The Agile Times

A quarterly publication of the AgileAlliance

VOLUME IV ISSUE 1

A 100+ page issue of the *Agile Times*? Quality content by people who are shaking and making, rather than just hypothesizing and theorizing? That is Volume 4 of the *Agile Times*, which owes its existence to the efforts of the finest team of editors that anyone could ask for. Just wait until you see Volume 5 late this spring!

There is so much rich information available about Agile processes that the *Agile Times* has been expanded into a number of subject areas. Each subject area has its own editor who writes, solicits articles for that area, and pulls all the content together for publication.. These areas are listed on the following pages along with each area editor's email address. Contact the editors directly with comments or to offer submissions of your own. The table of contents reflects the content for each of the areas.

--Ken Schwaber

THE SPREAD OF AGILE AND AGILE CONFERENCES

This has been a great year for Agile conferences. The marketplace has become more interested in deeper information about Agile. Interest has picked up as word has gotten out that Agile can deliver better, quicker, and more than other approaches. A survey done for ThoughtWorks by Liz Barnett, a Forrester Research Vice President, indicates that the aspects of ThoughtWork's implementation of Agile that are most important to ThoughtWorks customers were efficiency and early delivery of business benefits. Check out the study at www.thoughtworks.com/forrester_tei.pdf.

As Agile Alliance membership requested in a survey, the Agile Alliance Board of Directors has formed a Conference Committee that is bringing the Agile Development Conference and the XP/AU Conference together as one conference for 2005. Thanks to many people to mention for making this happen. In addition, a number of other conferences are springing up, including various XPDays, Scrum Gatherings, and the 2nd Agile Canadian Network Workshop.

Scrum Gathering, April 20-25, Vienna Austria
gathering.scrumalliance.org

XP2004, June 6-10, Garmisch-Partenkirchen, Germany
www.xp2004.org

Canadian Agile Network Workshop, June 20-21, Banff, Canada
www.agilenetwork.ca/ws2004

Agile Development Conference, June 23-26, Salt Lake CIty, Utah
www.agiledevelopmentconference.com

XP/Agile Universe ,August 15-18, Calgary, Alberta, Canada
www.xpuniverse.com

Volume IV; Issue 1

Inside This Issue

Table of Contents

Volume IV; Issue 1
February 2004

To work effectively with Agile Methods you must have some basic skills. How well you master these skills can determine how successful you are in implementing your Agile Process. This section will help you to understand the traditional "foundation" skills that Agile methods build on.

PREREQUISITES FOR AGILE SOFTWARE DEVELOPMENT

Agile methods do not tell us everything about how to develop software. Agile methods rely on their users having some basic software development skills. To be effective at using Agile methods, you need to know more than just the practices and principles of the Agile method of your choice. This section is for explicitly discussing what some of these basic skills are, and how to apply them in an Agile environment. This article provides a roadmap to the topic. Brad Appleton's article Extreme Locality talks in detail about how locality of reference can minimize and simplify documentation and traceability in an Agile process.

SO WHAT?

At first glance, this seems to be stating the obvious; Agile software development is software development after all. There are a few reasons that I feel this topic is worth discussing:

- In the excitement about using a new approach to developing software, the team can forget that they are not building from scratch.
- The success of an Agile project depends as much on the execution of basic skills as the Agile practices. When an "Agile" project fails, customers are as likely to blame the method, which is new, as they are the underlying skills of the team.
- Agile software development methods tend to favor generalists over specialists, so an Agile software developer needs to have a larger toolbox of basic skills. A developer may have a certain skill to offer the rest of the team, and it is important for developers to share that experience with others on the team. The team will work better if everyone has enough knowledge of basic issues to know what they do not know. There is much power in knowing enough to know when you need help.
- Agile approaches use skills in different ways than traditional ones. Having a good understanding of foundation concepts will help you understand how to adapt them. This knowledge will also help you help others on the team make the transition to Agile approaches by bridging the cultural divide.

Judging from some recurring threads on the extreme programming mailing list (among other places), a lot of people seem interested in learning how to apply certain skills in their Agile projects. We need to understand better what we can learn from the traditional practices, and what we should discard or adapt. In this article I hope to:

- Describe some of the basic skills that one needs in addition to, say, the practices that one might find described in a book on Extreme Programming [1] or Scrum [2].
- Provide pointers to places to learn about some basic skills
- Briefly discuss how to apply these skills in an Agile team. Future columns will discuss this issue in more detail for specific skills.

In this issue I'll provide a list of some of the skills I believe need more discussion in the context of Agile software development. I hope that this list can start a discussion. I'd also appreciate hearing your views on this topic.

WHAT COMPRISES THE FOUNDATION?

Any list that I make here will be incomplete, but some items seem to be common topics of discussion, often because they are skills that are a bit of a mystery even for those on non-Agile teams. These are skills

Given a list of disciplines, the issue is more about how applying a discipline differs in an Agile team, than whether or not the skill is useful. This often gets lost when the focus of conversation is how to apply an Agile method. Part of this has to do with our use of language; "Change Management," "Version Control," "Databases" and the like have developed very process-heavy meanings. Many Agile developers feel that "process" can get in the way of developing software, which is why we subscribe to the Agile principles, especially "Individuals and Interactions over Processes and Tools." Yet, many processes facilitate interactions among individuals, for example. And the lack of certain processes and tools can make your development move more slowly.

Agile Teams practice change management, Version control, and use databases. And in many cases these tools enable agility. We should understand how to use some of these processes in a way that furthers, rather than hinders, agility. Some of the areas that I believe are foundation skills that are often ignored are:

- Configuration and version management (this includes change management)
- Database design and schema management
- Requirements gathering and management and customer expectation management skills. Since these items are the once most often viewed as "not Agile," I'll use them as a start for a list of techniques that Agile teams should not ignore.

All of these areas have established knowledge resources. While most of us understand that one can't just buy a copy of *eXtreme Programming eXplained* [1] and then start building software in an Agile fashion, there are some who need some guidance about how to learn how basic skills fit in with XP and other Agile methods. We often treat many of the skills needed to build software as tacit skills. We can benefit from making them explicit.

BRIDGING THE GAP

Agile software development methods are effective at addressing issues common to many environments: uncertainty and risk. While Agile practices are well suited to the reality of many work environments, they do not reflect the way that most environments currently work. Just as being Agile is about change and adaptability, being effective at using Agile practices in an organization requires that you move others in the organization to a new way of thinking. Having a better understanding of commonly "misunderstood" areas can only help the Agile advocate promote Agile software development. What follows is some brief information on four topic areas that all Agile developers should understand.

SOFTWARE CONFIGURATION MANAGEMENT

Many Agile developers often think of software configuration management (SCM) as a very heavy weight discipline that adds overhead and complexity. This perception is often the result of bad experiences in an organization where SCM was used to control change, rather than track change. In fact, SCM practices can enhance the ability to make changes. Proper version management discipline can, in combination with good unit testing practices make one feel safer about attempting change: if something isn't working you can back off to the last version that worked. You can think of codelines as integration points for your team.

Branching is one of the most feared SCM practices because it can add complexity and work to a project. But when done properly, creating a branch can be the simplest way to free a team to work on new projects in an Agile manner, while still supporting older code. (Of course, you want to stop supporting the it eventually.)

practices are essential for coordinating the work of people on your team. To learn more:

- The book *Software Configuration Management Patterns: Effective Teamwork, Practical Integration* [3], which I wrote with Brad Appleton, covers the key practices team must use to work effectively together. The pattern language in the book shows how version control practices integrate with test, integration, and build practices. Www.scmpatterns.com has links to other useful resources.
- *Pragmatic Version Control*, by (Dave Thomas and Andy Hunt: As the title says, it's a pragmatic guide to using CVS usefully. [4]
- *Configuration Management Principles and Practice* [5], by Hass is in the Adison-Wesley Agile Series, but talks about CMM and other non Agile things. It is worth a scan to learn how to communicate with the more process heavy people on your team.
- CM Crossroads News (http://www.cmcrossroads.com/newsletter/) has a monthly column on Agile Software Configuration Management that Brad Appleton, Steve Konieczka and I write in which we discuss how to bridge the gap between "formal" and "Agile" SCM environments. The CM Crossroads site has numerous discussion forums of interest to those concerned with Configuration Management.

UNDERSTANDING CUSTOMER NEEDS

Agile processes emphasize interacting with, and getting feedback from, customers. Developers often don't always know how to use the feedback effectively. Agile methods have excellent mechanisms for allowing development teams to manage the expectations of their customers, but the presence to feedback loops isn't always enough; you need to know how to use them effectively. To learn more:

- *Are Your Lights On? How to Figure out what the Problem REALLY Is* [6] by Gause and Weinberg is a very readable book that demonstrates the importance of really understanding your customer. This book explains the difference between "wants" and "needs" and how to help your customer figure that out so that you can spend your time working on useful tasks.
- *Requirements by Collaboration : Workshops for Defining Needs* [7] by Ellen Gottesdiener is a wonderful book about how to emphasize customer collaboration over contract negotiation by getting the customers together to harness the power of customer collaboration and how to successfully facilitate customer meetings.
- *Managing Expectations* [8] by Naomi Karten (Foreword by Gerald M. Weinberg) is all about managing customer expectations. Her more recent work *Communication Gaps and How to Close Them* [9] is more generally about managing expectations and perceptions and clarifying misunderstanding (both with customers as well as with team members and other project stakeholders).

DATABASE ISSUES

Applications use databases, either because they need to interact with data that already happens tobe in a database, or because using a database (correctly) just makes things simpler (really!). Databases can be surrounded with mystery because data and databases are treated differently than code. To learn more about how to work with databases:

- *Joe Celko's SQL for Smarties: Advanced SQL Programming* [10] by Joe Celko is a very pragmatic book on how to work with the language of databases.
- *Agile Database Techniques: Effective Strategies for the Agile Software Developer*, [11] by Scott Ambler, is a guide to working with databases. What makes this book especially interesting is Scott discusses how to navigate the gap between the formal "DBA" centric world, and the Agile team.
- The article "Evolutionary Database Design" by Martin Fowler and Pramod Sadalage is a concise

summary of the issues on working with databases and DBAs in an Agile environment. http://martinfowler.com/articles/evodb.html.

· Brad Appleton, Steve Konieczka and I wrote a column in a the January 2004 CM Crossroads News: "Applying Agile SCM to Databases" that discusses the intersection of Agile SCM and Agile databases: www.cmcrossroads.com/newsletter/articles/Agilejan04.pdf.

IN SUMMARY

This is a big topic, the details of which seem boring when compared to the excitement of adapting a team to use Scrum or XP. But moving to an Agile method will more likely than not fail if the team expects that having a copy of Extreme Programming Explained [1] around is all that is necessary for success.

REFERENCES

1. *Beck, K.,* eXtreme programming eXplained : embrace change*. 2000, Reading, MA: Addison-Wesley.*

2. *Schwaber, K. and M. Beedle,* Agile software development with scrum*. Series in agile software development. 2002, Upper Saddle River, NJ: Prentice Hall. xvi, 158 p.*

3. *Berczuk, S.P. and B. Appleton,* Software Configuration Management Patterns : Effective Teamwork, Practical Integration*. 2003, Boston, MA: Addison-Wesley.*

4. *Thomas, D. and A. Hunt,* Pragmatic Version Control using CVS*. 2003, Dallas, TX: The Pragmatic Bookshelf.*

5. *Hass, A.M.J.,* Configuration Management Principles and Practice*. The Agile software development series. 2003, Boston, MA: Addison-Wesley. xiv, 370.*

6. *Gause, D.C. and G.M. Weinberg,* Are Your Lights On? How to Figure out what the Problem REALLY Is*. 1990, New York, NY: Dorset House.*

7. *Gottesdiener, E.,* Requirements by Collaboration : Workshops for Defining Needs*. 2002, Boston: Addison-Wesley. xxvi, 333 p.*

8. *Karten, N.,* Managing Expectations*. 1994, New York, NY: Dorset House.*

9. *Karten, N.,* Communication Gaps and How to Close Them*. 2002, New York: Dorset House. xiv, 362*

10. *Celko, J.,* Joe Celko's SQL for Smarties : Advanced SQL Programming*. 2nd ed. 2000, San Francisco: Morgan Kaufmann. xxi, 553 p.*

11. *Ambler, S.W.,* Agile Database Techniques : Effective Strategies for the Agile Software Developer*. 2003, Indianapolis, IN: Wiley. xxvii, 447 p.*

12. *Constantine, L.L. and L.A.D. Lockwood,* Software for Use : a Practical Guide to the Models and Methods of Usage-Centered Design*. 1999, Reading, Mass.: Addison Wesley. xvi, 579.*

I used to believe in "best practices." But, after enough years in the field I no longer believe. Rather than thinking in terms of "best practices," I now think in terms of "good practices in context." A "good practice in context" is a set of activities or behaviors that have been demonstrated to work when applied within a certain context. Holding a daily standup meeting with all team members assembled in one room is a good practice that has worked well for me in the context of relatively small, co-located teams. The exact same practice would not work well for 100 developers: the room they'd need would be too big and it would take too long for each person to comment on what they did yesterday, what they're doing today, and what problems they're facing.

Even worse, the term *best practice* is a dangerous one. To call one practice "best" implies that all others are inferior. They may not be; one practice may be best in one context while another practice is better in a different context. For example, I have seen teams excel with pair programming *and* with rigorous Fagan-style code inspections. I don't know which practice is "best." I do, however, know that each has worked for me in the past when applied within specific contexts.

Best practices is also a dangerous term because it brings to mind a collection of practices that can simply be combined into a "best process." But this doesn't work. There is no guarantee that a collection of so-called best practices will result in even an adequate overall process, let alone a best one.

With this mind, this inaugural installment of the Best Practices section of the *Agile Times* features an article by Giovanni Asproni in which he looks at the role of motivation in agile processes and considers the impact of an agile process on teamwork. In a second article, Paul Oldfield takes a look at one aspect of how a mix-and-match approach to practices must be managed in order to lead to an adequate process.

If you have suggestions for future Best Practices columns, or would like to contribute an article, please contact Mike Cohn at mike.cohn@computer.org.

Motivation, Teamwork, And Agile Development

Giovanni Asproni

INTRODUCTION

Motivation as defined by the Merriam-Webster dictionary 11th edition is "1a: the act or process of motivating b: the condition of being motivated 2: a motivating force, stimulus, or influence: incentive, drive." The fact that motivation is the most important factor for productivity and quality is not a new discovery. It was pointed out for the first time by the studies conducted by Elton Mayo around 1930. Since then there have been several studies that confirmed the same results in several industries including software development [2], [4], [10], [14]. Nevertheless, until recently the main focus has been on *process-centric* methodologies, the ones that Jim Highsmith calls Rigorous Software Methodologies (RSM) [5].

The basic assumption behind RSMs is the same as that behind scientific management—that is, that to improve productivity and quality, it is necessary and sufficient to improve and formalize the activities and tasks of the development process. In this kind of methodology, people have to adapt to processes. The advent of Extreme Programming first and the Agile Movement later on has put people back at the center of the development activities. In these kind of methodologies, "people trump process" [3]; the processes have to be adapted to the needs of the people involved.

According to Jim Highsmith Agile development methods appeal to developers because they reflect how software really gets developed [5]. In this article I claim that Agile methods also appeal to developers because they reflect how they really *like* to develop software. Since nowadays most software is developed by teams, I have taken the approach of showing the strong connections between motivation and effective teamwork and then showing how Agile development methods are related to the latter.

MOTIVATION

The classic experiments that demonstrated the influence of motivation on productivity were conducted by Elton Mayo between 1924 and 1932 at the Hawthorne Works of the Western Electric Company in Chicago [8]. The phenomenon that these studies discovered is known as the "Hawthorne Effect." The original purpose of the experiments was to find the effects of illumination on productivity. The results were quite surprising:

- When illumination was increased, productivity went up.
- When illumination was decreased, productivity went up.
- When illumination was held constant, productivity went up.

After seeing these results, Mayo and his associates began to wonder what kind of changes in the work environment could influence productivity. They set up an experimental group by pulling six women from the relay assembly line. The group was isolated from the rest of the factory and put under a supervisor who had management style akin to a leadership-collaboration style [5].

The experiment consisted of introducing some variations to the work conditions. For example, the reseachers reduced the number of working hours and increased the number and length of pauses during the workday. The researchers introduced the changes always keeping the experimental team informed and asking for advice or information and listening to complaints. No matter what changes the researchers introduced, productivity always went up. Eventually, all the improvements were removed; at this point, productivity was the highest ever recorded.

The researchers' final conclusion was that the six women formed a team that cooperated spontaneously and wholeheartedly to the experiment. The team had considerable freedom of movement, was not pushed by anyone, and was involved in every decision that could affect its work. Under these conditions the workers developed a higher sense of responsibility that induced them to do a better job, and at the same time, feel happier and more satisfied.

These experiments evidenced for the first time that workplaces are *social environments*, where people are motivated by many factors other than economic interest. Mayo concluded that recognition, security, and a sense of belonging are more important to productivity and morale than the physical environment. He also determined that a friendly relationship with the supervisor is very important in securing the loyalty and cooperation of the team. These studies represented a breakthrough. In fact, at the times when the studies where conducted, the prevailing theory was Taylor's scientific management [12], which was based on the assumption that the main motivational factor for workers was high wages. Mayo's findings clearly undercut Taylor's theory.

MOTIVATION THEORIES

After the Hawthorne experiments, several theories have been developed to try to characterize motivation. Each of them has strengths and weaknesses. None of them is general enough to be applied in every situation. The factors that influence motivation can be identified in two main categories: *intrinsic factors* and *extrinsic factors*. The intrinsic factors come from the work itself and the goals and aspirations of the individual, (e.g., achievement, possibility for growth, and social relationships), while the extrinsic factors depend on the surrounding environment, or basic human needs (e.g., salary, office space, and responsibility).

Some prominent motivation theories that can help in explaining what motivates software developers are Abraham Maslow's *hierarchy of human needs*, Frederick Herzberg's theory on *motivators and hygiene factors* [4], and David McClelland's *achievement motivation* theory [9]. Maslow's hierarchy of human needs classifies the human needs in five levels. According to this theory, higher level needs become motivators only when the lower level ones are satisfied. The hierarchy, ordered from from the lowest to the highest level, is:

- Physiological (e.g., salary, office space, appropriate facilities, and lightning)
- Safety (e.g., job security, pension scheme, medical insurance, and sick leave)
- Social (e.g., interactions with colleagues and customers, and teamwork)
- Self-esteem (e.g., reputation and the recognition of colleagues, subordinates, and supervisors)
- Self-actualization (e.g., the realization of the full potential of the individual, or "What a man can be, he must be" [7].)

Herzberg's motivators and hygiene factors theory relies on different assumptions than Maslow's does. According to Hertzberg, there are factors that have a positive impact on the increase of motivation: the motivators, which Herzberg identifies with the intrinsic factors and other factors that have to be present in order to avoid de-motivation but by themselves cannot increase motivation; and the the hygiene factors, which Herzberg identifies with the extrinsic factors [4]. According to this theory, motivators derive from

"that unique human characteristic, the ability to achieve and, through achievement, to experience psychological growth" [4]. They are, in order of importance, achievement, recognition, work itself, responsibility, advancement, and possibility of growth. Instead, hygiene factors are a consequence of humankind's animal nature and relate to the basic biological needs. For example, the need for food makes money a necessity. Hygiene factors include company policy, office space, supervision, personal life, and salary.

McClelland's achievement motivation theory characterizes the motivation of a particular class of people—the ones who have a strong desire to achieve. According to this theory, achievement-motivated people have the following characteristics:

- They like difficult, but potentially achievable, goals
- They like to take calculated risks
- They are more concerned with personal achievement than with rewards for success
- They have a need for concrete job-relevant feedback and want to know how they're doing.

These three theories are related to each other. Herzberg's extrinsic factors correspond to the lower levels of Maslow classification, whie intrinsic factors correspond to the higher ones. Achievement-motivated people tend to be more motivated by Herzberg's intrinsic factors. Achievement itself is an intrinsic factor. In general, in the workplace, intrinsic factors tend to be much more effective than extrinsic ones in motivating people [13].

SOFTWARE DEVELOPERS' MOTIVATION

The first ten motivational factors for software developers—in decreasing order of importance—reported by Boehm [2] are:

- Achievement
- Possibility for growth
- Work itself
- Recognition
- Advancement
- Technical supervision
- Responsibility
- Relations with peers
- Relations with subordinates
- Salary

The data on which the list is based is more than 25 years old, but I think it is still valid, as it matches well with my experience. Achievement is the strongest motivator for software developers, furthermore, most of the other ones are intrinsic factors as well. So McClelland's and Herzberg's theories are suited to explain what motivates them.

TEAMWORK

Nowadays, most endeavors are so complex that they can be accomplished only by a team, so it makes sense to know what make teams effective. Larson and LaFasto [6] undertook a three-year study to determine the characteristics of successful teams. The teams studied ranged from football teams to the team that built the Boeing 747 airplane. There weren't any software development teams. They found that all the highly effective teams always had these characteristics:

- A clear, elevating goal
- A results-driven structure

- Competent team members
- Unified commitment
- A collaborative climate
- Standards of excellence
- External support and recognition
- Principled leadership

From this list is evident that effective teamwork has a strong relationship with motivation.

A clear, elevating goal is absolutely necessary for achievement. A goal is clear if it is possible to concretely and unequivocally verify that the goal has been achieved. An example of clear goal is "The executable must not use more than half gigabyte of RAM at any given time." In contrast, here is an example of a less clear goal: "The executable must not use too much RAM." A goal is elevating if the team considers it important or worthwhile. For example, it might be a technical challenge that stretches the skills of the team to the limits, or it might instill a sense of urgency. People want to be involved in something that gives them an opportunity to make a difference, so if the goal is clear but not elevating, achieving it could be difficult, since it might be perceived as uninteresting or even worthless.

A team has a results-driven structure when it is organized according to the goal that it has to attain. Team structure comprises the process, the communication channels, the roles, and the skills of the team members. It is an hygiene factor. In fact, its presence makes achievement possible but doesn't motivate people, and its absence is certainly demotivating since it can make achievement at best difficult and at worst impossible.

Competence has an important influence on achievement motivation. Achievement-motivated people like challenging but potentially attainable goals. Lack of competence can make the goal impossible to reach. There are two types of competencies, both of which are equally important: technical competencies and personal competencies. Technical competencies refer to the knowledge and skills necessary to achieve team goals. They are clearly necessary. Personal competencies refer to the personal skills of the individual plus the ability to work effectively on teams; they can make the real difference in team performance. A team of star developers who cannot work well with each other is generally outperformed by a team of average developers who work well together.

Unified commitment is not easy to define. It is "team spirit" and involves individuals feelins a strong identification with the team. It happens when all team members are willing to devote time and energies for the achievement of their common goal pulling together in the same direction. It is when the team has its own identity. Unified commitment can be fostered by first of all establishing a clear, elevating goal and then by involving the team in all the phases of the project. Involvement enhances commitment. If unified commitment is lacking, the possibilities of success are severely reduced.

A collaborative climate is described by the phrase "working well together." It is important to foster unified commitment, a sense of belonging, and to give team members a possibility for growth. In order to have a collaborative climate is necessary for team members to trust each other. In this way, they can focus on the attainment of the goal. Furthermore, communication and coordination are more efficient, and the quality of the outcome is greatly improved.

A standard defines an expected level of performance. It defines expectations of the skill levels of team members, of the initiative and effort they are able to demonstrate, of how the results are to be achieved, and so on. A standard of excellence defines a standard in which the expected level of performance is very high. A consequence of setting high standards is that the expectations on the team become high as well. This positive enforcement can bring the members to exert pressure on each other in order to keep up to

expectations creating a whole that is more than the sum of its parts. Consequently, the self-esteem of team members receives a big boost and so do motivation and product quality. Standards are hard work and require a great discipline, so the best way to make them easier to follow is to make them concrete. They should not be stated as general principles like "the code must be of excellent quality," but rather should be defined in terms of what can be done concretely in order to follow them. For example, they can mandate the usage of unit tests, refactoring, and pair programming as techniques to keep the quality of the code high.

It is interesting to notice that the factor that managers tend to use most to motivate employees—salary—is listed in the last position. Actually, I have yet to know a good developer who is really motivated by salary. Certainly I know some very good ones who recently refused highly paid job offers because the work was not "interesting enough." Of course, money is important, but it becomes a strong motivator or de-motivator only when the availability is respectively very high or very low.

External support is about giving the team the resources it needs to get the job done. In motivational terms, it is an hygiene factor. Without sufficient external support it is very difficult to achieve any goal. Furthermore, it gives the team the message that their work is not very important (making the goal less elevating), with consequent drops of motivation and morale. Recognition are the rewards linked to achievement. The rewards must be tied to performance and viewed as appropriate by team members. Recognition is a strong motivator for software developers.

Leadership is one of the most critical factors for effective teamwork. A very effective leadership style is what Larson and Lafasto call a principled leadership [6], and Highsmith [5] calls leadership-collaboration. Principled leaders don't give orders, they inspire and influence people, they trust their followers to get things done, and use power only sparingly. Effective leaders, according to Larson and Lafasto [6] "(1) establish a vision; (2) create change; and (3) unleash talent." In a team with this kind of leader there is a great opportunity for responsibility, technical supervision, and advancement.

In conclusion, effective teamwork are strongly tied together. Most of the characteristics of effective teams are motivators or hygiene factors, and the remaining ones have a direct effect on it.

AGILE DEVELOPMENT AND TEAMWORK

In this section I'll show how Agile development methods have the basic characteristics that make effective teamwork possible.

A CLEAR, ELEVATING GOAL

Iterative and incremental development along with user collaboration plays a central role in keeping the goal visible and clear. The usage of incremental development allows the developers and the customer to define and work on smaller but clear goals. An important part of the increment is the definition of—often automated—acceptance tests. Their definition is what really makes the goal clear to the team. The usage of, preferably short, iterations allow the team to have the feedback necessary to understand if what they have done is what the customer expected. In fact, acceptance tests are very

The usage of(preferably short) iterations allow the team to have the feedback necessary to understand if what they have done is what the customer expected. In fact, acceptance tests are very helpful, but the experience of using the software gives the customer a better understanding of her needs. Often, this leads to a refinement of the goal without a loss in clarity.

Making the goal elevating is not an easy thing. There are no sure recipes for an elevating goal. Certainly, having a customer who works closely with the team can be of great help. Such a customer can

continuously remind the team of how important the software is to her or to her company, or she can instill a sense of urgency making it clear why the product absolutely must be ready by a certain date.

Making the goal elevating is not an easy thing. There are no sure recipes for that. Certainly, having a customer who closely works with the team can be of great help. In fact, she can continuously remind the (great) importance the software has for her or her company, or she can instill a sense of urgency making it clear why the product must be absolutely ready for a certain date.

Some often successful techniques are to create a challenge by setting up tight, but achievable, deadlines or to give the team the possibility to learn new skills. In a project I'm currently involved, my teammates and I used both of these techniques to make it interesting. Up to now it has been a great success—we and the customer are both satisfied.

A RESULTS-DRIVEN STRUCTURE

Agile teams are structured in order to deliver valuable software on time and on budget in a context of frequent changes in requirements. An effective team structure has four necessary features [6]. First, there are clear roles and accountabilities. For Agile development some of them are defined by means of the rights that the customer and the development team have. The customer has the following rights:

- The right to an overall plan that defines what can be accomplished, when, and at what cost.
- The right to receive the maximum value for each iteration.
- The right to change or substitute priorities without incurring in exorbitant costs.
- The right to be informed about schedule changes so as to change scope
- The right to cancel the project and still have an useable system.

The developers have the following rights.

- The right to clear requirements with clear priorities.
- The right to always produce quality work.
- The right to request and receive help from their peers, their managers and the customer.
- The right to create estimates and to update them as the problem becomes better defined.
- The right to accept their responsibilities rather than having them assigned to them.
- The roles and accountabilities inside the development team depend on the methodology used.

Second, there is an effective communication system. Agile development puts an emphasis on face-to-face communication—arguably the most effective communication channel between human beings; the team members tend to be located close to each other, possibly in the same room—so the speed of communication is optimized; the customer is encouraged to interact closely with the developers—so the feedback loop is shortened and the goal remains visible and clear.

Third, there is a way for monitoring individual performance and providing feedback. In Agile development this is a consequence of the high level of interaction between the parties involved. If someone is not doing his best, it becomes very clear to everybody very early on.

Fourth, all the judgments are fact-based. Agile development methods submit all activities and products—including software—to an usefulness test: they must contribute in some way to the achievement of the goal, otherwise they are dropped. The process is streamlined by executing only the activities that simplify the work of the team. Documentation is written only if there are people willing to read it. Software is kept as simple as possible, so it is easier to change. Future extensions will be examined when the need will arise. Gold plating is loathed and avoided. Code quality is kept as high as possible. Finally, technology is used only when necessary—very often using a whiteboard or CRC cards during a design

session is more effective than using the latest CASE tool. All these techniques allow the development team to travel light and focus only on what matters for the achievement of the goal.

COMPETENT TEAM MEMBERS

Agile development methods need people with both technical and personal skills. In my opinion, the latter are the ones that are more important, since at the end make the real difference. This goes against the common belief that Agile methods require a higher proportion of expert developers in the team than RSMs do. In fact, in my experience it is more important for Agile software developers to have the ability to *learn new skills, adapt to changing situations,* and *apply acquired skills in new ways* than it is for them to have strong technical skills. Furthermore, the high level of interaction required by Agile methods requires people who can collaborate effectively with others—practices like collective design sessions, pair programming, and collective code ownership could be impossible to implement otherwise.

UNIFIED COMMITMENT

Agile methods tend to involve the entire team in all phases of development. The customer is in close contact with the team so that everyone can better understand the requirements. Design sessions make extensive usage of techniques—such as CRC cards and whiteboards—that bring the whole team together to discuss design and programming issues. All these things help greatly to foster unified commitment.

A COLLABORATIVE CLIMATE

Agile methods put a strong emphasis on collaboration. The first value of the Agile Manifesto clearly states that individuals and their interactions are more valuable than processes and tools, and customer collaboration is preferred to contract negotiation. The setting of a collaborative climate is one of the main objectives of several Agile practices. The emphasis on face-to-face communication, shared or at least very close office spaces, common design sessions, collective code ownership, customer collaboration, and incremental and iterative development are all factors that help in creating a climate of trust and collaboration.

STANDARDS OF EXCELLENCE

The level of performance at which Agile methods aim is quite high: satisfy customers with continuous, on time, and on budget delivery of valuable software and write technically excellent software—that is, code that is self-documenting, fully tested, simple, and modifiable and has few and minor bugs. To make standards easier to follow, Agile methods make use of some concrete practices such as iterative and incremental development, extensive usage of testing, code refactoring, and coding standards, and keeping the code simple and self-documenting. Some methods—for instance Extreme Programming—also use pair programming and promote the concept of collective code ownership. These two techniques are a powerful incentive to keep up to the set standards, since every line of code can be, potentially, read and modified by any member of the team.

EXTERNAL SUPPORT AND RECOGNITION

Agile methods recognize explicitly the importance of external support. In fact one of the principles of the Agile manifesto states "build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done." They recognize that without appropriate resources software development is simply not possible. The allocation of office space, putting developers in close with each other, the emphasis on face-to-face communication, the availability of appropriate development tools, and close customer collaboration require a great deal of external support to be implemented. As far as recognition is concerned, when it comes from an happy customer it has a powerful effect. The simple act of showing appreciation for job well done is a very powerful motivator. It helps in increasing the self-esteem of the developers and the level of trust between them and the customer. This in

turn leads to better communications and better software.

The usage of iterative and incremental development can be instrumental in increasing the external support and recognition. A successful team that delivers early and continuous results is more likely to receive the support it needs from all the stakeholders.

PRINCIPLED LEADERSHIP

The leadership theme is not dealt with directly in the Agile Manifesto and its principles. Generally, each method has its own way to deal with it. There is a common theme that is clear from the literature: leadership is important, and it must be a principled one [5] [11] [10]. A principled leadership is a natural fit for Agile development, in fact is the most suitable for environments where change is the norm [5], and is the only style compatible with the Agile values and principles. A command-and-control style wouldn't be a good fit for an environment in which "individuals and communications" are valued more than "processes and tools." [1].

CONCLUSION

I have shown why I believe Agile methods can be more productive and appealing for developers. They leverage the most important factor for productivity and morale: motivation. The role of processes and tools in this context is still very important, since they are used to streamline all the repetitive tasks, letting the developers focus on what really matters: to satisfy the customer by producing valuable software.

ACKNOWLEDGMENTS

I thank Mike Cohn for sharing his ideas with me, and for reviewing the article. I also thank my colleagues Alexander Fedotov, Rodrigo Fernandez, Federico Garcia-Diez, and Renato Mancuso for their help and support.

ABOUT THE AUTHOR

Giovanni Asproni is an Italian software craftsman currently working as Senior Software Engineer for the European Bioinformatics institute, near Cambridge, UK. He is a member of the AgileAlliance, the ACM, and the IEEE Computer Society. He can be reached by e-mail at aspro@acm.org, or, through his website, at http://www.giovanniasproni.com.

