

Do It Yourself Agile

Damon B. Poole

September 29th, 2009

Copyright © 2008-2009 Damon B. Poole

All Rights Reserved

Portions of this book appeared previously in the following publications: “Developers Can Learn a lot From Salespeople” CMC Media, “Breaking the Major Release Habit” Copyright 2006 ACM, “Think Globally, Code Locally” Copyright 2007 Dr. Dobbs.

Table of Contents

Table of Contents	3
About the Author	10
Notes To the Reader.....	11
Updates In This Version	11
Acknowledgements	12
Introduction.....	13
We've Heard it All Before	13
Evaluating Agile Development.....	15
Think of Your Software Development Process as a Software Product	15
Agile for Mainstream Adoption.....	16
From Mainstream to Standard	17
Keeping Things in Perspective	17
A Quick Overview	18
The Primary Benefits of Agile Development	20
What is Agile Development?	20
Defining Agile via Benefits	20
Six Features in Six Months	21
Problems with Quality	21
The Misperception of the Value of Traditional Testing	21
Low Visibility	22
The Agile Approach.....	22
Agile Allows for More Flexibility and Options.....	22
Agile Provides Faster Realization of ROI	23
Agile Delivers the Highest Business Value and ROI	23
Agile Produces Higher Quality	23
Agile Gives High Visibility	24
Sustainable Pace: Supply and Demand.....	24
Primary vs. Secondary Benefits of Agile.....	26
Why Bother Changing?.....	26
Benefits of Adopting Agile	27
The Magic of Agile Development	28
Reinvest in Your Development Engine by Improving Your Work Environment	28
Your Development Process is Part of Your Work Environment.....	29
The Problems with Traditional Development.....	30
Traditional Development is a Chinese Finger Puzzle.....	31
Problems Hide Longer Than Necessary	32
The Testing Gauntlet.....	32
Code Churn: The Hydra Syndrome	33
Short Iterations, The Key to Agile Development	34
Feedback	34
Accidentally Agile	34
Short Iterations Reschedule Traditional Development Tasks.....	37
Say Goodbye to Feature Creep	38

Final Qualification of a Release.....	38
Remote Feedback.....	39
Dealing with Uncertainty.....	40
Software Development is Inherently Unpredictable.....	40
Sales are Inherently Unpredictable.....	40
Predicting the Future.....	41
Sales Techniques for Managing Unpredictability.....	41
Sales Pipeline.....	41
The Software Development Pipeline.....	43
Sales Quarters.....	45
Short Iterations.....	45
Sales Forecast.....	45
Sales Process.....	47
The Business Side of Agile Development.....	47
Art vs. Manufacturing.....	47
Creativity vs Infrastructure.....	48
Agile Provides a Manufacturing Platform for Creative Activities.....	48
The Software Development Lifecycle.....	49
All Phases of Software Development are Variations on a Theme.....	49
Requirements Gathering is Not Design.....	49
Specification is Design.....	50
Specification is the First Approximation.....	50
Estimation is Design.....	50
Planning is Design.....	51
Design is Design.....	51
Implementation is Design.....	51
Final Validation of the Design.....	51
Software Development is a Search Algorithm.....	52
The Search For Value.....	52
The Max For the Minimum.....	52
Searching For Value in All The Right Places.....	52
Responding Rapidly is Better Than Providing Complete Feature Sets.....	53
Minimally Useful Feature Sets.....	54
Getting Customer Feedback Faster.....	55
Fitting Work into Short Iterations.....	56
Divide and Conquer.....	56
Using Multiple Tracks of Development.....	58
Breaking Functionality Down to the Essentials.....	59
Breaking Features Down Even Further.....	60
Scaling Agile.....	61
Agile Exposes Scaling Problems.....	61
Process Perspective.....	62
There is no Bug.....	62
Think Of Your Development Process as Software.....	63
Creating a Robust and Scalable Process.....	63
A Simple Process.....	64

A Single Process	65
One Virtual Site	65
A Shared Process Model	66
Create and Maintain a Process Document	67
Continuous Process Improvement	68
Policies and Procedures	69
Frequent Feedback	69
Your Process Has Bugs.....	70
Shift Into Hyperdrive	72
Automate Everything.....	72
Always Act Like This is the Release	73
Customer Interaction.....	75
Product Management	75
The Business Value of Software Development	75
Tighter Requirements.....	76
Requirements	76
User Stories.....	77
Use Cases	78
Design and Architecture	79
Software Design Basics	79
Future-Proof Architectures, Major Releases and Large Feature Sets	79
Building Architectural Bridges to the Future.....	80
From Oversimplification to Rube Goldberg	80
Refactoring.....	82
Serendipity	83
Planning	84
Product Backlog.....	84
Estimation	85
Using Story Points For Estimation Instead of Units of Time	86
Velocity	86
Estimation Using Planning Poker	87
The Basics	87
The Benefits	88
Whole Teams	88
User Stories Simplify Estimation	88
Planning Poker Reduces Both Risk And Waste	88
Big Stories Can Hide Problems	88
If a Story Feels Like a Research Project, It Probably Is	89
If You Don't Have Enough Information, Don't Try to Make it Up	89
Planning Poker Fosters Good Habits	89
Quality.....	90
The Value of QA.....	90
Separation of Development and QA	91
The Role of QA in Agile Development	91
Maintaining the Balance Between Development and QA	92
End of Iteration Touch-up.....	93

Requirements Based Code Coverage Testing.....	94
Writing Tests Early	94
The False Economy of Big-Bang Testing.....	94
Automated Testing.....	97
Avoid “Random” Testing	99
Iteration Reviews, aka Demos	100
Usability	101
Integration	103
Big-Bang Integration	103
Continuous Integration.....	103
Self Integrity	104
Moving From Known Good to Known Good.....	105
It's All for One and One for All	106
Multi-Stage Continuous Integration	107
Distributed Integration	108
Adopting Multi-Stage CI	110
Releasing Your Software	111
Sink or Swim Maturation Process	111
Customers Don’t Want Frequent Releases	114
Size of Customer Base	115
Overhead Associated with Producing and Consuming a Release	115
Supported Versions	115
Customer Burnout From High Frequency of Releases	116
Frequent Releases for a Single Customer	116
It is Better to Find Customer Reported Problems as Soon as Possible.....	116
Beta	117
Limited Availability Releases.....	117
Moving to Frequent Releases Takes Time.....	118
Release Options	118
Release Option 1: Support all Versions in the Field.....	118
Release Option 2: Always Upgrade to the Latest Release.....	118
Release Option 3: One Release For All	119
Release Option 4: Dual Release Trains	119
We The People.....	120
Whole Team.....	120
Collocation.....	121
Transitioning to Agile Development	122
Focus on the Goal	122
Stage 1: Getting Ready	122
Self Education	122
Scope.....	123
Scouting	123
Prepare The Organization	123
Transition Development and QA Together.....	123
Keep Your Healthy Skepticism	124
Don’t Throw Out the Baby With the Bath Water	124

