Are Parametric Techniques Relevant for Agile Development Projects?

White Paper

Abstract
This paper addresses the following question: Can I use a parametric model to model my agile software development and if I can how do I need to adapt the model? The first section contains an overview of agile software development, along with a brief description of its principles and practices. In the next section we tackle the issue of 'sizing' an agile project for a parametric model. The following section contains some cost driver guidance around some common agile practices.

PRICE Systems

PRICE Systems provides more than 40 years of cost estimating software expertise across a variety of government and commercial market applications and business needs. PRICE cost models benchmarked from 16,000 major projects, ongoing PRICE research into refining predictable cost estimates, and new PRICE frameworks for integrating client-specific cost history, put PRICE Systems at the forefront of cost estimating and cost management solutions. For more information, visit PRICESystems.com

Are Parametric Techniques Relevant for Agile Development Projects?

Contents

Are Parametric Techniques Relevant for Agile Development Projects?

Wikipedia offers a nice overview of the history of software engineering describing the stops and starts that have led us to where we are today. [1] In the beginning, the complexity of the applications being developed was almost overshadowed by the logistics of the actual implementation process. People didn't have personal computers and feedback was not instantaneous – and it was impossible to make predictions about when a project would be complete. As technology and tools improved, software was used to solve increasingly complex problems so the logistics demurred to the solution as the bottleneck – and it was still almost impossible to make predictions about when a project would be complete. The 'software crises' describes a twenty year period from the mid 60's to the mid 80's during which the industry was beginning to understand issues around productivity, quality and good architecture. This 'crises' spurred projections of a plethora of 'silver bullet' solutions such as structured programming, Computer Aided Software Engineering (CASE) tools, objected oriented design and programming, standards, defined processes, formal methods, the Capability Maturity Model (CMM) and many others. All of these tools, techniques and methodologies have made improvements to the overall productivity and quality of software. Then along comes the Internet, significantly increasing the complexity of things that could be accomplished with software. In light of emerging technologies and growing complexity, the formal processes, methods and standards were viewed by many as an impediment to progress rather than supporting projects' productivity and quality goals. Many smart software development professionals began to trend toward lightweight methodologies or what we currently refer to as agile development.

In February 2001, a group of these smart software development professionals sat down to talk about these lightweight methodologies. The result of these discussions was the Agile Manifesto [2]. This manifesto introduced the following tenets for software development:

We are uncovering better ways of developing software by doing it and helping others to 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*

Agile development processes rely on experienced, highly skilled people communicating with clients and each other to deliver software that provides the clients with the most value for their money. This requires both developers and consumers of software to accept the reality that things will change over the course of the project and that the software that is eventually delivered may not be the same as what was envisioned when the project began.

At any given time, the agile software development team is only working on the feature ranked most valuable by the customer. Estimation is performed by the team and is only focused on the feature that is currently on deck. At the end of an iteration, the customer may review the implementation to date and may reprioritize remaining features based on the current state of the software or changes in their business or expectations. Estimating beyond the current iterations doesn't really make agile sense.

Unfortunately, the fact that estimation doesn't make sense for the agile team does not mean that it doesn't make sense for the business. The business needs to see the forest not just the trees. Customers need an idea when their software will be delivered, businesses need to prep the market for new features, and they need to be able to optimize resource allocations across many projects, etc. This of course begs question of how to perform an estimate for software when you really don't know exactly what software you're going to build.

Parametric techniques are particularly suited for this task, especially for an organization that has some historical data from previous agile projects. Parametric estimating models provide a repeatable framework through which an organization can study their past performance on similar projects and use what they learn to perform an estimate on the project at hand. Team history can be used to map Story Points or User Stories to a size measurement such as function points, use cases or source lines of code. Analysis of performance on past projects can inform decisions about productivity and other project drivers. This information can be used to drive the parametric algorithms to develop estimates for cost, effort and schedule.