REFERENCES

[1] Beck, K., et al., *The Agile Manifesto*, http://www.Agilemanifesto.org
[2] Boehm B. W., *Software Engineering Economics*, Prentice Hall, 1981
[3] Cockburn, A., *Agile Software Development*, Addison Wesley, 2002
[4] Herzberg, F., *One More Time: How Do You Motivate Employees?*, Harvard Business Review, 1968
[5] Highsmith, J., *Agile Software Development Ecosystems*, Addison Wesley, 2002
[6] Larson, C., E., LaFasto, F., M., *Teamwork: What must go right / what can go wrong*, Sage Publications, 1989
[7] Maslow, A., *Motivation and Personality*, Addison Wesley, 1987
[8] Mayo, E., *The Human Problems of an Industrial Civilization*, Macmillan, 1933
[9] McClelland, D., *The Achieving Society*, The Free Press, 1967
[10] Peters, J., Waterman, R., H., *In Search of Excellence: Lessons from America's Best Run Companies*, Harper, 1982
[11] Poppendiek, M., Poppendiek, T., *Lean Software Development*, Addison Wesley, 2003
[12] Taylor, F., W., *The Principles of Scientific Management*, Dover Publications, 1998
[13] Thomas, K., W., *Intrinsic Motivation at Work: Building Energy & Commitment*, Berrett-Koehler, 2003
[14] Weinberg, G., M., *The Psychology of Computer Programming: Silver Anniversary Edition*, Dorset House Pub-

Not everyone favours mix and match approaches to Agile methodologies. In some respects this attitude is quite correct: there is no need for another methodology consisting of pre-selected best practices. In other respects it is incorrect: existing methodologies are the result of such a mix and match effort, or alternatively advocate the selection of best practices as an ongoing process. Possibly of greater significance, those truly Agile developers who decide how to do things as they proceed are, in effect, mixing and matching from their own repertoire of techniques and best practices, on the fly. Some of the ideas and techniques that get included into the mix may appear to have nothing to do with the development process but instead address, say, the environment or the social aspects of a team. Anything that helps delivery of working software is fair game for mix and match.

The question arises; how do we manage the mix and match process to arrive at an appropriate development process? We can start to answer this question by considering what would make a development process appropriate. There are three primary properties that such a process needs.

It must be *Adequate* --the processes must be capable of producing the required products and meeting at least the most important subsidiary goals. A process that fails to detect and correct errors, for example, would produce a product that falls short of the requirements. It must be *Efficient* – the total cost of enacting the process should not be any higher than it needs to be. A process that permits change, but only at great cost, is not efficient in cases where change is common and unavoidable. It must be *Attainable* – the people available to enact the process must have the skills and ability to enact the process. A process that relies on refactoring must include people that can refactor and perform all the techniques that enable refactoring to be performed in safety.

It follows that any mix and match effort should produce a process that has these required properties. We may judge how good the resultant process is by how well it fits the required qualities, and thus we have a handle by which we can evaluate the mix and match efforts with a view to improvement. An inadequate or unattainable process will result in failure unless corrective action is taken.

Of the processes that have these two required properties, we may compare their efficiency when selecting the most appropriate.In this article I address the frequently neglected topic of Attainability and what consequences this has when considering the jigsaw of best practices. Essentially, all processes contain some parts that are defined. Even the most lightweight process, for example, may define when and where source code is stored. A heavyweight process, on the other hand, may attempt to define every possible action taken by any developer. The best way to ensure that a mix-and-match process is attainable is to pay attention to the gaps between the defined parts of the process.

All development processes have gaps. This is a distinction between development processes and production processes. Unless a product can be described precisely, in advance, the process to build it cannot be described precisely. These gaps in a development process are important, they enable the fixed parts of process to flex to fit the product being developed. They allow the developer to choose how to bridge the gap.An attainable process is one where the people available to enact the process can enact the predefined parts of the process, and can also bridge the gaps between these predefined parts.

At the extreme Agile end of the agility spectrum, a process will be all gaps; there will be no predefined parts to the process and the developer will have complete discretion over how the work is done. At the other extreme of the spectrum, a process definer may attempt to eliminate the gaps entirely so that the work can be done by the almost completely unskilled, emulating the factory production line. However, software development is not a production environment; such efforts are doomed to failure.

How is it possible that the developer can make these decisions that are needed at the extreme Agile end of the spectrum unsupported by the framework of a defined process? It is possible because the

developer has his own internal framework, built up from a combination of experience and learned knowledge.

It follows that when a developer does not have this internal framework, where it is insufficient, incomplete or poorly constructed, then the developer will need outside help to bridge the gaps and inadequacies of his internal framework. It is also important to allow for internal frameworks that may be strong in some areas and weak in others. This is common where experienced developers gain all their experience in one environment.

There are various approaches toward providing support. The more Agile approaches have the support coming from team members that have the relevant experience, or from mentors brought in to augment the team. Where there is a large shortfall and not enough available experience to make do, then it is time to add written, pre-defined elements of process.

Note that this is not the only reason to have written process; it may also be needed for coordination purposes. When there are differences in approach, and these differences risk undermining the work done by other people, then it is time to agree on elements of process to prevent conflict. Configuration management, for example, is one area where there is a broad consensus on the need for some agreed process.

The dilemma that a defined process attempts to solve is that the people defining the process hold the knowledge about process, while the people enacting the process are the people who hold the knowledge about the situation that determines what process would be appropriate. The ideal toward which all truly Agile approaches aspire is that the people enacting the process hold sufficient knowledge about process to define the process.

Before discussing how knowledge of attainability helps us select appropriate process, I would like first to take a step back and think about the development process. In many respects, this is like any other product; the stakeholders have requirements that apply to process. A typical set of requirements may be the 200 plus goals of CMM; with suitable re-wording to remove non-Agile assumptions, they may suit our purpose nicely. Yet if these are the requirements, the process that actually gets enacted is the design. All the Agile principles and practices that we apply to design of software may also apply, perhaps with some modification, to design of process. As with all analogies, some parallels will be useful and instructive, some will not. Defining process 'just in time' is a very useful parallel; 'continuous integration' and 'refactoring' are parallels with little immediately visible benefit. This probably stems from the difference that process gets used and produces a product that never needs to be produced again, so that particular bit of process never needs to get used again. Of course, something remarkably similar may get used again, and it is this similarity that leads some people to believe there may be a possibility of designing process up front.

Let us consider the case where there is a shortage of experience, a lack of communication, and a lack of trust in the developers to make the right decisions with regard to process. For inexperienced teams the process needs to leave fewer gaps, and the process described needs to be more in terms of solutions to problems. More feedback loops need to be placed in the process explicitly to detect those occasions where the process seems to be inadequate, where the goals are not being met by the developers following the defined process. Where problems are detected, a problem resolution strategy needs to be brought into play, bringing the available expertise to bear on the problem. Yet the process still has gaps, the developer still needs to use initiative.

Some people see these gaps as risks. A long series of process improvements may attempt to fill all the gaps each time a developer's initiative was misplaced. The resultant process would be so hidebound and inflexible that it would have the developers doing precisely what was asked even when this was known

to be the wrong thing to do. Here, any success would be in spite of the process, where developers ignore the process and do what they think is right.

Now let us consider the case where there is considerable experience, good communication, and trust in the developers. For those teams with a wealth of experience, the process can be defined initially as a set of goals. The team as a whole, and individuals within the team, can choose how the process they enact will meet the goals. Such a set of goals may be, for example, the aforementioned goals of CMM, suitably modified. Here, the stakeholders are defining the requirements for the development process rather than defining the design of the development process. We could attempt to fill these requirements by defining a process up front, as in the example for inexperienced teams. Alternatively, we could design the process in an Agile manner, much the same way as we would design a system in an Agile manner. A typical Agile manner may be to do some initial work up front to establish a direction for the process, then to let the detailed design of the process evolve as the needs dictate. The important thing is that the goals are met; the flexibility that comes through fewer constraints on how the goals are met gives the opportunity for an efficient solution to be found. Alternatively, it gives the opportunity for an inexperienced team to go astray.

Finally, let us consider the extreme situation, where every member of the team is highly experienced. Here the team may be trusted to establish their own list of goals and act on them without reference to any defined process at all. In all cases, they know what is the right thing, and do it. Where they don't have the relevant information or expertise they will seek it out, of their own volition.

There are various opinions on how to characterise a situation and its suitability and need for agility. DSDM, Appropriate Process, Crystal, and Boehm and Turner all have variants with a strong similarity; the DSDM view was shown in the third issue of the *Agile Times*. In general, there is a scale of suitability for agility, positions at one end being ideally suited for Agile approaches, positions toward the other end requiring additional robustness of approach. It is useful to note that the different criteria have different impacts on agility. Some, such as the degree of change (alternatively familiarity with business, familiarity with technology) determine the need for agility, while others such as criticality determine the need for robustness. One of the criteria considered by all these approaches is that of the skills and ability of the available people. The more the team has in the way of appropriate skills, the greater is the degree of agility that can be attained. Let us look at a few benchmark situations for an individual within a team. At the outset, we all start as beginners, with no techniques available for use, no experience in using them, and no ability to select appropriate techniques. Hopefully this situation has been improved before the student takes his first employment, but a person at this stage needs firm guidance in a technique, and mentoring to be able to use it at all. Effective use of the technique is still in the future.

With some experience, the apprentice developer gains the ability to use a few basic techniques without guidance, to enact those parts of the life cycle in which he is permitted to participate. For each case he will have a single technique available, and will be starting to learn a few wrinkles about how to apply it in different situations. There is still no opportunity to select between alternate techniques, because there are no known alternatives.

As the developer progresses through journeyman status, he will typically expand his competence to different areas of the life cycle. He may also learn alternative techniques. He may use these all the time, abandoning the old techniques, or he may start to choose one technique for one occasion, another for others. The developer now has some capacity to select process; he has the first opportunity to become Agile with respect to process. Eventually, the better journeyman developers may reach the stage of mastery, and a depth of understanding that allows them to shortcut the evaluation and selection of tech

niques, instead appearing to invent techniques precisely tailored to the situation as they go.

The learning of process can include learning a single technique for new areas of process, learning additional techniques for existing areas of process, and learning how to choose between alternative techniques where there are multiple known ways to achieve the current goal. To this, we may add learning the goals that are important when designing process.

When considering how much process to design in advance, we need to consider what gaps the developers can cope with given their current abilities, what gaps they can cope with given the aid of their team members, how to ensure this aid is given effectively, and what further support in learning about process should be given.

. In considering attainability, consider also the possibility of changing the membership of the team to enhance the overall attainability; say by adding people with relevant skills or buying in mentoring support. Consider also the rate of churn, and the possibility of losing the expertise that is currently available and that is built up as the project progresses.

One of the conclusions that is probably more startling to the traditional process mind-set is that as the team matures, process improvement should take the form of removing constraints and permitting more flexibility, rather than adding constraints to prevent recurrence of problems. A second conclusion that may surprise some of the agile adherents is that mix and match happens anyway. If it is not done up front by a process specialist, it will be done 'at the code face' by the experienced developer selecting techniques suitable to deal with the immediate problems, or by a huddle of experienced people trying to find a way to deal with the situation the project finds itself in.

A third conclusion is that if we are to shorten the time it takes to get individual team members to the stage where they can make sensible decisions with respect to the elements of process that are appropriate, then some time spent by them, explicitly considering what makes a process appropriate and how to select appropriate techniques and approaches, might be a sensible option. And finally, unless the developer can choose between at least two ways to achieve his goal, he cannot be agile. Where one has no choice, one cannot adapt to circumstances; one cannot be agile with respect to that goal.

ABOUT THE AUTHOR

Paul Oldfield is a member of the Appropriate Process Movement, a group who believe that the recognition of what makes process appropriate and how to select appropriate techniques and approaches is a key skill in agile software development. These ideas are developed further on their site at www.aptprocess.com. Paul can be reached at paul.oldfield@aptprocess.com.

IF YOU'RE STUCK FOR ANSWERS, YOU NEED SOMEONE TO ASK YOU EFFECTIVE QUESTIONS

Do you know that sinking feeling that comes when you think you've bitten off more than you can chew? There might be so many things to do that you can't choose where to start, or there might be so many hurdles to overcome that failure seems inevitable. Either way you feel paralysed by the sense of impending doom. But don't throw up your hands in despair! You probably already have the answers to your problems: all you need is for someone to ask you some effective questions.

WHAT MAKES A QUESTION EFFECTIVE?

An effective question cuts through the mental logjam that we create when things go wrong. Answering it gives us an insight into our problems and stimulates our search for solutions. An effective question is phrased in an open-ended fashion that does not suggest a particular answer and warrants a response of more than just a few words. It usually starts with a "what" or a "how", as "why" questions are easy to misinterpret and can trigger a defensive response. Questions that are intimidating, repetitive, or rambling are obviously ineffective. As an effective questioner it is important to listen without interrupting, waiting respectfully for each answer and considering it when it arrives before giving an opinion or launching into another question.

WHAT DO EFFECTIVE QUESTIONS ACHIEVE?

Effective questions clarify our goals and help to steer us from a state of confusion or aimlessness toward a set of concrete actions. The dialogue that effective questioning initiates also increases the number and quality of interactions within a team and makes it more effective. The following examples illustrate some categories of question that I have found effective while working as a coach for a software development team. These can be used in any sequence that makes sense for your team and your situation.

- "Where are we going?"
- "Where are we now?"
- "Have we been here before?"
- "Whose input are we missing?"
- "What steps ought we to take?"
- "How should we start?"
- "Which are the real obstacles?"
- "Are we ignoring something?"

"Where are we going?"

If we are totally floundering then effective questions will explore the motivation behind our current activity and focus us on the desired outcome. Simply asking "What are we trying to achieve?" or "How will we know when we're done?" can be enough to get us back on track. Posing Dale Emery's value question[1] - "If you had that, what would that do for you?" – is particularly powerful, as it diverts attention away from a single solution and back to the root of the problem.

"Where are we now?"

As programmers well know we can come to new insights by simply retracing the unsuccessful steps we have taken. Asking "What seems to be causing the problem?" or "What have you tried so far?" can elicit this kind of productive 'aha!' moment.

"Have we been here before?"

Problems will recur and we can often benefit from effective questions such as "What was the solution last time?" or "What happened before when...?" These questions remind us of the likely effect in the

present of repeating actions that were taken in the past.

"Whose input are we missing?"

We can often benefit from connecting with people on different teams (or on our own team!) whose strengths and achievements we are perhaps unaware of. Effective questions such as "Have you talked to [her]? What did she say about...?" help to overcome this kind of limiting distance. Alternatively if there is a discussion in which some people remain silent then asking them "So, what do you think?" at appropriate intervals will encourage alternative points of view to be raised.

"What steps ought we to take?"

Sometimes we can see both where we are and where we want to get to. In this situation we need effective questions that help to bridge the gap, such as "How could we achieve that when we are starting from here?" If this prospect is too daunting however then it is more helpful to ask "Can we break this down into smaller chunks?" or "What's the first step we are able to take? Then what could we do next?"

"How should we start?"

If there are many tasks to be done then we need effective questions that identify our priorities. Asking "If you could only do one thing today then what would you do?" or "What is most important right now?" will help us to focus on the most urgent courses of action.

"Which are the real obstacles?"

Sometimes we can seem to hit a dead end where there is nothing more that we can possibly do or try. When this happens we need effective questions that give us courage and allow us to determine which of the perceived constraints are real and which are the product of our fear, miscomprehension, or haste. By asking "What if this was true?" or "What if that were to happen?" we can create options where previously none seemed to exist.

"Are we ignoring something?"

If a particular course of action seems too difficult or if we are prone to procrastination then we need effective questions that challenge us to confront what we would otherwise ignore. These questions run the risk of being provocative but if handled thoughtfully then asking "What are you going to do about this?" or "How would you justify doing nothing about that?" can overcome many causes of inertia.

ASKING FOLLOW UP QUESTIONS

Sometimes the first answer we give or receive is too hasty or tentative. Effective follow up questions explore whether a response has been fully thought through, for example "Can you explain how that would work?" or "How can you be sure that...?" Follow up questions that require our understanding to be confirmed can also be effective, such as "So if I heard you correctly then what you're saying is..."

SUMMARY

Software development is a difficult activity and as such it is easy to get bogged down in its problems. But we can get unstuck by asking effective questions that facilitate the discovery of solutions. Effective questioning can also benefit software development teams by promoting discussion, co-operation and collaboration.

ABOUT THE AUTHOR

Tim Bacon works as a consultant for ThoughtWorks Inc in the United Kingdom. The views expressed are his own and are not necessarily those of his employer. You can reach Tim at tbacon@thoughtworks.com.

Self-organizing teams are undiscovered country for most software development professionals. What does it mean to say Agile teams are self-organizing? If a team is truly self-organizing, can we lay off all the managers? How does a shift to Agile methods shape the roles of team members and managers? What can team members and leaders expect when working with Agile teams on the way to self-organization? Each of the questions above deserves a complete examination that is more in depth than we have room for in this article. However, here are some short answers.

When we say an Agile team is self-organizing, we mean that a group of peers has assembled for the purpose of bringing a software development project to completion using one or more of the Agile methodologies. The team members share a goal and a common belief that their work is interdependent and collaboration is the best way to accomplish their goal. Empowered team members' reduce their dependency on management as they accept accountability, and the team structure places ownership and control close to the core of the work. Rather than having a manager with responsibility for planning, managing and controlling the work, the team members share increasing responsibility for managing their own work and also share responsibility for problem-solving and continuous improvement of their work processes.

If the team is assuming responsibility for managing the work, can we get rid of the managers? In short, no. Managers are still needed. Not so much for their planning and controlling ability, but for the important job of interfacing on the team's behalf with the rest of the organization. In addition, a team self-organizes over time and usually follows a stepped approach to assuming responsibility for self-managing. The manager plays several important roles, including the incremental letting go of management tasks as the team becomes more adept at performing them.

Agile methods inherently drive the team in a self-organizing direction. As alluded to above, this causes a shift in the roles of managers from planning, controlling, directing, and managing to new roles like building trust, facilitating and supporting team decisions, expanding team capabilities, anticipating and influencing change. Managers become facilitators, liaisons and network builders, boundary managers, resource allocaters, team champions and advocates, and in most cases, still have responsibility to watch the budget. Team members' roles change too as the group takes on increasing ownership of work processes and Agile practices. They become decision makers, conflict managers, innovators and conveners of spontaneous standup meetings that can bring production to a halt!

As with any organizational system change, the transition to self-organizing teams can be daunting – rocky and confusing. However, the process can be made easier if team members and leaders learn the fundamentals of the 'care and feeding' of teams. In particular, Agile practitioners reap benefits for their project by paying attention to three aspects of team dynamics:

- Tracking the team's progress toward self-organization
- Giving the team a good start
- Applying practices to encourage effective team dynamics
- Choosing strategies to move the team from the predictable impasses into higher productivity, satisfaction and success

TRACKING SELF-ORGANIZING TEAMS

The secret to successful teamwork lies in understanding, and then taking action based on, two factors: 1) the dynamics of team growth and development, and 2) the conditions that foster effectfacing teams change over time as the team moves through the well-documented stages of group development, first proposed by B.W. Tuckman and validated through research and empirical observation for nearly 40 years.[1] In each stage a team encounters typical team issues and interacts with six elements of teamwork:

performance, working agreements, shared responsibility, sense of purpose, communication and commitment. These elements evolve through the stages of development as the team learns to perform together.

Note that the stages are not stair steps in the sense that teams start and proceed in an orderly, linear, predictable manner onward and upward until they achieve their top performance level. The stages are stair steps more in the Shirley Temple/Bill Robinson tap-dancing mode (for a visual example, see the 1935 movie, *The Little Colonel*): two steps upward, one step down, three steps up, five steps back down, continuing until they reached the top. Teams start at the first stage often going up and down a number of times and some never settle permanently at the top. Internal issues and external forces will affect every team's developmental progress. These forces may include a large-scale organizational change, lack of supporting organizational systems, losing or gaining team members, the nature of the tasks or goals, break-up and reforming of teams, and others. Sometimes the steps through the stages seem more like a print by Escher, circling around and going nowhere in particular.

In addition, when dealing with teams, we must remember that they are rarely in a static state, but more likely to be moving along the development continuum and at any given time may exhibit characteristics associated with more than one stage. Through informed observation, team leaders and members can distinguish whether the team is making progress or has hit an impasse. That's how you know when your team could use some immediate help. Team leaders can learn about specific challenges to expect along the way, how teams can get sidetracked, how to recognize the sidetracks and strategies for re-direction.

GETTING THE TEAM STARTED RIGHT

As teams initially form and begin working together in the forming stage, individual team members often have a sensation similar to being the new kid in class. They are not fully committed to working with others yet want to feel included, or rather want not to feel excluded, by the rest of the group. Individual team members get busy gathering the information they need to orient themselves to the new Agile practices, to feel safe in the new territory and to produce software to their personal standards. In this stage the team may accomplish less concerning its task goals than managers would like. With the right support, most teams can move more quickly through this phase.

Both team members and team leaders take on new roles when Agile teams become self-organizing. The roles of team leaders change from traditional planning, controlling, directing responsibilities to ones that require being a facilitative leader, team advocate, resource allocator, boundary manager, and generally increased savvy about managing the organizational changes. Most professionals who have had leadership roles before have used some of these skills. Leading a self-organizing Agile team requires that the degree of focus shifts more toward their use. Subtle changes also occur as the role of individual contributor changes to the role of self-organizing team member. A greater degree of attention is needed to the well-being and effectiveness of the team as a whole, as well as stepping up to the accountability and sense of empowerment to make as a team decisions formerly handed down from management.

ENCOURAGING EFFECTIVE TEAM DYNAMICS

An understanding of the role of trust and communication in teamwork is fundamental for teams to develop and mature into truly self-organizing status. This means team members take more notice of process dynamics – a shift that some developers may consider a distraction from the real work. However, in addition to transmitting information and data, communication also serves a number of other purposes in developing a team that can develop efficiently and effectively. Skilled team communication serves a hierarchy of needs. It builds the foundation of trust so essential to efficiency. The presence of trust in working relationships translates into the rational commitment to the work of the team as well as the emotional commitment leading to loyalty to one's colleagues – the glue that holds the team together. That glue

shows its importance when disagreements and inevitable conflicts in opinion arise. A mutual acknowledgement of shared commitment to the work and to each other all allows team members to work toward solutions to conflict constructively without avoidance. The fear of uncomfortable angry, distrustful confrontations is lessened or absent. As a team develops the ability to surface and work through conflicts, its capacity for innovation and creative problem-solving skyrockets, leading inexorably to full self-organization, high performance and true agility.

For example, the promulgating the adoption following six communication tools are effective in providing a work environment that generates trust:

1. Credibility: Be consistent and reliable, follow-through
2. Tune In: Show the other person you listened
3. Self-disclose: Lift your "Mask" (even a little helps)
4. Empathy: Put yourself in the other person's "shoes"
5. Stretch: Express interest in the team and team mates
6. Communicate: Seek and give effective feedback

KEEPING THE TEAM MOVING IN THE RIGHT DIRECTION

Depending on their stage of development, teams encounter predictable challenges that can sidetrack attention and affect performance. Addressing those challenges and ensuring the team is developing appropriate quality and interaction skills along the way, endows the team with a greater ability to stay on course toward the goal. For example, after the team has worked together enough to understand each other's commitment to getting the job done, they may be willing to acknowledge more difference of opinion and surface more conflicts. On the way to self-organizing, during this stage team members test the extent of their power and control over their work processes. A wise manager will expect the increased tension and stress this can cause, recognize it as progress in team development and be ready to support team members in getting the information and support they need from other parts of the organization. In addition, tracking and celebrating the early small successes promotes team cohesion and ramps up momentum toward results for the customer.

The potential rewards of Agile self-organizing teams are great; however, results are not achieved without the investment of focus by team leaders and members – focus on the skills to be honed at each stage of the team's development. Ask yourself, "When I think of all my experiences as a part of a team, whether as a leader or member of the team, what stands out for me as the highpoints? When have I been a part of a team (or teams) that really clicked together and accomplished its purposes?" When you have the answer to that question firmly in mind, consider the factors that led to your sense of success. What can you do to replicate those conditions for your next team? How can you involve others on the team in creating a body of knowledge about team success? (I suggest project retrospectives as one technique.) Take the time to learn more about what makes Agile teams move to self-organized, high performance. Your projects and your teams will benefit.

AUTHOR BIO

Also known as the "Industrial XP Change Goddess," Diana Larsen is a senior organizational development and change management consultant with Industrial Logic Inc. (www.industriallogic.com). A specialist in the "I" of Industrial XP (www.industrialxp.org), Diana conducts readiness assessments and facilitates processes (including project chartering and retrospectives) that support and sustain change initiatives, attain effective team performance and retain organizational learning. Diana is a certified Scrum Master, writes articles on XP management and organizational change, and frequently speaks at Agile/XP conferences. Reach Diana at www.industriallogic.com, diana@industriallogic.com, or 503-288-3550.

There are a number of ways to tell that an idea is truly revolutionary: it is simple, elegant and powerful; it combines existing techniques in a novel way; it changes the way we do things; it has some kind of strange, unexpected impact; and most importantly, when you see it, you ask yourself, "Why didn't I think of that?"

In 2001, a survey of Extreme Programming practitioners revealed that Acceptance Testing (or Customer Testing, as we now call it) was one of the three most difficult practices to implement effectively. Another survey indicated that a majority of respondents did not automate their acceptance tests. I share this perspective, having never been on a project that has done Customer Testing well. There have been two underlying problems: end-to end tests are expensive to maintain over the long term and customers do not know how to write executable tests themselves. Since the customers do not write their own tests, the programmers have to do it, which takes time and elevates the risk of incorrectly translating the customer's requirements into code. When the programmers do write the tests, they usually test the application from end to end. As the application evolves, this growing collection of tests becomes increasingly costly to maintain. These two problems together have encouraged me to be much less vigilant than I could be in turning the Customer Testing dial up to 11.

And then one of those revolutionary ideas appeared. Ward Cunningham launched fit.c2.com, a site devoted to his new testing framework, FIT. In his own words, FIT is about "tests people can read." Ward'snew tool makes it possible to turn documents into executable tests by focusing on tables. Many business processes areepresented in documentation by decision tables. Commercial shipping rules are represented by tables. Employee source reductcion amounts are represented by tables. What better resource do we have that these tables when it comes time to verify whether we have correctly implemented a business process, or a shipping module, or a source deduction calculator? With FIT, we write code to interpret these tables, then pass our customer's documents through the FIT test runner. Table cells change color: green means success, red means failure, and yellow means that something went wrong. Once the table-interpreting code is in place, which FIT calls a fixture, customers can add their own tests by adding rows to their tables. With FIT, the customer can finally own their tests.

Browsing Music

The music browser starts up looking at the whole library of songs. We specify the library (an advanced feature) so that we know what we are talking about in this document.

fit.ActionFixture		
start	eg.music.Browser	
enter	library	Source/eg/music/Music.txt
check	total songs	37

This is a the file that library reads. It is tab separated text. Try downloading it and looking at it with a spreadsheet.
http:Release/Source/eg/music/Music.txt

We can pick songs and see details of our selection as we go.

fit.ActionFixture		
enter	select	1
check	title	Akila
check	artist	Toure Kunda
enter	select	2
check	title	American Tango
check	artist	Weather Report
check	album	Mysterious Traveller
check	year	1974
check	time	3.70
check	track	2 of 7

ActionFixture interprets the words in the first column. The actions operate on fields and buttons on the Browser *screen we started in the*

In 2003, Diaspar Software joined the ranks of companies using FIT to enable their customers

to write their own tests. The results have been very positive so far: on a project with 500 programmer tests, our customer wrote over a dozen tests of their own, verifying over 200 individual cells of data. What's more, our customer was able to add these tests entirely without help from the programmers.This was especially useful for late-changing and misunderstood requirements. One business rule started out with 19 individual test cases, each its own row of a FIT fixture. Our customer identified two entire special cases they had neglected to tell us during our initial story conversation. In past projects, the customer would have informed us of the problem then we would have spent a few hours reviewing the previously-stated requirements, understanding the new requirements, then writing additional tests to determine whether the feature already handles these new special cases. This time, however, the customer simply added four rows to the table, executed the tests, saw a sea of green and sent a polite e-mail saying, "Just to let you know that we missed a couple of special cases, but we tried them out and they already work, so don't worry about it." This is just one way that FIT has saved us time and effort in our work.

Not long after FIT appeared on the testing landscape, Bob Martin of ObjectMentor began to use it, and quickly grew tired of its command-line test runner. He wanted to execute FIT tests he had already written using Wiki, the open web-based collaboration platform that has itself transformed the way many organizations share project information. He wanted to execute his FIT tests "with finesse," and so became involved in developing FitNesse, a standalone Wiki capable of executing FIT tests.

FitNesse is simple: you can download it, start it and begin using it in seconds. Instead of clipping text descriptions of acceptance tests to story cards, you can write each story as a Wiki page, then annotate the story with its customer tests! It is this very feature that has allowed Diaspar Software to amplify the effectiveness of a customer who cannot remain on site. The mobile customer can visit the Wiki from

[redacted].fixture.PlayerEligibilityRule

dateOfBirth	schoolType	academicGraduation	eligibilityYear()	comment
7/21/1983	none	null	2004	Turns 21 on draft day plus 44 days exactly
7/22/1983	none	null	2004	Turns 21 on draft day plus 45 days exactly
7/23/1983	none	null	2005	Turns 21 on draft day plus 46 days exactly
7/22/1986	none	null	2007	Turns 21 by estimated 2007 draft day plus 45 days
12/31/1989	high school	5/2004	2004	High school senior, even though he's pretty young
12/31/1989	high school	6/2004	2004	High school senior, even though he's pretty young
12/31/1989	high school	7/2004	2004	Looks like a high school senior in a strange school that graduates late
12/31/1989	high school	8/2004	2004	Looks like a high school senior in a strange school that graduates late
12/31/1989	high school	9/2004	2004	Looks like a high school senior in a strange school that graduates late
12/31/1989	high school	6/2005	2005	High school junior
12/31/1989	high school	6/2006	2006	High school sophomore
12/31/1989	2-year college	6/2004	2004	Last year at 2-year college
12/31/1989	2-year college	6/2005	2004	First year at 2-year college
12/31/1989	2-year college	6/2006	2005	Taking three years at 2-year college
12/31/1989	4-year college	6/2004	2004	4-year college senior
12/31/1989	4-year college	6/2005	2004	4-year college junior
12/31/1989	4-year college	6/2006	2005	4-year college sophomore
12/31/1989	4-year college	6/2007	2006	4-year college freshman

anywhere, check up on us by executing the tests, and add their own tests by navigating to the story's Wiki page. While it is not a substitute for the on-site customer, it compensates better for the lack of an on-site customer compared to other techniques we have tried in the past.

While we can go on extolling the virtues of FIT and FitNesse, we do need to mention that these tools do not eliminate the need for programmer involvement in writing customer tests. You still need programmers to write the fixture code that sits behind these FIT tables. Without these fixtures, tables are just tables. It is only when you add fixture code that these tables become tests.

There is a certain learning curve as programmers try to turn their customer's data into a row fixture, a column fixture or an action fixture. Also, since tables contain text, fixture code needs to parse text into objects and format objects back into text. The good news is that if your application has any kind of user interface, the programmers will already need to parse text and format objects, so your FIT fixtures and your application can share those services. These fixtures sit just under the look-and-feel part of your user interface, but use the logical part of your user interface, killing two birds with one stone. First, FIT helps you test the most important parts of your user interface: navigating from feature to feature, interpreting user requests and choosing the right response. Next, FIT tests avoid the common pitfalls of end-to-end tests: they do not depend on your application's look and feel, so changing the position of controls on the screen or the text on your buttons does not cause false failures in your tests. This is one area in which FIT has had an unexpected impact on application design: we have noticed a cleaner separation between presentation services related to technology (web interface? desktop graphical interface? XML messaging interface?) and presentation services related to the application (currency, number and date/time formats), resulting in even lower coupling and higher cohesion than we already get from practising Test-Driven Design. In addition to enabling our customers to own their tests, FIT has made us better at designing user interfaces. We wonder what FIT is going to teach us next!

Ward Cunningham noticed that his customers had all these documents containing tables. He noticed that he was writing software to behave according to those tables. He thought, "Wouldn't it be great if we could just execute those tables somehow?" And at that moment FIT was born. Such a simple idea – one that holds the promise of helping customers truly own their tests. One that paves the way to effective customer testing, arguably the most important of the Extreme Programming practices. With FIT in our toolbox, we are a more successful organization. Why didn't I think of that?

ABOUT THE AUTHOR

J. B. Rainsberger is the founder of Diaspar Software Services, a Toronto-based Agile software firm. He participates actively in the Toronto Agile and XP community as a speaker, programmer, writer and trainer. His book, "Programmer Testing with JUnit: Recipes for Better Java Code" is due to be released in May 2004 by Manning publications.