Create a Process Document	124
Stage 2: Establishing a Natural Rhythm	124
Short Iterations.....	125
Create The Initial Backlog	125
Define “Done”	126
Transitioning to Agile QA	126
Recognizing Success.....	126
Update the Backlog and Plan for Future Iterations in Parallel	128
Iteration Retrospective	128
Celebrating Your First Iteration.....	128
Repeat	128
Keep Your Process Document Up to Date	129
Stage 3: Your First Release.....	129
Release	129
Release Retrospective	129
Celebrating Your First Agile Release	129
Stage 4: Continuous Process Improvement	129
Manage Resource Imbalances.....	130
Reinvest in the Engine	130
Recommended Order of Practice Adoption.....	131
Process/Planning	131
Build.....	132
Test/QA.....	132
Integration	132
Releasing.....	133
Going Agile When Nobody Else Will	133
Scrum Master of One	133
Reverse Engineered User Stories.....	134
Divide And Conquer	134
One Thing at a Time	134
Product Owner For a Day – Creating the Backlog	135
Talking to Yourself	136
The Moment of Truth - Did it Work For You?.....	136
In Retrospect, That Was a Good Idea!.....	136
Unit Tests	136
Refactoring.....	137
Growing Your Agile Team	137
Process Improvement: Creating a Process Document	138
What is Your Process Today?.....	138
Process Improvement: Complimentary Practices	140
Respecting Domain Knowledge	140
Anticipating Unintended Consequences	140
Self-Organizing, Self-Managing, Self-Directing Teams	141
Group one.....	142
Group two	142
“No-Brainer” Self-Management Practices in Agile.....	143

Simplifying and Redistributing the Management Load.....	143
Shared Code Ownership	144
Pair Programming	144
Meritocracy	145
Stand Up Meetings.....	146
Using 3x5 Cards to Capture User Stories, Plan Iterations, and Manage Backlogs	148
Outsourcing.....	149
Process Improvement: Infrastructure Considerations	150
Use Off The Shelf Automation	150
Project Management, Defect Tracking, and Requests for Enhancement.....	150
All Electronic Artifacts	152
Version Everything	153
Atomic Transactions	153
Infrastructure Support For One Virtual Site	154
Asynchronous Coordination	154
Process Improvement: Advanced Techniques	155
Decoupling Iteration Activities From the Iterations	155
Decoupling Iteration Meetings and Story Point Estimation	155
Decoupling Retrospectives	155
Decoupling Iteration Reviews.....	155
Simplification and Delegation	155
Appendix A – Example Process Document.....	157
Process Document for Acme Widgets Inc.	157
Requirements Gathering	157
Iteration Planning.....	157
Test Plan.....	157
Development	157
Integration	158
Exploratory Testing and Verification	158
Iteration Completion	158
Release Process	158
Appendix B – Case Studies.....	159
AccuRev’s Transition to Agile	159
The Development of AccuWorkflow at AccuRev	159
Little & Co.	161
The Iterative Design of a Transforming Lego Sports Car	164
The Challenge	164
Transforming Lego Car, Iteration 1	165
Thinking Outside The Box.....	165
Sometimes Flexibility is Constraining.....	166
Inspired by Elegant Design	167
Sliding Into Home	167
Developing a User Interface: Traditional Development vs Agile Development.....	171
Appendix C – Mainstream Practices.....	172
Mainstream Practices	172
Current Mainstream Practices	172

Preparation	173
Development	173
Quality.....	174
Releasing.....	174
Recommended Reading	175
Business	175
Product Management	175
Requirements, User Stories, and Use Cases	175
Lean.....	175
Estimation	176
Software Development.....	176
Usability.....	178
Software Quality Assurance and Testing.....	178
Software Configuration Management.....	179
Index	180



About the Author

Damon Poole is Founder and CTO of [AccuRev](http://accurev.com), a leading provider of Agile Development tools. Damon has nineteen years of software development methodology and process improvement experience spanning the gamut from small collocated teams all the way up to 10,000-person shops doing global development. Damon is President of the Agile Bazaar (<http://agilebazaar.org>) in Boston and is a Certified Scrum Master. He writes frequently on the topic of Agile development and as one of AccuRev's product owners works closely with AccuRev's customers pioneering state of the art Agile techniques such as Multi-stage Continuous Integration which scale smoothly to large distributed teams. Damon has spoken at numerous software development and Agile-related conferences, including SD Best Practices, Software Test & Performance, Q-Con, Deep Lean, Agile 2008 and 2009, and Agile Development Practices and also at numerous companies including American Student Assistance, Ford, Intercall, ITA Software, The Mathworks, Texas Instruments, and Verizon Wireless. He earned his BS in Computer Science at the University of Vermont in 1987. His "Do It Yourself Agile" blog is at <http://damonpoole.blogspot.com>.

Notes To the Reader

I had originally intended to publish this as an offline paper book. However, the web feels like a more “Agile” medium to me and so I’ve decided to provide this content primarily via the web. Translating a book to the web takes time, so I’ve also decided to make this document available via pdf in the meantime. As the material has made its way to the web, it has often been rewritten. Some of those changes have made it into this document, and some haven’t. Please see [Do It Yourself Agile](#) for the latest updates.

The current text represents only the “finished parts” of the book. There are some parts that are not yet finished and have not yet been incorporated.

This book was originally titled “Mainstream Agile.” I have since changed it to “Do it Yourself Agile.” While the title has changed, and the focus of my blog has changed, this document does not yet entirely follow that message.

I still believe that the quickest route to Agile success is to get people involved who have done Agile before. But, that option is not always available and even when it is, it may not be possible to retain an Agile coach for the multiple years that it will take to reach your full Agile potential. The purpose of this book is to give you a resource to lean on as you move to Agile on your own. If you run into problems along the way or feel that there is something missing in this book, please let me know via my blog. I will definitely incorporate your feedback. I am also very interested in your DIY stories. What worked for you and what didn’t work?

Updates In This Version

Thanks for all of the great feedback that people have provided on the first version! It is great to see Agile in action in the course of writing a book. As a result of the feedback I’ve removed a couple of sections that didn’t add any value and added a bunch of new material from recent blog posts. For those of you who have already read the first version and would like to peruse just the new sections, here are the major updates:

Using Story Points For Estimation Instead of Units of Time

Velocity

Estimation Using Planning Poker

Going Agile When Nobody Else Will

Process Improvement: Advanced Techniques

Acknowledgements

I would not have been able to produce this material without the help and support of my family, everybody (and I do mean everybody) at AccuRev, Inc, and the many folks who have commented on my blog. To all of you go my thanks and gratitude.

Thanks also to the many folks who have blogged and tweeted about the book's release!

Introduction

Keeping up with the rapid pace of technological change can be a daunting task. Just as you finally get your software working to meet yesterday's needs, a newer technology is introduced or a new business trend comes along to upset the apple cart. Whether your new challenge is Web services, SOA (service-oriented architecture), ESB (enterprise service bus), AJAX, Linux, the Sarbanes-Oxley Act, distributed development, outsourcing, or competitive pressure, there is an increasing need to shorten the development cycle time, respond to user needs faster, and increase quality all at the same time.

