

Agile Project Management (or, Burning Your Gantt Charts)

Embedded Systems Conference Boston (Boston, Massachusetts)
ESC 247-267, October 2008

Michael Karlesky
Atomic Object
karlesky@atomicobject.com

Mark Vander Voord
X-Rite
mvandervoord@gmail.com

Atomic Object, 941 Wealthy Street SE, Grand Rapids, MI 49506
<http://atomicobject.com>

X-Rite, 4300 44th Street SE, Grand Rapids, MI 49512
<http://xrite.com>

ABSTRACT – Embrace change. Schedules, budgets, competitive environments, and requirements change. Unanticipated, difficult problems are discovered during development. Traditional embedded project management usually hinges on eliminating change and designing out uncertainty up-front; such an approach is fantasy. Agile Project Management offers solutions to common, persistent problems: poor estimates, slipped timelines, products languishing in an almost-done state, and Gantt charts rarely reflecting reality. Here we discuss: usable definitions of “customer”, “feature” and “done”; prioritization; iterations; estimation; burndown charts; documentation; and risk & scope management.

Introduction

PROJECTS change. This simple fact is not fundamentally due to a lack of planning or incompetence on the part of project managers and software developers. Rather, change is an inherent characteristic of any growing entity. Embedded projects grow as much as they are built. Living things adapt to their environment. The environment surrounding any embedded project is ever in flux. Budgets change. Resources change. Schedules change. Competition changes. Customer needs change. Even if this changing environment could be eliminated, another form of change would continue to affect embedded projects. A project learns as it grows and must change in response to this learning. That is, as features come to fruition, the developers, users, customers, and managers become more fully aware of the project’s reality. That reality is a project’s limitations, its capabilities, what is needed of it, and how users desire to interact with it – all these are discovered over time. These discoveries will inevitably drive project change.

For the purposes of our discussion of project management we make the following distinctions. First, by “traditional project management” we refer mainly to any methodology where software development is viewed as a specialized version of manufacturing or as a construction project. This type of project management is identified by its sequential phases of design, implementation, and testing (the “waterfall” approach) planned out through critical path analysis (usually represented via Gantt charts). Second, we address here only those projects that include any sort of variability or unknowns in their requirements. Certain “black box” projects

limited to well-defined data manipulation or I/O transformations may not fall within the realm of the philosophies and techniques we describe here. New product development, R&D development, and projects requiring user interface work (i.e. the majority of embedded software projects) are well suited for Agile project management.

Overview of Traditional vs. Agile Project Management

Traditional project management views change and rework as the most expensive aspects of software development. As such, it attempts to drastically limit, even prevent, change through extensive upfront planning, design, and documentation. Conventional project management wisdom holds that if change happens during a project, insufficient planning, design, and documentation occurred. Traditional techniques advocate for a development path that moves orderly from laying subsystem foundations through implementing middleware layers and finally on to feature integration.

Conversely, Agile project management (based in Agile software development techniques[1]) views project failure as the most expensive aspect of software development (e.g. software that never ships, slipped schedules, marketable features never realized, and quality failures). It holds that change happens because change happens; change is something to be managed rather than avoided. Agile project management views planning, design, and documentation beyond the minimum necessary to be waste. It focuses on delivering working features to a paying customer as soon as possible, building supporting subsystems and refactoring the code-base as needed to support said features along the way.

Key Agile Concepts: “Customer”, “Feature” and “Done”

The philosophy of Agile project management is oriented around a handful of central ideas. All practices, metrics, and decision making come back to these core concepts. To lay the groundwork for explaining the techniques of Agile project management, we must answer three questions. Who is a customer? What is a feature? When is a feature done?

Customer Defined

The customer pays for software developed; this much is quite obvious. In Agile project management, the customer’s role encompasses more than this obvious definition. A customer is the single point of contact in making decisions on direction, prioritizing features, and answering domain questions. They are as close to the development team as possible; ideally they are present as a full-time member of the team providing decisions, priorities, exploratory testing, usability feedback, and research. Though many people may be involved in these activities, a single person acts as the voice of the customer.