INTRODUCTION

The idea of unit testing seems to always evoke a strong reaction in people. For those that buy into the concept, they have unanimously stated that good unit tests are difficult to write, and some question whether the tests they have written were really worth it while others rave about their effectiveness. On the other hand, there is also a large community that guffaws at the idea of unit testing, especially the concept that "the code is good when it passes the unit tests." When all the hoopla dies down, unit testing may one day be relegated to the dusty shelf of "yet another programmer too If this fate is to be changed, unit testing has to be embraced by both the community and the tool developers. The next version of Microsoft's Visual Studio will include tools to automate refactoring. It seems obvious to me that tools that automate unit test generation would not only address some of the issues concerning maintenance and cost, but would also introduce the concept to a much wider audience.

However, to achieve this acceptance, unit testing must be formalized so that it becomes a real engineering discipline rather than an ad hoc approach that relies on the dubious capabilities of the programmer. After all, the unit test is supposed to test the code that the programmer writes. If the programmer writes bad code to begin with, how can you expect anything of better quality in the tests? Of even more concern is the concept that the unit test should be written first, before the code that is to be tested. To a certain extent, this implies that not only does the programmer have to consider what the code will *do*, he/she has to consider how the code is *designed*. Both drive the interface. This is why many people balk at the idea of writing the unit test first—it places them in the uncomfortable position of having to do up front design work without consciously recognizing that this what they are doing.

So, we are faced with a double edged sword. First, there is no formal unit test engineering discipline established in the community that provides a guide to the programmer and works to ensure some level of unit test quality. Second, the prerequisite that the design has to be somewhat formalized before any tests can be written causes difficulty for many programmers because they either don't have formal design experience or simply don't like up front design work. Aggravating this situation is the idea that up front design work can be replaced under the guise of "refactoring".

In order to blunt this sword, two things are needed—a formalization of unit testing by establishing unit test patterns, and the early adoption of object oriented design patterns in the developing application to specifically target the needs of unit testing. This article will paint a picture of this two pronged solution with some very large brush strokes. The intention is to whet your appetite and hopefully begin a dialog amongst yourselves that will lead to a more formal unit test engineering process, similar to object oriented design, design patterns, and refactoring.

As you read this article, keep in mind that one of the goals is a tool suite that can be used to automatically generate unit tests, both as a reverse and forward engineering process. With the latter, it should be possible to generate the method stubs for the code under test. After all, one of the benefits of unit testing is that it provides the implementer with some documentation as to the expected structure and behavior of the code under test. Also, to keep this article in the general reader category, there are no code examples.

PATTERNS

The patterns that I have identified so far can be loosely categorized as:

- pass/fail patterns
- collection management patterns
- data driven patterns
- performance patterns

- process patterns
- simulation patterns
- multithreading patterns
- stress test patterns

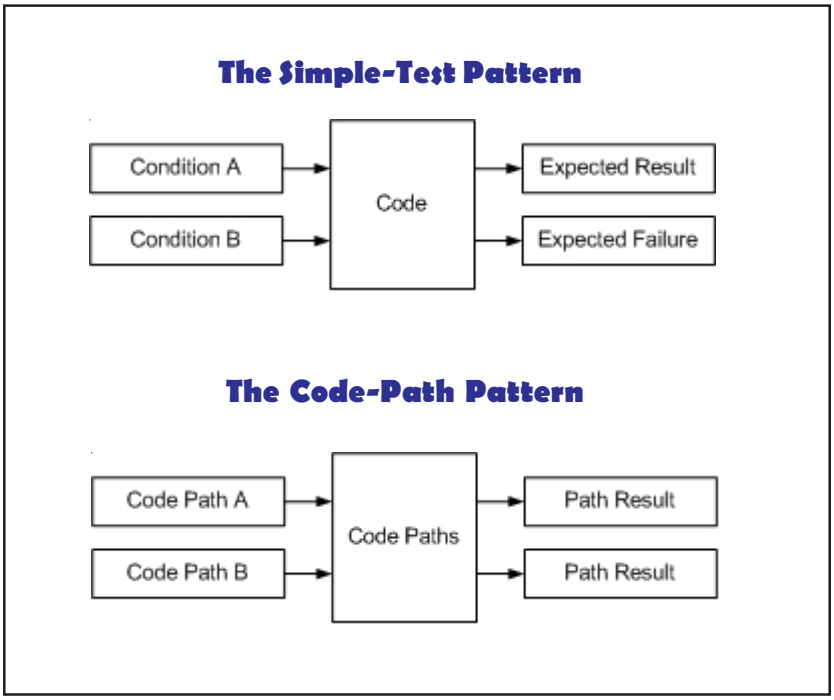
Again, let me emphasize that these are broad brush strokes. From my research, this appears to be quite new territory.

PA$$/FAIL PATTERN$

These patterns are your first line of defense (or attack, depending on your perspective) to guarantee good code. But be warned, they are deceptive in what they tell you about the code.

The Simple-Test Pattern

Pass/fail unit tests are the simplest pattern and the pattern that most concerns me regarding the effectiveness of a unit test. When a unit test passes a simple test, all it does is tell me that the code under test will work if I give it exactly the same input as the unit test. A unit test that exercises an error trap is similar—it only tells me that, given the same condition as the unit test, the code will correctly trap the error. In both cases, I have no confidence that the code will work correctly with any other set of conditions, nor that it will correctly trap errors under any other error conditions. This really just basic logic. However, on these grounds you can hear a lot of people shouting "it passed!" as all the nodes on the unit test tree turn green.



The Code-Path Pattern

The Simple-Test pattern typifies what I call "black box testing." Without inspecting the code, that's about all you can do—write educated guesses as to what the code under test might encounter, both as success cases and failure cases, and test for those guesses.

A better test ensures that at least all the code paths are exercised. This is part of "white box testing"—knowing the inside workings of the code being tested. Here the priority is not to set up the conditions to test for pass/fail, but rather to set up conditions that test the code paths. The results are then compared to the expected output for the given code path. But now we have a problem—how can you do white box testing (testing the code paths) when the code hasn't been written? Here we are immediately faced with the "design before you code" edge of that sword. The discipline here, and the benefit of unit testing by enforcing some up front design, is that the unit test can test for code paths that the implementer may not typically consider. Furthermore, the unit test documents precisely what the code path is expected to do. Conversely, discipline is needed during implementation when it is discovered.

The Parameter-Range Pattern

Still, the above test, while improving on the Simple-Test pattern, does nothing to convince me that the code handles a variety of pass/fail conditions. To do this, the code should be tested using a range of condi

tions. The Parameter-Range pattern does this by feeding the Code-Path pattern with more than a single parameter set. Now I am finally beginning to have confidence that the code under test can actually work in a variety of environments and conditions.

DATA DRIVEN TEST PATTERNS

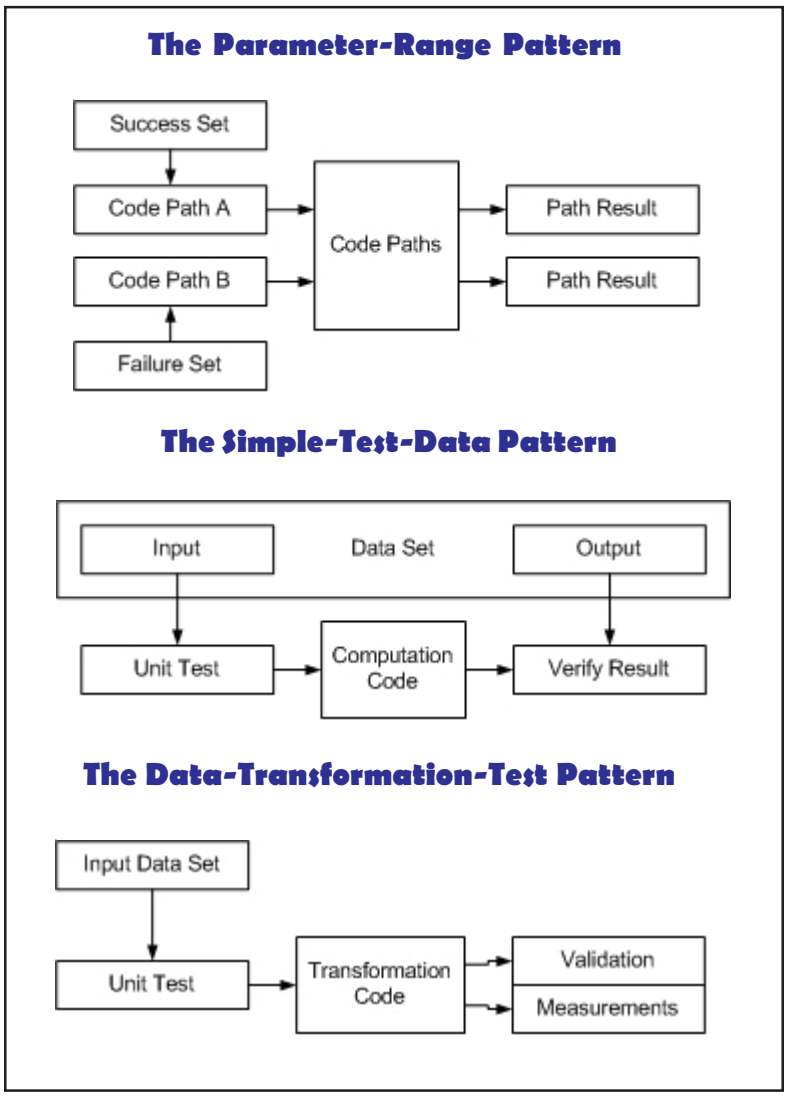
Constructing Parameter-Range unit tests is doable for certain kinds of testing, but it becomes inefficient and complicated to test at a piece of code with a complex set of permutations generated by the unit test itself. The data driven test patterns reduce this complexity by separating the test data from the test. The test data can now be generated (which in itself might be a time consuming task) and modified independent of the test.

The Simple-Test-Data Pattern

In the simplest case, a set of test data is iterated through to test the code and a straightforward result (either pass or fail) is expected. Computing the result can be done in the unit test itself or can be supplied. In the simplest case, a set of test data ishrough to test the code and a straightforward result (either pass or fail) is expected. Computing the result can be done in the unit test itself or can be supplied with the data set. Variances in the result are not permitted. Examples of this kind of of Simple-Test-Data pattern include checksum calculations, mathematical algorithms, and simple business math calculations. More complex examplesinclude encryption algorithms and lossless encoding or compression algorithms.

The Data-Transformation-Test Pattern

The Data-Transformation-Test pattern works with data in which a qualitative measure of the result must be performed. This is typically applied to transformation algorithms such as lossy compression. In this case, for example, the unit test might want to measure the performance of the algorithm with regard to the compression rate vs. the data loss. The unit test may also need to verify that the data can be translated back into something that resembles the input data within some tolerance. There are other applications for this kind of unit test—a rounding algorithm that favors the merchant rather than the customer is a simple example. Another example is precision. Precision occurs frequently in business—the calculation of taxes, interesting, percentages, etc., all of which ultimately must be rounded to the penny or dollar but can have dramatic effects on the resulting value if precision is not managed correctly throughout the calculation.



Data Transaction Patterns

Data transaction patterns are a start at embracing the issues of data persistence and communication.

More on this topic is discussed under "Simulation Patterns." Also, these patterns intentionally omit stress testing, for example, loading on the server. This will be discussed under "Stress-Test Patterns."

The Simple-Data-I/O Pattern

This is a simple data transaction pattern, doing little more than verifying the read/write functions of the service. It may be coupled with the Simple-Test-Data pattern so that a set of data can be handed to the service and read back, making the transaction tests a little bit more robust.

The Constraint-Data Pattern

The Constraint-Data pattern adds robustness to the Simple-Data-I/O pattern by testing more aspects of the service and any rules that the service may incorporate. Constraints typically include:

- can be null
- must be unique
- default value
- foreign key relationship
- cascade on update
- cascade on delete

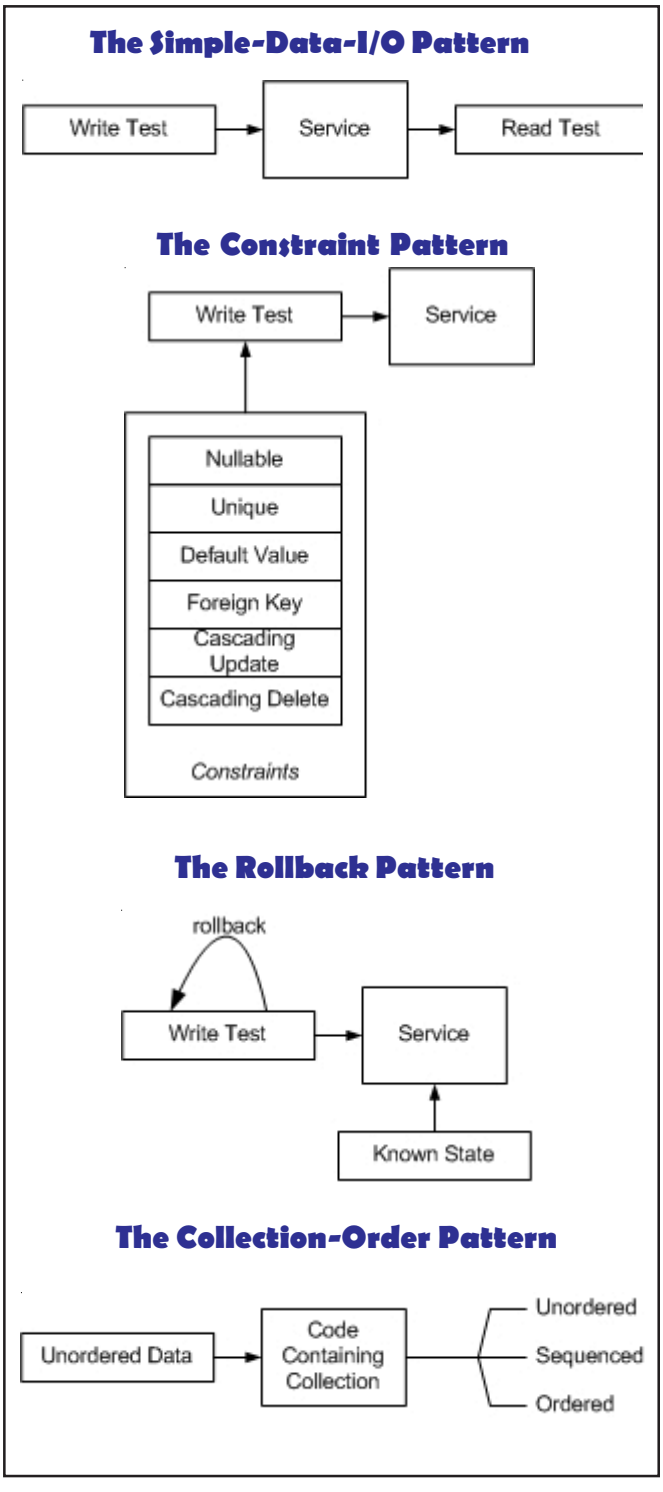
As the diagram illustrates, these constraints are modeled after those typically found in a database service and are "write" oriented. This unit test is really oriented in verifying the service implementation itself, whether a DB schema, web service, or other model that uses constraints to improve the integrity of the data.

The Rollback Pattern

The rollback pattern is an adjunct to the other transaction testing patterns. While unit tests are supposed to be executed without regard to order, this poses a problem when working with a database or other persistent storage service. One unit test may alter the dataset causing another unit test to inappropriately fail. Most transactional unit tests should incorporate the ability to rollback the dataset to a known state. This may also require *setting* the dataset into a known state at the beginning of the unit test. For performance reasons, it is probably better to configure the dataset to a known state at the beginning of the test suite rather than in each test and use the service's rollback function to restore that state for each test (assuming the service provides rollback capability).

COLLECTION MANAGEMENT PATTERNS

A lot of what applications do is manage collections of information. While there are a variety of collections



available to the programmer, it is important to verify (and thus document) that the code is using the correct collection. This affects ordering and constraints.

The Collection-Order Pattern

This is a simple pattern that verifies the expected results when given an unordered list. The test validates that the result is as expected:

- unordered
- ordered
- same sequence as input

This gives the implementer crucial information on how the container is expected to manage the collection.

The Enumeration Pattern

This pattern verifies issues of enumeration, or collection traversal. For example, a collection may need to be traversed forwards and backwards. This is an important test to perform when collections are non-linear, for example a collection of tree nodes. Edge conditions are also important to test—what happens when the collection is enumerated past hte first or last item in the collection?

The Collection-Constraint Pattern

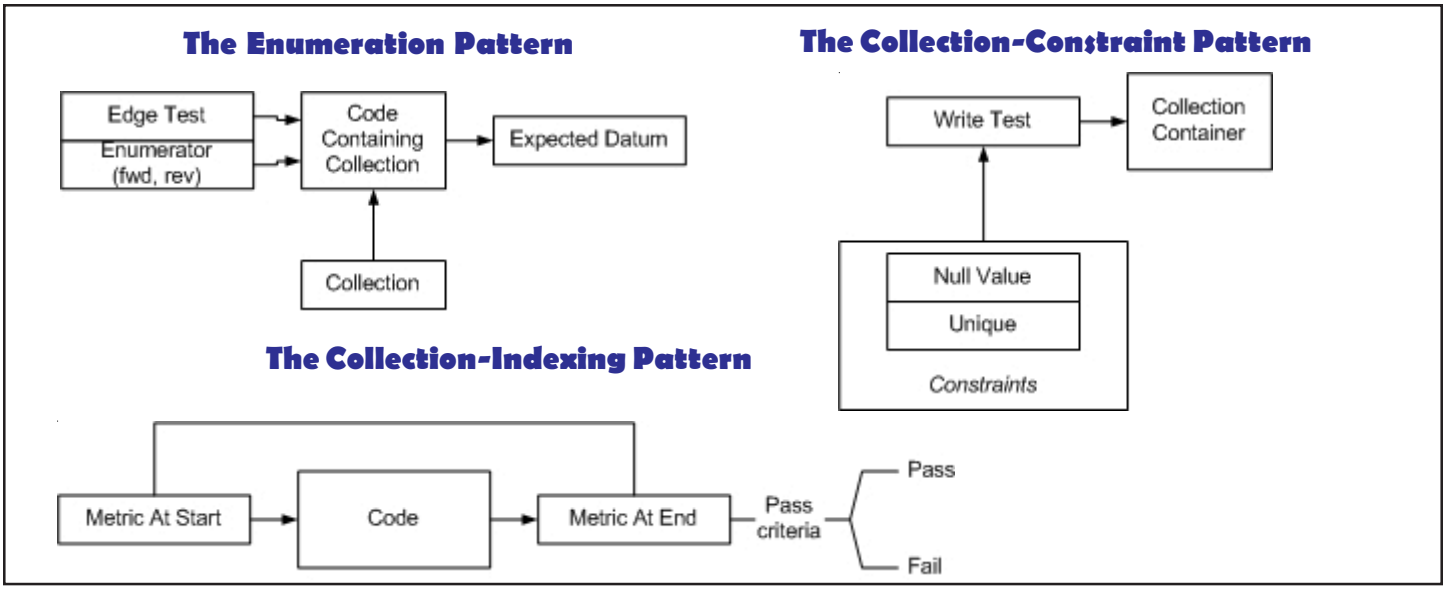
This pattern verifies that the container handles constraint violations: null values and inserting duplicate keys. This pattern typically applies only to key-value pair collections.

The Collection-Indexing Pattern

The indexing tests verify and document the indexing methods that the collection container must support—by index and/or by key. In addition, they verify that update and delete transactions that utilize indexing are working properly and are protected against missing indexes.

RFORMANCE PATTERN$

Unit testing should not just be concerned with function but also with form. How efficiently does the code under test perform its function? How fast? How much memory does it use? Does it trade off data insertion for data retrieval effectively? Does it free up resources correctly? These are all things that are under the purview of unit testing. By including performance patterns in the unit test, the implementer has a goal to reach, which results in better code, a better application, and a happier customer.



The Performance-Test Pattern

The basic types of performance that can be measured are:

- Memory usage (physical, cache, virtual)
- Resource (handle) utilization
- Disk utilization (physical, cache)
- Algorithm Performance (insertion, retrieval, indexing, and operation)

Note that some languages and operating systems make this information difficult to retrieve. For example, the C# language with its garbage collection is rather difficult to work with in regards to measuring memory utilization. Also, in order to achieve meaningful metrics, this pattern must often be used in conjunction with the Simple-Test-Data pattern so that the metric can measure an entire dataset. Note that just-in-time compilation makes performance measurements difficult, as do environments that are naturally unstable, most notably networks. I discuss the issue of performance and memory instrumentation in my fourth article in a series on advanced unit testing found at http://www.codeproject.com/csharp/autp4.asp.

PROCESS PATTERNS

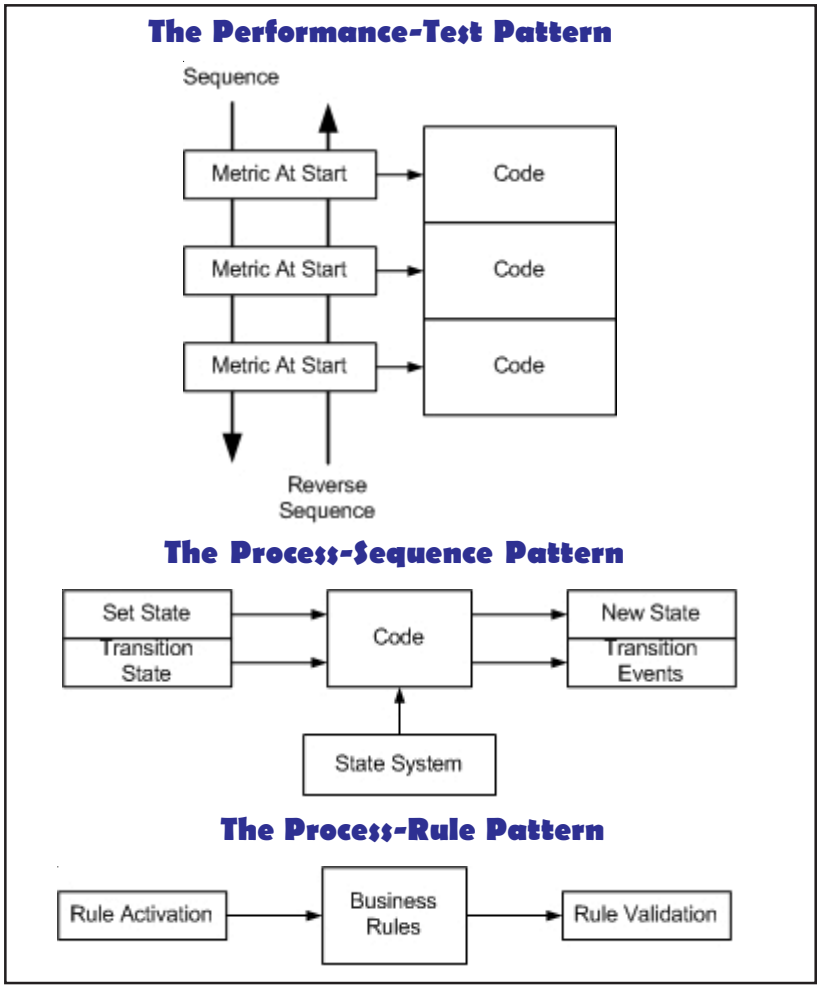
Unit testing is intended to test, well, units...the basic functions of the application. It can be argued that testing processes should be relegated to the acceptance test procedures, however I don't buy into this argument. A process is just a different type of unit. Testing processes with a unit tester provide the same advantages as other unit testing—it documents the way the process is intended to work and the unit tester can aid the implementer

The Process-Sequence Pattern

This pattern verifies the expected behavior when the code is performed in sequence, and it validates that problems when code is executed out of sequence are properly trapped. The Process-Sequence pattern also applies to the Data-step, improving performance and maintainability of the unit test structure.

The Process-State Pattern

The concept of state cannot be decoupled from that of process. The whole point of managing state is so that the process can transition smoothly from one state to another, performing any desired activity. Especially in "stateless" systems such as web applications, the concept of state (as in the state of the session) is important to test. To accomplish this without a complicated client-server setup and manual actions requires a unit tester that can understand states and allowable transitions and possibly also work with



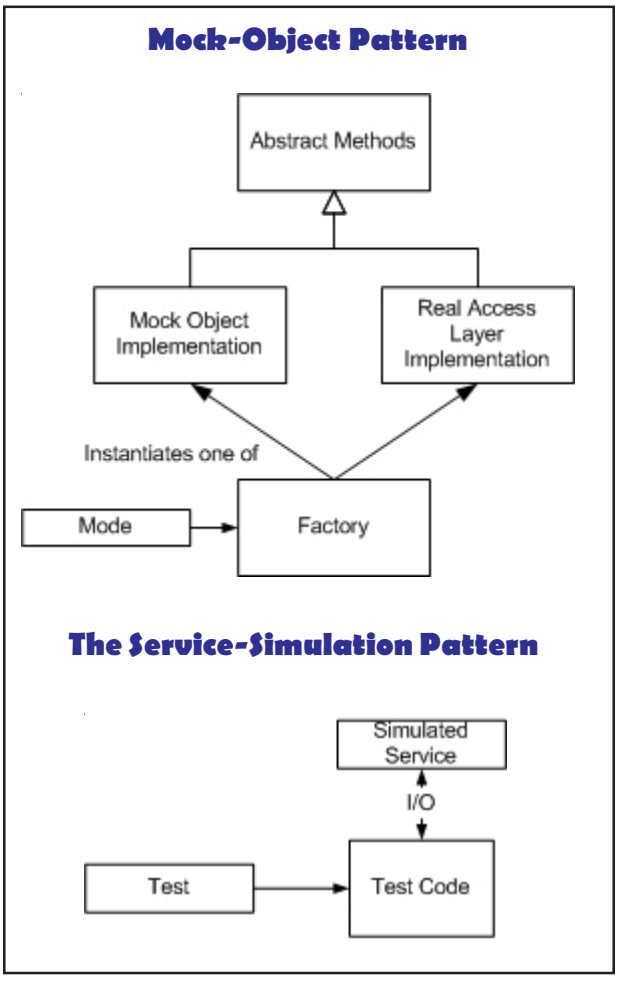
This test is similar to the Code-Path pattern—the intention is to verify each business rule in the system. To implement such a test, business rules really need to be properly decoupled from surrounding code—they cannot be embedded in the presentation or data access layers. As I state elsewhere, this is simply good coding, but I'm constantly amazed at how much code I come across that violates these simple guidelines, resulting in code that is very difficult to test in discrete units. Note that here is another benefit of unit testing—it enforces a high level of modularity and decoupling.

SIMULATION PATTERNS

Data transactions are difficult to test because they often require a preset configuration, an open connection, and/or an online device (to name a few). Mock objects can come to the rescue by simulating the database, web service, user event, connection, and/or hardware with which the code is transacting. Mock objects also have the ability to create failure conditions that are very difficult to reproduce in the real world—a lossy connection, a slow server, a failed network hub, etc. However, to properly use mock objects the code must make use of certain factory patterns to instantiate the correct instance—either the real thing or the simulation. All too often I have seen code that creates a database connection and fires off an SQL statement to a database, all embedded in the presentation or business layer! This kind of code makes it impossible to simulate without all the supporting systems—a preconfigured database, a database server, a connection to the database, etc. Furthermore, testing the result of the data transaction requires another transaction, creating another failure point. As much as possible, a unit test should not in itself be subject to failures outside of the code it is trying to test.

Mock-Object Pattern

In order to properly use mock objects, a factory pattern must be used to instantiate the service connection, and a base class must be used so that all interactions with the service can be managed using virtual methods. (Yes, alternatively, Aspect Oriented Programming practices can be used to establish a pointcut, but AOP is not available in many languages). The basic model is as shown in this diagram. To achieve this construct, a certain amount of foresight and discipline is needed in the coding process. Classes need to be abstracted, objects must be constructed in factories rather than directly instantiated in code, facades and bridges need to be used to support abstraction, and data transactions need to be extracted from the presentation and business layers. These are good programming practices to begin with and result in a more flexible and modular implementation. The flexibility to simulate and test complicated transactions and failure conditions gains a further advantage to the programmer when mock objects are used.



The Service-Simulation Pattern

This test simulates the connection and I/O methods of a service. In addition to simulating an existing service, this pattern is useful when developing large applications in which functional pieces are yet to be implemented.

The Bit-Error-Simulation Pattern

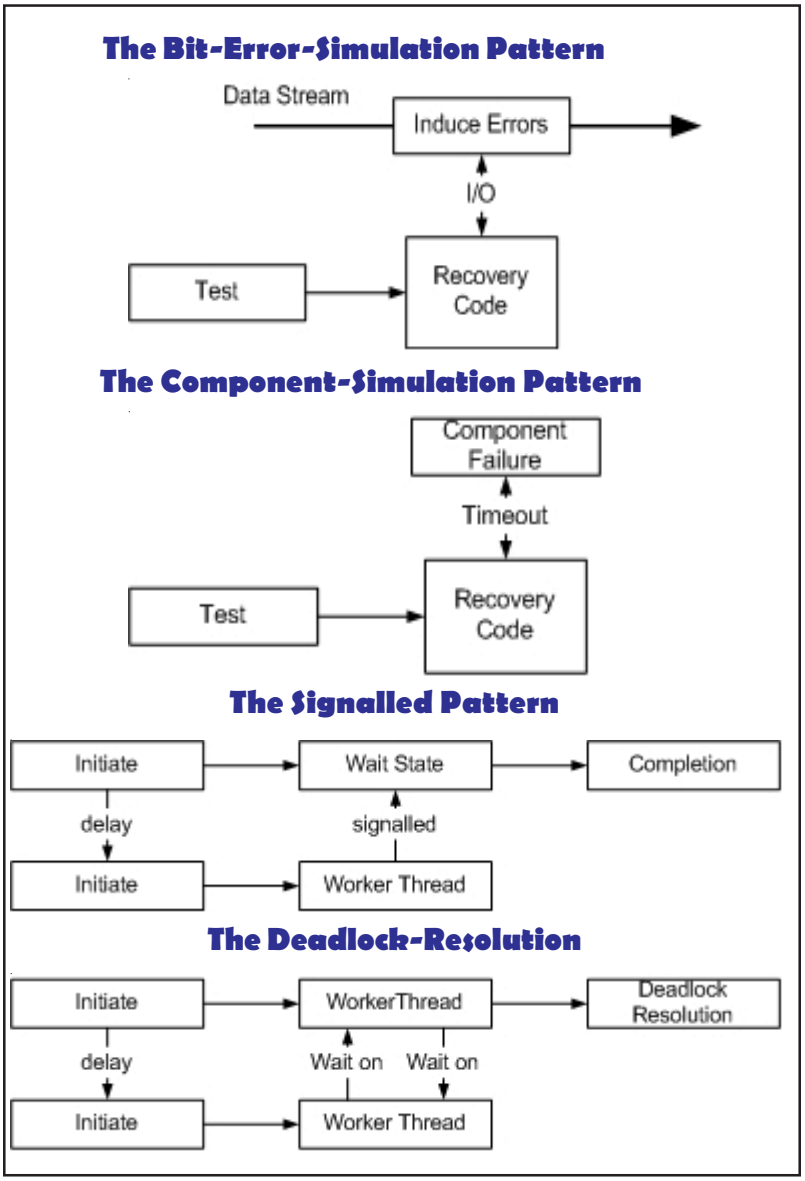
I have only used this pattern in limited applications such as simulating bit errors induced by rain-fade in satellite communications. However, it is important to at least consider where errors are going to be handled in the data stream—are they handled by the transport layer or by higher level code? If you're writing a transport layer, then this is a very relevant test pattern.

The Component-Simulation Pattern

In this pattern, the mock object simulates a component failure, such as a network cable, hub, or other device. After a suitable time, the mock object can do a variety of things:

- throw an exception
- return incomplete or completely missing data
- return a "timeout" error

Again, this unit test documents that the code under test needs to handle these conditions.



MULTITHREADING PATTERNS

Unit testing multithreaded applications is probably one of the most difficult things to do because you have to set up a condition that by its very nature is intended to be asynchronous and therefore non-deterministic. This topic is probably a major article in itself, so I will provide only a very generic pattern here. Furthermore, to perform many threading tests correctly, the unit tester application must itself execute tests as separate threads so that the unit tester isn't disabled when one thread ends up in a wait state.

The Signalled Pattern

This test verifies that a worker thread eventually signals the main thread or another worker thread, which then completes its task. This may be dependent on other services (another good use of mock objects) and the data on which both threads are opearting, thus involvingother test patterns as well.

