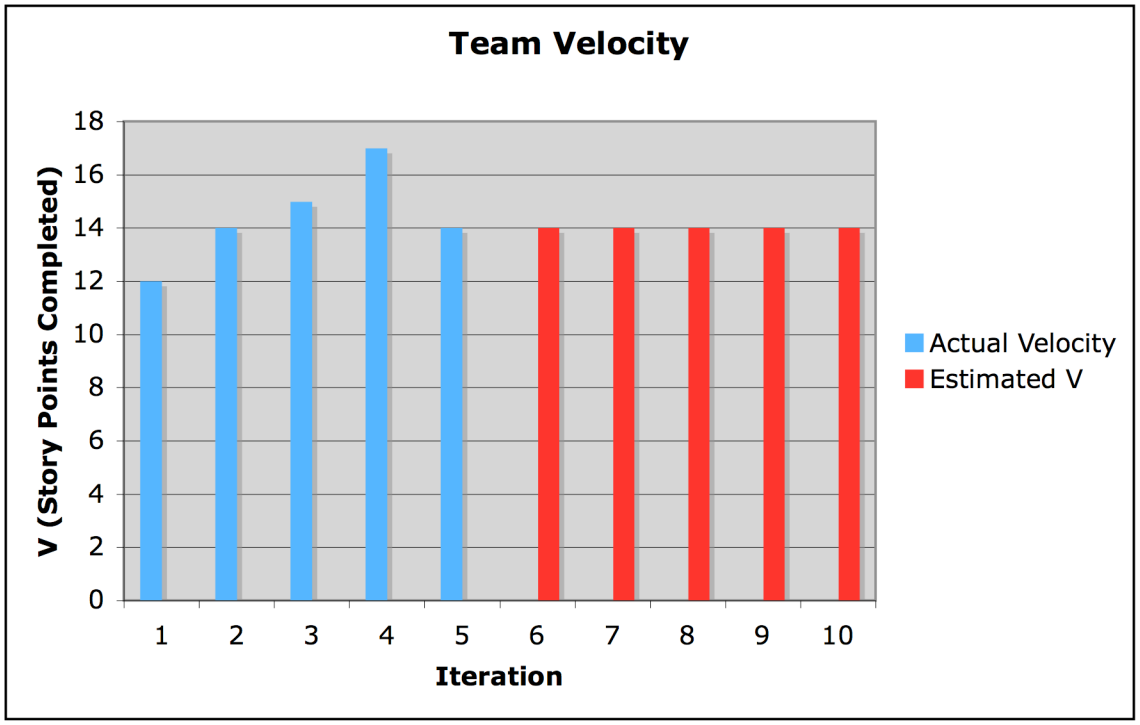
Figure 3 – Release Plan



After a few iterations the team will develop its historical velocity, which is the total of the points of stories actually completed by the team in an iteration. The teams' velocity is the critical metric that provides feedback on the team's measured progress. Figure 4 shows a team's velocity tracking chart.