In developing complex technologies or creating software for complex business situations, more complex definitions of customers can be necessary. Multiple customers may exist internally in systems engineering, production, hardware engineering, marketing, etc. Each of these customers’ decisions cost money and require that value be delivered to those customers. In such cases, a project manager must coordinate priority decisions among these individuals, maintaining a single course for the development team to navigate.

The end user of a product and the customer may very well be different people. This generally occurs when a company develops a product to be sold to a particular market; it is less common

for projects consumed internally. In these cases, it becomes critical to fold user feedback into development early and often thus requiring features to be delivered as soon as possible (more on this in later sections). Bringing end users as close to the development as possible engenders understanding of their needs and aids the customer in making appropriate decisions.

Feature Defined

We define a feature from the customer's perspective. A feature is a unit of functionality: (1) described by the customer in his or her own words in terms of system behavior rather than implementation details (2) verifiable in its completion to the satisfaction of the customer (3) deemed valuable enough to the customer to be paid for by said customer. Valuable software meets the needs and desires of the paying customer. As such, in Agile project management, all efforts are centered on delivering features to the paying customer.

Features are not tasks, nor are they modules or subsystems. In Agile project management, features are short, high-level narratives capturing the customer's expectations of system behavior (often referred to as "stories"). As such, features are not concerned with implementation details, only user recognizable functionality and value. A well formed feature or story should take no more than a week or two to implement. An example might read: "When a user holds the power button for three seconds, the device should power down."

Tasks exist solely in the realm of a developer's day-to-day activities. A collection of tasks comprises the technical implementation details of accomplishing a feature. A progression of completed tasks leads to the realization of a feature.

Modules, subsystems, and architecture are implementation details that support the delivery of features. This, in itself, is not a departure from traditional project management. However, in Agile project management, we invert the understanding of building software. Concentrating on working features is given priority over building out supporting subsystems. We will elaborate on this point in a later section.

Done Defined

On its surface, "done" appears to be such a simple idea as to be unworthy of elaboration. Yet, with traditional project management practices determining when a feature or even a project is done often requires extraordinary effort and generally produces little exactness. For instance, "done" is often such a poorly grasped idea that development teams speak in terms of "done" and "done done." The former describes when a developer believes a feature is complete; the latter describes when the feature is truly complete (though this state of completeness is likely to be quite indeterminate as well). Anecdotes abound of Gantt charts indicating 95% project completion, yet an amount of time far exceeding the final 5% of the project timeline is required to finish the embedded software development effort.

In Agile project management "done" has a specific definition and represents a measurable state of completion. A feature is done when it has thorough test coverage and passes all unit, system, and acceptance tests. Ideally, the unit and system tests are automated and can be run as a regression test suite. Unit tests and system tests are created by the developers in parallel with production code (ideally, the tests precede the production code in the manner of Test-Driven Development). Acceptance tests are performed by the customer; these can range from scripted manual operations to variations of end-of-line production test equipment.

Practices

Like all project management methodologies, Agile project management is a set of practices and philosophies. These form a symbiosis among themselves. As they say, the whole is greater than the sum of its parts.

Testing

We have defined done in terms of tests. Without tests and this measurable means of determining the state of a feature, many of the practices of Agile project management become impossible or ineffective. One of the most effective forms of testing is Test-Driven Development (TDD). In TDD a developer uses automated, executable, regression tests to capture the effect of production code to be implemented before said code is implemented. With those tests in place, the developer can then confidently fill in the production code. This approach is best applied in both unit tests (tests written in the same environment as the production code exercising the system's innards) and system tests (tests that exercise the system externally). Testing of this sort is an advanced practice[2] well worth the benefits in quality, design, confidence, and metrics it provides.

Iterations

In general, humans are better at managing highly detailed work over short intervals of time than long ones. That effectiveness is increased even more when those short intervals of time are well defined and repeated in such a way as to develop a rhythm. Traditional project management often stages work in phases lasting many weeks or months. These periods are simply too long to effectively monitor progress, react to changes and new knowledge, and take action with sufficient time as to prevent schedule problems. Agile project management uses development iterations to break up long projects. With these short, defined, repeated periods of time (on the order of one or two weeks), metrics can be gathered and used to predict and manage schedule changes after completing only a small number of iterations. Estimation, metrics, and forecasting (all interconnected to iterations) are addressed in later sections.