Traditional software development techniques have not adapted well to this pressure. Attempts to deliver higher ROI in a shorter timeframe, increase quality, produce releases more predictably, and get finer-grained visibility into project status and progress have mostly served to highlight the problems with traditional software development. The more a team tries to improve their results by adhering more tightly to traditional techniques, the worse off they become.

An emerging response to this challenge is an approach called Agile software development, the common theme of which is taking a traditional development process with a single deliverable at the end and splitting it into a series of small iterations, each of which is a re-mixed microcosm of the full process and each of which produces working software. Rather than trying to reduce and constrain change, Agile embraces change as a fact of life and includes techniques that transform what would otherwise be hurtling at high speed out of control with danger at every turn into the equivalent of regularly scheduled coast-to-coast jet flights.

We've Heard it All Before

The Segway, UML, Virtual Reality, TQM, object-oriented databases, Artificial Intelligence. Agile's no different, right? Just the latest fad? That's what I thought.

Invasion of the Body Snatchers

My first real encounter with Agile development was at the Software Development Best Practices conference in September of 2005. Up until then I had managed to be all but ignorant of it. I was at the speaker's reception, and unbeknownst to me I had been dropped into a den of Agilistas. They all looked like perfectly normal people to me. Conversely, they all seemed to assume that I was "one of them" and they started talking to me about 3x5 cards, pair programming, collocation and the like.

I wondered why these folks thought it would be a good idea to send poorly tested software to customers on a weekly basis. And what were they thinking doing their planning by shuffling around 3x5 cards with notes scratched on them? These folks were talking about writing software while sweating hip to hip with a fellow programmer all day as part of a two-person programming team in a small cramped space in close proximity with lots of other sweaty two person teams. And when I objected I was treated as a heretic that "just doesn't get it." To top it all off, this stuff is all written up in a Manifesto!!

I wasn't just skeptical, I was outraged that all these people were trying to push what seemed like nonsense. Keep in mind that I work at a company that provides software tools to companies that are looking to improve their process. I believed that traditional development could be made to work just fine. Sure, some companies needed a tweak here or a tweak there and some companies were more mature or more disciplined than others, but all in all traditional development was just fine. Ok, so many companies complain about predictability, quality, visibility, and rework but that's just the cost of doing business, right?

I decided that it was time for somebody to step forward and do something about this Agile nonsense. I nominated myself. To gather evidence I read the books, talked to the experts, and attended the seminars. The more information I gathered, the more I felt that Agile went against everything that I believed about software development.

But then, one day a funny thing happened. I inadvertently reached critical mass on Agile and I realized what had kept me from seeing the value of Agile up to that point. My objections were based on assumptions formed by doing traditional development. Many of the concepts in Agile development, taken individually, do not work in a traditional framework. So, if you examine the concepts one by one, it is easy to dismiss them one by one and conclude that Agile won't work. But if you suspend your disbelief long enough to absorb the critical mass of individual concepts in Agile and then see the new framework that Agile provides, the individual concepts of Agile make sense.

There is no simple way to show that Agile is worthwhile. The only route that I know of is to spend time learning more about it based on the promise that it will be worthwhile. For me, all of the pieces of the puzzle came together in an "aha" moment and I realized there might actually be something of value buried under all of the rhetoric. I wasn't completely convinced yet, but I saw enough value to give it a try.

Agile is Jolting

At AccuRev in April of 2006, we conceived of a new product called AccuWorkflow. At around the same time we were thinking about what to enter for the 2007 Jolt Awards. We decided that AccuWorkflow would be a big part of that but then we also quickly realized that if we wanted to have a shot at the Jolt Award for 2007, we had to go from PowerPoint to shipping product in just 7 months. It seemed like a good opportunity to give Agile a try.

Our hard work and the use of Agile paid off. In December we shipped the first version of AccuWorkflow. In January we were announced as a finalist and in March of 2007 we won the Jolt Product Excellence award for AccuRev 4.5 with AccuWorkflow. I am 100% sure that the primary reason that the version of the product we submitted for consideration was ready on time with high quality was Agile development. As I accepted the award I realized I was hooked on Agile and there was no going back.

Evaluating Agile Development

Our prior experience, encapsulated in the form of habits, beliefs, expectations and assumptions shapes our perceptions and forms a framework for evaluating new ideas. This simplifies our daily routine. We can't be going through each day questioning every assumption and changing everything we do. We would never get anything done. As a result we tend to have a bias against change. We are inherently skeptical of change and require the advantage of any change to be much higher than the effort required to make that change. Disruptive changes can be particularly difficult to decide to adopt because they require us to evaluate not only the new idea but also the framework which we use to evaluate new ideas.

When considering a transition to Agile development, there are at least four approaches you can use: believe the hype and hope for the best, look for documented ROI studies, find people that you trust at a company with similar circumstances that have done it successfully and learn from them, or learn all you can about it and decide for yourself.

Unfortunately, we've heard similar hype before, there is not yet a large and diverse body of evidence showing that Agile development is generally better than traditional development, finding folks with similar circumstances that have done it before is difficult, and most of the available literature concentrates on how to be Agile rather than how and why Agile works.

As technical folks, we love to know how and why something works. When it comes to a new way of coding something, we want to know what the algorithm is and how it works. Often, there are two or more possible paths forward and we don't have the time or resources to implement multiple approaches to find out which one works best. In that case, we analyze and compare algorithms and architecture. So, why not apply that same approach to Agile development?

As technical folks, we're good at this sort of thing and we do it as part of the regular course of doing business. It is part of what makes software development such a creative endeavor and we're very comfortable with it. We like making decisions based on a rational analysis. We prefer meritocracy. We like to know "does this make sense?"

Think of Your Software Development Process as a Software Product

In order to apply our technical evaluation skills to the job of evaluating Agile development, we have to be able to look at the process of software development itself in an algorithmic and architectural way.

The main goal of software is to automate and simplify what would otherwise be accomplished using a manual process. This gives the users of the software leverage to do more with less. Instead of balancing our checkbook by hand, we can use Quicken to do it faster and more accurately. Instead of maintaining the records of millions of people's financial transactions with paper and pencil, banks use mainframes.

By thinking about the software development process in the same way as you would think about any manual process that requires automation, you can leverage your skills as a developer and apply them in

a new way. The process of developing software is, in effect, an algorithm implemented with people, process, and tools.

Agile for Mainstream Adoption

When considering something new, one of the questions we ask is “will it stand the test of time.” We’d rather not invest our valuable time in something that is going to turn out to be just a fad. That is, we’d prefer to invest in something that is either mainstream now or has the potential for mainstream adoption. The approach of this book is to look at everything from the perspective of mainstream adoption.

As it relates to software development, I define mainstream as something that

- provides value significantly greater than whatever it replaces
- can be adopted by any software development team regardless of size
- uses existing team composition
- can be adopted piecemeal
- will work with any physical distribution of team members
- can be used for development in any domain
- the how and why it works is well understood by everyone using it

Quite frankly, some Agile practices are either unlikely to become mainstream or only apply under special circumstances. For instance, it is unlikely that pair programming will ever become mainstream. Thus, this book refers to it as an optional practice and shows how you can integrate it, but it is not considered central to Agile success.