The Deadlock-Resolution Pattern

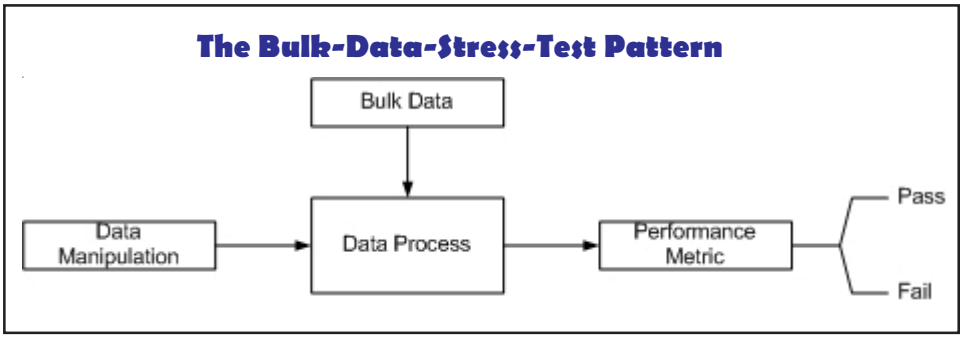
This test, which is probably very complicated to establish because it requires a very thorough understanding of the worker threads, verifies that deadlocks are resolved.

STRESS-TEST PATTERNS

Most applications are tested in ideal environments—the programmer is using a fast machine with little network traffic, using small datasets. The real world is very different. Before something completely breaks, the application may suffer degradation and respond poorly or with errors to the user. Unit tests that verify the code's performance under stress should be met with equal fervor (if not more) than unit tests in an ideal environment.

The Bulk-Data-Stress-Test Pattern

This test is designed to validate the performance of data manipulation when working with large data sets. These tests will often reveal inefficiencies in insertion, access, and deletion processes which are typically corrected by reviewing the indexing, constraints, and structure of the data model, including whether code is should be run on the client or the server.

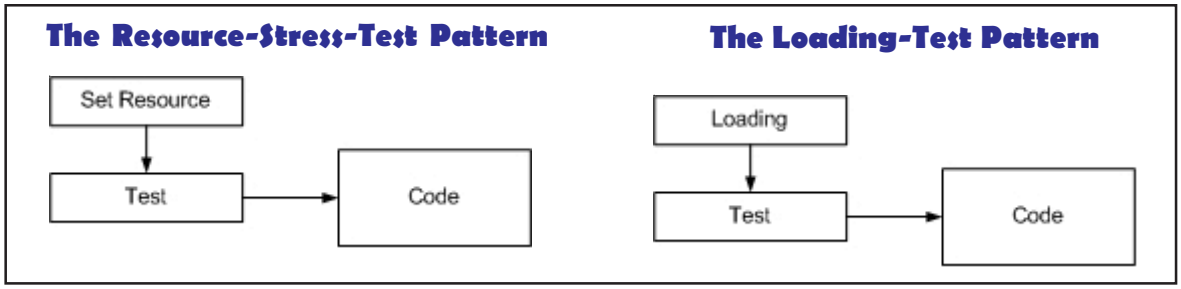


The Resource-Stress-Test Pattern

Resource consumption stress testing depends on features of the operating system and may be served better by using mock objects. If the operating system supports simulating low memory, low disk space, and other resources, then a simple test can be performed. Otherwise, mock objects must be used to simulate the response of the operating system under a low resource condition.

The Loading-Test Pattern

This test measures the behavior of the code when another machine, application, or thread is loading the "system", for example high CPU usage or network traffic. This is a simulation only (which does not use mock objects) and therefore is of dubious value. Ideally, a unit test that is intended to simulate a high volume of network traffic would create a thread to do just that—inject packets onto the network.



CONCLUSION

This article has described 24 test patterns that hopefully bring the technique of unit testing closer to a more formal engineering discipline. When writing unit tests, reviewing these patterns should help in identifying the kind of unit test to write, and the usefulness of that unit test. It also allows the developer to choose how detailed the unit tests need to be—not every piece of code needs to be stress tested, nor is it cost effective to do so.

As you can see from this article, we desperately need some tools that generate unit tests automatically. I'm not much of an advocate of code generates, but I can see that unit testing would really benefit

from code generation. It would all but eliminate the arguments against unit testing based on cost and effectiveness, and could also be used as an application generating tool—given information on the unit test, the code generator can also create the application code classes, structure, and stubs.

ABOUT THE AUTHOR

Marc Clifton lives with his son and girlfriend in Rhode Island and works as an industry consultant and developer. He is currently writing a book on the application of Agile Methods in the .NET framework, has made numerous contributions to The Code Project website, and has started an Advanced Unit Testing project (http://aut.tigris.org/) on Tigris.org. To contact him, please send email to webmaster@knowledgeautomation.com.

Extreme Locality
Brad Appleton

In Agile methods, we often hear a lot about the importance of simplicity: simple design; do the simplest thing that could possibly work; simple tools; minimal documentation... Much of the documentation and artifacts created in larger software development methods is for the sake of capturing historical knowledge: the rhyme and reasons behind why something is there, or is designed a certain way.

The desire for such information is often used to justify the need for formal traceability and additional documentation for the sake of maintainability and comprehension. Some very powerful and sophisticated tools exist to do this sort of thing. And yet, there are basic fundamental principles and simple tactics to apply that can eliminate much of this burden.

WHENCE FORMAL TRACEABILITY?

The mandate for formal traceability originated from the days of Department of Defense (DoD) development with very large systems that included both hardware and software, and encompassed many geographically dispersed teams collaborating together on different pieces of the whole system. The systems were often mission critical in that a typical "bug" might very likely result in catastrophic loss of some kind (loss of life, limb, livelihood, national security, or obscenely large sums of money/funding).

At a high level, the purpose of formal traceability was three-fold:

1. Aid project management by improving change **Impact Analysis** (to help estimate effort/cost, and assess risk)
2. Help ensure **Product Conformance** to requirements specs (i.e. ensure the design covers every requirement, the implementation realizes every design element and every requirement)
3. Help ensure **Process Compliance** (only the authorized individuals worked on the things [requirements, tasks, etc.] they were supposed to do)

On a typical Agile project, there is a single team of typically less than two-dozen. And that team is likely to be working with less than 10 million lines of code (probably less than 1 million). In such situations, many of the aforementioned needs for formal traceability can be satisfactorily ensured without the additional rigor and overhead of full-fledged formal requirements tracing.

Rigorous traceability isn't always necessary for the typical Agile project, except for the conformance auditing, which some Agile methods accomplish via test-driven design (TDD). A "coach" would be responsible for process conformance via good practices and good "teaming," but likely would not need to have any kind of formal audit (unless obligated to do so by contract or by market demand or industry standards).

NEWS

Page A12 / News

Agile Consultancy ThoughtWorks Adding to Worldwide Staff

The increasing acceptance of Agile methods by Global 1000 companies has helped drive continued success at ThoughtWorks, Inc. Company leaders disclosed today that they are hiring professionals

"Global leaders are quickly realizing that Agile offers distinct advantages over traditional methods," explained Roy Singham, ThoughtWo president a

ThoughtWorks has long been a leading practitioner and advocate of Agile methods. Our successful application of Agile has been documented in *CIO, ComputerWorld, Software Development* and elsewhere. Our people—including Chief Scientist Martin Fowler—are known in the Agile community as leading speakers, authors and practitioners.

Today, we're enjoying continued success. As a result, we're looking for Agile-minded professionals to fill a variety of positions in our offices around the world. While our most immediate need is for analysts, architects, project managers and developers, we welcome inquiries from qualified personnel in all roles.

If you're interested in joining a company that's using Agile methods to deliver on some of the most complex projects in the world, and setting higher standards for the delivery of quality software, please send your resume to work@thoughtworks.com.

ThoughtWorks®

The art of heavy lifting.℠

Chicago • New York • San Francisco • Nashville • Calgary • London • Melbourne • Bangalore

W W W . T H O U G H T W O R K S . C O M

Agile methodologies turn the knob up to 10 on product conformance by being close to the customer, by working on micro-sized changes/increments to ensure that minimal artifacts are produced (and hence with minimal reconciliation) and that communication feedback loops are small and tight. Fewer artifacts, better communication, pebble-sized change-tasks with frequent iterations tame the tiger of complexity!

THE PRINCIPLE OF LOCALITY OF REFERENCE DOCUMENTATION (LORD)

Not all software projects fit into the ideal smaller-scale environment with closely collaborative project communities. These larger projects require more artifacts. More artifacts, means more things to trace, and more differences to reconcile, and more effort to track and maintain them. Here, the principle of locality of reference can be applied to documentation (as well as to a configuration item and the configuration identification that describes it). The *Principle of Locality of Reference Documentation* (LoRD) [1] states that:

> The likelihood of keeping all or part of a software artifact consistent with any corresponding text that describes it is inversely proportional to the square of the *cognitive distance* between them.

A less verbose, less pompous description would be simply: *Out of sight, out of mind!*

Agile methodologies address artifact traceability by minimizing the number of different artifacts produced (especially non-code artifacts). In the extreme case, LoRD says that when the distance between two things is effectively zero, then there is nothing to trace. (Note that *cognitive distance* is basically the same thing Constantine and Lockwood mean by *visual distance* [2]..

For example, in an extreme programming (XP) project, how do I trace a story to its tests and vice-versa? An extremist might say:

> "Simple! Just flip over the index card that contains the user story. They're on the same physical artifact - problem solved because the pieces of information to trace were never split up into separate physical elements in the first place." Regarding tracing requirements (stories) to code changes ... if it's required (for whatever reason), might not a checkin or checkout comment simply identify the corresponding story? Seems to me that would do it for tracing to unit-tests too. So that takes care of the cards and code.

Now maybe not all Agile projects use index cards as the sole means of requirements capture (cards are often just an initial capture mechanism, with a tool being used to store and track/sort/report the requests for features and fixes.), but the basic idea is the same: *Placing the related (traceable) information in the same "storage" container minimizes the burden of maintaining linkages.*

LEVERAGING LOCALITY AT MULTIPLE LEVELS

This is the essence of applying the LoRD principle! Ideas such as "Literate Programming" [4] and the ability to declare variables in C++ and Java just before their use (instead of "up front" at the very beginning) are all based on this same principle of locality of reference! Other applications of LoRD include:

- *The Document is in the Code* - Anyone who has done literate programming [3] or used javadoc or Perl POD or the embeddable documentation systems for Python or Ruby code should be familiar with this. Such systems have even been extended on occasion (e.g., with a "Doc-let") to cross-reference a use-case name or other "traceable entity" that is somehow auto-hyperlinked.
- *The Document* is *the Code* - (or "The Source Code is the Design") [4] where the names of variables and methods are so readable and clear, and the statements, structure and interface are so completely and clearly intention revealing as to make detailed design documentation and even detailed code comments unnecessary

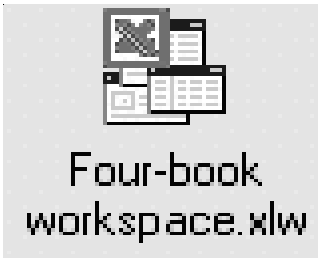
- *User Guide as Requirements Spec* - For those whose teams are responsible for the end-user documentation as well as the development, the practice of using the user guide as an initial draft of requirements and evolving it into the actual user documentation has been a tried and true practice over the years
- *Interface/Implementation Colocation* - in C/C++, many would put their corresponding header files and source file "pairs" in the same source directory (or at least appear that way, even though they might be physically stored in different locations for reasons of build-time performance). Ada practitioners often did the same with package specs and their bodies. Languages like Java and Smalltalk go one better by not requiring them in separate files in the first place.
- *README per Directory* - many a project source tree has used a file named "README" in each of its directories and subdirectories to give a quick overview of its contents. The collection of README files in the source tree effectively replaced the need for a document describing the same information. Such READMEs can even be auto-generated/appended (from a Make/Ant file) from the initial contents of each of its files.
- *Interspersed block comments* - even for those agilists who do resort to comments, many eschew the practice of putting all/most of the commentary for a method at the very beginning. Instead, the method-comment might contain only interface information and a brief "headline" description or paragraph; any necessary gory details would appear in block comments immediately before the

The Four Ways To Organize A User Interface

Martha Lindeman, Ph.D.

A system's user-interface is where a user interacts with the system to complete tasks and achieve goals. Thus developers tend to organize a user interface in terms of the tasks done by users. However, "Tasks" comprise only one of four possible types of organization for a user interface. The other three types of organization are "Time," "Tools" and "Things." At first glance, these may seem very similar, but the distinctions among them can yield very different user interfaces containing the same functionality. Current user interfaces tend to incorporate more than one type of organization, and understanding the differences among the four types can help create better user interfaces.

A true "Tasks" organization contains multiple workspaces, with each workspace customized for one or more specific tasks. The simplest example of a "tasks" organization is the "Save Workspace ..." command in Microsoft Excel. With this command, a user can save information about all open workbooks, such as their locations, window sizes, and screen positions in one .xlw file, such as:



In this example, the workspace contains four Excel workbooks: Book 1, Book 2, Book 3 and Book 4. A user may change the size and position of the Book 1 display when using it by itself, but opening the .xlw file will display the size and position of all four books as originally saved in the workspace. The workspace does update the display to include the saved contents of the workbooks so that the information is current.

A large, complex system may contain multiple default workspaces for some or all of the tasks supported by the functionality. Allowing customization of the default workspaces, and/or creation of new workspaces, provides flexibility so that the user interface can change as the users' tasks change. The provision of workspaces allows users who always do specific tasks to start and resume their work with a minimum of keystrokes.

For tasks that are not clearly defined, a "Tools" or "Things" organization may be the best organization for a user interface. A Tools organization normally has one fundamental type of user object and many tools that can be used to operate on that object. For example, Painter and Adobe Illustrator display a canvas or artboard surrounded by palettes containing many different types of tools. Thus a primary indicator of a Tools organization is frequent use of "tool palettes" or "toolbars" that may be predefined or user customized.

In a Tools organization, the user interacts with a fundamental object via some type of software tool (e.g., using a 'pen' to draw on the canvas). In a "Things" organization the user operates directly on the fundamental object(s) without an intermediary. For example, the user types characters into a document or drags-and-drops a document icon onto a printer icon. Excel provides workspaces, but Excel itself is organized as a "Things" user interface with workbooks, worksheets, rows, columns and cells as the primary things. Tools, such as the Chart Tool, are available but they are not the primary focus of the user-interface organization. Unless shown in the toolbar, tools are hidden under second-level menus, such as to insert objects or do data analysis.

The final type of organization is Time, which focuses on software control of one or more real-time processes. For example, a user interface might show the flows of liquids through a refinery. There is no single 'thing' in the refinery because the crude oil is transformed into numerous other 'things,' such as heating oil and gasoline. The process is a pre-defined set of possible states and changes, and the users control, trouble-shoot, maintain and supervise the on-going operations.

To successfully control a large-scale process, users must understand the 'global' (e.g., entire refinery or nuclear reactor) consequences of local actions (e.g., shutting a valve to change the flow of coolant). In these types of situations, a user interface focused on local 'things' rather than the Time-based global picture may have disastrous results (e.g., nuclear-reactor meltdown), because users are too overwhelmed to do the requiredglobal integration. When the time-based consequences are visibly supported by the user interface, the users can devote more of their cognition to trouble-shooting and decision-making.

From a user's perspective, each of the four types of organization requires creating a somewhat different type of mental model. When the four types of organization are mixed together within one user interface, either the developer or the user will have to integrate the models. For a developer, creating an 'intuitive' user interface means doing the integration in a way that is consistent with, or at least builds upon, the users' existing mental models.

Agile Distributed Teams

Raghu Misra

NTRODUCTION

I have been using Agile methodologies for over 5 years now. I was introduced to them while working with Mr. Marshall Gibbs (currently CIO of Information Resources, Inc.) in 1998. So, at the outset I would like to dedicate this article to him.

When I was introduced to agile methodologies, it was a cool thing and was very different from most of the way projects were handled in large companies. Since we were involved in product development we had the need and the luxury to be innovative and try different approaches to the way we delivered software products. For us, as with any product development firm, time is of the essence and generating demonstrable functionality as quickly as we can was of highest importance.

SCRUM AT SHIPXPRESS

Agile works best when project sponsor/owner and senior team members believe in it and the capabilities of the team. Thankfully it was an easy sale at ShipXpress where we have been successfully using it since our inception. We are regularly closing new customers using Agile methodologies. And I am proud to mention here that we got few customers primarily because of Scrum. We are so excited about Scrum that we are adopting the philosophy in our Sales and Marketing processes.

Scrum is a easy to sell to Senior Management of the customers too – primarily because:

- It increases the transparency of the development and delivery processes
- Customers get to steer priorities / deliverables and control the development and delivery process
- Customers get immediate benefits from immediate deliveries

Agile methodologies are perfect for small companies with low budgets (since there are no funds to spend on the whole vision anyway. These companies are forced to produce salable functionality very soon without wasting too much money or time.

DISTRIBUTED TEAMS A REALITY

As everybody already know by now, distributed teams are a reality. I would like to define a distributed team as a team where the members are separated geographically and are operating in different time zones. A distributed team is not necessarily a team that is both onsite and offshore.

Earlier whenever the teams were distributed – the activities were split across the locations. There used to be a Onsite team which was in-charge of all customer interaction (requirements gathering etc.) and prepare the functional specifications for the application / system being built. These specs used to be sent to offsite teams for coding and delivery to the onsite team for implementation.

In some of my projects the Product Owner was remote (customer site), QA was remote (offshore), UAT was remote (customer site) and core development was happening in Jacksonville, FL from where all the activities were coordinated.

I completely agree with Mr. Martin Fowler's view about separating functionality and not activity while having distributed teams. I have been following this philosophy for quite a while in many of my projects with great success. This will makes life easier in terms of dependencies across the team members in terms of communications and build level dependencies. Each sub-team can concentrate on building demonstrable functionality as if it is operating in a island (as far as this piece of functionality is concerned).

IMPORTANCE OF TOOLS

I cannot stress enough the importance of having the right tools accessible to the team members so that they can be productive in a distributed development and delivery environment. Tools like Groove, Intranets.com, MS SharePoint, IM, e-mail, web and video conferencing etc. are very valuable and effective for communication with each other and keeping everybody on the same page.

Luckily, I learned early in my career that documentation by itself does not generate any Business Value but it is the system(s) that get built that have value. People pay for functionality and not pages of documentation.

The development team definitely loves Scrum because there is less documentation to be done. In general the techies hate documentation with a passion (including myself), but it is necessary to have some kind of documentation which is adequate to give a decent idea of what the functionality / code / test cases etc are about without looking at the code itself. One very positive thing I have noticed with distributed teams is there is documentation generated automatically in the form of e-mails or word documents or IM chat messages while we are communicating with the guys who are part of the team.

DOES 24-HOUR DEVELOPMENT WORK?

I should say, 24 hour dev works in *some cases* - like bug fixes, small / trivial updates and Testing / QA etc. There has been a lot written about advantages and dis-advantages of time zone differences (primarily between USA and India). The dis-advantages are easy to overcome by having a sufficient enough overlap time between the teams here in the USA and in India. The big advantage is having a team that will work like a 2nd / 3rd shift of USA business hours. This will work under the assumption that everyone (USA and Indian team) is on the same page with the existing code and what needs to be done. Sometimes you need someone awake here during the wee hours to answer the questions of the offshore team. This has worked for us many times where-in we impressed our customers with new functionality as well as bug fixes updated to the production environment overnight. But, we don't operate on this notion on a daily basis. We do this only to handle some urgent cases like new reports / functionality for demos, bug fixes etc.

Scrum Development Vs. Scrum delivery

Do not under estimate the Culture shift that is required for end users too – end users in a large stock brokering house where I was consulting CIO did not want even bi-monthly releases. They preferred one big bang delivery of the entire application after 9 months to a year. The reason being end user training time, disruption to end users during the implementation process, the requirement to use a very hardened application (since it was mission critical) which is like a work-horse which ideally does not break and does not change.

Scrum / Agile make great sense for ASPs / Hosted applications and not necessarily where installation, configuration and legacy integration needs to be done at customer location(s) etc. I have seen almost all IT teams at our customers to ask Big Bang (One Time) implementation whenever there is any kind of integration with legacy applications in involved. Its Ok even if the software is intelligent enough to pick new versions/updates from a central location and auto update itself (something like a Java application using Java Web Start or .Net's Web Forms etc.).

WEEKLY / BI-WEEKLY DELIVERIES VS 30 DAY DELIVERIES

The key to short delivery cycles is to get the base architecture in place first - so that additional functionality is bolted on as it gets ready. We have delivered additional functionality in weeks over and over again with great success in a distributed team environment.

WHERE DISTRIBUTED TEAMS WORK AND DON'T WORK

Communication is the key: Can the remote resource take their cue from an explanation which is a 1-line bullet point or do they need a 10-line explanation or even a 5-page explanation? It helps a lot when the team members have worked together on earlier projects too. The better the understanding between the team members, the better the delivery quality and time. Its also important that all the team members are aware of the Agile methodologies and are completely sold on the concept.

Everybody should understand that Agile / Scrum is a constant pressure environment as the delivery dead lines are on a very frequent (monthly or shorter) basis. This can be compared to the education systems in India and the USA. Indian education system is Bing Bang. There will be examinations at the end of the year. Ifthe student studies well in the last few months he/she can easily get good grades and move on. In contrast Agile/Scrum are like the education system in the USA where the student is graded throughout the year at regular short intervals.

Also, projects which are componentize-able are ideal for distributed teams. If the project is being delivered using OO practices its ideal as opposed to Mainframe type applications.

CLOSING THOUGHTS

Distributed Teams are a reality and are here to stay. Agile techniques make them more viable because of the inbuilt risk management (because of visibility to project status) capabilities and flexibility to change course / functionality as the business environment changes. Currently people are offshoring for commercial, cost cutting reasons. As people who are already who are already doing offshoring will tell you, traditional offshoring comes with lots of process and infrastructure overheads. The trick is to continue to use offshore/near-shore teams for what they are good for (cost cutting and 24 hr development and support) while using Agile techniques and methodologies.

ABOUT THE AUTHOR

Raghu Misra is an experienced IT / product strategist and a Certified Scrum Master. He is currently the Co-Founder & CTO of Shipxpress (a supply chain solutions company), based in Jacksonville Beach, FL. He can be reached via e-mail at raghu@shipxpress.com.

Agile Project Management
Kent McDonald

I responded to Ken Schwaber's request for editors for the Agile Times in mid November and volunteered to be an editor for a section on Agile Project Management. My thought was this would be a great way to get involved with the Agile Alliance, and to put some purpose to my learning efforts about Agile Software Development.

My contributions to this section are under the title the Pragmatic Project Leader. I chose that title because in my opinion project leadership is primarily a matter of using the most appropriate project management tools for a particular project. In other words, do what works for your particular environment. The columns under this title will study the role of project management and the practices that agile approaches introduce to the traditional project management tool set. If you have some thoughts about Agile Project Leadership or Agile Project Management and would like to submit an article for this section in a future issue of Agile Times, please send me an email at kent@madsax.com.

The Pragmatic Project Leader
Kent McDonald

INTRODUCTION

Over the past several months I have pondered the question that serves as the title of this article. There is a great deal of personal interest in this question and the corresponding answer because my role on software development projects is that of analyst and project manager. After reading the various Agile mailing groups, it seems that most of the focus of Agile approaches has been on the developers and practices that help them be more effective developing software, and for good reason. Unfortunately, an unintended result of this focus is that, until recently, the role of project management in Agile approaches has been marginalized. This marginalization is apparent in the Agile Manifesto where most of the items on the right side of each statement, which are important but favored less than those on the left, are items typically equated with traditional project management.

With that in mind, I believe there is still definitely room in Agile software development for project management. The key thing to understand is that project managers find themselves taking on different responsibilities and using different practices in Agile projects when compared to traditional projects. This is

key for the concept of the pragmatic project leader because the vast majority of projects exist in an environment that calls for a combination of tools from traditional project management and the practices introduced when managing Agile projects.

In this article, I examine how the values of the Agile Manifesto impact the role of project management and provide pragmatic project leaders with new practices for leading Agile, and not so Agile, projects.

INDIVIDUALS AND INTERACTIONS OVER PROCESSES AND TOOLS

Agile approaches are based on the assumption that software development is a highly creative and intellectual exercise that does not always follow a predictable path, due to the large number of ways that the same problem can be solved through software. Because of this assumption, Agile approaches stress the importance of the people on the project team and the communication between them as critical to the success of the software development project. Instead of relying on a manual full of processes, Agile methods focus on principles and a small set of rules that project teams can use to guide them in development of software. This leaves the project team to focus on the core activities of solving the business problem instead of wading through the myriad of processes that do not always provide business value.

Project managers find that instead of managing the team to make sure that processes are followed, they make sure that the people on the project have the appropriate knowledge and skills to complete the project. They take on the role of a visionary, making sure that the team stays focused on the ultimate goal of the project. They take on the role of roadblock remover, finding out what is preventing the project team from focusing on the right activities and removing those roadblocks, which in some cases may be the processes that the project manager so faithfully upheld. They may even find themselves taking on roles usually held by other members of the team, such as analyst, architect, developer, or tester depending on the needs of the project. These new roles arise because the project manager is not spending their time managing the variety of processes and instead can focus on what needs to happen to build software that provides value to the customer.

One of the key practices that allow the project manager to focus on the development effort is the use of simple tools that are appropriate for the task at hand. The tools mentioned in the Agile Manifesto are typically thought to refer to development tools, but I think it could apply to project management tools as well. There is a widely held belief that you have to build your project plan in Microsoft Project or a similar project planning tool. I am of the opinion that a plan in Microsoft Project can often be overkill. If all you need to track is who is doing what and when it needs to be done, that tracking can be done by simple tools such as charts on a wall or a spreadsheet. Status reports are important as well, but it has been my experience that many status reports provide the wrong information, and often too much of it. Status reporting from the team to the project manager is much better communicated verbally in quick daily discussions, such as the Daily Scrum utilized by the Scrum methodology. Status reports to those outside the team should focus on what the team has accomplished, what the team plans to do next, and what issues are getting in the way of the team that the audience can help out with. A caveat with that last item is that the audience should already have been notified about any issues upon which they can have an impact as soon as they became apparent.

WORKING SOFTWARE OVER COMPREHENSIVE DOCUMENTATION

Traditional project managers are familiar with heavily documentation focused processes. These processes can feature phases where the main output is documentation with no actual work done building software. This reliance on documentation is due to the number of handoffs between different groups of people with different specializations working on the same project. Projects also produce a great deal of documentation for the purpose of communicating status or recording decisions for future reference.

Project managers in Agile projects reduce the reliance on documentation that is handed back and forth between the project members and instead instill an environment that supports and encourages face to face communication between the team members. One way they do this is eliminate the specialization that occurs on projects and to eliminate the phased approach to software development. Scott Ambler introduced the concept of the generalizing specialist (www.Agilemodeling.com/essays/generalizingSpecialists.htm) that is able to perform several different pieces of the software development puzzle. By having a project team made up of individuals who can perform various tasks, the need for handoffs is reduced, as the same person or group of people can discuss requirements with users, establish a design based on those requirements, and develop the code to implement those requirements.

Another difference in the way project managers operate in Agile projects is the manner in which the project is planned and progress is measured. Feature Driven Development (FDD) and SCRUM encourage organizing the project plan based on features instead of tasks. As a result, progress is tracked based on what features have been coded, tested, and are working as the user had expected rather than on which documents have been written, reviewed, and signed off. At the end of the day, the purpose of the project is to produce a software application, not binders full of documentation, so the project manager should measure progress based on what matters.

CUSTOMER COLLABORATION OVER CONTRACT NEGOTIATION

Above I stated that project managers need to foster an environment that encourages face to face communication among the team members, but I left out the phrase "and customers too" for a reason. Customers should be part of the team. Agile methods stress the importance of involving the customer throughout the life the project. There are many reasons for this, but one of the more important includes shortening the feedback cycle between developed code and customer review. The sooner a customer is able to review the software, the sooner the customer can provide direction for continued development. Another reason to encourage continuous customer involvement is that because requirements are not always clear at the beginning of a project. Including the customer as part of the team minimizes confusion or misunderstandings when discussing the requirements of the application.

Perhaps the most important aspect of this value from the Agile Manifesto is the role of trust on a project. One of the reasons for all of the approvals and sign offs that traditional software development methodologies prescribe is because there is an environment of distrust between the customer and the development team. The customer does not trust that the development team will build what they actually want, and the development team does not trust the customer to tell them everything they need to know in order to develop the right system. This mistrust is bred mostly because of a lack of true communication. I have been involved in projects where the development team upon discovering a problem spent a great deal of time discussing what to tell the customer, if they told them anything at all. After having lived through some of those projects, my practice now is to tell the customer the truth, as soon as possible. I have always found that the truth is much easier to remember, and bad news does not get any better with age.

The best way to ensure that the customer and developers feel part of the same team is to have all of them sit together while they are working on the project. While this is an ideal goal, it vary rarely happens in practice. In most cases the project manager has to take an active role in establishing clean lines of communication amongst all the team members, whether they be developers, analysts, or customers. These lines of communication need to focus on face to face conversations whenever possible. The face to face aspect is very important because it is the best way to ensure that both parties in a conversation understand each other. Emai lis a good tool to use for communication, especially when there is a need to

relay information to a number of people, or when there is a need to maintain that information for future use. Unfortunately email can often lead to miscommunication and misunderstandings and should not be used as the primary communication mechanism between project team members.

RESPONDING TO CHANGE OVER FOLLOWING A PLAN

This value is perhaps the biggest departure from typical project management philosophy. The main role of a project manager is to manage the project according to the plan and to manage change as much as possible. The importance placed on this role is based on the assumption that the cost of change increases as the project progresses, so change should occur as early in the project as possible. Ironically, the response to that assumption actually validates the assumption. The more effort that a project team puts forth to determine the requirements and design the application up front, the higher the "sunk cost" incurred by the project, and therefore the more expensive a change becomes.

Software is often relied upon to help solve business problems, so it is victim of the uncertainty and quick pace inherent in the business world of today. This rapid change often throws carefully laid plans into disarray when the various unexpected events arise. Agile software development approaches accept that change is inevitable and look for ways to reduce the cost of change so that it can be harnessed for the benefit of the development project. One example of how responding to change is beneficial to a project are the changes that occur as the team learns more about the problem domain during the development process. In my experience, the act of designing, developing, and testing the solution often leads to a clearer understanding of the problem and solution, which can often result in changes to the software. Some project managers incorrectly label this "scope creep," when it is just a clearer understanding of what the project is trying to accomplish.

To adapt to Agile approaches, project managers change their outlook on the nature of planning and change. Planning becomes a matter of leading a project based on what is being developed, instead of how it is developed. The project manager's outlook on change goes from something that should be controlled to something that should be harnessed for the benefit of the project. This new outlook on the nature of planning and change in a project allows the project manager to be more flexible in the manner in which they lead the project and enables them to better respond to the customer's requirements.

CONCLUSION

Agile approaches propose a small set of simple rules that provide the project team with boundaries, but allows them to do their best work and focus on providing real value to the customer. The focus of the project falls on work by the developers to finish the software, instead of the work by the project manager to ensure that processes are followed, plans are maintained, and changes are controlled.

There is room for a project manager in Agile software development, but that project manager my find themselves doing things differently than they have in the past. The project manager works for the project team instead of the project team working for them. The project manager does whatever they can to make sure that the team is able to focus on delivering working software; this includes quickly eliminating roadblocks that stand in the team's way and keeping the team isolated from political battles within the organization. The project manager fosters an environment that supports communication, collaboration and honesty amongst the team. Finally, the project manager finds ways that they can contribute to the team in other ways, such as doing analysis, design, development, or testing. By playing a more active role in the details of the project, the project manager has a better understanding of the project and is better positioned to help the team make decisions and deal with the constant change to which the team is responding.

This article focuses on project management practices useful for Agile projects, but not all project managers have the advantage of working in an environment that fully supports Agile development processes. The Pragmatic Project Leader picks the practices that fit best with their environment and utilizes them to do what works. Often this will be a combination of practices from Agile software development as well as traditional project management practices. Future articles in this series will look more specifically at the different project management knowledge areas to see how practices from Agile projects can be useful to pragmatic project leaders in a variety of different environments.

DSDM
Mike Griffiths

Event: Applications for DSDM Certification Close
Date: 20 February, 2004
Type of event: Exams
Location: UK
Organized by: DSDM Consortium
Contact: info@dsdm.org
More information: http://www.dsdm.org/en/training/accreditation.asp

Event: DSDM Congress 2004: Business Performance, powered by DSDM
Date: 11 Mar 2004 until 12 Mar 2004
Type of event: Conference
Location: Amsterdam, Holland
Organized by: DSDM Benelux
Contact: conference@dsdm.org
More information: http://www.dsdm.org/en/benelux2004/information.asp

Event: London Agile Conference 2004
Date: Autumn 2004
Type of event: Conference
Location: London, UK
Organized by: DSDM Consortium

DSDM Ten Years On: RAD Relic Or Agile Advocate?
Mike Griffiths

As January 2004 marks the 10th anniversary of the formation of the DSDM Consortium, we examine the role and relevance of DSDM today.