This paper addresses the following question: Can I use a parametric model to model my agile software development and if I can how do I need to adapt the model? The first section contains an overview of agile software development, along with a brief description of its principles and practices. In the next section we tackle the issue of 'sizing' an agile project for a parametric model. The following section contains some cost driver guidance around some common agile practices.

Agile Software Development

When discussing agile development it is important to understand that agile is a set of tenets. Any project that is adhering to these tenets can be considered an agile project. There are no rules or standards emerging from these tenets - they just propose an overarching philosophy with which organizations should approach software development.

A traditional software development projects begins with a set of customer or user requirements. These requirements are analyzed, architecture and design are created and the set of requirements are implemented, tested and delivered to the customer. With such an approach it takes months or years to get to the point where there is usable software for the customer to use and evaluate. If there was misunderstanding or confusion about the requirements it may not come to light until much software has been developed. This has led to more than a few software failures in industry.

Agile development projects develop small usable chunks of software in short periods of time, evolving the software incrementally over time without ever being too far away from releasing a usable piece of software. Software is delivered in short increments or sprints lasting anywhere from a week to a month. The end user or customer works with the agile team so that misunderstandings and confusion around requirements can be addressed quickly and proactively. Agile teams are self organizing, cross functional and expect and embrace change.

A brief (simplified) example of a typical agile project is appropriate here. An agile project also starts with a set of requirements. User stories are developed to describe the features necessary to implement a requirement. User stories should deliver business value and should be small enough that they can be completed in a single iteration or sprint. The agile team collaborates with the customer to reach a level of understanding of each story and, as a team, develops a high level estimate for each story. These estimates are generally done using a notional size unit that is only relevant to the agile team. The stories are then prioritized by the customer into a backlog and the team begins development on the highest priority item. At the end of the iteration the team delivers working software that contains the high priority items. While there is often not an actual release of the software with each iteration, it is available for walkthrough with the customer so that the customer has a chance to determine whether the requirements were properly understood and implemented and to give the customer a chance to change the priorities. The team meets to perform a retrospective on team performance during the last iteration and to begin planning the next iteration which will tackle the next items in the backlog. Teams rather quickly determine a team velocity (number of story points they can complete in an iteration) and can plan subsequent iterations based on realized velocity. A release occurs when all the requirements have been met or a sufficient amount of capability has been added to make a release sensible.

This is a high level concept of how an agile project might progress but the reader should be aware that there are many forms of agile and many different ways to incorporate the agile tenets into a software project. An agile approach embraces some or all of the following principles:

- Customer satisfaction through customer involvement and rapid delivery of useful software
- Focus on business need and business value
- Time box development
- Constant collaboration and communication between business, developers and customer
- Development at a constant, sustainable pace
- Adaptation to change
- Self organizing teams
- Collective ownership and responsibilities
- Commitment to measurement

Different agile teams have different ways of applying these principles. One should think of agile as an umbrella, not a methodology. Various methodologies have been developed proposing a set of practices to realize the agile tenets. Scrum, Extreme Progamming (XP), Lean, and Crystal are examples of such methodologies. Some of the practices within these methodologies include:

- Pair programming or other forms of peer review
- Continuous integration with automated testing
- Test Driven Development
- Daily standup meetings
- Co-located teams
- Code refactoring
- Small releases
- Customer on team
- Simple design

Each of the methodologies available incorporates some or all of these practices, though the terminology may differ from method to method. It is important to note that implementing organizations often do not apply all of methods practices but rather pick and choose those that make sense for the company culture and types of software development projects they normally work on.

Agile Size Estimation

From one perspective, the estimation of an agile project is a no brainer. At the beginning of a project you have a high level estimate of story points and you know how many story points the team can accomplish per iteration. If you also know when the software has to be delivered you just do the math and you know the cost and effort for the current set of requirements. Unfortunately it's not often that simple. The estimates from the development team at the beginning of a release are top level and are based on a measure that is entirely notional.