As a result, this book does recommend a specific approach to implementing Agile. There is a commonly held belief that there is no “one true way” of developing software. I agree that this is true in principle. However, in my experience, from the narrow perspective of the software development process, there is much more in common between any two software development shops than there are differences. Yes, there is a small chance that in your organization one of the differences in how you do your software development process may actually be a business advantage for you. However, there is a much greater chance that there is a much larger benefit in making improvements that are well known to provide benefits and are universally applicable.

The looser the description of a methodology is, the harder it is to actually implement it. Those wishing to implement it will have to spend a lot of time interpreting, deciding, designing, and experimenting which reduces the amount of time spent on the main goal of producing software for the customer.

As an example, it is possible to read a book on the merits of object oriented programming and then write object-oriented code using C. But it is much more productive to write in C++ which was specifically designed for object-oriented development instead of having to create conventions on your own that may not be shared by another development group.

From Mainstream to Standard

When something is mainstream, there is another benefit. Eventually it becomes a standard. Standards give us the canvas, paints, and brushes for creatively solving the problems at hand. They give us the common language for expressing, communicating, and absorbing information and ideas.

Consider how you benefit from standards any time you start a new job in software development. Let's say that you are a Windows Java developer. You know that when you sit down at your desk you will have a Windows PC with a standard keyboard, an Ethernet connection, and access to all of the resources of the world wide web rendered in html and transmitted via http. Since you are fluent in Java, you will be able to read the Java code, figure out what it does and make changes (probably using Eclipse.) Furthermore, you will likely have wireless access via 802.11g and use standard tools for producing and manipulating XML and SOAP. All of these standards have provided an enormous amount of value simply because they are standards.

In order for a standard to be useful, it must be invisible and unspoken. It must be pervasive and tacit. It cannot be something which is done with conscious thought. If you had to refer to a manual as you drove, driving would not be as popular and as widespread as it is today. The implementation of a standard must be a whisper on the wind: silent yet ever present.

Contrast this with software development as it is done today. Not only is the process present, it is loud, obnoxious, and irritating. It is confusing, vexing, and painful. It is wasteful, taxing, and overly complex. It will likely take you at least a year and probably two before you know exactly how the process works that your company uses to produce the software that it sells. Any new job and often any new project will mean learning a completely different process which is probably not written down and maintained in one place. You learn the process from your manager and co-workers, and through painful trial and error. Everybody has a slightly different view of how the process works, and the cycle time of the process is so long that it is difficult to learn through repetition.

Keeping Things in Perspective

Software development is all about perspective. There are two key perspectives. The first perspective to keep in mind at all times is the customer's perspective. When you are working on something and you say "it's done," it really isn't. It isn't done when it builds for you. It isn't done when it is integrated. It isn't done when QA says it is done. It isn't even done when it ships. And believe it or not, it isn't done when an end user says "I like that!" Work can really only be considered done when it has been in production for a while without any problems.

This may surprise you to hear, but the other key perspective is yours. I'll bet that you identify with most or all of the following statements:

- When working on a software project on your own, you make changes and use the new version right away.
- One of the reasons that you got into software development is the fact that you can make a change and see the result right away.
- When you have something new working for the first time, you want to show it to somebody.

- Even when creating something for yourself, it isn't easy to create what you want the first time.
- It isn't easy to create something that your customer thinks is exactly right the first time.

Most software projects are part of a larger scope, even if we are the sole developer or most of the time we are concentrating on a single task that we alone are responsible for. Somebody has to test the software, and there are usually organizational standards that must be adhered to. In any case, you will spend a fair amount of time developing according to the rules of the organization ("the process") rather than the way you would do it naturally if it was an individual project just for yourself. The more the way you would naturally develop software conflicts with "the process," the more you have to slow down and think about the correct next step and the greater the hit to productivity.

With traditional development, there is generally much more process in place to cope with the greater complexity of trying to develop a much larger increment of functionality over a much longer time period. Because of the large gap between how one would develop software on one's own and how one develops software as part of a much larger process, the steps in the process are often counter-intuitive and thus it is much harder to establish a natural rhythm.

I have found that everything your organization needs to do in order to become the best possible development organization is entirely contained within the best practices that you as an individual practice and the things that you need and do in your day to day work. By looking at everything from the perspective of an individual, your organization stands to benefit a great deal. Conversely, everything that the organization does as a result of thinking this way will benefit you directly.

A Quick Overview

This book contains everything you need to get Agile adopted on your project. It is written with the expectation that you are either skeptical of Agile development or that you will need to counter skepticism as you advocate the use of Agile.

The book starts with a description of Agile development and briefly describes how Agile works. This is followed by an examination of the problems with traditional development and shows how the root causes of these problems can also cause resistance to Agile adoption. Awareness of these problems will simplify your adoption of Agile.

Software development involves a large portion of work on tasks which have not been done before using technology that has not been used before. Add in the fact that creating something that satisfies users is more of an art than a science and you have lots of unpredictability. This is often cited as an unsolvable problem that is the major contributor to project failure. However, there are many ways to increase predictability. Some come from process changes such as separating tasks into those that are predictable and those that are not, and others come from examining how other professions succeed in the face of unpredictability and translating those techniques back to software development.

The steps you follow to develop software are themselves algorithmic in nature and the same skills and techniques used for designing scalable and easy to use mainstream software can be applied to the software development process itself. By thinking of your process as a software product, you can apply all

of your software development skills and experience to improving your process. As a result, you will produce a much more robust and efficient software development engine.

The last section of the book is a step by step guide to successful Agile adoption. Making changes and adopting new practices is hard. Anything that requires sweeping changes usually encounters a lot of resistance or is only done when things are so hopeless that “it couldn’t possibly get any worse.” As a result, the adoption section introduces Agile gradually via a series of incremental process improvements. There is no need to start with a brand new project or to make wholesale changes in an existing project. It doesn’t matter if you are currently using a waterfall process, Extreme Programming, Scrum, a hodgepodge of practices, or “no process at all”, you can start down the path of becoming Agile today and adopt as much as desired at whatever speed feels comfortable.

The main adoption section is just enough to get you started. There are follow-up sections that provide additional techniques that you can use based on the needs of your specific circumstances.

The Primary Benefits of Agile Development

You may have some skepticism about Agile. That's perfectly understandable. This book assumes that you take nothing for granted about Agile and want to know exactly how and why the various aspects of Agile are a good idea. Let's start by explicitly stating the exact benefits that you will get from implementing Agile as described in this book.

For the moment, let's assume that the claims are absolutely true and will be fully substantiated later. Whether they are true or not, if the claims have no relevance to your situation or you are not in need of any of the benefits, then there is no point in reading any further. On the other hand, if the benefits to you personally are compelling, then why stop now? If the benefits seem appealing, but something tells you that it goes against everything you believe in, consider that it is just possible that a change of perspective may be in order.