SOME HISTORY

In the early 1990's, interest in customer focussed, iterative development began gathering momentum in the UK after Boehm (1986), Gilb (1988) and Martin (1991) outlined many of the basics approaches. While these publications gave pointers towards how to make it work, none provided the total solution.

In January 1994, a group of 17 companies formed a consortium to study, define and promote a public-domain standard for rapid, iterative, customer focussed development. The consortium was comprised of a variety of organizations including large IT consultancies, tool vendors, and user organizations.

At the time of the consortium's formation, I was working as a team-lead at one of the founding companies, Data Sciences Ltd (a consulting company of 2000 staff later acquired by IBM Global Services). I vividly remember using the various DSDM defined techniques on early projects, including a few non-conventional approaches that did not make it into DSDM Version 1, published a year later. The main thing that struck me was the increased involvement of the customer throughout the lifecycle. It made perfect sense to engage the arbitrators of project success, but this tactic was contrary to existing beliefs that the IT staff knew how best to develop a system once some initial, sketchy requirements were gathered.

Despite learning as we went and making all manner of mistakes, the results were very clear. End users and project sponsors preferred the approach, they were now in control of the project's business direction and the resulting systems were more closely matched to the true business requirements rather than the initially stated (often flawed) requirements. While follow on projects frequently refined the DSDM techniques applied, no clients ever asked to go back to a non iterative/incremental approach to development. The principles and techniques associated with DSDM methodology became an integral part of how clients expected and wanted their systems to be developed.

RELIC OR RELEVANT?

Looking at the DSDM principles today, 10 years since their original definition, it is reassuring to see how well they align with current best practices. The original DSDM principles were:

1. Active user involvement is imperative
2. DSDM teams must be empowered to make decisions
3. The focus is on frequent delivery of products
4. Fitness for business purpose is the essential criterion for acceptance of deliverables
5. Iterative and incremental development is necessary to converge on anaccurate business solution
6. All changes during development are reversible
7. Requirements are baselined at a high level
8. Testing is integrated throughout the lifecycle
9. A collaborative and co-operative approach between all stakeholders is essential

These principles align very well with the Agile Manifesto Principles as shown in the table below. Actually, this close alignment is not just good fortune or testament to some "great truth". One of the attendees of the February 2001 Snowbird, Utah meeting that defined the Agile Manifesto was Arie van Bennekum, a member of the DSDM Consortium. Arie proposed the DSDM principles as a starting point for discussions and ideas for the Agile Manifesto Principles. These ideas, along with many other great inputs, led to the definition of the agile principles so familiar to us today.

STAYING CURRENT

So, while it would seem that DSDM is still applicable and current in 2004, some people have reservations and are put off by its age or association to RAD practices. In an industry where technology advances so quickly, it is easy to see why new adopters look to the latest approaches as embodiments of today's best practice. Why adopt a method that dates back to RAD prototypes when there is a plethora of more recent alternatives? DSDM distills the practical learning's of its membership in an iterative fashion to meet the challenges of software development for a myriad of business scenarios.

DSDM is continually evolving and adopting best practices, it is also no stranger to fighting stereotyped perceptions. With the release of DSDM version 3 in 1996, the terms "RAD" and "JAD" were dropped and the term "facilitated workshops" introduced to describe the high-bandwidth, face-to-face communica

Agile Manifesto Principles	DSDM Principles
1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.	4. Fitness for business purpose is the essential criterion for acceptance of deliverables. 3. The focus is on frequent delivery of products.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.	6. All changes during development are reversible.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale.	5. Iterative and incremental development is necessary to converge on an accurate business solution.
4. Business people and developers must work together daily throughout the project.	1. Active user involvement is imperative 9. A collaborative and co-operative approach between all stakeholders is essential.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.	2. DSDM teams must be empowered to make decisions.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.	[DSDM makes extensive use of Workshops.]
7. Working software is the primary measure of progress.	3. The focus is on frequent delivery of products.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.	
9. Continuous attention to technical excellence and good design enhances agility.	8. Testing is integrated throughout the lifecycle.
10. Simplicity--the art of maximizing the amount of work not done--is essential.	4. Fitness for business purpose is the essential criterion for acceptance of deliverables.
11. The best architectures, requirements, and designs emerge from self-organizing teams.	2. DSDM teams must be empowered to make decisions.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts behavior accordingly.	

tions approach preferred for collaboration. The evolution continues, the consortium is now driven by over 1000 members and they are keen to see new approaches incorporated.

DSDM version 4.2 released last year contained guidelines for using DSDM in conjunction with XP. DSDM remains a leading agile method in Europe; its adoption in North America has been slow but is now advancing more quickly. While the DSDM Consortium is driven by its members, it will continue to be a powerful advocate of agile best practices, who knows where the next 10 years will take it.

The Unpredictable Element: People
Barbara Roberts

Customers often ask me what kind of problems I have encountered on DSDM projects and do projects ever go wrong. Now I could give them a frighteningly positive impression about DSDM: superhuman team members calmly delivering projects to meet business requirements on the due date with little mention of the troubles encountered. However, the key to success with DSDM is user involvement and as we all know people are unique individuals and all to often unpredictable. So, with reckless abandon, and little thought about the impact on my career as a DSDM consultant, I have decided to come clean, and reveal a few of my own DSDM people nightmares, partly to capture what I've learnt, and partly as therapy!

THE DSDM FANATIC

I was running through the Suitability Filter to evaluate potential DSDM projects with a Project Manager. This Project Manager was wildly enthusiastic about DSDM and determined to use it regardless. The reply to every question about his project was "Yes" and "Low" risk – the perfect project. Fortunately, I'd done my own preparation on the project beforehand and recognized that this bore no relation to the same project I'd read up on. I was able to follow up with more searching questions and I explained that you don't need to have a clean sheet Suitability Filter to use DSDM. The Suitability Filter identifies potential risks that must be addressed to decide whether they are manageable or not, before making the decision 'Does DSDM add value or does it increase the risks?'

Learning: Beware the fanatic – over enthusiasm is as bad as a lack of enthusiasm

THE WORKSHOP ABSENTEE

We were planning a series of key workshops, and achieved buy-in from all departments except one. Historically, this department had been very powerful within the organization, and they informed us they were all far too busy to attend, but would expect to see the workshop results and then add their comments or veto the results, as appropriate. With the approval of our sponsor, we introduced and publicized a special rule for this initial workshop, "Absence = Silence = Assent" – i.e., this is your chance to be heard, use it or lose it. They didn't attend, assuming (wrongly) that exceptions would be made for them, and were horrified to see the decisions made on their behalf. After that they always made sure they had their best representative at every subsequent workshop.

Learning: Do not allow a small minority to stop the collaborative, co-operative process.

THE MISSING AMBASSADOR USER

This business-critical project (aren't all DSDM projects business critical?) had a full-time Ambassador User assigned. Two months into the project, he became very ill and was signed off for several weeks. It was not the sort of illness where we could give him a mobile phone, a bedside table, laptop and modem (although we did discuss this as an option). However we could not afford to put the project on hold for several weeks either. As a short-term option to keep the project moving for-ward, we decided to bring in

the Visionary and give him plenty of support from the Advisors. We had to be very firm that the Visionary could not rework anything previously approved by the Ambassador, and once this was clear, our improvised solution worked well. However, when our Ambassador returned from his sick bed, we then ran into problems because he felt that the Visionary's solutions did not fit his view. To avoid re-working the last few weeks' work, we used MoSCoW to prioritize his concerns and agreed that we would only revisit those that were actually wrong – i.e. that must be changed.

Learning: If circumstances force staff changes, manage the handovers carefully and use DSDM controls.controls too risky. Empowerment and rapid decision-making would involve a major culture change (not Berserk Techies). I was rolling DSDM out into an organization where technical excellence was the main (only?) criterion for recruitment and promotion. Developers with the ability or willingness to communicate with other humans were rare, and commercial awareness was minimal. Successful DSDM in this type of culture always presents a major challenge, and part of my rollout plan was raising the company's awareness of the wider impact of DSDM - on recruitment profiles, appraisals, grading and promotion. DSDM is not just about IT – it affects the whole culture. The ideal is to select team members for their soft skills as well as IT ability – skills like active listening, team working, time management, commercial awareness, etc. Training can help, but it can be extremely difficult to teach soft skills to staff that feel these are unnecessary. My last resort was to veto any staff that seemed totally unsuitable to sit near a customer – it would be unkind to both parties!

Learning: For DSDM team members, communication and soft skills are paramount.

DSDM BY COMMITTEE

On this project, DSDM had been previously confirmed as the chosen approach, but problems arose when I tried to agree user roles and representatives. I outlined the Ambassador User responsibilities, and received an email containing about 35 names. There was also a postscript explaining these were only the first thoughts, and that probably other names would need to be added. I re-stressed the responsibilities of the role (especially decision-making and business empowerment) and asked for a list reduced to single figures (preferably one or two). A second email cancelled all the names on the first list and offered a single name – the Department Chief Executive – the only person with the authority to make the sorts of decisions we needed. At this point it was very clear that getting the necessary time commitment and close involvement would be impossible, and the use of full DSDM would probably be achievable overnight). In order to maintain the sanity of the team, we decided to use a partial DSDM approach.

Learning: Without the supporting culture, a full DSDM approach may be impossible

THE MAIN LESSONS

- Always do your homework and don't always believe everything you're told
- Negotiate but be prepared to stand firm when it is important
- Always, always, always contract for the minimum user involvement (even on internal projects)
- Check up-front how the Principles will be applied. Discuss this with the customer, to check their understanding and to get their agreement.
- Define very early on how empowerment will work in practice (preferably as part of the Project Terms of Reference)
- Remember that use of DSDM is a sliding scale – all DSDM, some DSDM, a few DSDM techniques – and it can change during the life of a project

In a previous life within Xansa I led the DSDM Consultancy Practice where one of the service lines on offer was to troubleshoot/health check DSDM projects. These projects were either undertaken by in-house departments or by software houses. By far the most common reason for failure was an ineffective Business Study. The urge to commence prototyping/ coding without undertaking the Business Study proved impossible to resist for many. So how do organisations new to DSDM ensure that the Business Study is carried out effectively? The bottom line is to ascertain whether the objectives for the Business Study have been truly satisfied. In order to find out we must demonstrate answers to the following questions:

1. Do we have an agreed scope of the business processes to be supported by this project, and the benefits expected from it?
2. Do we have the 'names in the frames?' For instance, do we know who does what and when they do it? And how do we know we have finished a prototype? Can we still use DSDM? If not, why not? And what are the associated risks?
3. What technical infrastructure do we need to:
    - Build it?
    - Test it? (all aspects of testing!)
    - Deploy it?
    - Support it? – both sustain and optimize

All of the above need to be designed to meet the 'must have' non-functional requirements such as security, performance, maintainability and so on.

SHORTCUTS DON'T PAY

Projects and project teams, which take short cuts or pay lip service do so at their peril and risk the wrath of their organisation and customers. In theory this stage should be easy to do and applying the Business Study Products Quality Criteria is the way to check that it is being done properly. Please remember that quality criteria are not there just to apply when reviewing a product, but are needed when planning the creation of that product.

I know it may be seen as boring to do it according to the book, but so is undercoating before painting. It's only common sense but how common is sense?

It always amazes me when I review DSDM Business Studies that we seem to ignore things that have been proved to work, like problem analysis and determining the cause and effect. I have found many projects don't challenge users who articulate the requirements except in regard to prioritisation. In effect the users are coming with a solution defined in business terms and then that is translated into a technical requirement. In my experience it has often been beneficial to the project when we have asked the user to take a step back and define the problem (this is not as easy as it sounds and requires a strong facilitator).

When we do this the whole project team then understands the problem we are attempting to address. On more than one occasion changes to business processes have solved the problem rather than an IT development that the users were insistent upon.

When a Business Study is undertaken effectively the main benefit is that the tough questions are asked early in the project lifecycle. Questions which during the traditional approach are left until much later, such as, how are we going to test and how are we going to deploy it? This quite simply allows us more time to resolve the issues, and reduces the potential for unpleasant surprises later in the lifecycle. I mentioned earlier about prioritisation during the Business Study workshops. Early signs of dissatisfaction can become apparent, as prioritizing can be a very stressful time. This is nearly always brought about by the lack of a quantitative business case. It is difficult to achieve true prioritisation when requirements are prioritised on politics, or seniority or just good old manipulation.

When I trained in DSDM I always said that a 'must have' requirement must have a direct bearing on the business case of the project. In other words, if a piece of functionality could not be delivered, the project's return on investment would be diluted. Then and only then could the requirement be deemed a 'must have.' Ialways ensure that the rules on prioritisation are agreed prior to the requirements workshops and if a quantitative business case doesn't exist, I will create one with the Visionary either during the Feasibility Study or at the start of the Business Study.

GETTING DOWN TO TECHNICALITIES

So you now have the business side of the project sorted out but what about the technical side? What about the delivery? You know what you want and who is going to do it, but you don't know how you are going to do it or when you are going to do it by.

Which takes us to the System Architecture Definition (remember good design principles here). Recall that in the Business Study we move from estimate to quote. We provide cost and delivery dates for each increment so it's essential to undertake real design work. It is also important to remember that the quote you give should preferably include all associated costs including such items as hardware and license costs.

I strongly recommend that a separate Test Strategy and Configuration Management Strategy are produced and that the Implementation Strategy and Development Risk Analysis Report are brought forward from the Functional Model Iteration in order to provide a truly effective foundation for the rest of the project. This leaves the Development Plan, I believe it is important that all stakeholders underwrite the plan and that the plan reflects activities and responsibilities of all involved in the project. On many occasions I have seen this co-operative and collaborative approach just reflect the contribution of IT! So prepare your surface properly with an undercoat and the top coat will go on much more smoothly!

Agile Europe

Marco Abis

Many things are happened in Europe in the last few months and many others will in the next ones. Beside the growning of single usergroups meetings I think five events in particular can be taken as an indication of the Old Continent Agile environment.

XP Day Benelux 2003 on 21th, November in Breda, The Netherlands: a one day conference about all aspects of Extreme Programming and other agile software development methods like DSDM, Scrum, Feature-driven development, and Crystal with 20 sessions (!) and 24 presenters (!!).Session presentations and workshop transcripts are available at http://www.xpday.net.

XP Day London 2003 on 1st and 2nd December: the third edition of the two days international conference for project managers, developers, and testers with 23 sessions and speakers from the UK, USA, and Europe in general. All the presentations are available at http://www.xpday.org

From Agile to Lean: the day after XPDay London another great meeting, this time in Scotland, with Mary Poppendieck, Martin Fowler and Ammar Kaka as keynote speakers attracted 75 people! A nice report can be read at http://www.agile-scotland.org/

And more! Between April 20th and April 25th, 2004 in beautiful Vienna, Austria there will be the first Scrum Gathering Europe, a grassroots conference developed for the IT community to promote research and to educate individuals about the Agile methodology known as Scrum. More info at http://gathering.scrumalliance.org/

Of course we cannot miss to remember that the Fifth International Conference on Extreme Programming and Agile Processes in Software Engineering (aka XP2004) will be held between June 6th and June 10th in Germany: http://www.xp2004.org/

Acceptance of Agile processes has required a bit of a leap of faith for most who manage software developers for various reasons. One that strikes me as extremely important stems from the need for Executives and Managers to quantify and qualify project status either to their senior management or investors.

The questions asked in the boardroom don't translate well into "Agile" terms and, as such, understanding between engineering and executives can fail. Although the bottom line improves over time, the day-to-day status is harder to visualize. When emphasis is properly placed on the people and product some amount of project status measurement may suffer. Process for the sake of process is a bad plan, but some means of measuring status, while leaving the technology talent to the art of building a software product, is critical to keeping a project funded and supported by Executives, Marketing and Sales.

A functioning Agile "Code Factory" monitors status by managers gathering data manually. They then create reports in terms of budget, schedule and feature that will be used by outside factoring groups. This data gathering takes up approximately 25% of a project manager's and 15% of a developer's time. These are expensive resources and are better suited for other tasks. When considered as a part of the overall cost of a project, this status collection is expensive, and serves limited purpose. This single fact is both a great reason to use Agile and the largest reason for not adopting the methods.

The balance between the need for a pure creative process in development and quick, accurate and objective reporting of status for management has been an impediment to acceptance of Agile in the software industry. The concept of Agile was created by developers for developers, without a strong consideration of management needs. Management overhead had become an unbearable burden for the creative staff to bear. When a process is developed without consideration of all sides of the bargain, a void is created. This void has made it harder for managers to sell Agile to their executive staff, as the justification of spending and staffing in an IT department becomes much more obscure. In all fairness it must be said that "Agile" is intended to shorten measured project intervals and be iterative in nature, with the intention of minimizing management overhead in a project. However, this places the burden of reporting onto the Project Manager, although it is still based on the opinion of technical staff. Every project status report is thus subjective, and may or may not be a true reflection of the actual status.

The question might be raised, how is project status tracked in an "Agile" development team. As the chart on the following page shows, project tracking might not be addressed directly in an "Agile" method. Standards of good project management, of course, apply. The problem with these precepts is that they are only a loose set of guidelines. Theories come and go, are applied differently by one manager to the next, but in the end it becomes a question of what gets measured, understood and communicated is based on the skill and perception of the PM. "Agile" shows a way for developers to write better code cheaper, but acceptance by executives and managers who pay for software development remains an issue that must be solved. (See chart below)

I believe that becoming "Agile" is the most promising path for the software industry and a method to measure progress without impacting the developers that is based in facts gathered from objective sources is the key to "Agile" gaining traction. It therefore becomes essential that we establish a set of standard measurements that can be collected objectively to determine actual and comparative status of a project, its' features and sub-features. As long as executives and managers believe "Agile" is the fad of the day, and does not take their needs into consideration, acceptance will be hard to obtain.

Agile Method	Project Management Or Tracking Method
Scrum	Scrum has a very mature project tracking method. Each Scrum team is responsible for building and maintaining a "Product Backlog" document. This backlog is a list of all features and technology used. A daily meeting during a Scrum "Sprint" at which any issues or blocks are discussed and resolved. The meetings and the Backlog are the foundation for tracking progress and status.
Dynamic Systems Development Method (DSDM)	DSDM has been use effectively since 1994. DSDM is highly iterative with minimal documentation. The feature set is at only the very highest level and is extremely fluid. There is no objectively defined method for tracking the progress of a project under DSDM, so it for the Project Manager to facilitate communication within the development team and with the end-user community.
Crystal Methods	Crystal is actually a family of methods. The core of the method is that the development process should be self-adaptive, i.e., the development team can promote or demote aspects of the method as the support or block the process. Crystal has no defined process for tracking the progress of the project other than subjective and informal meetings between the team.
Feature Driven Development (FDD)	FDD is characterized as being among the most lightweight of the Agile methods - and one of the most successful when properly scaled. FDD has no defined objective process for tracking the project. In FDD the Project Manager is given a great deal of autonomy to set priorities and schedules. The PM's knowledge of the status of the project is based on their relationship to the project team.
Lean Development	Lean Development is another very mature Agile method. It stresses strategic, long-term thinking and is organizationally driven. Lean Development places great emphasis on Risk Management, therefore the Project Manager gathers a great deal of information on a regular basis from the development team. It has a robust template oriented system for gathering metrics on project progress. The Project Manager spends much of the time gathering largely subjective opinions from the development team, with the single objective data point being what feature has actually been delivered.
Extreme Programming (XP)	XP is probably the most popular Agile method right now. It is heavily developer focused and has the lightest emphasis on documentation and project artifacts. XP is defined by an emphasis on group commitments to values and principles. Communication and feedback from the team tell the Project Manager the status of the Project. Other than working software produced, there is no objective standard for measuring project status.

develop a means to do those things within your work-group or company. No current set of standards, or means to gather standards is in vogue.

In my development group, we choose to measure information gathered from code versioning, bug tracking and project management software. We calculate a measure of progress toward a short duration, highly iterative goal contributing to a feature or sub-feature frequently, often more than once each day. We graph this progress and feed the data back to developers to help them understand when the "painting" will be finished, and to the executives to help them make wise choices about the business. We have become an "Agile" code factory without ever stating the fact, and since the production stays on schedule and budget we are measured as a success.

We have the advantage of being a small group of developers working on a common goal with relatively easy communication and interaction. So, can we translate our success into the need for a standard? My opinion is that we can. In one company that I studied, I found that they undertook over 28 thousand projects each year, some small and some large. They had adopted an "Agile" approach to developing code and had placed the burden of reporting status on the PMs. The staff was over 300 project managers, meaning each manager was responsible for more then 93 projects per year. They had no standard methods and expected each manager to decide how each project was to be managed. Each PM was burdened with a detailed status of all of their projects each week in a large status meeting that took an entire day. They gathered the information on Friday, using most of the day to gather it by speaking to each developer and record it, than on Monday, they build a status chart and a list of open issues and constraints. This all was used at the Monday afternoon meeting where status was gathered by their Directors who were expected to give a status of each project on Tuesday to the Senior Executives who could then hand down guidance. It appears to me that nearly1/10 of the developers time, 1/3 of the PMs time and ¼ of the Directors and VPs time was being misallocated. "Agile" seeks to remove the overhead, but in this case they had replaced the overhead with a burden almost as hard to carry. What is the cost of this method of accounting for status? In this organization with they had 10 VPs, 300 PMs and 4,500 developers. The total cost of time misdirected equals approximately $62 million per year or about 10-15% of the total IT budget. This significant amount mitigates the overall savings from using "Agile" processes. Application of a standard set of measured metrics directly affects the cost of gathering this data, and allows automation of some of the process needed to gather the required data. Using a standardized set of reports and charts, created as a part of a normal process can further reduce the overall costs. The real key to getting "Agile" into a development group is going to be minimizing the cost of generating the needed reports while maximizing the accuracy and objectivity of data provided to managers.

"Agile" minded Project Managers and Executives must start building an acceptable set of objective standards and the means to efficiently communicate the data collected to all parts of the company. Letting the creative development group be free to design, and build innovative software products while giving the Executives, Sales and Marketing the information needed to build a market for those products and effectively manage the overall process on-time and on-budget with the desired features.

"Agile" processes are the means to making innovative software products efficiently. Standard reporting of status is the means to manage funding, planning and selling those products. To have a successful software company both of these critical functions must be enabled.

In this first column, I'm trying out a different format. Martin Fowler gave a thought- provoking keynote talk at XP Day in London last month, and I want to pass it along to a wider audience. The following was written from my notes taken at the event, and OK'd by Martin afterward. I've added some commentary on the topic from myself and from a colleague. I'd be interested in your thoughts too.

Design In Agile Processes
Martin Fowler

I have a confession to make – I'm bored with Agile and XP.

In the mid 80's I got involved with software methodologies, but my main interest has always been design. I'm used to building things the engineering way, with two stages – design and build.

Software drawings (e.g. UML) correspond to electrical schematics, but software is not physical so it's not obvious how best to represent it in drawings. We use processes to create software from the drawings but you cannot really say which software process is better than another if they both produce working code.

In the 1890's scientific management ideas began, led to Taylorism, and eventually to Lean development. There are areas where a single process cannot be defined. For example in health care there are so many variations in the processes needed by doctors and nurses that you simply can't define one process for health care.

Steve McConnell is a thoughtful critic of Agile – I've had many talks with him. We each have totally opposite processes for writing a book. He will make a rough outline and then keep refining it into more and more detail over a period of 10 months or so. Then he'll spend maybe 3 months filling in all the prose, and it's complete. I just can't do it that way. My method is to simply start writing prose, and writing more and more of it. Eventually I organize it into chapters. We've both had many best-selling technical books. So which way is right? Both and neither.

Similarly with software development processes – we cannot measure our output well. Lines-of-code is meaningless. Our output could be called "Value the customer realizes". You cannot prove that one software technique is better than another. There is a similar problem with the medical system – you cannot prove that therapy A would benefit a given person better than therapy B. Once they're given therapy A, we can't turn back time and try therapy B in the same person under the same conditions. Therefore we must make our own judgment based on our own experience and that of colleagues.

A better question to ask is: What makes for a good software design? What is the place of design in the flow of software work? The Agile community does not separate design and build in programming; there is a very big overlap.

- You can have programming without design.
- You can have design without programming.
- But it's better to interleave the two.

As a manager how can you tell whether design has taken place? If you have an explicit design phase, ok – but is the design any good? If design and build are interleaved, a non-technical manager cannot tell whether design is happening. Early on, the issue of "programming without design" was addressed by the Software Engineering community by instituting a separate design phase with reviews held before the build phase could start. When design and build are interleaved as they are in agile development, it can appear from the outside to look like the old "programming without design". I think that's where some of the opposition to Agile is coming from.

SWEBOK presents one view of how things ought to be done. But there are many valid ways. The Smalltalk community blends design and build. The LISP community blends design and build. The Software Engineering community separates design from build. In the UNIX/ open source community, design and build are interleaved and evolutionary – you don't need to know the end goal. Eric Raymond has written a book "The Art of UNIX Programming" which records the story of a successful community rather than tell us all what we should do. We could all do with more examples of what has worked than with continuing to battle over methodologies.

Testing, Refactoring, and Continuous Integration are the practices XP has to ensure that design gets done even though there is no explicit design phase. These parctices are meant to ensure that design converges. The real reason that design converges is because someone on the team has the will and skill to make it happen. You've got to have the *active desire* to look at how the code is coming together *and* an active desire to do something about it. Where does the will come from? A demotivated team can't do it.

For non-technical managers working with Agile teams, how do you tell if design is happening? If no code is being thrown away, design is probably not happening. If the team is unhappy, design is probably not happening.

COMMENTARY FROM RON MORSICATO OF XP EMBEDDED:

Martin pointed out Testing, Refactoring, and Continuous Integration as being the practices that make XPers honest, design-wise. But you can add the Planning Game to that mix. You need to design if you're serious about estimating. Although for the most part you're decomposing requirements into specifications and specifications into tasks, it's still a hierarchical design activity that out of necessity will involve some degree of object oriented analysis. In fact, breaking down the design into smaller and smaller subtasks, allowing for shorter iterations, is the key to accurate estimation. It follows that if the software team is estimating accurately, it must be designing. This is another example of how the agile practices reinforce each other.

In a typical XP room there are design artifacts hanging all over the walls. This means that design in XP is meant to be visible. These things tend to be marked up, showing that they are living documents and design is happening with each iteration.

There are situations where it's proper to do some up-front design in XP, and not expect that all design will be done after the fact via refactoring. An example is when the project has multiple teams working in different locations. In this case it may be preferable to do up-front design for the interfaces between teams. This allows for more communication within teams than between teams. So the bulk of communication can be done in person. Granted, you lose something if the distributed teams don't talk as much but you're valuing personal communication over electronic communication.

COMMENTARY FROM NANCY VAN SCHOOENDERWOERT OF XP EMBEDDED:

Martin's thoughts reminded me of the extra things I was constantly doing to keep the design structure clean throughout my most recent XP project. I knew we weren't doing as well at testing as we should. Too often the tests weren't written before the code, and so developers would then be reluctant to bother writing them. We did fairly well on Continuous Integration. Refactoring is all too easy to postpone under the daily pressures of project work, and I was keenly aware of the ugly spots in our code that needed refactoring. There weren't a lot of them but they are always malignant. If I couldn't make time to clean them up, I was at least determined not to let more cruft build up on top of them.

The truth is that even if we were doing a perfect job on those XP practices, I probably would still do my "extras" to watch out for the design. These things included:

- Creating a high level diagram of the current software design, and how it should look at the end of the current iteration – just a single page sketch. It allowed developers to absorb the ideas at their own pace, and gave us a means to talk about design.
- When bugs were found during integration or later, I kept a log and wrote up a description of how the bug got created. This usually pointed either to lax practices or a design faux pas.
- When code comments weren't sufficient to document a complex area of the code, I created mini documents – one page explanations of the feature that leaves the details to the code and just covers enough for a developer to recall the design concept.
- Have a brief review of the design approach developers were using when estimating new stories.

Most members of my team were inexperienced in one or another skill, and they were eager to learn more about software design. The combination of practices worked well to help them learn without getting overwhelmed.

Is it possible for a team to follow XP's practices alone, and have a design fail to emerge from the activity? I cannot answer that question from my own experience. Maybe I should re-phrase the question: "...have a *good* design fail to emerge...". Martin seems to think so. I did extra things to ensure a good design. I might not have needed to if I had had a more seasoned team. But that confirms Martin's point: that it is individuals with the will and the skill who ultimately make sure good design happens.

Selling Agility To Senior Management
Scott Bogartz

As an "Agile Evangelist" in several different organizations, I have often found the most difficult challenge is to win the support of senior level management. To me, it is clear that the so called traditional processes are insufficient to handle the rapid pace of change in both the technical and business environment of today. However, for many other executives there is a perceived safety in following these approaches. In order to influence such executives I often need to reference other "objective" third party information so that it does not seem like I am simply presenting my own unsupported opinions.

Upon review of the first draft of this article, a colleague pointed out the contradiction in the use of the concepts of evangelism and objectivity in the preceding paragraph. He pointed out that evangelism of a particular process or technology often distracts us from our real (professional) purpose which is to deliver software that is useful to the businesses we serve. The very idea of such distraction is un-Agile at its core, so how can I be an "Agile Evangelist." It was difficult to reconcile the contradiction, but just as I was about to give up, I considered the values put forth in the Agile Manifesto. I 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"

I don't see adherence to these values as a conflict with serving my business. By being true to the first value alone, I cannot allow a process to distract me from delivering the best possible results to my business. I am comfortable evangelizing these values.

As a community we must sell our values at the senior management/executive level in order to truly succeed in our mission of "uncovering better ways of developing software". While bottom-up initiatives to introduce Agile practices are a good start, top-down support is vital to the successful mainstream adoption

of Agile values. Top-down support is required for a variety of reasons.

A primary reason for the need of top-down support is the impact that adopting Agile values has across the entire organization. Product management is one of the most heavily impacted areas, perhaps even more heavily than product development. In an Agile organization, product management has to accept much more accountability for the ultimate delivery of a product than in a traditional environment. Product management must be willing and able to make clear cut priority decisions, stay actively engaged with the development team and have the fortitude to actually release the product at some point. This requires a strong mandate from senior management. Downstream areas like publications, training, support and marketing are also impacted by the adoption of Agile values by the development group. These groups are used to having a pre-defined plan (specific feature set to be available on a specific date) to which they manage. With Agile values, the full feature set of a product may not be known until much later in the game, and this causes anxiety in the downstream areas. Agile activists observe that traditional approaches provide only a false sense of predictability to the downstream areas, so they really need to be more flexible regardless. However, the downstream areas must be strongly encouraged towards (and mentored in) a more priority driven, incremental approach. Again senior management is needed to provide such direction.

As an "Agile Evangelist" in several different organizations, I have often found the most difficult challenge is to win the support of senior level management. To me, it is clear that the so called traditional processes are insufficient to handle the rapid pace of change in both the technical and business environment of today. However, for many other executives there is a perceived safety in following these approaches. In order to influence such executives I often need to reference other "objective" third party information so that it does not seem like I am simply presenting my own unsupported opinions.

Upon review of the first draft of this article, a colleague pointed out the contradiction in the use of the concepts of evangelism and objectivity in the preceding paragraph. He pointed out that evangelism of a particular process or technology often distracts us from our real (professional) purpose which is to deliver software that is useful to the businesses we serve. The very idea of such distraction is un-Agile at its core, so how can I be an "Agile Evangelist". It was difficult to reconcile the contradiction, but just as I was about to give up, I considered the values put forth in the Agile Manifesto. I 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"

I don't see adherence to these values as a conflict with serving my business. By being true to the first value alone, I cannot allow a process to distract me from delivering the best possible results to my business. I am comfortable evangelizing these values.

As a community we must sell our values at the senior management/executive level in order to truly succeed in our mission of "uncovering better ways of developing software". While bottom-up initiatives to introduce Agile practices are a good start, top-down support is vital to the successful mainstream adoption of Agile values. Top-down support is required for a variety of reasons.

A primary reason for the need of top-down support is the impact that adopting Agile values has across the entire organization. Product management is one of the most heavily impacted areas, perhaps even more heavily than product development. In an Agile organization, product management has to accept much more accountability for the ultimate delivery of a product than in a traditional environment.

Product management must be willing and able to make clear cut priority decisions, stay actively engaged with the development team and have the fortitude to actually release the product at some point. This requires a strong mandate from senior management. Downstream areas like publications, training, support and marketing are also impacted by the adoption of Agile values by the development group. These groups are used to having a pre-defined plan (specific feature set to be available on a specific date) to which they manage. With Agile values, the full feature set of a product may not be known until much later in the game, and this causes anxiety in the downstream areas. Agile activists observe that traditional approaches provide only a false sense of predictability to the downstream areas, so they really need to be more flexible regardless. However, the downstream areas must be strongly encouraged towards (and mentored in) a more priority driven, incremental approach. Again senior management is needed to provide such direction.

Also, senior management must be willing to stay the course. While adopting Agile practices is a logical, common sense step for many organizations, it is not always an easy one. The first few projects may encounter turbulence, and if senior management is not fully supportive the organization may slip back towards the imagined safety of more traditional methods. In more than twenty years, traditional methods have produced more failure than success, so organizations should not be easily deterred from experimenting with new approaches.