There are several ways of characterizing the kinds of estimation that an agile team performs. At the start of each iteration, planning takes place where the development team breaks the user stories for just that iteration into tasks. As this is done the team estimates the effort hours required to accomplish each task. These estimates generally tend to be very good.

Estimation at the story level is generally done using Story Points (some people refer to this as estimating by Fibonacci[4]). Basically the team creates a relative scale against which to rank each story relative to the every other story. For example if a team is using the Fibonacci series (0,1,2,3,5,8,13,21,34,55,89,144) and they assign a 3 to a story that means that story is 3 times more complex than a story assigned a 1. One can think of story points as a convenient way to merge the concepts of software size and software complexity. A Story Point estimate may be high because there is a lot of code to write to deliver the story or because the small amount of code delivers some highly complex functionality. Story points are notional and relevant only to the team that is estimating them. There is no way to look at a delivered User Story and 'count' the story points required to deliver it. There is no standard in the industry for story points and to try to create one would hinder not help agile projects.

Estimation at a release or epic level is the most challenging. One way that teams accomplish this is with t-shirt sizes. Everyone knows what a t-shirt is and we all can visualize the difference between a small and an XXL. Teams still need to assign story points to t-shirt sizes so there are consistent definitions within the team but they find them to be an excellent tool to facilitate communications with less technical sides of the business so that the organization can get a handle on the scope of the requirements. Clearly this estimate tends to be the one with the most risk since it is high level and based on a less than clear picture of the final product. Sound familiar? This is where parametric cost estimating excels.

A frequently asked question from estimators is how to translate Story Points to a more standard size measure that makes sense to a parametric cost models. Clearly there must be some sort of a proxy to Function Points or Source Lines of code so the top level information from the development team can be translated into an estimate early on in the project when the business needs to understand scope and plan campaigns and resource assignments. Well, the answer is yes and no. Remember that Story Points encompass two facets of the project – the amount of code that needs to be written or touched and the overall complexity of that code.

A study was conducted at PRICE systems to study the agile data collected from our projects in order to find out what kind of relationships we could see between our story points and more traditional parametric cost drivers for software. We routinely collect information for each iteration including velocity (number of stories points completed), number of lines of code added, changed, and deleted, and hours spent by the team. Comparisons of ESLOC to Story Points and Story Points to Effort offered no interesting or useful trends. What we did find was a very close trend between our ESLOC per hour and an input in the PRICE model that we call Functional Complexity. This input indicates to the model the degree of complexity in the code based on the how complex or simple the functions the code performs are determined to be. All parametric cost models for software have one or more inputs intended to express this complexity. Through a calibration exercise we were able to map Functional Complexities of the code with ESLOC per hours and were able to make some very useful observations. Figure 1 shows the relationship between ESLOC/Hour and Functional Complexity and Figure 2 shows how this calibration creates a good method for trending cost with story points.