What is Agile Development?

Giving a concise definition of Agile is far from easy, probably because Agile is actually an umbrella for a wide variety of methodologies and because Agile is officially defined as the four values in the Agile Manifesto:

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

But how would you translate this into a dictionary definition? It may be pointless to define Agile development. It may be better to just talk about the concepts and the practices and say "these things are worth learning about regardless of what you call it or how you define the overarching concept." I still think it is worth a shot so as to establish a starting point for discussion. So, let's discuss the definition of Agile a bit and then move on to the nitty-gritty.

Because there are so many practices associated with Agile development, a simpler way to define Agile is to do it in terms of the benefits. It is not a perfect way to define something, but it is currently the best way that I know. There would be no point in doing Agile development unless it was better, so I define the benefits in relation to traditional development. I consider waterfall to be part of traditional development, but I use the phrase "traditional development" instead of waterfall because there are many shops which would say "we're not doing Agile, but we're not waterfall either."

Defining Agile via Benefits

I define Agile development as that which, in comparison to traditional development, provides the benefits of more flexibility, higher ROI, faster realization of ROI, higher quality, higher visibility, and

sustainable pace. Let's learn more about these benefits and the differences between traditional development and Agile development by looking at an example development scenario.

Six Features in Six Months

Let's say you want to add new features to stay competitive. In a traditional project, you have a known timeframe for major releases. Too short and you'll spend too much time on overhead, too long and you'll miss opportunities. For the sake of argument, let's pick a timeframe of six months and say that allows you to provide six "big features" including time for all preparation and testing. Marketing says the six features with the highest ROI are a Facebook plug-in, a Second Life plug-in, an RSS feed plug-in, and three other features. During the six months, the business value of the planned features may change. If it goes up, that's great, but if it goes down, there's very little you can do about it.

Midway through the design process, marketing announces that the Second Life plug-in is not as marketable as they had hoped and iPhone support is showing signs of becoming very lucrative. You think, "oh well, nothing we can do about that now."

Just after you finish coding, marketing declares that the Second Life plug-in is going to be a complete flop and wants to know when can they get iPhone support?

Now that the functionality has settled down, QA finishes their test plan, and starts writing test cases and running them. Planning and development took longer than expected. Originally, there was a month reserved at the end for testing, but now the release deadline is looming with just two weeks left in the schedule. The time for testing is compressed, QA concentrates on the most critical stuff and gives the rest a spot check. Once the find/fix rate gets down to an acceptable level, you declare victory and deliver the new release. In the end, the functionality that was asked for at the beginning is delivered a month late and doesn't have the originally anticipated value.

Problems with Quality

In a traditional project, the elapsed time from start to finish from the perspective of any individual work item is very long. There is no consistent process applied to each and every work item. Instead, work items are fused together into "the release" and the quality of "the release" is measured. One consequence of this is that QA gets a big dump of functionality near the end of a release and has to figure out how to make the best use of the time remaining which is often less than originally planned. That means taking shortcuts. The "most important" new features get very thorough testing and the rest get a "spot check." Another problem with traditional projects is that since much of the test case writing, test automation, and test plan execution is left to the end, problems can hide out until just before the release and thus there is a long time between the introduction of a problem and the detection and fixing of that problem.

The Misperception of the Value of Traditional Testing

Here's a mystery. If running through your full test plan takes two days, why does testing take a month? The answer is that you aren't really doing a month's worth of testing, you are doing the same testing over and over while a month of time goes by. Problems have been creeping in all along the way that you

are just now finding out about and it takes many test/fix cycles including repeating some or all of that test plan over and over again to expose and fix the problems. It is only the final full run of your test plan that gives you the measure of the quality of the product, so in the end you've really only gotten the value of the test plan. If it takes two days for that final run, then you have two days of testing, not a month.

Low Visibility

Time after time, with traditional development, progress is measured based on progress against a plan. The problem with that is that customers don't buy Gantt charts, they buy shippable software and the progress that is measured by Gantt charts is not directly connected to shippable software. Just because you have finished 100% of the requirements which is 20% of your plan, that doesn't mean you are 20% done. In fact you are 0% done from the perspective of the customer because they can't benefit from that progress until you ship. In a traditional project, you only know how much ahead or behind plan you seem to be based on the progress that people claim, but you don't really know how much ahead or behind plan you actually are from an "is this shippable" perspective. It is almost impossible to see at a glance what the real project status and progress are. You know that you won't start getting information about where you really are until sometime after code freeze.

It is hard to know what to do next if you aren't sure where you are. Do you feel like you know exactly where you are and what to do next at every step of the process? Or do you feel pulled in a million different directions at once and lose track of what you've already done and what you need to do next? If you do feel like you've finally "got" how software is developed in your organization, how long did it take you to get to that point?

The Agile Approach

Now let's try Agile development using the same scenario. Marketing says the three features with the highest ROI are a Facebook plug-in, a Second Life plug-in, an RSS feed plug-in, and three other features. You start with the Facebook plug-in. You plan. You design. You create a test plan with test cases while the code is being written. You discover potential problems and deal with them. You automate the test case while the code is being written. As the code is written, it is integrated, built and tested continuously; problems are found and fixed immediately. At the end of development, the only problems that remain are the ones that could only be found at the end of development. If you wanted to, you could cut a new release with the Facebook plug-in with very little overhead.

Agile Allows for More Flexibility and Options

With Agile development, you develop your software in short iterations where "short" means a month or less. At the end of each iteration you have a new increment of functionality which is shippable. That means that at the end of each iteration you have the option to easily change plans before the beginning of the next iteration in order to take into account new business requirements or information gained during the previous iteration.

At this same point in time, the traditional project would just be finishing up their preparations for starting development and would not yet have anything to show for their progress other than plans. With traditional development the software looks like a construction site until just before the release: some parts are done, others are not, but the building is not currently usable at all. Changing plans during the development cycle means ripping out work in progress and possibly work that is done but depends on other work that is not done.

In the Agile scenario, since there is nothing in progress, the organization can now reevaluate which work will produce the most business value. Marketing determines that the Second Life plug-in is not as marketable as they had hoped and iPhone support is showing signs of becoming very lucrative, so they tell engineering that the new ranking of features is RSS feed support, iPhone support, the Second Life plug-in, and three other features. Since RSS feed support is ranked the highest, you start on the RSS feed support.

Now marketing says the Second Life plug-in is worthless but iPhone support is hot. That means that the new ranking of features is RSS feed support (which you are working on), iPhone support, three other features, and then the Second Life plug-in. So, you finish the RSS feed support and then work on providing iPhone support. You continue this process indefinitely.

Agile Provides Faster Realization of ROI

Since you have shippable software at the end of every iteration you can start realizing the ROI whenever you like instead of having to wait.

Agile Delivers the Highest Business Value and ROI

Because Agile gives you more flexibility you can change your plans to take into account the current market conditions instead of working on things which seemed like a good idea when you started working on the release but have now lost their appeal.

In the end, the traditional team produced a release which had less business value than originally expected, and they missed out on the opportunity to deliver iPhone support early which turned out to have very high business value. The Agile team on the other hand produced more business value than was originally expected.