The purpose of this section is to provide a forum for exchanging information that will help members of this community promote Agile values at the top management levels of their organizations. As Scott Ambler discusses in his article "Proof Positive" (*Software Development Magazine*, November 2003), the hard proof that senior management often demands about the benefit of Agile methods is beginning to emerge. I invite those involved in, or aware of, empirical studies of Agile practices to provide summaries and links to additional information here. Since, as Ambler also discusses, it may be many years before there is enough empirical data to be considered definitive, I also invite contributors to provide philosophical arguments and/or anecdotal evidence of the benefit of agility. In fact, there is a significant argument to be made that after more than twenty plus years of mediocre results, the traditional methodologies must be justified against new approaches instead of the other way around. Again, I invite articles along these lines. Since we value results over process, we must be willing to discuss situations in which Agile methods (at least as they exist today) are not applicable. Such discussion might lead to new approaches that embody our values or enlighten us to other practices and values that we have not considered.

My goal is that the information provided in this section helps the members of this community advance our values to the "outside world". I would like to spur members to think about the issues in new ways, provide proof of value to those who need it or just give a member the confidence to present his or her ideas to senior management. The first article for this section is my own, and it is philosophical in nature. I look forward to collecting and presenting your submissions in the future.

ABOUT THE AUTHOR

Sott Bogartz is the Director of Data Products Development at Reed Construction Data in Norcross, GA. He is an experienced software developer, architect and project manager whose background includes consulting and in-house development. He is also a Certified Scrum Master, an XP practitioner.

Do you have a Scrum Success Story to share? Have you found an interesting or novel way to apply Scrum to a non-software project? Perhaps it's a spectacular failure, but one that we could learn from. If so, send it to Michael Ivey mdi@iveyandbrown.com for possible inclusion in a later edition of this column.

Scrum Success Stories
Michael Ivey

At Ivey & Brown, we've had some success with an alternate approach to "fixed-price" projects and Scrum. The technique we've been using specifies a price on each sprint (instead of one price for the whole project), and has the added benefit of immersing customers in Scrum.

The traditional approach to fixed-price Scrum projects could be called "maximum-cost, latest-date." This variation of "fixed-price, fixed-date" gives customers the most important functionality up front, and offers them the opportunity to substitute any new feature for a feature that hasn't been started that is of equal estimated effort. With this method, customers can halt development before completion if they feel the project is "good enough."

In projects like these, the level of exposure to the behind the scenes Scrum process varies from customer to customer. Sometimes customers don't know (or care) what process the developer is using, and sometimes customers know they are playing a role called "Product Owner" but not much else. Customers on fixed-price projects are usually not fully involved in the workings of Scrum.

We at Ivey & Brown wanted a way to involve our customers more in the Scrum process and still give them a fixed price. In one particular case, we had a customer who had a large number of rapidly changing requirements and virtually no time to prioritize and specify those requirements. This customer also didn't want to spend a lot of money at one time, so we would only have a few part-time developers working on the project. We identified a Product Owner who took on the responsibility of managing all of the fluctuating priorities and items for the Product Backlog. At the beginning of each Sprint, we ask, "How much do you want to spend this month?" Based on the answer, our team selects an appropriate amount of work from the Product Backlog. Instead of the usual negotiations over price, the customers set the price, and therefore the pace. If one month they want to spend a mint, they get a lot of new software. If the next month they only want to spend a couple hundred dollars, they can get a little bit of software. It's completely up to them.

Of course, this won't work in every situation, but "fixed-price" sprints are certainly an option if you come into contact with customers who have circumstances similar to the above.

> "*Drop wisdom, abandon cleverness, and the people will be benefited a hundredfold …See the Simple and embrace the Primal*" –Tao Teh Ching

Over twenty-five-hundred years ago, the Chinese philosopher Lao Tzu wrote the Tao Teh Ching, a 'guidebook' for achieving happiness by living in harmony with the universal laws of nature. To Lao Tzu, the best way to live a contented life was to understand these laws, and make the best of the circumstances which they created. The more man interfered with these laws the more unhappy he became. By imposing abstract and arbitrary rules, leaders set the stage for strife and struggle instead of teamwork and success.[1]

This article examines the "natural laws of software development" and how man's interference with these laws has also caused much suffering by preventing the successful completion of many projects. It provides high level insight into how the practices of Agile development[2] more closely mirror the natural laws than do the dogmatic approaches of more traditional methodologies. The use of philosophy and "natural laws" to describe best practices for software development is a somewhat whimsical idea and is intended to present a more interesting twist on the information. However, there are more serious underpinnings for this approach. Software development is an activity built upon complex, interdependent relationships that mirror those within a natural organism or ecosystem. It may not be too far fetched to propose that such an activity is governed by a set of natural laws. A similar argument can be made for the applicability of philosophy to the study of software development. Interactions between humans, and the manner in which different humans interpret a given set of information, are the factors that drive software development. A philosophical view delves into those factors.

This article does not attempt to provide hard proof of the benefit of Agile methods. The assertion is that in many organizations, the case for continuing to follow the same old practices requires more justification than the one for trying something radically different. It is a fairly well documented fact that the traditional methodologies have done little to improve the reliability and predictability of professional software development. Studies indicate that nearly two-thirds of all software projects either fail to deliver at all or substantially overrun their cost and/or time estimates.[3] These statistics offer proof that the traditional methodologies are not in harmony with the natural flow of software development. A less tangible, but no less interesting, indication of this divergence is observed in the writings of the thought leaders of the Agile movement. The use of the word "uncovering" in the opening statement of the Agile Manifesto is highlighted as a deliberate and fascinating selection by Jim Highsmith and Martin Fowler.[4] While the stated intent of this selection was to indicate that the process was still in process, one can infer deeper meaning from the choice of uncovering instead of other verbs like creating, inventing or crafting. The use of uncovering suggests that that the Agile methodologies are helping to remove the clutter that has obstructed our view of the "better ways of developing software" that have existed all along. This theme of returning to a better approach (instead of creating a new one) is also repeated in introduction of Agile Software Development with Scrum. While the Agile methodologies are often described as radical in relation to the traditional methodologies, the Agile methods are really more common sense than revolution. They represent a return to the natural approaches that have been obscured over the past 20 years.

SEEKING WISDOM AND CLEVERNESS – THE MYTHS OF TRADITIONAL METHODOLOGIES

Just because there is a natural flow of software development, does not mean that software development is easy or that anybody can do it well. The natural flow of software development does not always lend itself to the business needs of a software organization. It is natural and legitimate for a business manager to want to know what features will be in a product by what date and at what cost. Even for those with a very deep understanding of the natural flow of software development it is very difficult to

provide this kind of information with any degree of accuracy (at least beyond a few weeks into the future). As we will discuss later, those who do understand the flow know how to manage the process to make the best of this situation.

The traditional methodologies set out to fight against the inherent uncertainty of software development. They asserted that they could provide concrete answers to the vexing questions and increase development productivity at the same time. A cynical view of this approach is that these methodologies were created for the commercial gain of their owners and not for the greater good of the industry. A process that claimed "we can answer all of those difficult questions" was much easier to sell than one that admitted "we cannot answer all of the difficult questions, but we can tell you to make the best of the uncertainty." A more forgiving explanation of the traditional methodolgies' attempts to impose control on the software development process is that they were born of manufacturing management practices. The early methodologists, for example "Big 6" consultants,[5] were heavily influenced by manufacturing practices. In addition the clients who first attempted to apply these methodologies were mostly manufacturers looking to improve the software they developed for internal use. Since the manufacturing process could be controlled very precisely, it must have seemed logical that similar practices could be applied to software development.

Unfortunately, software development is very different from manufacturing (at least as it was practiced when the traditional methodologies were born). Where manufacturing mass produces identical units after they have been concretely defined and completely designed, software development starts with a high level concept and customizes a solution around that concept. More importantly manufacturing is best optimized by improving task efficiency while software development is much more dependent upon creativity, communication and problem solving. Many of the techniques that can be used to increase task efficiency either have a neutral or adverse effect on tasks that require intense problem solving. For example, adding more people to perform the tasks in a manufacturing effort can be a very good way to speed up production. However, due to the complex interrelationships of the tasks and the need for communication among team members adding more people to a development effort often slows down software development.[6]

Flawed as they were in their assumptions about the nature of software development, the traditional methodologies offered answers to a desperate industry, and the industry bought in to their promises. This combination of circumstances gave birth to a number of myths about software development, and the myths over time became conventional wisdom.

Probably the most significant myth is that of predictability. The statistics mentioned in the introduction of this paper demonstrate that traditional methodologies do not deliver on their promises of predictability. A very good software development team might be able to accurately estimate a month's worth of effort after working together on the same problem space and technology for some time. However, managers have been led to believe that an entire project can be accurately estimated. This disconnect leads to serious psychological problems across the organization. Developers may pad estimates to make up for the known uncertainty. Managers become wary of high estimates or past failures, and pressure teams to "come up with better numbers". Mistrust and apathy are commonplace. While developers may initially feel guilt about missed deadlines, any motivating effect of this guilt quickly reverses itself as the developers begin to feel they are always up against unreasonable deadlines based on inaccurate estimates.[7]

Another damaging myth created by the traditional methodologies is it is possible to completely analyze and design a software product (with non-software artifacts) before beginning to actually construct that product. The traditional methodologies pushed paper based deliverables as the primary means of understanding and specifying a system. These documents were used as the basis for estimating the

development effort and were typically seen as a "hand-off" from the business specialists to the technical team. Again, this myth was born of the manufacturing background of the traditional methodologies. In manufacturing it is typical to have a completely accurate specification before mass production begins. However, for most complex products the design process involves much more than just documentation. Typically the process involves the creation and refinement of prototypes of these products. These prototypes allow analysts, designers, and customers to interact with the product, identify previously misunderstood requirements, and adjust priorities based on feedback. The traditional methodologies used the concept of analysis and design without using the same level interactivity. The benefits of a purely paper based analysis and design diminish quickly, and the traditional methodologies overly emphasized these activities without paying enough attention to the more productive interactive techniques (prototyping).

EMBRACING THE PRIMAL – THE AGILE ANSWERS

Where the traditional methodologies struggle against the inherent complexities of software development, Agile methods tend to adapt to them. Agile methods do not try to force software development to be predictable. Instead they provide organizations with a means to achieve the best possible results given the unpredictable nature of the activity. This does not mean that the results of an Agile development effort are totally unpredictable. With an Agile method, you can be confident that highest priority features identified at a given point in time are the ones which will be completed first. This way it is easier to manage scope as a release date approaches in order to hit the date (with your highest priority features if not all that you originally envisioned). Also, Agile methods do not try to use "complete" paper based analysis and design to conquer the intricacies of human to human (and human to system) interaction. Agile methods do not advocate abandoning design. Instead they encourage more natural and productive design activities by increasing the interaction between customers, developers and the system.

CONCLUSION

As we move to introduce Agile practices into new environments we must be prepared to overcome a variety of objections. While many of these concerns and requests for hard evidence about the value of Agile practices are legitimate, we must recognize when the basis of an objection is the myth of another approach instead of its reality. We must not give in to the pressure and try to sell our approach as a way to conquer the complexity of software development. In admitting that we cannot eliminate this complexity, we establish our honesty and realism. From this base of understanding, we can help our organizations move towards adapting to the natural laws instead of fighting against them.

FOOTNOTES

[1] Hoff, Benjamin, The Tao of Pooh, pp. 4-6, E. P. Dutton, Inc., 1982

[2] While terms such as Agile development, Agile methods and Agile practices are used interchangeably throughout this article in general reference to the various processes currently defined under the Agile umbrella, they should be considered in broader terms to refer to any practices that reflect the core values expressed in the "Agile Manifesto".

[3] Schwaber, K and Beedle, M, Agile Software Development with Scrum, pp. 1-2, Prentice Hall, 2002

[4] Fowler, M. and Highsmith, J., "The Agile Manifesto," Software Development, 9(8): 28-32, 2001.

[5] Schwaber, K p. 107

[6] Brooks, F, The Mythical Man Month, p. 19, Addison–Wesley 1995

[7] Schwaber, K. p. 100

In 2001, the Agile methodologists had great stories to tell about coding, team organization, and interaction with the business world. With the exception of unit testing in XP, testing thinking noticeably lagged behind. What's the state of things today?

Let me illustrate my impressions by referring to the high-tech adoption curve that marketing guru Geoffrey Moore describes in *Crossing the Chasm*. For him, adoption starts with *technology enthusiasts* who try things out because they're cool. They're not bothered that those things don't actually work well. The *visionaries* listen to the enthusiasts in order to hear about new technologies that can lead to order-of-magnitude improvements in their business. They are the ones who force the technologies to work. After the visionaries comes mainstream adoption by *pragmatists* who want incremental improvements and assurance that the technology works. There are two later stages that don't concern us here.

Test-driven code development (TDD) has made it to the pragmatist stage. Evidence? Here are three titles on my bookshelf: *Test-driven Development: A Practical Guide* (Astels), *Test Driven Development: By Example* (Beck), and *Pragmatic Unit Testing* (Hunt and Thomas). TDD is being taught in university classes. Support is integrated in at least the Eclipse and Intellij IDEA development environments. TDD is safe technology: not universal, but adopters need not be pioneers.

Importantly, the role of testing in TDD is shifting. It's by now a truism that test-driven development is more about thinking through a design by way of examples than it is about finding bugs. People are exploring how tests help communication, team alignment, and documentation. They're embedding testing into the larger development process, rather than considering it a thing apart.

Visionaries are experimenting with **test-driven product design**. How can business experts, testers, and programmers collaborate to produce examples that drive entire iterations? As was the case with test-driven code design (which gained critical momentum with the release of JUnit), tool support has been a motivating force. Many are experimenting with Ward Cunningham's FIT table-driven testing tool (fit.c2.com). Along the way, they're learning more about how business experts can collaborate with development teams.

Exploratory testing is the province of the technology enthusiasts. James Bach defines exploratory testing as "simultaneous learning, test design, and test execution". Unlike the previous two techniques, it's done after the fact, when there are finished features to play with. My limited experience leads me to think that exploratory testing fits nicely as part of closing up an iteration: demonstrate this iteration's progress to interested bystanders, have the whole team and those bystanders explore the new features, move on to the end-of-iteration retrospective, then prepare for the next iteration.

Exploratory testing seems a good match for the reactive and improvisatory nature of Agile projects. Programmers seem to enjoy it, so long as it's clear that exploratory discoveries are potential tasks for later iterations, *not* punishments for mistakes in the last iteration. (Programmers should not be chastised for not doing what no driving tests told them to do.) To learn more about exploratory testing, I recommend James Bach's articles at www.satisfice.com/articles.shtml.

A variety of test sub-disciplines like configuration testing, load testing, usability testing, and the like seem much the same in Agile projects as in conventional projects. This **non-functional, para-functional, or "ility" testing** is either solidly established in the pragmatist stage or it awaits a technology enthusiast to find a new approach.

2003 was a year of great activity in Agile Testing. 2004 promises to be even better.

This issue of the Agile Times Newsletter marks the beginning of a regular section that will discuss issues and experiences around the topic now called "Embedded Agile".

EMBEDDED SOFTWARE – JUST DIFFERENT

Developing software on embedded platforms (also known as "firmware") has always been considered different from "traditional" software development. Some of the reasons make sense, namely:

1. Embedded systems are function specific and usually unique.
2. Embedded programmers are not normally classically trained programmers. Many, like myself, have degrees in Electrical Engineering. This is because knowledge of electronics can be essential or at least very beneficial to an embedded programmer.
3. The amount of code on a typical embedded platform is usually smaller, often dramatically so.
4. Real time constraints often put performance at a very high premium.
5. Code space limitations (and item #4) often lead to frequent hand coding in assembly language and other compromises.

Embedded Systems Programming magazine reported that in 2002, "6 billion processors of all types were sold (and) ... only about 100 million (just 1.5%) became the brains of PCs, Macs, and UNIX workstations; the rest went into embedded systems.". This does not mean that 98.5% of all software written is for embedded platforms. It's likely that most software written is for non-embedded applications. Still, there appears to be a lot of embedded software out there.

Further, I have noticed that embedded programmers tend to be reluctant to adopt a lot of the traditional tools of software professionals. I was around when C compilers were first being used on embedded platforms and the phrase "pulling teeth" really understates the unwillingness of embedded developers to adopt a high level language. Items 4 and 5 above drive the rational part of this mind-set. Much of the aversion is, however, irrational. If you know of an embedded programmer with the hiccups, sneak up behind him and shout "C++!." The ensuing anxiety attack should cure him. So here then is the paradox: While traditional software developers often seem irrationally eager to adopt the next new thing, embedded programmers are often irrationally averse to anything new. That is why both groups need good managers.

It should come as no surprise, then, that the level of experimentation and adoption of Agile practices is much less common in the embedded world. The following table lists the Google hits generated when certain things are typed in:

- "Extreme Programming" + software 177,000
- "Extreme Programming" + embedded 22,200
- "Extreme Programming" + firmware 1,640

Note that the word "Agile" it a bit too generic to yield meaningful results, but when I typed in "Agile embedded" I got exactly 3 hits and 2 of them had nothing to do with this topic. Also, a brief survey of yahoo user groups yields dozens of groups devoted to things Agile (mainly XP) with thousands of members. However, as far as I can tell, there is precisely 1 user group devoted to the topic of applying Agile techniques in the embedded world and it has exactly 20 members. Truly, then, the Agile ball has yet to really start rolling in the embedded domain.

WILL IT WORK?

There is every reason to believe that Agile techniques will be as successful in the embedded domain as they are in the traditional programming environments. There is a lot of parallelism in the challenges and

frustrations faced by both groups. First and foremost, the embedded environment is just as prone to ever changing and evolving requirements as any other engineering field. Predictably, embedded personnel are just as likely to conclude that change is an unpleasant, expensive thing and take steps to legislate it away. Merely declaring that change is not allowed fails as miserably for us embedded folk as it does everywhere else. Here then is the heart of the debate. Traditional project management holds that change is not inevitable, it can be eliminated with careful up front analysis and design. Agile claims that change *is* inevitable so project work should be driven by techniques to *make change cheap.* This includes:

- Regular delivery of a working subset of the system. This serves as the best catalyst to uncover the true set of requirements and give us great feedback on our proposed solution. The faster we know these things, the less the cost of the project.
- Invest heavily in software that is architected to be flexible and agile when change comes along. This drives down the cost of changing the software.
- Invest heavily in automated testing to produce fast, high quality feedback on the quality of our work. This drives down the cost of testing.
- The non-software portions of the project should be designed with flexibility around the high risk and least known aspects of the system. This drives down the cost of changing them.

One of the great things to come out of the Agile movement is that it has turned all of us down to earth engineer types into philosophers, namely epistemologists[2]. Agile folks are normally a happy and optimistic bunch, but we are very proud of our epistemological skepticism. Agile philosophy observes that all decisions on a project are made with imperfect information. Further, we claim that our information is *most* imperfect at the *beginning* of the project, and the picture will get clearer with each passing day. If this is true then is does seem nonsensical to insist on hard decisions for all aspects of the project at the beginning. Traditionally, though, this has been the strategy to minimize, and blunt the impact of, change. As Jim Highsmith has said, "Agile managers understand that demanding certainty in the face of uncertainty is dysfunctional[1]." Also, Agile claims that discussion and documentation are poor predictors of requirements validity and tell you little about the integrity of the proposed solution. Only by building the system can we uncover the flaws in our up front thinking.

Lest you think that uncertainty is something that only plagues engineering, note that the problem exists at the highest levels of government:

> "...because as we know, there are known knowns; there things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns — the ones we don't know we don't know." -Donald Rumsfeld

That's easy for you to say, Mr. Secretary.

BUILDING THE HARDWARE ALONG WITH THE SOFTWARE

The particular point that seems to get raised immediately when I discuss implementing Agile in an embedded domain is that fact that quite often the hardware is being developed in parallel with the software. To a large extent, I think this is the biggest difference. For if the hardware is already finished and known to be good, then it seems to me that you have a pure software effort and all theory regarding best practices and methods that apply to any software effort should apply. You might have different priorities, especially items #4 and #5 above, but all in all this scenario parallels traditional software development very well. It is when the project starts with no hardware and no software that things get meaty. Unsurprisingly, the instant reactions to applying Agile techniques in this situation are as follows:

- "Hardware is much more expensive to change later, so we must do our homework up front"

- "The lines between what is a hardware and what is a software requirement is vague and interdependent, so all must be decided up front"

Piffle. Mary Poppendieck[2] has done a tremendous job of refuting all of these arguments. I asked her this specific question and this is what she had to say:

> "I am convinced that people who think intelligent hardware systems are developed waterfall-style haven't developed many of them. Very often when such a system is being developed, the hardware is on the bleeding edge of known technology. This is certainly true of weapons systems and cell phones, and quite often it is true of medical instruments. The idea that the hardware is designed 'up front' rarely matches reality."

On the other hand, it is true that hardware design gradually gets more fixed and formal as time goes on. Thus you can't do the kind of refactoring at the end of a hardware design cycle that you can do at the end of a software design cycle. I think that the best hardware design starts with a lot of options and big tolerances, and then narrows the funnel as development proceeds. Thus you have to gradually increase the formality of the software design as the hardware design gets closer to being fixed."

She describes in her work how the Japanese were able to drive down the cost and time of developing a new car by using this technique. That is, they started new designs with big tolerances in the areas that were predicted to be most likely to change and gradually narrowed their tolerances as time went on. This is why the Japanese were able to compete better in the '80s than American car makers. Now *all* cars are developed this way. Needless to say, if a *car* can be developed without deciding everything up front, *and if such a technique can dramatically lower the cost of development*, then surely the same technique will work with the electronics on an embedded project.

PROJECT PLANS ARE FINE

There is a well-known axiom attributed to Dwight Eisenhower that goes "War plans are fine, until the fighting starts." Because of this, some have concluded that Eisenhower was a critic of war planning. Ironically, this was not the case. As a general and as a president he supported vigorous military planning. At the same time, though, he knew the limitations of such planning. He knew that war plans needed large amounts of flexibility and a smart, empowered chain of command to make fast battlefield decisions based on new information that was being obtained on a minute-by-minute basis. There are many other examples of situations that must be managed this way. Coaching a football game requires much up front planning, but those plans can often get turned upside down or thrown out altogether on the first play of the game. Management of this type requires much elasticity, many options, and good, quick feedback on the merits of any approach.

This, then, is what it means to be Agile. To manage your project as if surprising, new information will present itself at any moment and must be worked into the plan. To manage as if the up front plan was created by flawed, imperfect beings who do not have the benefit of hindsight. Embedded projects have all of these characteristics, and I am convinced these techniques when applied enthusiastically to the embedded domain will yield the same spectacular results that other fields are seeing. Let me close by plagiarizing Dwight Eisenhower and offer the following axiom:

Project plans are fine, until development starts.

This section of the Agile Times Newsletter will gather experiences and discuss the issues surrounding implementing Agile techniques in the embedded domain. This edition contains three such articles. If you would like to share your real-world experience or wisdom, please contact me at dan@embeddedeng.com.

ENDNOTES

1. Highsmith, Jim, What is Agile Software Development, Crosstalk, The Journal of Defense Software Engineering, October, 2002.

2. If you are unfamiliar with her work, you should stop reading this immediately and go read everything you can. A lot is available for free on www.agilealliance.com and www.poppendieck.com

I have yet to buy her book and I am hoping that flattery such as this might garner me a free copy!

Transitioning To XP In An Embedded Environment

Nancy Van Schooenderwoert

About five years ago I was technical software lead for an embedded project that I and my team transitioned over to XP. The move was successful, but that success depended on the state of the project beforehand. I'll give some background on the project, then explore the success factors.

PROJECT SETTING

Our challenge was to build a mobile ruggedized spectrometer, using a new board, new sensor hardware, new CPU (final silicon net yet available), operate two communications protocols simultaneously, and fit the architecture into a partner company's existing architecture (a network of communicating instruments). Oh, and implement a very sophisticated mathematical algorithm that was still under development and would be changing all through the project and afterwards as well. End users would receive periodic updates to the algorithm in the form of calibration tables.

We were about a year and a half into development of the spectrometer when I learned about XP and wanted to try it. We had created unit tests for most of the modules we wrote but they weren't automated. The software could be run on the target CPU and also on top of Windows NT, to isolate hardware problems. I had successfully instituted collective ownership of the code, and had communicated the overall design vision to everyone on the team. They all understood how the code they were writing fitted into the big picture. We were using code reviews, coding standard, continuous integration, and had easy access to our customer.

Prior to adopting XP we were not doing anything like the Planning Game. Requirements came from multiple sources, and there wasn't the sense of accomplishment you get when you have a clear, achievable release goal and you all focus tightly on it. Pair Programming and Test First (now TDD) were missing. Refactoring was being done but not enough.

SUCCESS FACTORS

1. Dual Targetting

Our application ran on a desktop PC as well as on the target CPU. We maintained this capability throughout development, even after we had good hardware. With so many hardware components at early stages in their own development, we simply could not risk having to troubleshoot with multiple unknowns. This would have been tough for the most seasoned people, but half the team were new to embedded work and would've been overwhelmed without the ability to quickly isolate hardware problems. We were using Nucleus PLUS operating system which comes with a version of the OS that runs on top of Windows NT. This allowed us to fully exercise the application logic on WINNT before moving to the target CPU. The interface between the OS and our application was the same for both platforms – we'd just have the build link in one or the other library. Dual Targetting isn't a defined XP practice, but I nominate it as a practice for Embedded XP. Without it we'd have been lucky to finish the project at all,

2. People - the Right Stuff

A spirit of cooperation among the team members cannot be underestimated. Without it, these crucial practices would have been undermined: Coding Standard, Collective Ownership, Code Reviews, Continuous Integration, Metaphor. One can argue that these are not prerequisites to adopting XP – they *are* XP, a big part of it at least. My point here is that you need a team capable of these practices. If there is interpersonal strife going on, it has to be solved. We didn't have this issue but if we did and couldn't solve it, there's no way we could do XP.

3. Unit-Test Code Already in Place

At the time we changed to XP we had a large amount of the code in place. It's final size was approximately 30,000 lines (comments excluded). If we hadn't already written good unit tests for most of this code, there was no way we could've found time to add it at that point. I know *now* that I can bring legacy code to an XP project, but it would have been just too big a hurdle then. I was learning XP, trying to teach others XP, and the company culture was big-process waterfall mentality. Without those unit tests in place our move to XP would have failed.

The only other practices we were doing before XP that I haven't ranked here are these:

Trouble Log: A "bread crumb trail" that's always "on" so it doesn't distort your troubleshooting by having to be enabled.

Stand-alone Module Execution: Each loosely-coupled domain of the application should build and execute alone on either target. For example, we had a serial communications domain (made up of many functions) and we could build the embedded system with only this domain running – by sending it messages and commands we could find out whether delays were in the serial communications other parts of the system. This capability for isolating problems at the domain level is useful in any software system, but it can be critical to troubleshooting an embedded application. Likewise for the trouble log.

The above are my other nominees (in addition to Dual Targetting) as practices for Embedded XP. They are not new. Most of XP is not new either, just amazingly effective.

REFLECTIONS

When I began this piece, I was going to rank the Unit Tests first, Dual Targetting second, and People third. On reflection, they all moved position. In my team's situation, the Dual Targetting was essential. But it wouldn't be for a system with stable hardware. Because we had a lot of code in place before moving to XP, the existing Unit Tests were very important to our success. Again, that's more to do with our situation and trying XP for the first time. It's not a universal principle. That leaves the People issue. That does apply universally.

It's not surprising that we were able to transition to XP, given that our existing practices were essentially a subset of XP – the right subset. For our situation, anyway. What does this mean for other projects? Is there some test one can apply to tell whether a given existing project can transition successfully to XP? Aside from having a harmonious team, the key issue is testability. Whatever the state of the software and hardware, can it be made testable? That question has an extra dimension when you're talking about embedded software. Sufficient testability cannot usually be achieved by just adding software unit tests (assuming you have the time). The hardware may not have the necessary hooks. Can the software generate a CPU reset? Are there spare I/O lines so you can monitor selected activities using an oscilloscope? Are there spare interrupt lines? No matter what hardware "hooks" I might mention, they won't apply universally because embedded systems vary so much.

That leaves "testability" as a quality to be judged by the engineers involved. How much testability is necessary to make your system Agile? Can that testability be achieved? Do you have the time and resources to achieve it? No easy answers. That's embedded development.

BIO

Nancy Van Schooenderwoert is a founder of XP Embedded, a consulting company dedicated to bringing Agile practices to the development of embedded software and large-scale real time software systems. She brings a systems perspective to software engineering for real-time and embedded systems. She has extensive experience in building large-scale, real-time systems for flight simulation and ship sonars, as well as software development for safety-critical applications such as factory machine control and medical devices. Nancy has held positions in electrical, systems, and software engineering, as well as in test design. She has a BS in Computer Engineering from RIT and a private pilot's license. Contact: nancyv@xp-embedded.com; website: http://www.xp-embedded.com

Progress Before Hardware

James Grenning

A common problem facing embedded software engineers is the concurrent development of hardware and software. The embedded software engineer does not have a test bed for their work often until late in the project. I have seen too many project plans that show an integration and test phase late in the project where hardware and software are brought together. Those integrations usually end up turning into seemingly endless debug sessions. We may tell ourselves that this project will be different, that we can integrate, test and ship in two weeks. But we'd be kidding ourselves.

Embedded systems expert Jack Ganssle says "The only reasonable way to build an embedded system is to start integrating today... The biggest schedule killers are unknowns; only testing and running code and hardware will reveal the existence of these unknowns."[GANSSLE] Jack goes on to say that "Test and integration are no longer individual milestones; they are the very fabric of development."

Does the lack of the target platform mean we cannot test our code? Does that keep us from following Jack's advice and the advice from the Agile development community? The answer to these questions is a resounding "No!". In this article I'll describe how to make progress prior to hardware availability.

EMBEDDED SOFTWARE DEVELOPMENT

Developing software is hard. Too often projects are late, with poor quality and inadequate feature sets. Embedded software development shares some of the same problems with non-embedded software development, but it also presents some additional problems. The development machine architecture and operating environment are often different from the target machine. The hardware for the target machine is usually developed concurrently with the software, and therefore not available until late in the project. The hardware may go through several iterations, changing in ways that confound the software systems. There may be real-time constraints, concurrent processing, and safety issues. Typical human-computer interfaces are not used and the computer operating the machine is hidden from the user. Resource constraints such as limited memory space or processing power are the norm.

PRACTICES

Test driven development and object oriented design are two practices that can help make concrete progress early in the embedded software development cycle. Test driven development is an incremental

technique for concurrently writing and testing code. In this article we'll look at applying TDD to embedded development.

Object Oriented Design is not a new technology, but it is a poorly understood and therefore an underused technology in the embedded development world. Object oriented languages like C++ or Java really enable this technology, but the ideas behind OOD ideas can be implemented in procedural languages such as C.

TDD and OOD can give the embedded software engineer some advantages. One specific advantage is designing, coding and testing prior to target hardware availability. I'll discuss how you can make significant progress by testing on your development machine. This implies using a portable programming language. If your environment is so constrained that you must develop in assembler you may not be able to use all the advice in this paper.

DEVELOPMENT ENVIRONMENT AND EXECUTION ENVIRONMENT

In embedded systems the development environment usually differs from the target execution environment. I can buy a development environment at the local computer store or on the net. I can buy compilers, debuggers, source control tools, word processors and other tools for my development environment. Development environments are relatively cheap. On the other hand the target is custom made. Maybe the target is a cell phone, an engine controller, or a high speed color printer. I can't go down to the local computer store to buy that platform. Target systems are limited and expensive.

I've seen prototype hardware that cost over $1 million. This results in the engineering team having a one to many ratio of target machines to developers. A limited resource means sharing and sharing means waiting. Waiting kills productivity. Even with access to target hardware development time is slowed whenever we test on it. Downloading and running in the target takes time, and it's a tough environment to debug in.

That said, testing in the target is necessary, but not always possible or practical. Fortunately, there are alternatives. You may be able to run in a simulator, a limited hardware prototype, or your development system.

SIMULATORS

Simulators can be very expensive and complicated. Simulation can be done at many different levels. We can simulate the processor. We can simulate the behavior of the environment. A comprehensive simulator can rival the target platform in complexity. Later in this paper I'll describe an alternative that I call a scenario simulator.

LIMITED HARDWARE PROTOTYPE

If you cannot have the full-fledged prototype, a limited hardware prototype is very useful. The limited prototype would be very close in design to the target, but would not have all the capabilities of the target system. Maybe it's the target processor with none of the special IO.

Using a prototype can have a very positive impact. Only part of the IO is available, so it will be necessary to build hardware independence into your design. This is one way that Object Oriented Design fits in. OOD allows the definition of interfaces, isolating one part of the system (the main application logic) from some other part of the system (the hardware implementation).