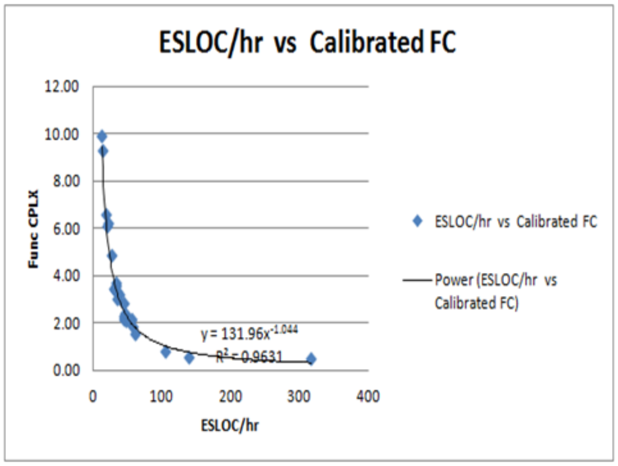


Figure 1: ESLOC/Hr vs FC

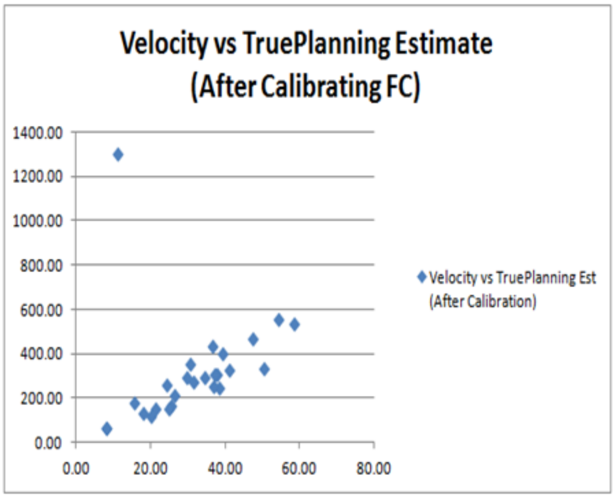
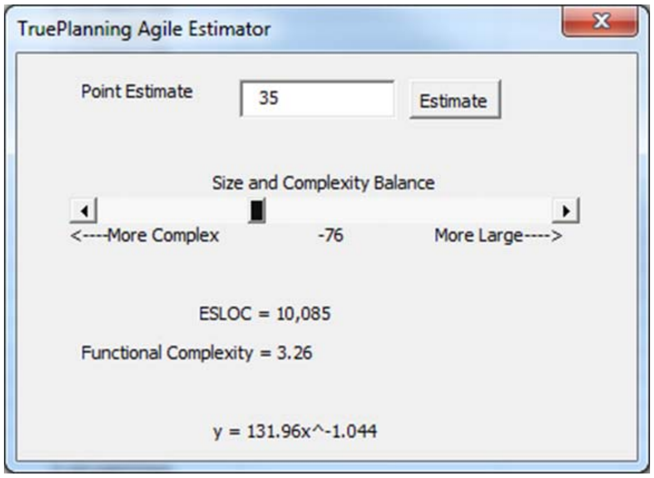


Figure 2:Velocity vs cost estimate

Armed with this information the next step was to create a method for using this forward. Figure 3 is a prototype of a tool that uses this relationship to create size and complexity pairings based on the number of story points and the estimators judgment of the relative complexity of those story points to the average complexity of the software that that the organization delivers. As the slider moves towards more complex, the size is decreased as the complexity increases and conversely as the slider moves towards more large, size is increased while complexity decreases.



Point Estimate 35

-76

ESLOC = 10,085

Functional Complexity = 3.26

y = 131.96x^-1.044

Figure 3: Estimate Size and Complexity from Story Points

Because story points are so specific to an organization, this tool will only work for the development team at PRICE but the process and methodology for creating this tool will work for any organization collecting agile metrics and using a tool that has a method for quantifying functional complexity.

Agile Cost Drivers

Because agile development is a paradigm not a methodology it is disingenuous to suggest that there should be an input to a parametric model that indicates that agile development is being done. There is no way for a mathematical model to understand whether the fact that a project is agile is going to increase, decrease of have no effect on the cost of a software development project. There are of course some practices common in agile projects that should be consider as cost drivers are being determined.

Agile teams tend to be highly skilled. There are several reasons for this. First of all, unskilled or unmotivated workers stand out in an agile environment more so than they would in a more traditional environment. It's not easy for a slacker to hide in an agile environment where each member is expected to take ownership of the success of the project. If pair programming is utilized, new members of the team are quickly brought up to speed on the projects function. If this phenomenon exists the input parameters indicating team experience and skill should be adjusted appropriately.