Agile Produces Higher Quality

In an Agile project, the elapsed time from start to finish from the perspective of any individual work item is very short. Test case writing, test automation, and test plan execution are done throughout the iteration and each work item is given the amount of QA resources that is appropriate for it. Because problems are found and fixed faster, there is less chance of the quality of a project being both unknown and poor for long stretches of time. Code for each iteration is written on the stable base of the previous iteration and is more likely to be stable itself because there will be accurate and timely feedback on the results of the changes.

Agile Gives High Visibility

Short iterations solve the problem of misleading progress reports. With short iterations, you know at the end of each iteration exactly how much progress you have made. Instead of the traditional "we're still on target" all the way up to just before the end of the release, you know at the end of every iteration exactly where you are. Whatever work was done during that iteration is done and potentially shippable. You know exactly how many work items were fully completed, how many weren't started, how many test cases remain to be written and automated, how many test failures remain to be fixed, and how many defects were filed on work done during the iteration. You get the kind of information that is usually only available just prior to release on a monthly or even weekly basis.

In a release plan consisting of six iterations, each iteration is $1/6^{\text{th}}$ of your overall plan. When you've finished your first iteration, you know you are not only $1/6^{\text{th}}$ of the way through your plan, you also have $1/6^{\text{th}}$ of the work actually done and shippable. When you've finished three iterations you are now 50% of the way through your plan and you also have 50% of the work done.

It may well be that you encounter a problem. Let's say that in your fourth iteration you run into a major problem and you only get $1/2$ of the work done that you planned for that iteration. Well, at least that work is done and you know for sure that you are now both $7/12$ ths of the way through your plan and you have $7/12$ ths of the work done and shippable with a 2 week slip in plan.

You know immediately that you are behind schedule and you can respond immediately instead of finding out near the end and having a much shorter runway to respond.

Sustainable Pace: Supply and Demand

People are much more productive and much less prone to error when they work at a constant and sustainable pace. This also means that when there are unforeseen circumstances, people are more likely to be able to respond well.

In a traditional project, the demand for resources from the four major aspects of software development see-saws dramatically over the course of a project. These aspects are project management and planning, architecture and design, development, and QA. You need resources on hand to serve the peak demand level, but during periods of low demand those resources will either be idle or used for activities which have a lower ROI.

A common circumstance is that there are insufficient resources on hand for the peak demand level and so people end up working in "crunch mode." During crunch time, people tend to make more mistakes. Agile levels demand out over time and removes this see-saw effect which simplifies resource planning and removes the need for crunch time.

With traditional development, delays during development compress most of the testing to the end of the process which then requires taking shortcuts due to schedule pressure. I used to think that one way of compensating for insufficient QA resources was to delay the release until QA finishes. On the surface it seems to make sense. But only if the folks writing code sit on their hands while QA does their work.

Ok, so you have multiple projects and the developers work on another project. But then they finish that. Now QA starts on the second project and the developers move to the third. The problem is still there.

On the other hand, as a result of the need for increased QA resources during testing, you may have two other problems. If you have enough QA resources to handle the pressure of the endgame, you may have too many QA resources during the rest of your development cycle. Alternatively, you may bring on additional QA resources on a short-term basis to compensate. Both of these options are obviously undesirable.

There's a natural balance between the amount of effort required for developing something and the amount of effort required to QA it. No matter what you do, if you have the wrong ratio of development resources to QA resources, it will cause problems. If development creates more than QA can absorb, you will create a backlog of QA work that will always grow.

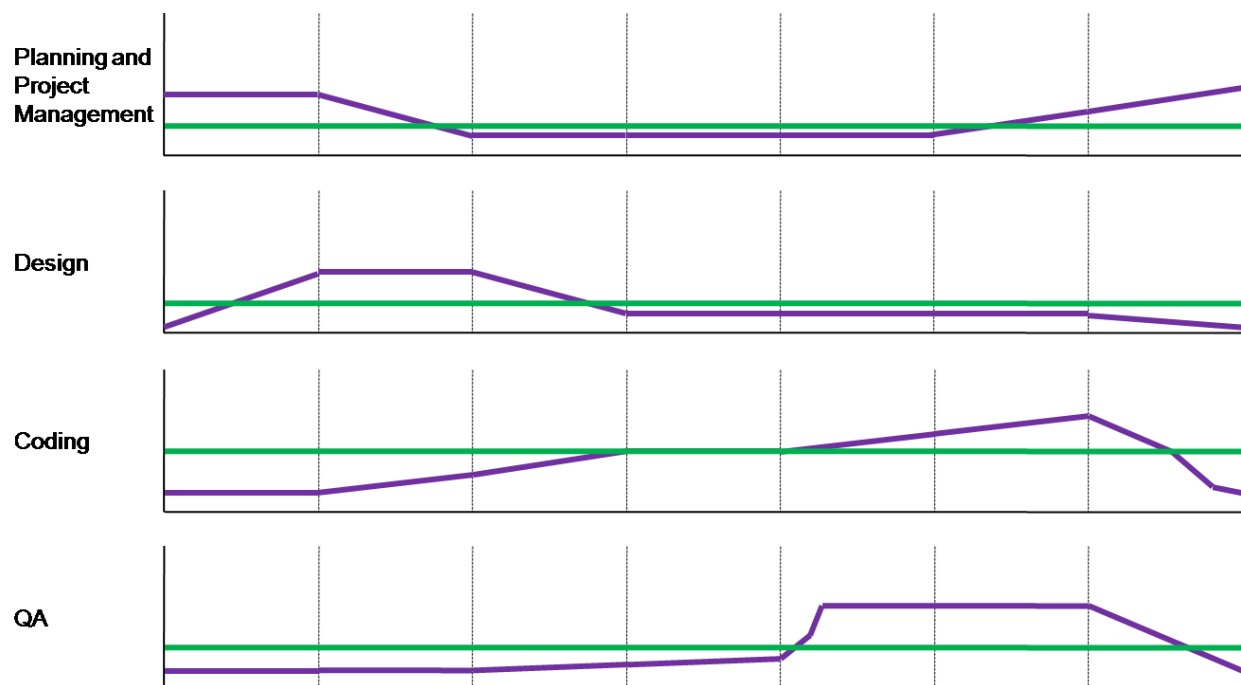


Figure: the straight green lines represent resource consumption on an Agile project. The purple lines represent a traditional project.

This natural balance holds between all four aspects of software development. Depending on your organization, there may be an imbalance between supply and demand at any stage in the pipeline.

When using short iterations, resource imbalances are easier to detect and correct. Having balanced resources means that all development activities are done consistently and on a regular basis and there is no need to take the shortcuts that are typical of traditional development.

Primary vs. Secondary Benefits of Agile

Agile is generally talked about as a single package: if you adhere to the principles, you get the benefits. There is another way to look at Agile. On the one hand, Agile introduces a whole new set of practices to the development toolkit. These practices include: product backlog, pair programming, on-site customer, continuous integration, refactoring, test driven development (TDD) and many others. While all of these practices have been either associated with or created as a result of Agile, their application and resulting benefits can be applied completely independent of any specific methodology.