Feature-Driven Development

Our goal in software development is to deliver working features for which a customer is willing to pay. We deliver features not architecture or subsystems. Software architecture and subsystems exist only to support features. All project management, therefore, should be oriented around delivering features. Agile project management does just that.

In Agile project management, we ask the customer to prioritize features each iteration. This guides where programming effort is directed and allows priorities to shift each iteration as needed in response to circumstances. Delivering features early and often allows end users to work with and test software long before the final test phase of traditional project management.

This approach invites users to offer feedback that can be folded into iteration planning and feature prioritization. In learning iteratively as the project grows, features of little value can be cut from the program (saving time and money) while those that offer the most value are discovered and implemented. This technique can radically change the initial set of requirements. However, by regularly re-evaluating feature priorities, we can guarantee that what is most important to a customer is always accomplished first. Should schedule or budget changes cut the project short, the most important and valuable features are those that have already been

accomplished.

Simplest Thing That Could Possibly Work

Unneeded complexity is unneeded cost. Complexity increases the chance for introducing error, complicates testing, slows progress, and obfuscates code (thus increasing maintenance and documentation effort). The simplest thing that will achieve the goal at hand yields cleaner code, faster progress, and greater system efficiency than a more interesting, complicated solution.

Simplicity has more enemies than merely complexity. The easiest solution to a problem is not necessarily the simplest solution. Ease of implementation can drive code duplication, overloaded functions, excessive nesting of conditionals, etc. Striving for simplicity yields the most elegant solution to a problem with unneeded complexity and a clean implementation.

Driving development via unit and system tests (i.e. Test-Driven Development) provides a suite of regression tests. Those tests act as a safety net protecting against bugs and broken code giving confidence to developers to refactor code towards simplicity.

You Ain't Gonna Need It

Traditional views of software development and project management strive to avoid change. Developers and project managers who subscribe to this viewpoint build subsystems and software architecture to meet every foreseeable need of the layers of software to follow. In reality, much of the functionality provided by such foundational layers of software will never be consumed by the software developed later in the project. Any functionality built early in a project that is unneeded later in the project is waste. Further, the unneeded effort taken to introduce unnecessary complication creates barriers to change the software and slows progress toward the ultimate goal of delivering valuable features.

Agile project management takes a different approach. Developers implement only the software needed at the time they are creating it. Relying on suites of regression tests, developers can refactor existing code and add functionality when actually needed at later stages of the project.

Estimation

Planning Poker

Predicting the future in any sort of meaningful way is difficult. Thus, software estimation is difficult. A high percentage of traditionally managed projects do not meet their time estimates and thus either fail or are completed with serious budget overruns or lacking important features.

Agile project management addresses the failings of traditional software estimation by recognizing an inherent limit in humans. We are not particularly capable at estimating how much time a given task or group of tasks requires. This inability is only compounded in large projects comprised of a significant number of tasks. However, humans are quite good at estimating relative complexity (e.g. A is twice as complex as B). In the place of time-based estimation, Agile project management employs complexity-based estimates.

Planning Poker is a technique used to facilitate arriving at complexity estimates within a development team. A numbering scheme is arbitrarily selected; popular choices include powers of two (1, 2, 4, 8...) and a simplified Fibonacci sequence (1, 2, 3, 5, 8...). The high end of the scale is capped close to 10. Individual features (suitably decomposed to reasonably fit within an iteration timeframe) are announced and discussed followed by each developer revealing a card with his or her complexity point estimate. To prevent influence, these estimates are not discussed

until after they are first revealed. Any divergence in the numbers spawns conversation until general agreement is reached on a complexity points value.