A limited prototype is a very valuable and necessary tool when the full target is not available. This prototype may suffer from the same problems as the actual target. It may be expensive, not ready,

buggy, or slow for download and test. What's an engineer to do? Perhaps we can focus our testing efforts on the development system.

DEVELOPMENT SYSTEM AS A TEST BED

I'll start out with a claim: significant progress can be made on the development system. With isolation from hardware and operating system dependencies much of your embedded application can be tested on your development system. You'll need to be able to compile and generate executables for the development system as well as for the target.

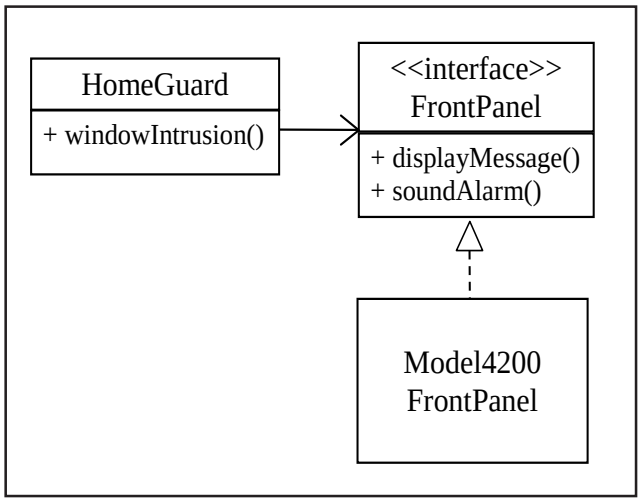
How does this work? The development system does not have the specialized IO that the target has. How can you test it? What does running it on the development system mean? One key to solving this problem is to design in hardware independence using OOD. The idea we started talking about a few paragraphs ago. The second key is Test Driven Design.

OBJECT ORIENTED DESIGN

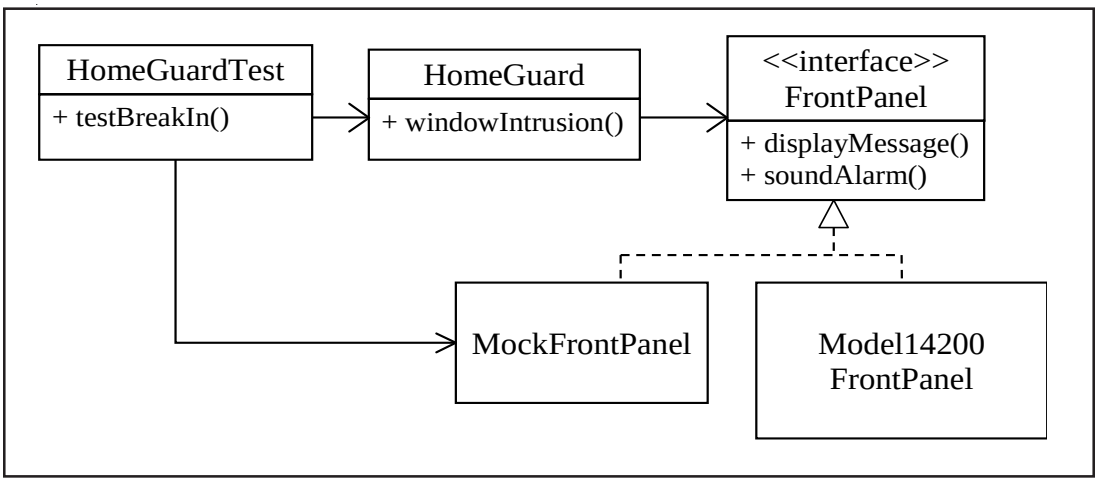
When thinking about Object Oriented Design (OOD) think *interfaces*. An interface can be defined that describes how to interact with some hardware provided service. The code in the layer above the hardware isolation layer can be designed to have very limited knowledge of the underlying hardware. In C++ a class is defined that specifies the calling conventions of the interface and reveals none of the details. The main application code interacts with the execution environment through a set of interfaces. The application code can interact with the real hardware or some stand-in for the hardware that obeys the same interface.

This UML diagram illustrates part of a home security system called HomeGuard. The logic in the HomeGuard class understands what it means to be a home security system. It knows the incoming events (window intrusion) and it knows how to report the security system state to its front panel. It does not know that when you write a one to address 0xFDAF00, bit 3 that the alarm will start sound. The presence of the FrontPanel interface means we can create other implementations of the FrontPanel. For instance we could create a LoggingFrontPanel that prints the changes to the FrontPanel to a log file Test Driven Development Cycle

Test Driven Development is a state-of-the-art software development technique that results in very high test coverage and a modular design. In TDD we try to test each function in isolation and incrementally build larger groups of collaborating functions and classes to provide the desired functionality. Tests come in layers. The need to test in isolation means we have to decouple one part of the system from another. Interfaces are one of our tools. Interfaces are used to decouple the parts of the system from each other. Notice the structure of the test code and application code below. The HomeGuardTest class defines the tests (only one shown by name). HomeGuard encapsulates the security system rules. The FrontPanel describes what can be asked of a front panel. The Model4200FrontPanel implements the FrontPanel interface and knows how to interact with the hardware. But what is a MockFrontPanel? It is a test stub. It is part of the test code. When HomeGuardTest wants to test the break-in scenario, it binds HomeGuard with a MockForntPanel. The MockFrontPanel can intercept messages meant to go to the front panel so HomeGuardTest can see if HomeGuard has responded



per the requirements. The test can interrogate the Mock Object[MACKINNON] to see what state it is in. The practice of testing helps to improve modularity. Modules are tested in isolation and in collaboration with other modules. Between the test and the Mock Object we are creating a simulator for a specific scenario.



The window intrusion test looks like this:

```
TEST(HomeGuard, WindowIntrusion)
{
  MockAlarmPanel* panel = new MockAlarmPanel();

  HomeGuard hg(panel);

  hg.arm();
  hg.windowIntrusion();
  CHECK(true == panel->isArmed());
  CHECK(true == panel->isAudibleAlarmOn());
  CHECK(true == panel->isVisualAlarmOn());
  CHECK(panel->getDisplayString() == "Window Intrusion");
```

EMBEDDED TDD CYCLE

Kent Beck, author of *Test-Drive Development by Example*[BECK] describes the TDD cycle as:

1. Quickly add a test
2. Run all the tests and see the new one fail
3. Make a little change
4. Run all the tests and see the new one pass
5. Refactor to remove duplication

This cycle is designed to take only a few minutes. Every few minutes you find out if the code you just write is doing what you want. Is such a rapid feedback cycle feasible in embedded development? Let's look at some possibilities.

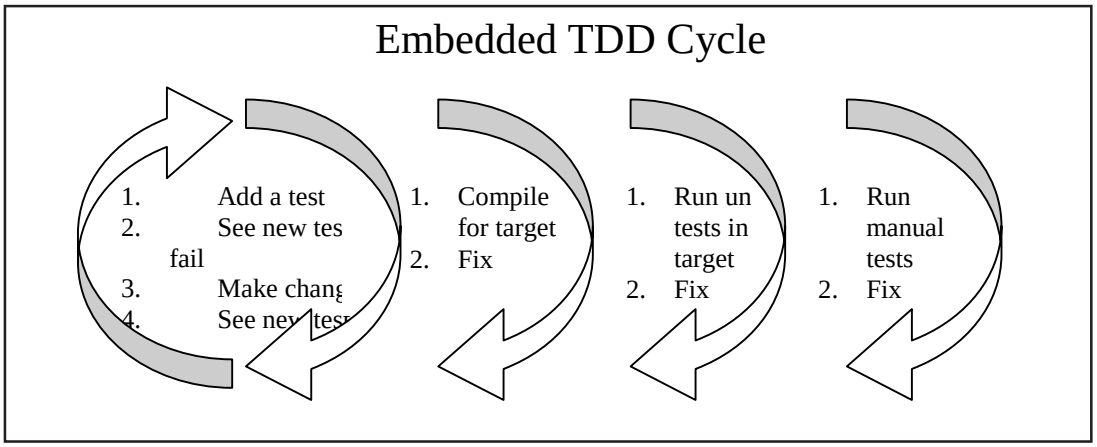
When and where are these tests run? The short answer is as often as possible and anywhere you can. Let's look at a few different situations: prior to target hardware, limited prototype hardware available and full target hardware available.

If it is early in the project cycle and we do not have target hardware, we could run our tests on our development system, with interfaces mocked out to isolate the application from the hardware. We could use the development system's native compiler. This sounds dangerous due to compiler variation; so, I would add another step to the embedded TTD cycle: periodically compile with the target's cross-compiler. This will tell us if we are marching down a porting problem path. What does periodically mean? Code written without being compiled by the target's compiler is as risk of not compiling on the target. How much work are you willing to risk? A target cross-compile must be done at least before any check in, and probably whenever you try out some new language feature you have not compiled before.

Once we have a limited prototype, we'll continue to use the development systems as our first stop for testing and periodically compile for the target as above. We get feedback more quickly and have a friendlier debug environment. Now we'll periodically run the unit tests in the prototype. This assures that the generated code for both systems works the same. The test should be run at least prior to check-in, and more frequently based on how long it takes and how much work is being risked.

If some of the real IO is available on the limited prototype we'll start to add some tests for the hardware or that use the hardware. Automated tests are more difficult to create when the real hardware is being used.

The tests may involve external instrumentation or manual verification. We want to make our tests easy to run or they will not be executed. This leads to a design where the hardware dependent code is very thin. Our goal is to automatically test most of the system.



The discussion for the full target hardware is much like the discussion for the limited prototype; except that now we can do end-to-end testing. Ideally the end-to-end testing would be automated, but this is often difficult to achieve. One big challenge in end-to-end testing is running the system through all the scenarios it has to support. Rare scenarios have to work, but how do we get the system into a particular state and have the right triggering event to occur? Controlling the state and triggering certain events will be easier in our test environments. Our Mock Objects can be instructed to give any response needed to exercise the code. A common place to end up is that the end-to-end test is a subset of all the supported scenarios that demonstrate that the parts of the system are talking to each other properly. A combination of automated and manual tests is needed. The development systems tests never become obsolete, even though the real test bed is available.

SUMMARY

Using Object Oriented Design Test Driven Development can provide embedded software engineers a valuable test bed for their software. These techniques can be used almost out of the box for embedded software development. But some additional steps are needed. If I have made this sound too easy, keep in mind that there are some significant challenges that have not been covered: issues of concurrency, timing constraints, testing a large application and how this fits in the bigger picture. I'll address these issues in another paper.

ENDNOTES

Gansle, Jack, The Art of Designing Embedded Systems, Butterworth-Heinemann, Woburn MA, p.48

Tim Mackinnon, Steve Freeman, Philip Craig, Endo-Testing: Unit Testing with Mock Objects (tim.mackinnon@pobox.com, steve@m3p.co.uk, philip@pobox.com)

Beck, Kent, Test Driven Development By Example, Addison Wesley, 2003, P.1

Using Agile Testing Methods To Validate Firmware

Bill Greene

This paper describes applying Agile software development practices to the development of firmware for the Intel® Itanium® processor family, particularly in the area of testing. It describes several unique testing challenges presented by embedded firmware development, as well as our experiences using Agile methods to address them. We found Agile approaches well-suited for our embedded project, though most Agile methodologists are from very different object-oriented and pure software backgrounds.

1. INTRODUCTION

Since you are reading the *Agile Times*, I can assume you agree with the statement from the Agile Manifesto that "we value working software over comprehensive documentation" or other project artifacts. I think it is important to emphasize the word "working" in that statement, and to acknowledge that quality working software requires a focus on testing. Agile methodologies provide this testing emphasis in several ways. XP advocates unit testing and the "test first" approach to development. Other Agile methodologies advocate iterative development, where testing isn't a phase you get to when you're done with coding (and don't have much time left), but is rather more tightly coupled with the code development itself.

I am currently leading a team of seven developers who are implementing the firmware for a next-generation Intel® Itanium® processor. Our code has evolved along with this new processor architecture to now consist of 300,000 lines of Itanium® assembly and 30,000 lines of C code. This is very different from projects typically cited as examples for XP or other Agile methodologies. Embedded firmware has a number of unique characteristics, but there are also similarities with other pure software efforts, and in many cases Agile approaches are a perfect fit for the challenges of embedded development.

On a previous firmware development project, we faced a problem many software development groups encounter: embarrassing test escapes. A root cause of this problem can be the "code and fix" development style, where developers write code and then proceed to fix problems that are found by a testing group or from customers in the field. In our firmware development, the cost of this approach is prohibitive, as it becomes very difficult and time-consuming to root cause firmware problems when the problems often require elaborate hardware configurations and system timings to be reproduced.

Test escapes are the most frustrating when they could have easily been found with simple focused tests, and this indicates lack of coverage in some areas. In our group there were several reasons for a lack of test coverage. Often there was too much reliance on a separate processor validation group, which gave a false sense of security that problems would be found before our firmware got into the hands of customers. After making a release, I observed an attitude in the group that since no one was reporting any problems, everything must be OK. Also, changes were sometimes made without being explicitly tested, resulting in more problems arising out of the code that was intended to fix the original problem.

2. WHAT MAKES FIRMWARE UNIQUE?

For some reason, many firmware developers are not keen on the practice of testing. There is also a common feeling among the general processor development organization that everyone wants to do design, but very organization that everyone wants to do design, but very few people would choose a validation job as their first priority. I joined the Intel processor validation team in 1994, and the testing group would not have been my first choice. But looking back, the value of the experience was that I now see the value of testing and am always thinking about how to test my code as I write it. I firmly believe that testing is a skill that can be learned, but for some reason we mistakenly assume people have this skill without any training. In the processor design group, it was understood that no feature can be said to be in the design until it is tested. This mindset didn't seem to carry over into the firmware team. This may be because it is much easier to make firmware changes if you make a mistake, but the cost of making a hardware change is much greater.The first silicon for a next generation processor design is not available for several years, so until that time the firmware must be ltested on an instruction set architecture simulator, or the actual processor logic design model itself. The biggest imitation of simulators is their speed: architectural simulators run about 100K instructions per second; the logic design simulator runs only 10 clock cycles per second! But a benefit of the architectural simulator is that it has powerful capabilities to automate the running of tests, through batch scripts and a Perl API that can be used to control the simulation and track pass/fail status.

Since the hardware design is being done in parallel with software development, and because of the high degree of interaction between hardware and firmware, we can't just test the firmware standalone because the underlying hardware is also changing. This forces us to at least test critical sections of firmware on the logic design model regularly.

The nature of assembly language also introduces a lot of potential areas for problems. Anything can go wrong – even your simple "for" loop can have bugs. Defects at this level are a non-issue in higher-level language designs. There are very few language-imposed organization or structure requirements in assembly, which can create unmanageable and error-prone code. There is an even greater need to test the low-level code implementation in assembly.

3. AGILE METHODS TO THE RESCUE

We applied a mixture of practices from Scrum and XP to our project. We have realized numerous benefits, but the biggest impact was the improvement in quality achieved by the **team focus** on testing. This focus improved everyone's motivation to invest more effort in test automation, tools, and infrastructure. This in turn made it easier to develop tests, and as more tests were developed people could see the quality increase, and the cycle fed upon itself. More about our experiences with Agile development will come in a future article.

The technique of writing tests first promoted an amazing shift in mind set. Tests were no longer an after thought which may or may not find problems in completed code, but became a tool to support the coding itself. The tests provided an intermediate level specification, with more detail than the API defined

by the Itanium® architecture. It forced us to think about what the code should be doing before coding, and made the coding process itself flow easily once the tests were in place.

On previous projects we had only developed functional tests which exercised the architected interfaces to the firmware. Unlike other languages for which test harnesses are widely available, there is no "AssemblyUnit" testing framework. For our current project, we had to write our own tools which allowed us to run setup code, branch to an arbitrary starting point in our firmware code, execute to an arbitrary ending point in our code, and then branch back to the test's checking code. With this capability it became easy to test any snippet of assembly code on the architectural simulator. The ability to code unit tests gave us the capability to test components at a low-level and then put them together into more complex structures with confidence. And the feelings of confidence reinforced the value of the testing.

Scripts were also developed to automatically run and check the results of our regression suite. The suite was broken into a 2-hour nightly regression and a less comprehensive 10-minute suite that is used by developers as they code. The ability to run a group of tests with a simple command allows tests to provide timely feedback.

4. CONCLUSION

The focus on testing that Agile methodologies promote has made all the difference in our project. The "test first" and unit testing practices support this focus and provide the real-time feedback to create quality code from the start. Our investment in tools and automation to support the testing effort has really paid off, as it enables quick and easy test development. Now that testing is made easy, developers actually enjoy writing tests and seeing improvements in the robustness of their code. It's a vicious circle of increasing quality!

Hard Questions For Hard Projects
Mel Pullen

INTRODUCTION

Here I am, a neophyte ScrumMaster, trying to get an organisation to use Scrum and Ken asks me to edit the section that everyone will turn to. I could have refused, but I know it's a way of keeping me going on this uphill struggle. Hard projects? Trying to get Scrum accepted is a hard project in itself. Every project is going to be a hard project, unless it's already a Scrum site.

Well you can help. I'm writing this section for the first time in this newsletter. Through subsequent editions of the newsletter I'll report on progress in my company and I'll cover any points I've thought about. In future, I'll write about other people's experience with difficult projects. So you can help with questions, answers and your perspectives on what I've written.

WHAT MAKES PROJECTS HARD

We know from the Scrum theory that variability is what makes feedback at many levels necessary. The major parts of variability are caused by:

People

In my company's case, like in many software companies, the culture encourages innovative people rather than teams. The management notice individuals who carry out heroic programming feats. We still remark on and reward the late night hack, the weekend work. Who wants to work in a team if they'll miss out on a cheque to pay for their next holiday.

More generally, despite software being a catalyst or an agent of change, developers are probably the most reactionary. They think they have a job for life and it had better be a damn easy one. There are lots of new ideas to play with, or to learn about, so they want time to meander around. Too many developers have heard of the instant millionaires when companies float. Too many have seen management imposed deadlines come and go and still the company continues. So what's the big deal about deadlines? They're made to break aren't they? If a team works too hard they'll only get more work. If they consistently miss deadlines, it has to be because they don't have enough resources.

Management don't want to scare off developers as they quite often hold the whole of a design in their head. A long development cycle means that a developer holds all the aces until they deliver some functionality. So management try to keep the projects under control the best they can and avoid too much conflict. They rarely assess the productivity of their staff nor compare it to any industry norms in case their company is shown to be low in the league tables. If they have to do that then the low numbers are justified in terms of other attributes of the delivered software. It is better because of the great quality, reliability, or stability of their product or how stringently the other non-functional or compliance requirements are applied.

So we have management that measure as indirectly as possible and developers that develop as indirectly as possible. The waterfall process control model provides an illusion of control, when in fact the only way this control works is by achieving next to nothing. Some deliveries are pitifully small, with the software written as an incidental to the long periods of functionality and resource negotiation. We end up with projects, project managers and project plans. And no project.

Technology

My realm is mobile device software. Smart phones. They eclipsed personal digital assistants. Now they are being eclipsed themselves by communicating ultrapersonal computers. As with all technological innovation, the next generation appears before the demise of the current one. Dinosaurs take some time to die out, but die out they always do. It takes nearly two years to get mobile phones to market, yet their window is between 3 and 9 months. The end point is fixed so you get more life out of a product the earlier you get it to market. So far, every smart phone I've had experience of, has been late. Or canned.

Imagine readying a computer system with a general purpose operating system, with nearly every protocol that is in use at the moment and with the latest silicon to display video. All in a couple of years. Now imagine getting the OS ready for multiple customers when new protocols are happening so fast your whole company could spend all its resources attending standards body meetings. Each customer has their own agenda and if it's a competitive protocol they might not want the OS supplier to put it into the core because then the other customers won't get it. The battleground is what is not in the core. You know no customer has requested a specific technology. You know your product department has put it low down on the priority. You know it takes months to develop through the waterfall process you have. What do you do?

HOW TO DEAL WITH HARD PROJECTS

Leave, find another job. Simple, really. Well, I've considered it. A number of times.

I have been trying to get my company to use Agile technologies for a long time; over three years. I think the idea of the company is good, I think their strategy is good. The execution of the delivery of software can be improved. That's why I stay. I could leave and go to a firm that wants a Scrum Master, or find a consulting firm. Then I wouldn't be working in my specialist area, which is mobile communications.

When I first arrived at the company I was just a developer and my interest in Agile development methodologies was just part of the passion. I got teams to use CRC cards. I got people to try pair programming. A few times I've got people to do test first programming. They try them then go back to their comfortably numb way of life. I learned a vast amount by attending the Scrum Master course. I wouldn't have known to talk to management; project or line, about implementing Scrum without attending the course. I would still be talking to the developers. As I learnt more about Scrum I realised it was more than just software development, it's process control. So it takes a while to come to Scrum with an understanding of what it is, and what it is not.

This is important. The developers are likely to be the most resistant. If the project and line management staff are recruited by being promoted from developers then they are likely to come out of the same mould. These people can be talked to, but you'll get little active, honest support from them. I spent a long time getting teams to consider using Scrum, and they only considered it because they saw it as a bargaining chip. They want more resources (see above), so they are happy to trial a new way of working. The thinking perhaps goes along these lines:

> "If I offer to do Scrum I'll be able to ring fence some development resource. I can than say I'm slower on defect fixing in the rest of my team because resources are taken up doing the Scrum trial."

Or worse still, it may be:

> "I can offer to trial some of Scrum and do it half-heartedly. I'll use it as a pool of resources I can pull people from when I have an emergency in the real project I'm working on. At the end of the project I can show that Scrum didn't work so the whole company will be able to reject it and go back to the easy life."

This is from a developer or team leader's point of view. What about the middle management? Are they friend or foes? Unless middle management are wholly behind Scrum, have made an effort to find out about it, then they're against it. That's my opinion.

Scrum will make middle management look stupid unless they participate. More like a headless chicken than a chicken. Self managing teams make middle management redundant. If the managers have come from software development they may have to go back to it, if not they may be out of a job. A cruel irony when most organisations are so under-resourced they could re-use almost anyone who knows the business.

So, to end this section, some advice to myself. I'm writing this as I do battle myself. Make sure senior management know about Scrum. Make the message simple. Scrum helps your bottom line. You deliver what your customer wants. Catch them just after they've had a *fist banging on the desk* meeting with the customers. Offer them a solution. Better yet, offer them two; do Scrum in-house or do Scrum Offshore.

EXAMPLE QUESTIONS

When I was asked (press ganged) to submit material for the newsletter I polled people who had expressed an interest in Scrum. I'm not the first in my company to talk about Scrum and certainly not the most eloquent. However I was concerned about doing it rather than talking about it.

As Ken says, he doesn't tell the companies what to with the technology, they shouldn't tell him what to do with the process control. So I wanted the company to really try it, not pussy-foot around. My idea was that without doing it within the company we would have no comparative information. So, I'll get back to the comparative part later on. First I'll describe what I've done so far, then I'll look at the questions posed by the people who responded to my request.

How do you get a company wedded to the waterfall method to successfully use Scrum?

This is my experience trying to introduce Scrum to my company. This is part 1 because, assuming I ever get asked to contribute again, I'll update this section with parts 2, 3 and 4. I also use the word successfully because I don't want to introduce Scrum in such a way as to allow anyone to say it was a failure.

So, some background. I arrived in Symbian, a joint venture, in September 2000 after working in a couple of the owner companies as a contract programmer. I knew about agile development techniques and had used CRC cards. I also knew about throughput accounting or Theory of Constraints. When you work in lots of companies you learn about a lot of methodologies. Well, I tried to follow along the lines of the other people who have tried to introduce agile technologies before me. As I've mentioned I'm a pragmatist, so I wanted to do it rather than talk about it. Apart from which no-one makes decisions in most companies. You have evaluation projects and steering groups to ensure collective responsibility. Management by committee it's called.

I discovered the ScrumMaster courses and asked my company to send me to the first European one that was held in Milan in July 2003. I then started to make presentations about Scrum to anyone who would listen. I also put together a page about Scrum on our internal Wiki. I can't show these presentations outside of the company without written permission. Apart from which they just list the basic reasons for using Scrum; you know them all, or should do:

Scrum concentrates on exceptions (this comes from the Theory of Constraints). We leave cross functional teams to manage themselves. By continuously monitoring and correcting development we get:

- Improved quality
- Better match to requirements, defects and change requests
- Empowerment of development staff

Among the presentations I have made I made one to the eXtreme Tuesday Club (XTC - http://www.xpdeveloper.com/). So I could take the presentation public I put it on wimpypoint (http://philip.greenspun.com/wp/display/1704/). I won't duplicate it here, it's not got too much relevance here as it's mostly about general agile technologies.That's just about the end of part 1.

I've got one team leader interested running his team as a Scrum project. I've got a few project managers starting to take an interest. I've received words of support from some line managers. Myself and a colleague have put together a presentation and taken it to a customer.

Line managers don't bother to turn up to the presentations, the customer is more beaurocratic than we are and estimates it will take a few months to make a decision. So far it's been four months. The culture of the action item prevails As long as there is no meeting to follow up on an action item it need never be done.

Now I have to take the bull by the horns and present to the board. For that I'm waiting until they have a crisis on their hands. I'll report on progress later.

Oh, didn't I tell you? Companies that take on Scrum fall into two camps. Either they're pro-active or they're in crisis and will try anything that will help them get out of their mess. See if you can guess from the questions I list here which camp my company is in.

Project manager questions

This is asked by most people and is the one that most project managers ask.

> "How do you make a large, complex, project cope with cross functional dependencies at design and integration phases?"

Most project managers in our company think that writing an operating system is more difficult than

applications. As an aside I think it's easier because we rarely have to write a user interface, just throw together an API.

The people who think systems are difficult see the problems of interactions as one of the biggest. Certainly interactions between components in our operating system take their toll on our developer's time. However this is expected and should and is factored into plans. If multiple components are being changed simultaneously, of course there will be interactions.

Well, armed with Ken's latest book the answer to this is simple but I'll leave the problem here. The solution is also mentioned in the ScrumMaster training courses, so go on them if you haven't already. You put people from the various projects into the other teams. Either part or full time, or just to sit in on the scrum meetings.

The major part of this problem is solved by the basic nature of Scrum. Frequent inspections and adaptations. If code is released to interested teams every month then only a month's worth of work will be affected by an unforeseen interaction.

How it's done at the moment is for architectural designs and functional specifications to be written and then read by the teams who might be affected. The teams then plan to carry out integration work at various intervals. This involves some of the dependent team (the protocols and system layer code is assumed to be core) spending time running their test suite on a build with the new code. Together with finding bugs and re-submitting the code this can take up to a week. Mostly paper work, so the management know that the integration has succeeded.

To a project manager, just having integration as part of the backlog sounds like no design is being done. The essential feature in this case is the cross functional nature of the sprint meetings.

All of these rest of these questions came from one project manager. He's sufficiently motivated to ask me to present to his team. What do the questions say about his concerns?

> "How does Scrum provide a higher level of confidence in meeting the end date compared to traditional technologies?"

I think responding by saying this is a non question is too glib. The fact is this project manager needs to know how Scrum can give greater accuracy in estimates. It's not enough to say that the nature of complexity is such that any estimates will be inaccurate. What is needed here is to indicate that by delivering in increments the estimates can be verified along the way rather than having to wait until the end. With a waterfall approach it is unlikely that another round of estimation will be made unless the project hits delays.

> "Is Scrum better for low risk environments where we train people?"

I've not done my job by explaining that Scrum is actually good for high risk projects. Better than anything else. Because it is adaptive. I've also not explained that the self management epiphany the team experiences can't be trained.

> "How does Scrum impact the career path / does it mean a redefinition of traditional promotion values?"

This is a difficult question to answer honestly. In my opinion, the western programmers have become complacent. Hubris is the correct word and it is appropriate is considered one of the three virtues of a great programmer (Laziness, Impatience and Hubris). There are developers in Eastern Europe, the Middle and the Far East that can't wait to take our jobs from us. I looked at a web site just this month created by a company in Lithuania. It was a good site, fast, clean. And it cost the owner $470 to have it built. Offshore development is here. It's only a matter of time before they offer Scrum. Our senior managers

may take Scrum from an offshore company when they won't take Scrum in-house. At least they get to fly off to an exotic location for the Sprint Review and Spring Planning meeting.

I've had discussions with Offshore Development Companies (ODC) that offer project management as well as the execution. In the context of Scrum the Scrum Master really should be with the team to carry out the daily Scrum meeting. So the answer to this question is get on board. Scrum offshore will look increasingly attractive to senior management as the wage bill goes up and the productivity goes down.

> "In terms of a defined and unchangeable scope - can it reduce the timescale for development?"

We do have this in our organisation. In fact quite often. They're called protocols. We have to code a module that plugs into a framework that already exists and provide the new protocol according to the specification. This specification is subject to a standard. A good example is Real Time Protocol (RTP) or Motion Picture Experts Group (MPEG). I think it will reduce the timescale, in the long term. I think this because not everything will be thought of at the beginning.

Functionality that is demonstrable to the customer in terms of a protocol require that other artifacts are developed. In the traditional waterfall approach these artifacts may not be developed at the beginning because no-one appears to want them. Only at the end does the customer realise they may want extra functionality and then raises a few Change Requests. An obvious extra when considering protocol modules is the control – the ability to turn the protocol off forcibly. Battery operated devices have a habit of stopping.

> "How can Scrum be used to more effectively manage costs and benefits in a fixed cost environment?"

I really do think this is a simple question. Anyway it is adequately answered in both of the books by Ken Schwaber. Normally you will be bidding against a known set of requirements. You have to plan enough to be able to create a backlog in priority order from these requirements. You can offer a quote based on these estimates, knowing that some of them are inaccurate. That's no different from any other approach.

You indicate to the customer they will get a change to see progress on a monthly basis and to change the priorities of later deliverables. They will be able to decide to terminate early. You point out that the Pareto principle (80:20 rule) means that the customer may be happy with the functionality much earlier than with a one hit wonder; they can stop the development at the end of any sprint as there is potentially shippable work there.

If they drop their frills they get to save money. No other mechanism offers that option.

> "As a customer of a company who uses SCRUM how can I be confident that they will be better than my normal supplier using tried and tested traditional methodologies?"

Isn't Scrum tried and tested? I thought it was companies that were unreliable, worse yet, the people in them. Again, the incremental demonstrations of working functionality gives a customer greater confidence after just one increment. With any other company they have to wait until delivery. So due diligence prevails. Ask around.

As an aside and this is more a comment to the project manager from my company who raised this question. Senior management would manage the risk better if they went out to a company who has carried out Scrum development before than to train developers in-house. Programming is programming, only requirements make it different. Every technology is different, every new bit of functionality has a

learning curve so it's probably lower risk getting an outside Scrum company to do the first Scrum project. A sobering thought.

"Does Scrum improve system reliability and performance?"

If they are requirements, they are addressed. Need I say more? After an increment is demonstrated, the customer may decide to put these attributes higher in priority for the next sprint. That can't be done in waterfall.

"How costly would Scrum be in training to move people across to?"

One consultant to come in and train for two days. Is that cheap enough?

The real cost that this person is thinking about is the people cost. As I've mentioned before, developers are some of the most reactionary people. The cost of training will be high because many people will insist they are not capable of working in Scrum teams unless they have first been trained. Once they are trained and then discover they can't sit and surf the web for the best part of the day they'll want to find another job. We may get a lot of "attrition" shortly after Scrum development practices are instigated. It's too much freedom. They don't want to be responsible for their work. They've had project managers, team leaders, technical leaders, technology architects and line managers to tell them what to do. Why change and have to think as well?

"What skills sets need more emphasis to enable Scrum?"

Thinking. I think. People are going to be working together so their sense of humour may get exercised more. In terms of development teams, I think it is immaterial. Teams are cross functional in their composition. So a designer may have to help with documentation or testing.

"Which types of developers find SCRUM more enjoyable to work under?"

The one's who would like to work a real 37 ½ hour week then go home knowing they've done a good weeks work. Again this is likely to cause a culture clash. In cold, damp, drizzly London developers may prefer to sit in the nice, warm office for 10 hours a day. Then slide off to the pub for a few pints before a curry or a kebab and then bed.

I've already had developers refuse to work in a team, claiming that they work as a team yet each do their own development work. We come back to the heroic programmer mentality. I work late most days because I find it difficult to get the work done in an environment where people want to discuss their work. I used to arrive at a normal time (for England – 8:30 to 9:00).

I ended up doing 50 hours, so at the moment I come in late. That way I keep my hours down. My manager regularly starts before 8:00 and finishes after 18:00. Luckily the culture is changing and the newer developers don't feel so devoted to the company. They see it as a job. They're no longer being sold the myth that they're in a high tech startup that could make them rich.

People find it hard to let go of their old ways of doing things. If that means sitting at a desk and working on a piece of code, or worse still crafting a functional specification, then it's going to be hard to get them to work in a team where their progress is open to scrutiny every day of the project.