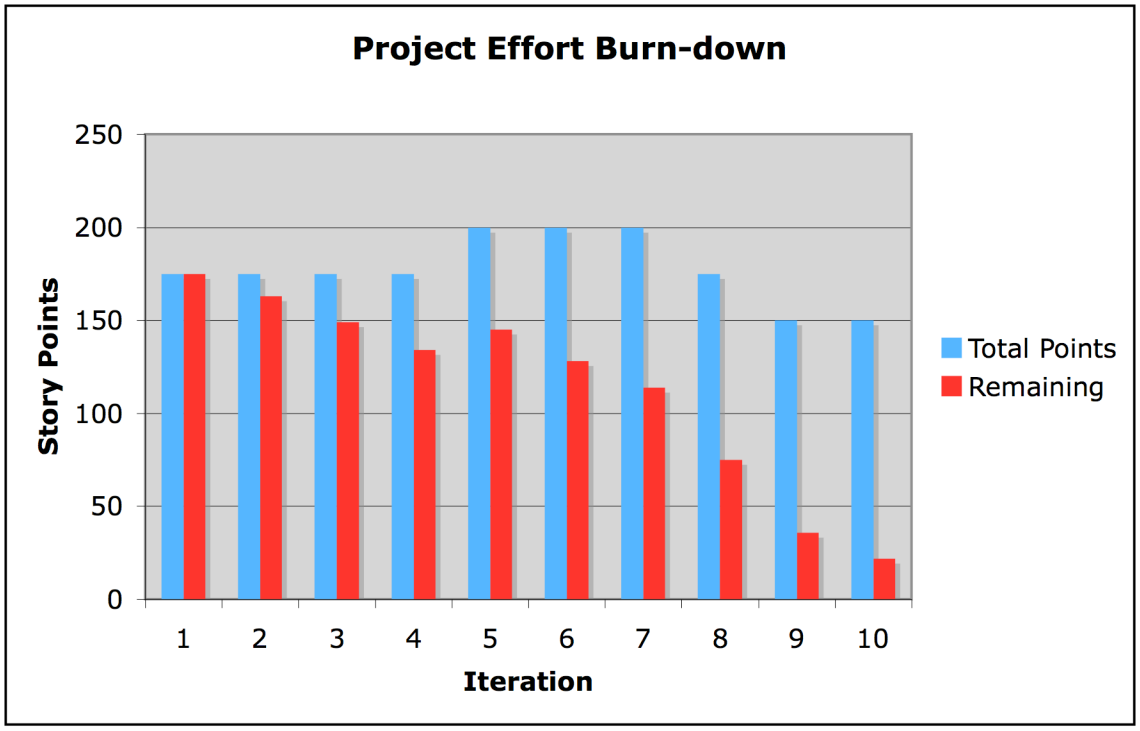
james@grenning.net

Figure 4 – Velocity Chart



If the estimated velocity was optimistic, as it usually is, that optimism is checked by the actual track record of the team. Optimism is good but the business needs to know is the plan is realistic. Figure 5 shows and burn-down chart which is used to monitor project progress.

Figure 5 – Burn Down Chart



The velocity tracking and burn-down tracking can provide an early warning system for schedule problems, and this early warning gives the team management time to take corrective actions, such as reducing scope, adding resources, extending the date, or possibly cancel the project. In this chart there is a new batch of requirements introduced in iteration 5. The velocity showed that the project was likely to be late, so in iteration 8 scope was reduced to 175 point and further to 150 points in iteration 9.

The most practical variable to control is project scope. With the behavior of the system having been broken down into stories, there are small bits of functionality that may be removed and rearranged in the plan. This is a highly visible planning process and this visibility can be used to remove higher-cost and lower-value stories. No one wants to cut scope, or delay the project, but knowing that the plan is in jeopardy is critical business information.

The nature of agile planning is that long term plans are less precise with more uncertainty than the short term plans. The upcoming iterations in the release plan are more detailed and the iterations that are farther out have less precision. As time goes by the plan becomes more complete and the confidence in the plan improves.

Final Words

This paper could really only do a broad-brush treatment of Agile Development. You probably recognize many of the practices of agile development. Agile is not new, some

james@grenning.net

of these practices have been around for decades and have been very successful. Developers, business stakeholders, and end users should see improved schedule performance and reliability. Developers should feel the accomplishment of regular feedback of iterative development, TDD, and spend a lot less time chasing bugs. Business stakeholders should see improved predictability and visible fact-based management data.

There are some other very good references on agile development practices.

- Beck, Kent, Extreme Programming Explained
- Cohn, Mike, Agile Estimation and Planning
- Cohn, Mike User Stories Applied
- Martin, Robert, The Principles, Practices and Patterns of Agile Software Development
- http://objectmentor.com/resources/publishedArticles.html

Agile development can also be found by other names, such as:

- Extreme Programming
- Feature Driven Development (FDD)
- Scrum
- Crystal Clear
- DSDM

[LARMAN] Larman, Craig and Basili, Victor, Iterative and Incremental Development, a Brief History, IEEE Software, June 2003 Cover article

[AGILEMAN] http://agilemanifesto.org

[AGILEP] http://agilemanifesto.org/principles.html

[FITNESSE] Open source Framework for Integration Test, www.fitnesse.org

[GREN-DES] Grenning, James, Object Oriented Design for Embedded Software Engineers, ESC-209, San Jose 2007

[BECK1] Beck, Kent, Test-Driven Development, Addison Wesley, 2002

[GREN-TDD] Grenning, James, Test Driven Development for Embedded Software, ESC-241, San Jose 2007

james@grenning.net