As features are estimated, they are placed in a backlog with their complexity estimates. Some amount of time at the beginning of a project is necessary to estimate all features known at that time. The result of planning poker is a comprehensive set of complexity points that taken together represent the entire complexity of the project. Correlating these point values to time requires measuring velocity and plotting predictions with Burndown charts.

As a project progresses and change inevitably occurs, features may be dropped from the backlog altogether while new ones are added. Each time a new feature is conceived and added to the backlog, a quick round of Planning Poker is played to estimate the complexity points of the new feature. Any errors in complexity estimates tend to average out to a net error of zero over the course of a project (i.e. some features are more complex than thought while others are less complex).

Velocity & Project Forecasting

To correlate complexity points generated in Planning Poker to a project timeline we must calculate velocity. Velocity is a simple ratio of complexity points completed per iteration. That is, for each feature marked as done (its tests all pass), its complexity points are credited to the iteration in which it was completed. Velocity can be calculated as a weighted historical average favoring recent iterations (as these are most representative of the current rate of progress looking forward) or as a simple average of the most recent two or three iterations.

This method requires at least a handful of completed iterations before meaningful project completion estimates can be generated. However, with that overhead out of the way, a meaningful and ever-current estimate of project completion is available. Project completion is forecasted simply by dividing the backlog of complexity points by velocity; this yields the number of remaining iterations in the project. Because features are truly done when all tests pass, we have confidence that past work is truly complete. Looking forward, we can accurately size the amount of work and resources available. If project completion extends past the budget or delivery date, resources, features, and priorities can be adjusted months before the difficult truth would become apparent using traditional methods.

This technique pins relative complexity to the reality of time. It also very naturally absorbs all overhead not directly associated with programming. That is, meetings, coffee breaks, unexpected absences, etc. are folded into velocity calculations. A great deal of overhead will be revealed in a slower velocity. Adding resources can increase velocity, but there are, of course, limits to this effect. This too is shown by velocity. By tracking the number of developers on a project and comparing that number to velocity over time, the natural optimum for the team and project become quite obvious with simple math.

Burndown Charts

Burndown charts are visual representations of the total backlog of complexity points and progress made against that backlog over time. It presents the raw data that yields velocity numbers in a way that allows quick visual analysis. Extrapolating lines on the chart easily reveals a great deal about project completion – even quite early in the project.

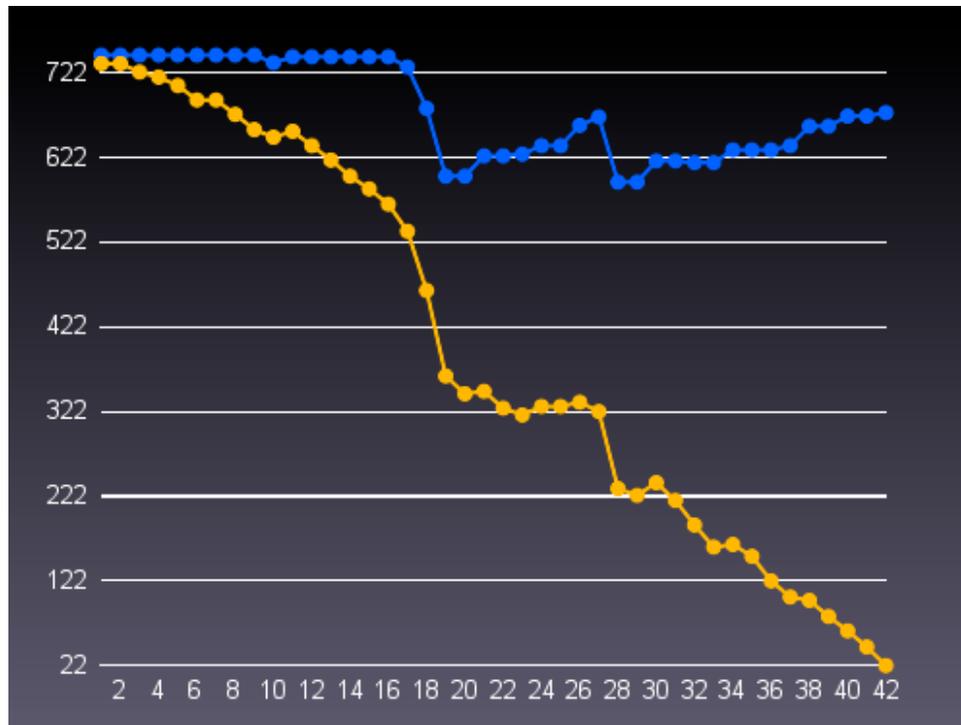


Figure 1 – Burndown Chart (Complexity points on Y axis; Iterations on X axis)