Whether you use Scrum or "Waterfall" you still get the benefits of the Agile practices. For instance, refactoring is a standard feature in Eclipse. Eclipse is an IDE and is completely orthogonal to your methodology. The benefits from these practices are secondary benefits when practicing Agile.

The primary benefits of more flexibility, higher ROI, faster realization of ROI, higher quality, higher visibility, and sustainable pace come from the single practice of short iterations.

Many people object to some of the Agile practices without realizing that they don't need to adopt them to get the primary benefits. You don't have to do pair programming, use 3x5 cards, collocate, only use small teams, or have a customer on site to get to short iterations. Depending on your circumstances, these techniques can be used to accelerate and reinforce your transition to Agile, but they are not the only practices that will. For instance, test driven development is very easy to adopt, contributes greatly to enabling short iterations and does not require any experimentation with personal boundaries like pair programming does.

Once you have successfully transitioned to short iterations and are receiving the primary benefits, there are many things you can do to fine-tune your process and continually increase the benefits of Agile development. Depending on your circumstances that may mean incorporating pair programming, working at the same site as the customer, collocating, using self-organizing teams, etc.

Why Bother Changing?

Aside from personal preference, the only reason to make a change to the way you develop software is to realize a benefit. It could be to increase quality, customer satisfaction, employee satisfaction, productivity, or profits. In the end, these should all result in increased profits. If profits are not an important part of your organization, for instance you work at a not-for-profit, then another way to look at this is reducing expenses. For simplicity, I will focus on profits.

Why should you care about the profitability of your company? There are a number of reasons. If you are a stockholder, you benefit directly. The more profitable a company is, the more likely it is to be secure. The more profitable a company is, the more likely it is to embark on exciting new opportunities which means more opportunities for you.

Agile Profits

In 2006, Litle & Co. landed at No. 1 on the Inc. 500 list with \$34.8 million in 2005 revenue and three-year growth of 5,629.1 percent. In 2007 Inc. magazine cited Litle & Co.'s 2006 revenue as \$60.2 million, representing a three-year growth rate of 897.6% over its 2003 revenue of \$6.0 million. How has Litle achieved these impressive results? One factor that they site is their use of Agile development.

Another reason to care about the profitability of your company is because your company is more likely to invest in its development infrastructure. If you were smirking while reading the previous sentence because you know that extra profits will never be invested into the development organization, perhaps you are working for the wrong company. Or perhaps nobody ever thought of investing more into the development organization and a suggestion or two in the right place is all that is needed. In either case, you can still take to heart the idea of re-investing profits into the development engine and work on strategies to make it happen.

Focus on Value

At Litle & Co., each developer has a quad-core workstation with 2GB of memory and a fast disk. They know that the less time developers spend waiting for build and test results or switching back and forth between the work that provides high value and other work, the more productive they will be. Not only will they be more productive, but they will also spend more of their time working on the highest customer value activities.

Benefits of Adopting Agile

The benefits of moving to Agile development can be split into two categories: benefits to the organization, and benefits to you personally. As a result of the high level benefits that Agile provides - more flexibility, higher ROI, faster realization of ROI, higher quality, higher visibility, and sustainable pace - Agile can provide the following benefits to the organization:

- Increased revenues
- Reduced costs
- Increased market share
- Higher customer satisfaction

Each of these benefits leads to a stronger organization which is then in a better position to reward you for your efforts both directly and indirectly. Some of these benefits include:

- Getting a raise and/or bonus – more discretionary income to buy cool stuff

- Improving your working conditions
- Actually using all of your vacation time
- The opportunity to spend more time working on cool stuff

In addition, Agile can provide the following direct benefits:

- A less stressful environment
- Less cancelled or shelved work
- Career advancement due to learning new skills
- Having the resume that gets you your dream job

The Magic of Agile Development

From a business perspective, the main reasons I appreciate Agile development are the benefits that I've described above. But from a purely personal perspective, the reason I enjoy Agile development is because it made my job more fun.

Today, thanks to Agile development, I interact with customers more than ever before. As a product owner, I do more demos and am able to provide new features that hit closer to the mark faster and more frequently than ever before. This in turn means more oohs and ahs from customers which is more fun for me and more profitable for the business.

Reinvest in Your Development Engine by Improving Your Work Environment

There are really only five ways to increase the profitability of a business based on software development: reduce costs via outsourcing, reduce headcount, reduce other expenses, increase productivity or increase revenues. Reducing expenses can only go so far. The most expensive part of software development is the people. Thus, one of the most successful ways to increase profits is to increase the productivity of the software development team.

The Agile Workplace

At Litle & Co., developers like the fact that Agile provides the additional challenge of solving business problems instead of just technical problems which requires thinking at a higher level. Developers at Litle report that they have a higher level of job satisfaction than in previous companies that were not using Agile because they see the results of their software development efforts installed into production every month. Also, they like the fact that there is much less work which amounts to "implement this spec."

Your development infrastructure is really no different than the general company infrastructure which includes your cube or office, the carpet, the artwork on the walls, the company cafeteria, your phone, your computer, and the company location. These are all part of your work environment. If you have a

computer that is 5 years old, your work environment is not as good as if you have a computer that is only 2 years old. If you are writing in C rather than C++, C# or Java, your work environment is sub-optimal.

The closer that your development infrastructure is to the ideal environment for your circumstances, the more productive your team will be. This principal extends to all aspects of the development environment, from development language, to build system, to build farm, to issue tracking system, to the process that you follow.

Your Development Process is Part of Your Work Environment

Your development process (regardless of how it is implemented), is also part of your work environment. If as a result of your development process you regularly end up redoing work because problems weren't discovered until just before the release, or projects get cancelled or shelved, then this is also likely to reduce productivity and job satisfaction. As this process improves, so does your work environment. The smoother it operates, the more pleasant your working environment will be.

There are many problems which you may think of as being unrelated to your development process. For instance, broken builds. Broken builds are simply the result of somebody making an idiotic mistake, right? Perhaps that's true some of the time, but most of the time it is due to the complexity of integrating many changes made by many people for software that has many interdependencies.

To be sure, a "perfect" process does not guarantee happiness, success, or the absence of problems. You still have to debug complicated problems, port to new platforms, deal with unforeseen circumstances, etc. However, the state of your process impacts the efficiency with which your effort is applied. For instance, if your process is perfect and completely frictionless, then 100% of your effort will be applied to the work that creates value. If it is rife with problems, it may mean that only 50% (or less!) of your effort will be applied to work that creates value. If there are problems with the process, then you are already expending effort which is essentially wasted. You would be better off investing some of that effort in removing the problems permanently instead of losing it to friction on a regular basis.

The Problems with Traditional Development

Have you ever thought “why do the companies I work at have so many problems with developing software?” I can tell you from interacting with literally thousands of software development organizations that most development organizations have problems with reliably and predictably producing high quality software. Over and over again people say “if only we could hire the right people” or “if only people could be more disciplined” or “if folks would just do X, Y, and Z it would be better, but I just can’t get people to see the light.”