Co-location of teams has several potential impacts on productivity. Agile seems to create a 'culture of interruption'. In an agile project when a test goes red one or more of the team members need to stop what they are doing and do whatever it takes to make the test go green. If a developer has a question about a requirement they will immediately ask someone (or pose the question to the room in general) and someone has to stop what they're doing to answer the question. One would think this would lead to a decrease in productivity, and in some cases it will. However, much evidence exists that teams that are distributed across companies or countries, have productivity associated with communication issues. Why wouldn't the opposite be true for teams when they're co-located? Additionally, many agile teams have a daily stand up meeting where they quickly discuss progress, plans and obstacles further improving the communication within the team. In cases where teams are co-located and that is working well or where daily stand up meetings are a practice, cost drivers that indicate multi-site development or communication issues in the project should be adjusted accordingly. Co-located teams also tend to act with high cohesion and function similar to an IPT, especially agile teams which have the customer as part of the team. Cost drivers intended to model team cohesion or use of an IPT should be set to reflect this.

There's overhead associated with being agile. The backlog needs to be maintained, developers need access to the current user stories for the iteration, velocity needs to be tracked, tasks need to be determined, documented and tracked, effort needs to be tracked against the iteration, tests need to be executed, etc. While all of these things can be dealt with manually, there are many good tools available that are specifically geared to help with some of the overheads created with certain agile practices. In general agile teams tend to have good toolsets around their agile processes. If this is the case then any cost drivers around tools or automation should be adjusted accordingly.

Continuous integration with automated testing requires that code be checked in often and that builds run regularly – either each time code is checked in or every few hours. These builds are then automatically tested and the team is informed when there is a failure. While this process is likely to cause frequent interruptions for the development team, these interruptions really occur at an ideal time. Because builds are occurring all the time, very little changes from build to build and it is easy to isolate the cause of the problem. Because the developer or pair responsible for the failure has just touched the code, they are going to have an understanding and clarity around the change that they may not have in a project where integration only occurs at the end. For these reasons it is quite possible that the integration activity will be easier than it would be in a more traditional project. This factor should be considered when evaluating cost drivers describe integration complexity.

Conclusions

The answer to the question 'Can I use a parametric tool to estimate an agile development project?' is a resounding YES. The answer to the follow on question 'But how?' is basically the same way you would use a parametric tool to estimate any software projects. First you need to study the history of the team and the organization and understand how they function and how that influences cost or effort on previous projects. Next you need to understand the nature of the software being developed and the practices commonly employed by the development team.

There is no cookbook or magic formula that says agile development projects are 20% more productivity or 30% less productive. The truth is some are more productive and some aren't. Sizing for an agile project is a challenge because common agile practices force the development team to estimate size using a completely notional scale. This scale is not easily translatable on its own to be useful for estimating activities, but an understanding of relative size and complexity of the user stories creates a method for translation into parametric input values. As with any other software estimate, once a measure for size has been established, the next step is to learn nature of the project and the practices the team will employ. If the team is agile they are likely to employ many of those cited above. If the team is not agile, they still may be employing some of the practices above or they may have a completely different set of practices which need to be evaluated as to their potential impacts on cost.

It is important to remember that agile just a paradigm. You can't estimate the impact of a paradigm; you can only estimate the impact of the practices that may occur as a result of that paradigm.

References

1. http://en.wikipedia.org/wiki/History_of_software_engineering
2. http://agilemanifesto.org/
3. Santillo,L, "Early & Quick COSMIC-FFP Analysis using Analytic Hierarchy Process, Proceedings IWSM-Metrikon 2000, Konigs, Wusterhasen, Berlin
4. http://agile101.net/2009/08/18/agile-estimation-and-the-cone-of-uncertainty/

About the Author

Arlene Minkiewicz is a software measurement expert dedicated to finding creative solutions focused on making software development professionals successful. She has over 29 years in the software industry, researching all aspects of the software development process and providing thought leadership to the software community. Ms. Minkiewicz is the Chief Scientist at PRICE Systems, LLC. In this role, she leads the cost research activity for the entire suite of cost estimating products that PRICE provides.

PRICESystems.com