Figure 1 shows an actual burndown chart from a large project. The blue line represents the total number of complexity points in the project. Note how it fluctuates over time. A drop in total complexity points demonstrates moments of project re-evaluation where features were removed from the backlog. An increase demonstrates re-evaluation where features were added. The drops seen in Figure 1 were specific efforts to simplify the project and pull in the completion date. The increases shown in Figure 1 were additions of features learned to be essential as the project progressed. Note that these decisions occurred months before the end of the project specifically because there was data that demonstrated these decisions had to be made to meet goals. The orange line in Figure 1 represents the completion of features over time. It tracks with the blue line but maintains a downward slope towards completion. Note how the graph shows an increased velocity at the tail end of the project (though the line has the same slope as earlier times in the project it occurred over a time period where many features were added to the project).

Continuous Integration

The technique of continuous integration regularly brings together a system's code and ensures via the regression test suite that new programming has not broken existing programming. Automated build systems allow source code and tests to be compiled and run automatically. Continuous Integration ensures the system's code-base is always thoroughly tested and has no integration problems among subsystems or sections of code. Integration problems are discovered early when it is cheapest to correct them. Further, any such problem will be discovered close to where and when the problem was created; here, understanding is greatest and good design choices are most likely.

Documentation

Traditional views of software development and project management favor extensive documentation. Extensive planning and architectural documents are written before the project commences. More documentation is added to the source code while it is written. More documentation still is created once the project is completed. The fundamental difficulty with software documentation is its short shelf life. Even small changes to the source code of a system can invalidate significant amounts of documentation.

Agile project management favors limited, flexible, just-in-time documentation. To the extent that contractual obligations will allow, initial and final documentation should be simple, high-level overviews of important features and subsystems that are unlikely to change. Unit and system test suites act as executable documentation on the system's source code behavior and architecture. As tests are updated, this living documentation is updated as well. Development teams can utilize flexible, collaborative documentation systems such as wikis to capture and easily update essential procedures, setup instructions, and command interfaces. Such systems are effective at communicating among the team itself and the larger organization. The RaPiD7[3] documentation technique delays static documentation and manual generation until the last possible moment. In this method all parties involved in a project are brought together for a single day documentation sprint to be edited and refined later by a single editor. This concentrates documentation effort, maximizes communication among the team, and ensures documents are as up to date as possible before they are shipped.

Risk Management & Scope Management

With Agile project management, managing risk and scope are, in fact, quite simple to accomplish. The riskiest portions of the project are prioritized to be completed first. As each iteration is completed, the burndown chart is updated and decisions are made on existing and new features. Velocity calculations give such foresight that resource and feature planning can be actively and preemptively adjusted to meet release schedule and budget constraints.

Conclusion

Traditional project management is insufficient to manage the inevitable change inherent to embedded software projects. Agile project management, however, is well equipped to aid project managers and software development teams in managing risk, scope, budgets, and schedules to create successful, valuable products.

References

- [1] Kent Beck. *Extreme Programming Explained*. Reading, MA: Addison Wesley, 2000.
- [2] Michael Karlesky, Greg Williams, William Berezka, Matt Fletcher. "Mocking the Embedded World: Test-Driven Development, Continuous Integration, and Design Patterns" *Embedded Systems Conference Silicon Valley*. San Jose, California. April 2007.
- [3] Roope Kylmäkoski, "Efficient Authoring of Software Documentation Using RaPiD7," *icse*, pp.255, 25th International Conference on Software Engineering (ICSE'03), 2003.