Developer Questions

Most developers don't know anything about Scrum apart from it being 'another software development methodology'. In our company they've heard about lots and the development process control system we have is supposed to allow teams to use whatever development methodology they like. How they do this when they get all of their requirements at the beginning and system test only test once they've delivered everything is possibly another hard question.

A deveoper named John really does know about Scrum. He's given a presentation about it. He's a PhD student, and the company allows him about a day a week to continue on his dissertation. He's passionate about everything in life, particularly programming. He gave me this response, well nearly this. I've tidied it up a bit, but not too much. John is not a native English speaker. English is not his mother tongue. By the way, that's one of the many virtues of working here. I can go to someone and ask them about their perspective on an Internationalisation problem. I can ask about the culture in a country. I can even get a translation done easily.

> "The major problem with organisations and projects during avalanches is momentum and inertia."

In that respect the most difficult problem is to first to stop and then to act after giving it some thought, then briefly stop again and iterate. Symbian projects are no different; first we need to stop and then do Scrum (or whatever). Now, as far as Scrum goes it is excellent because it allows you to steer momentum and inertia while it forces you to build and control them.

By far though, before we can even remotely do so, we need to stop in the middle of an avalanche... and that involves some serious physics" So, John likens the state of our company to being in an avalanche. That's an interesting perspective. Does this mean the staff don't have time to think? That they won't try anything risky? Well, we can't stop the avalanche. As John says, that involves some serious physics. So we have to plan quickly and adapt. Hey, that's Scrum.

LAYING THE GROUNDWORK

Are you going to be in for the long term? Are you going to keep trying to get your company to consider trying Scrum properly? If you are you have to make sure you're prepared. I'm quietly collecting metrics so I can create productivity figures of the existing development methodologies. I can then offer the assurance of comparing the productivity of the existing processes with Scrum development.

I've found a way to get development teams to make estimates using function points instead of just numbers. I can measure historic figures from our code base. I will correlate estimates against the time taken for each of the sub projects, so that by the code delivery phase of the next project (about 9 months) I should have some productivity figures. I'll also create some figures from working with the teams. Then I should be able to get senior management to at least consider Scrum if I can predict a huge productivity increase. I'm always learning, although I don't get given any time in work to do any of this. This page looks good on metrics: http://sern.ucalgary.ca/eeap/wp/bk-position-2003.html.

So a final suggestion. Should we start to collate productivity figures to be able to present management with success stories? I know I need them.

Implementing Agile
Lisa Crispin

Even if you have a lot of experience using Agile practices, it can be tough to introduce Agile practices and processes into an organization that is less than Agile. In this section of the *Agile Times*, contributors will share their successes and mistakes in getting their organizations to adopt Agile approaches. Some areas we might explore:

- What fears people have around 'Agile' and how to deal with them
- Showing developers how Agile practices will help them
- Measuring success of a new practice or process
- Patterns for getting your organization to adopt new ideas

- Potential pitfalls and how to avoid them
- Showing managers the ROI of implementing Agile software development
- Leading by example

Last year's Agile Development Conference had a well-attended and lively technical exchange on this subject. See the notes from this discussion at http://home.att.net/~lisa.crispin/Introducing_Agile.htm for a look at the types of ideas we might talk about in this section of the Agile Times.

We welcome your contributions on this subject! Please contact Lisa Crispin at lisa.crispin@att.net. Watch this space in the next issue for a stimulating article on introducing Agile!

Sociology And People
Boris Gloger

Kent Beck wrote "Embrace Change" and by all means, the Agile development methodologies try to change the way we do software development in this century. Weinberg had written "programming is a psychological activity" and he was right, because software development was seen as being done by individuals who where tight together by manager.

In these days, we know that creative work is most of the time done by individuals who work in teams. Teams can be far more creative and productive than a single individual can be. Agile software development is about creating business value by utilizing the creativity and abilities of teams.

Linda shows us in her brilliant way we can use "change patterns" to embrace and to enable change in our organizations. She gives us tools that will help scrum masters, project managers and last but not least software developers to get change done. Deb found a book by Tom DeMarco. She tells us the story why "Slack" - or in other words "space" is necessary to change things.

Sociology is the science of people in groups, the science of their interaction in groups and the science of the interaction of groups: or in short the science of the way people are working together. Agile software development addresses in its basics the way we think about the way people work together. I my own article I will try to give you an idea in which way social laws rules our daily behavior.

I call you to participate in this discussion. Send me your feedback and your ideas about the human factor in software development.

Patterns For Introducing New Ideas Into Organizations
Linda Rising

Volume IV; Issue 1
February 2004

Let's say that you've just read an article about a new, cool approach that you are certain could save your organization gazillions of <whatever currency> and that you're all fired up, ready to convince everyone on your team to try this <whatever technology>. Then reality sets in. Those people won't listen! Maybe you've tried talking about other new things in the past, only to be met with resistance. Or maybe you had limited success, a few were open, but the excitement died down when real work had to get done and then everyone got busy and forgot all about your new idea.

My good friend and colleague, Mary Lynn Manns, and I have been collecting patterns for introducing new technology into the workplace. We have been working for several years, capturing successful experience, submitting our work for review, continually revising, growing, and learning strategies from those who have introduced new ideas—including our own hard-won lessons learned. The book that describes these patterns and several case studies will be published by Addison-Wesley in 2004. An early edition of all the

patterns can be seen at: http://www.cs.unca.edu/~manns/intropatterns.html

I'm going to describe a few of these patterns—those that might get you going. I'm anxious for any feedback or stories you might want to share. These patterns are "alive" and will continue to grow long after the book is published.

Let's begin with the first, and most important pattern, Evangelist. The name has a "religious" flavor and that is intentional. We've found that unless you are truly passionate about the new idea, others will not be convinced to leave the tried and true ways and follow you. There's another piece to this rationale. If you don't have faith in your proposal, then you won't survive the bumpy road to grass roots adoption. There will be successes and failures along the way and you must celebrate the former and withstand the latter. Only a sincere and abiding belief will carry you through all this turmoil. You must have passion for the idea and share that with others.

As we collected information about these patterns, someone suggested to me that many of these are based on influence strategies. This area of soc ial psychology (the psychology of groups) was completely new to me, but now that I have read several books and articles, I can see the data that supports our patterns and that has helped me understand why they are patterns—why they are successful solutions to problems we face in introducing new ideas.

Let's look at some influence strategies for Evangelist. Research shows that we are more open to new ideas when they are presented by someone we like. Studies in this area show that we tend to like those who are like us and those who are attractive. While there's very little we can do about our physical attractiveness, there's quite a bit that we can do about being "like" others. Looking for commonality is the key. Don't set yourself apart, even within yourself. If you go into the situation telling yourself, "These guys are bozos. They won't get it. I've read all this stuff and I know the answers. Why don't they just listen to me?" you're doomed from the start. If, instead, your attitude is, "I can identify with what these guys are going through. They're smart and want to do the best job they can. If I share my story with them, I think they'll hear me out." That's it—tell them your story. Don't regale them with facts. Facts are good. They keep our minds busy, so our emotions can figure out what to do. [Dale Dauten, one of my favorite authors said that!] Share your experience and let the hearers see how you are just one of them and it will go a long way toward convincing them that you have something useful to share.

A pattern that can help in this regard is Just Do It. Instead of waiting for the right moment, experiment as much as you are able with the new idea in your own work and then report to others. Don't overpromise, but actual experience is convincing, much more so than a case study in an article. However, if the case study is from a respected author or company, the pattern External Validation says that this kind of information can be convincing, but, remember, it must be respected by the target audience. This, again, is based on influence studies—it's the principle of authority. Like it or not. Believe it or not. We are influenced by respected authority.

Let's take a moment to talk about influence. Let's address those fears that this is underhanded or some kind of dirty trick. Let's also address those who say, "Well, I'm not influenced by that!"

First, the underhanded or dirty trick part. The influence principles have been shown as a result of controlled research. They give us information about the way we are, about how we make decisions. When you set about to convince someone, you make assumptions about what is convincing. Most of us are left-brained geeks. We believe ourselves to be completely "logical." Therefore, our attempt at convincing strives to follow this model. We make a "logical" argument. Research has shown, however, that we make decisions based on emotion and justify with logic. If you create a Powerpoint slide with a list of bullets that logically leads the viewer through a reasoned argument and make no appeal to emotion and have no

emotion in your presentation, the only people you will convince are the ones who already agreed with you. All good salespeople know this. They are influence experts. They know that success depends on using the influence principles. If you want to convince, isn't it "logical" to understand the best way to go about it?

Now for the "that won't work on me" response. In these studies, there is always a control group – not only a control group for the sake of the experimental results, but also a control group that targets the issue of influence itself. For example, many studies have shown the benefit of having good-looking models in automobile ads. Advertisers know we transfer the beauty and desirability of the model to the cars. In one study, men who saw a new car ad with a seductive female model rated the car as faster, more appealing, more expensive-looking, and better designed than those who saw the same ad without the model (the first kind of control group). The second control in this case is to ask men if they would be influenced by the model, and, of course, they always refuse to believe that the presence of the model would influence their judgment! I see this as the most powerful part of influence strategies—that we don't believe we are influenced! Now that I have been studying this area for years, I see that I am surrounded by it, and, yet, even in the midst of it, I feel pulled to go the way of the influencer!

Since I've been studying social psychology, I've also been introduced to evolutionary biology. What I thought were patterns that experienced change agents had helped document—are based on influence principles that researchers in social psychology have studied for some time. But, underneath these principles—we're hardwired to behave in certain ways. This is scary stuff—but useful.

This is just the beginning. For those who are new to patterns, I'm going to follow this article with another one or two that will help you "read" through this large collection and make sense out of the information in a way that is meaningful for you. Good luck in your attempt to introduce your next new idea, download those patterns, and don't forget to send feedback. Thanks!

Computer Programming Is A Social Activity

"Sociology, the study of human social life, having as its subject-matter our own behavior as social beings" (Giddins), offers us a distinct and highly illuminating perspective on our human behavior. Every team leader, project manager, scrum master or line manager is confronted with us: human beings. With this article, I'd like to start a series of articles about the sociological aspects of Agile development teams; or, to say it in a different way, bring some new attention to team dynamics. I won't claim to be scientifically correct or to be writing scientific articles in this area. Rather, I'd simply like to draw your attention to aspects of team dynamics in a very accessible manner.

On the next pages I would like to show you that some of our daily conflicts between different teams are caused by very old rule sets. Then I would like to answer the question why we use these rule sets. After you have an understanding of these sociological rules than I will try to give you some ideas about how you can *manage* these rule sets. And I will give you a first insight into (at least) my observation that Scrum is working because Scrum deals with our sociological behavior very intuitive (if it is possible to say this from a process framework).

WE ARE ALL SOCIAL BEINGS

We are all damned. All of us are influenced by the fact that we are social beings. In fact, we often unknowingly use fairly primitive ways to communicate as we move from group to group, sometimes invoking roots of behavior that may not necessarily be nice but are effective all the same.

To be a little more specific, we all know about the classic problems we tend to encounter in interactions between, say, a Marketing department client and our software development team. For instance, the software team might conclude, "Those marketing people are different! They don't understand us because they don't have a clue about what really matters, TECHNOLOGY!" Or, we might say, "Those people talk a different language than we do." Often in these situations, teams, and the individuals in those teams, manage these disconnects with the client by simply creating a wall around themselves. And so, behind the wall, we surreptitiously decide that we - the technicians - will simply do what WE think should be done, even going so far as to ignore requests by our marketing clients. Why? Because we are so sure that, "Our ideas are clearly better than the ones from the Marketing Department."

Does this situation sound vaguely familiar to you? It might not be politically correct to say such things, but if you think about how our software teams will talk or complain around the office or in the cafeteria, you know that these thoughts are much more common than we might usually be willing to admit.

SOCIOLOGICAL INSIGHTS IN TEAM DYNAMICS

So, what are the reasons for these situations, these conversations and thoughts? I would like to conjecture a viewpoint or hypothesis that one might not normally expect. In short, one reason is our basic human behavior. There are some anthropological and sociological aspects we should consider if we want to think about reasons for these thoughts and actions.

Whether we want to believe it or not, there is one important need in all of us: the desire to belong to a group of people. The groups we belong to—family, school, sports clubs, work groups—change over our lifetime. But the anthropological fact is that we are truly social beings, and one very strong expression of this is that we want to belong to a group. One reason for this is that in former times, a single person would have had no chance to survive. The worst punishment was always to be outcast by your group. The other reason is that we are born into the world without the ability to survive on our own. If we did not have someone to protect us, feed us, and show us the world, we would die in a few hours.

This fear of not belonging to a group forces an interesting behavior. When people decide that they want to belong together, or when conditions bring them together, they tend to try to converge on one trait or similarity that all team members have. In our IT world, this is usually the application or project on which we are working, or we might be in the same training course together. Or we may congregate along race lines or nationality lines, where we are from China or Germany and therefore we try to find others from our country of origin with which to socialize. Or perhaps we base our sense of group on the department for which we work. And yet, if we meet a group from another company, suddenly it doesn't matter if our group is from different departments; now, we are from one company and this is a part of our identity!

The interesting thing is that software development teams behave in exactly the same way as all these other non-software groups. There is no exception in sociological behavior for software development teams. They try to find one similarity among the group and then they try to protect themselves by showing that they belong together in every aspect of their communications. They start to expand this similarity by using team languages, or they introduce specific rules or a specific way to work. Or, they try to identify themselves by saying that they are the "X" team because they are responsible for an application "X". There are a lot of possibilities. You can say that a group that really feels like a group starts to have group culture.

HOW CAN YOU OBSERVE YOUR "TEAM-CULTURE"

I would like to sharpen your view, to "see" or to "feel" your team culture. The easiest thing to see that your team has its own culture is to have a look around. How do your teammates express the fact that they belong together? Do you go to lunch together? You might have a specific team event every week or month, you might use specific abbreviations, or you identify yourselves with a specific application you are responsible for. Do you have a special self-image. A self-image: maybe boasting about the best application you build, or software only you use, or boasting about the methodology you use.

Please, you need to understand: nothing about this behavior is wrong. It is absolutely necessary and "normal" for humans to act in this way. But what is if you have now a closer look? Can you see the wall that the (your) group has already build around itself? Perhaps it is only a very thin wall, a wall you do not see. Maybe it is more an inner feeling that there is something that is between you and your neighbor team or department.

This wall is necessary! It protects the individuals and it gives the individuals the feeling that he or she belongs (to a group). You can call this a type of security. It is your social safety net. You know you can count on the people within you group. Or in other words: you trust each other. Trust is very necessary for the interactions and the performance of the team. Another aspect: Teammates that belongs together for a long time do have established a team culture based on their history. Their common history establishes a strong relationship. A relationship that the teammates feel they can count on: again we have trust.

But there is another side of the coin. Besides this advantages there are also some drawbacks. One drawback is the language the group creates. As any village has its own dialect of the common language (in Europe this might be more visible than in the US), groups create their own group dialect. We call this group dialect: sociolect. A sociolect is the specific language a group builds by their own. This is a form of code. A code with expressions and meanings no one else understands. For example the way a doctor is talking to his colleague show the sociolect of his specific profession.

Another possible drawback is the way they interact with each other. They might have a way that excludes other people. Maybe they have a special ritual or a way they start the morning or end the evening. And this ritual is seen as a very uncommon way of doing things by the outside.

In all cases, when a team starts to be seen as a team then sociological rules (laws) start to work. If a group has established their security walls, their "moat", then they might be seen as a "potential" problem from the outside. They are different from "us". Unfortunately this means for any social group: they might become an aggressor. You can observe this behavior quite easily if you have a look in your own company. Many conflicts are running about resources, more people, better equipment, more money or better projects. People do have questions in mind like: Why do they get something and not us? Why do *they* get a new team member and not *us*? They are mostly caused by these social aspects.

What happens if such walls are very easy to see? That means if everybody knows that there are borders. In most companies there is a "moat" between Marketing department groups and IT department groups. Fact is both groups usually have a clear self-image. These self-images distinguish both from each other. (And you can easily hear the different sociolects of each group.)

But both groups have to interact with each other. What happens usually? Most of the times we gain a lot of misunderstandings (Such misunderstandings are sometimes based on the fact that the same word has a different meaning in the sociolect of the other group). One reason is that they are not able to interact with each other on a basis both can accept. There is no trust. There is fear that the other group might

want to rule the situation. The other aspect is that each team members of each group will always try to identify him as belonging to Marketing or IT.

This lack of trust leads to a similar problem: The always wanted "cross-functional" project team that is made of people from different departments will need to deal with this lack of trust. A project team member cannot easily accept that he now belongs to a new project team. Each person in this project team will be loyal to its home group in the first time. He might not be able to leave the old group because he has a history with the old group and he has strong feelings for his "home" team.

Another aspect is that each human group always settles in a territory. This anthropological behavior that you can observe in the animals is valid in our daily work environments also. When you replace the word "territory" with the word "department" than you will see this aspect immediately. Every department wants to have the kingship/dominance over a specific aspect. Conflicts occur if someone violates such dominance. It is important to consider this kind of social territory during intergroup interactions. You will create very big conflicts by not understanding that a team gets its self-image from a specific task or responsibility. For example, you want to be nice and you want to help the other group with a task they usually are responsible for. Then they might get offended, because you scare them. What if you take over what they do? What will be their core then?

HOW CAN YOU OVERCOME SUCH PRIMITIVE WAYS OF BEHAVIOR?

What can a team leader or project manager does to avoid conflicts that were caused by the fact that we are social beings and that social beings follows very "old" rules. I will not give you a guideline or a handbook. There is no „How To Make ..." that is able to give you the guidance you might expect. But I can try to sharpen your understanding of these processes. I believe understanding is most of the times the first step to change. So please do not consider the next ideas as musts or tips.

First - Understand your type of walls. Start by observing your own team and its structure. What kind of team culture do you have? In which way do your team demonstrates that it is a team? Do you have a specific process your team is proud of? Do you have a specific communication style that separates your team? For example we are using English as team language in IT department of an Austrian company. Do you have a specific way to show that your office is your office (territory)? For example, we put a big picture of the Scrum methodology on the wall and we plotted our Sprint Backlog in a way that everybody can see it. It is viewable that we "own" this part of the office. (There are a lot of people around us that do not like this.)

Second – Increase you ability to communicate with the outside. If you know in which ways you segregate yourself from other teams, you might be able to see possibilities for increasing the communication with the outside. The goal is *not* to destroy the walls. That is not useful. Our social behavior wants to protect us. It might not be appropriate way of acting in a specific situation. But that does not mean that this way of action is wrong at all. On the other side you cannot neglect: Your group is different from other groups. You do not want to create a symbiosis with other groups.

On the other hand you want to establish and enforce a better way of interaction. One possibility is to identify the similarities that you and the other group have. There is always at least one. Based on these similarities you can start to create a relationship. The other group will see, that you are not so bad, because you are similar to them (at least in one specific thing). Imagine this first similarity as the first gates in your both walls. Now you can build a street between both gates. Maybe by using this street more often (you have more successful interactions) you will find out that there are much more similarities.

Another possibility is that you can create a common enemy. This is a very seductive track. It is working very fast (- the dark side of power). But as most of the fast solutions this is combined with big costs. What happens if the enemy is gone? And the problems of allies is always that the do not trust each other when circumstances are changing. It is a very fast but week way.

The next possible way is to structure the teams so that both teams want the same aim. This is the positive way instead of using a common enemy. This way is much harder. To go this way would mean that you have at least one working relationship. Based on this you can start to create a common goal.

Another way: The way of diplomacy. You manage the interactions between territories (or interests). You could try to define the roles of both groups and try to minimize interferences. I.e. marketing creates the requirements and IT builds, no discussion. That is the way to deal with the different territories correctly. I can only say this works but it needs not to interfere in the others domain.

Third – You need to build on your common successes, on your similarities. The above techniques are useless if you change your strategy to often. Stay reliable. It is hard work to try to overcome the sociological rules. For example if you are working in a cross-functional team it is very difficult to live the fact that you are now in another team.

It is obvious, that all these insight can be used to create an internal team spirit. A project manager can start building such a team spirit by saying that his team is outstanding, or all the others are incompetent, useless or something like this. In this case you use the fact that people stays together in case they have a common enemy. Or you use the fact that they feel good as long as they belong to the "more competent" group. It does not matter if this is objectively right. It will work as a foundation for team spirit as long as the team illusion holds. These points should not be considered as hints but as examples to make different kinds of behavior visible. All of us behave this way, sometimes unawares.

SCRUM AS ONE POSSIBLE WAY TO OVERCOME "THE WALL"

As an Agile developer (a PM for me is a developer also) you are already aware of self-dynamics and the possibilities to enhance the quality of communication with your clients, your teammates and your management. If you accept that we are social beings and when you are realize that we as social beings follow our basic instincts — very old rules — than you might be able to increase the collaboration skills with the "enemy" further.

Scrum is a very interesting way to overcome the "wall." Scrum deals with the territory aspect by clearly define roles and responsibilities (chicken and pigs). It deals with the trust aspect by showing at least every month what the project team has delivered. It deals with understanding and communication problems with Daily Scrums and an onside customer. Scrum deals with the problem of not understanding the viewpoint of the others by using the working software as visible product. Scrum creates a culture – a way of "how we would like to work" by establishing the Scrum framework (and by having the Sprint Feedback sessions).

OUTLOOK

Weinberg mentioned 25 years ago that programming is a psychological activity. He was right. In our days programming business software applications is not possible without the combined effort of many persons: so we need to rephrase this statement: programming is a social activity. The sum is more as its parts. But to be able to understand how this sum is working we need to understand our social behavior as much better. In this article I was able to show you only very view social aspects that leads to team dynamics we are not always able to control. I explained why people want to live in groups, why groups need to protect themselves against other groups, in which way groups create an identity for its members and in

which way these facts might lead to problems. In future we might be able to understand these dynamics much better and we might see that the "silly" behavior of groups is in fact very rational. In my next article I would like to show that Scrum is a kind of guidance that gives us some very simple tools to deal with the social aspects I described in this article. I hope you have found this article interesting – please sent feedback to boris.gloger@chello.at.

Introducing The Agile Enterprise

Andy Winskill

HOW DO WE CREATE THE AGILE ENTERPRISE?

So how to we integrate Agile development processes into an existing enterprise? This question has caused me sleepless nights. Take for example, a Systems Integrator (SI) that uses an existing development methodology, possibly even a number of them, for clients. The SI would probably have a project management process, too. But could the SI's client have their own processes impacting the delivery of software? If the SI was delivering a solution into another Enterprise the SI's client might use Six Sigma. The SI would have to integrate with its client's processes. With all these heavyweight processes, how do we integrate an Agile approach to software delivery into the Enterprise? How do we create the Agile Enterprise?

Recently I was engaged on an assignment in a technical capacity with a large SI. The SI was to deliver a £25M solution to a government department, a fairly typical scenario. Being responsible for the management and delivery of the project, the SI used their Prince 2 based project management methodology. A great deal of money was spent on the production of a PID, Functional Specification and other supporting documentation; some would say this was needless expense; however, when the SI was delivering to a fixed price, the PID and the Functional Specification became the project bible when the client changed the requirements. Aiming to practice Agile software development, we should welcome changing requirements. Indeed, on this particular engagement, requirements changing on a micro level – such as the minor alteration of the user interface or a change due to a clarification of the requirements – were welcomed. However, alteration of requirements that could impact on the delivery timescales, architecture or other related projects were put through the SI's change process. Is this an Agile approach?

Integration of Agile techniques into multi million dollar enterprise projects is far from simple. A couple of years ago I was engaged with a multi-national SI to help roll out RUP to a particular client. While RUP has its challengers, it can be used to deliver software, and this particular client had determined that RUP was its methodology of choice. I believe the client is still rolling out RUP. Is this a symptom of the client, the methodology or the intrinsic nature of an Enterprise? I firmly believe it is in the nature of an Enterprise to be slow to change: CEO's may make statements to the market on their "new direction"; the Enterprise has a shudder in response but rarely changes its direction rapidly. In fairness, CEOs' of large Enterprises realise this and the programme for change is rarely expected to deliver in the short term. But where do Agile processes fit in the programme for change? If a new software development process does not deliver immediate and observable benefits it, is often changed. How dowe generate the opportunity to introduce agile processes into slowly changing enterprise and maintain momentum behind them?

This column is called "The Agile Enterprise," where I'd like people to share their experiences of introducing agile processes into large projects or large enterprises. Experiences, both good and bad, would help a great number of us sleep better at night.

ABOUT THE AUTHOR

Andy is an independent consultant specialising in the building of large scale J2EE systems. You can contact Andy with your articles or article suggestions at andy.winskill@rosewoodsoftware.com

A constant debate rages in any public forum where Agile is discussed. Proponents talk about their projects and how using Agile has contributed to the success. Agile sceptics point to the sweet spot of small co-located teams with talented people and control of scope and wonder, "who wouldn't succeed"? While we are forced to agree that talented people are necessary for project success (whatever the methodology), ThoughtWorks has used Agile methods in a variety of different environments and has shown them to be successful in or out of that sweet spot.

ThoughtWorks projects come in all shapes and sizes.

In terms of size, we've used Agile extensively in its sweet spot – small, co-located teams. But we are known primarily for delivering very large, enterprise-transforming projects, stretching the boundaries around what kinds of projects can be done using Agile. Our smallest project is probably a one-person development for a client in Holland...our largest to date features around 100 people spread across 4 teams in 3 countries.

ThoughtWorks also uses Agile methods across a diverse range of technologies. The majority of projects use either Java or .NET, but we have used Agile methods in environments ranging from embedded software for handheld devices to banking software on Mainframes. We've introduced TDD into legacy environments and continuous integration into clients who previously did a complete build once a quarter.

ThoughtWorks is involved in all stages of the project lifecycle.

We're using Agile principles to define new practices for Analysis. Not every client wants to completely abandon upfront analysis and we've had to integrate Agile with existing methods and practices.

We've been working on expanding the range of metrics and tracking processes to allow clients to see how the project is progressing. New terminology like "stories" and "velocity" can be confusing and it's important that clients can track progress and see deliverables in terms they understand.

We've been working with testers to try and further define the role of a QA department in an Agile project. In banks, we've been bringing Security and Regulatory personnel into the project teams early on in the development process.

Over the coming issues of Agile Times, ThoughtWorkers will share their experiences with applying Agile methods in these varied project environments. We'll discuss what has worked, where the difficulties have arisen and what we've learned from both the successes and the failures.

Book Corner
Mike Cohn

Balancing Agility and Discipline: A Guide For The Perplexed, by Barry Boehm and Richard Turner, attempts to breach a conceptual divide between developers following Agile development methods and developers who taking what the authors call a "plan-driven" approach. The book does an excellent job of summarizing many Agile methods. The main premise of the book is that Agile and plan-driven methods each have a "home ground" for which they are the appropriate choice. The central part of the book is Chapter 5, "Using Risk to Balance Agility and Discipline." In this chapter the authors present a tailorable method that can be used to balance what they view as the two extremes of Agility and discipline.

Where the book fails is in the assumption that Agility and discipline are at opposite ends of a continuum. First, those of us who have worked on Agile projects know that we only achieve Agility through discipline. More importantly, however, the distinction between Agile processes and heavy-weight processes is not continuous and an effective process cannot always be achieved by balancing a little more of this with a little less of that.

For example, self-organizing teams are a fundamental component of what most of us consider an Agile process. I cannot add a little self-organization to a project. It's either added or it's not. I can lighten up on some aspect of my heavyweight, plan-driven process, but that doesn't make the process Agile all of a sudden.

Despite disagreeing with this main premise, I generally agree with the six main conclusions presented in Chapter 6. For example, the recommendation to "Build your method up—don't tailor it down" points out the main flaw I've seen with implementations of the Unified Process. Once something gets added to a process, it is just too easy to keep it. If we build our process up from nothing the results are much more satisfying than if we start with everything and slowly remove things.

There is much to like in this book. The book is very well written and edited. Illustrations are crisp and useful. Especially if you are not already using an Agile process, this book will help you identify ways to lighten your current process and may help you make a switch all the way to agile. While I disagree very strongly with the main premise of the book I recommend it highly. The book is enjoyable to read and you will come away from it knowing more about Agile processes.

Balancing Agility and Discipline By Barry Boehm and Richard Turner
Addison-Wesley; $29.99

Book Review: Slack
Deborah Hartmann

When I first saw the cover of the book "Slack", with its colourful slinky toy, I smiled because I'd just been lamenting the toll exacted on me by my current project. I thought "Slack - I wish!" Tom DeMarco makes the case that wishing for Slack is not enough, that we must create space for it in our software development workplaces because, in addition to improving the energy level and morale of our teams, it improves the products we deliver and ultimately the worth of our enterprises.

DeMarco writes from decades' experience managing software development, and is respected for his straight-forward wisdom in management books like The Deadline, Peopleware and most recently Dancing with Bears (these last two with Timothy Lister). DeMarco reminds us of things we know from simply "looking around", things which seem to run counter to the patterns of leadership we've been taught. The Agile movement encourages us to consider "a simpler way", and this book is a good place to start, for those of us striving to replace unproductive organisational patterns with more wholesome, more realistic ones.

It's a quick read, which is a good thing because by the time I'd reached page 11 I was listing the people with whom I'd like to share this small book. The writer's style is conversational but to-the-point, drawing conclusions through the use of metaphor, diagrams, and examples familiar to almost any workplace. And although accountability for a project's outcome ultimately lies with its managers, the book is not addressed solely to management. It is also intended for the knowledge workers who carry out the work, because the "slack" solution involves a cultural shift touching at all project participants.

"Section 1: Slack" addresses the inverse relationship between efficiency and flexibility, and the balancing effect of slack. After providing some simple and convincing examples, DeMarco summarises: "Making efficient use of workers in the sense of removing all slack from their day has an attendant cost in responsiveness and results directly in slowing the organisation down." And "It's possible to make an organisation more efficient without making it better. That's what happens when you drive out slack" (p. 11).

I enjoyed "Section 2: Lost, but Making Good Time", because it rang so true. Chapter headings here include: The Cost of Pressure; Overtime; Culture of Fear; and Process Obsession, in which he discusses the problems we've created by treating development teams like mechanised production lines. Here DeMarco further addresses the content of his sub-title (Burnout, Busywork, and the Myth of Total Efficiency). He reflects on those very issues we've all commiserated over - practices meant to improve production but which, in fact, stress teams and reduce their creativity, draining the energy and meaning from their work.

"Section 3: Change and Growth" addresses organisational learning. DeMarco maintains that only middle management has the proper perspective for this, as other layers of the organisation are too far from, or too close to, daily operations. Here he discusses leadership and laments the elimination of middle management, a common quick-fix when cost-cutting is mandated. He argues that the collaboration of middle managers, as a team, is key to growth and change in our organisations. He maintains that we must make the space for middle management to question, rethink, learn and improve, if we are to continue to succeed in the current climate of rapid change.

The book ends with at Risk Management, the "quantitative declaration of uncertainty." DeMarco proposes that the Risk Management approach is diametrically opposed to Plan-Driven project management, which he calls Plan for Success. "Plan for Success is a staple of management philosophy, particularly in our high-tech industries. It leads us to pour our desired outcomes into concrete, and make commitments on achieving those outcomes... It [also] puts an effective damper on risk-taking. An organisation that is expected to overcome *all* adversity can't afford to take on any but the most trivial risks. Conversely, an organisation that has suffered no important setbacks has in fact taken no real risks." (p. 187).

I found this last section interesting - since now "even the most staid of today's corporations know that risk is something they dare not run away from." (p. 187) It seems clear to me that this disconnect, between the very real need for risk-taking and the risk-suppressing philosophies by which we manage, is causing our enterprises pain. It is wearing out our valuable middle managers and the teams they represent. DeMarco cites corporate cultures that oblige us "to look ... bosses and clients in the face and lie rather than show uncertainty about outcomes" (p. 195). Realising this disconnect, I 'd say that we can no longer wait for the resulting problems to resolve themselves - they will persist as long as there's this essential mismatch between our mental model of how projects work and what really happens in our workplaces. It appears that we must either adjust our philosophies to match our realities, or reduce our expectations for the projects to which we are committed.

It is apparent to me that the left side/right side format of the Agile Manifesto physically demonstrates just such a philosophical shift, the very factor that causes it to be perceived as radical or, by some, extreme. For those of us who embrace the Manifesto, I think this book could be an important resource. Without ever mentioning the Agile Software Development movement, this book sows the seeds of openness to new ways of thinking. I think it could be a spark that starts our colleagues questioning "the way things have always been done". It might even give them the courage to work with us to make the space in our workplaces, the Slack, to allow better solutions to emerge – which I'd suggest is the very essence of Agility.

FURTHER READING

Waltzing With Bears: Managing Risk on Software Projects By Tom DeMarco and Timothy Lister
Dorset House

Managing Software for Growth: Without Fear, Control, and the Manufacturing Mindset By Roy Miller
Addison-Wesley Professional

Slack: Getting Past Burnout, Busywork, And The Myth Of Total Efficiency By Tom DeMarco
Broadway; $14.95

AGILE MANIFESTO VALUES:

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

While there is value in the items on the right, we value the items on the left more.

www.agilealliance.org