There are a finite number of software developers, so unless we find some magical place to hire more of “the right people” from, we are going to have to figure out how to use the people we have. Whatever discipline that exists today, that’s all you are going to get. So what’s left? Changes to the way we do things.

Let’s say there was some combination of techniques which would mean that most development groups could use traditional development to reliably and predictably produce high quality results. Let’s call it the “Niagra” method. Furthermore, you (or whoever wants to make this claim) get to define the Niagra method. It works every time. Just use Niagra and performance improves overnight.

If the Niagra method existed, it would have become widespread by now. It would have become the #1 prescribed solution to project performance problems. If it existed and produced the claimed results, people would start referring to it, and it would spread rapidly. That’s how C++, Java, and HTML became mainstream. People rapidly adopted them because they provided benefits that anybody could realize.

In fact, there are many proposed fixes for traditional development, but in the long history of traditional development, no Niagra method has emerged because the fixes only work under special conditions. Traditional development has resisted significant improvement. It is time to admit that it is highly unlikely that such a method will ever emerge.

This raises an obvious point. If traditional development is so bad and is such a big failure, then why has the software industry done so incredibly well? The software industry is an incredible success, that’s absolutely true. It is incredible how much of our daily lives runs on software and how much software has positively influenced our quality of life. The benefits of software, when it does finally ship and it does finally have a high enough level of quality are worth waiting for. Otherwise, of course, there would no longer be a software industry.

There have been incremental improvements to the software development process which have produced incremental improvements and have become widespread. Examples are: the use of software development tools such as source control and issue tracking, breaking down large tasks into small tasks, nightly builds, one-step build process, moving from procedural programming to object-oriented programming and many others. The important point here is that each of these improvements have become widely adopted and made things somewhat better, but still didn’t solve the root problem. The process is still unpredictable and unreliable.

This is akin to the $O(n)$ approach to problem solving. If you have an algorithm which takes $100 \cdot n^2$ operations, then getting that down to $50 \cdot n^2$ is a good thing, but changing the algorithm to be $200 \cdot n$ is much better. To date, changes to the software process have been of the first variety, shrinking the constant.

Traditional Development is a Chinese Finger Puzzle



Figure: Chinese Finger Puzzle

One of the difficulties in moving to Agile development is that much of the “knowledge” that we have from decades of traditional development makes Agile seem counterintuitive. This knowledge is embodied in habits, taboos, and ceremonies. There are things which we believe are facts in general which are actually only facts within the framework of traditional development. In effect, Agile is right in our blind spot.

While many aspects of Agile development are the same as traditional development, many of the practices of traditional development are actually accommodations for problems that only occur in a traditional project and thus are not necessary in an Agile project. We feel that we need these accommodations because they have always been there, but we need to recognize them for what they are in order to embrace Agile development.

In traditional development, a frequent response to many problems is adding more rigor, taking more time, adding more detail, and adding more policies and procedures. This is often expressed as “this time we’ll do things right!” It is determined that the reason for problems last time was a lack of discipline and so there is a call for more discipline. This is analogous to the Chinese Finger Puzzle. The solution appears to be adding more policies and procedures (pulling harder on the finger puzzle), but in fact the solution is to simplify (pushing your fingers together.)

Problems Hide Longer Than Necessary

Let's say that you have 600 planned work items for a six month release. Of course, the system test comes near the end of the cycle because you know it will be painful and you don't want to go through that process more than once. You may do all of the specification and design up front for all of the changes, or you may do it in batches. Perhaps you then move on to doing all of the coding and testing, or maybe you start coding as the batches of specifications and designs are finished.

Regardless of the exact process, you won't start to get information about how good the requirements, specification, design, and coding are until the system test starts. Now you may find that the requirement gathering was done very poorly because the testers can't make heads or tails of how things are supposed to work.

You now have to take more time than planned because of the rework required for many or most of the 600 work items, not just in the coding but also in revisiting the requirements, specifications, and design, and rewriting the test plans and automated tests. If, instead, you had broken the release into 6 iterations of 100 work items each, you would have found the problems associated with the 100 work items during the first iteration, corrected them, and as a result the impact of those problems would have been reduced by a factor of 5.

The Testing Gauntlet

Why does it take so long to qualify a release? Let's look at this a different way. What is the critical path time for a customer-down hot fix in a high-impact area of the code? Your company probably has a special process for this situation, and once an issue has reached this status, it undoubtedly gets top priority and all of the resources it needs. Thus, there will be no dependencies on other issues, no waiting for somebody to get back from vacation, or any other delays in the critical path other than the time directly associated with the sub-tasks required to produce the hot fix. Since this involves a high-impact area of the code, it should get the highest level of scrutiny that your organization has to offer: multiple code reviews, regression tests, manual testing, the addition of new tests, stress testing, etc.

This type of hot fix typically takes four to eight hours with a worst case of 24 hours. Because time is critical for this issue, testing is limited to a subset of the testing that is done for a regular release. For example, only platform-specific testing or data integrity testing is done first; then the full battery of testing is done post-delivery, which might take another day. Most organizations can turn around a high-quality release that contains just a small change in two days or less if they really have to.

The critical path exercise just described demonstrates that the actual overhead associated with a single small change is at most two days. Since most of that overhead is related either to fully automated testing or to stepping through a manual test plan, and needs to be run through only once, why is it that so many organizations perceive that there is a much larger overhead—sometimes as much as three months for a one-year release cycle—associated with getting a product out the door?

It boils down to two simple answers: breaking the habit of long iterations is difficult, and long iterations hide fundamental problems. To make matters worse, these two problems tend to reinforce each other.

The hidden problems help to keep the cycle time long, and the long cycle time helps to keep the problems hidden.

The many variations of hidden problems all stem from one root cause: feedback on process problems comes late in the cycle. As the saying goes, “An ounce of prevention is worth a pound of cure.”

The delays associated with process problems being detected late in the cycle occur every release, which is why so much time is reserved for the end game. It isn't the qualification of the release that takes so long—that still takes only two days. It is the process of getting the release to the point that it makes it through that two-day gauntlet, without any problems, that takes so long.

Code Churn: The Hydra Syndrome

Once system testing begins, there is enormous pressure to both fix problems and meet the deadline. Many people are making many changes to interdependent systems. Just as module A, which depends on module B, starts to work with the previous set of changes made to module B, the next set of changes to B comes along and now A no longer works. Plus, the set of changes to A breaks C, and so on. Since so much is changing, it takes a long time to do root-cause analysis to figure out how to solve problems and make fixes. Sometimes it seems that for every problem solved, two new ones appear.

Since everybody “knows” that it takes a long time to qualify a major version of software for release, you of course want to get as much in there as possible. This desire must be balanced against the fact that if you take too long to get the next release out the door, you may fall behind your competitors or miss out on opportunities. As the planned release date comes and goes, you get nervous that it is taking too long and that you're going to break all of those promises you made, so you look for shortcuts. Eventually, the release goes out, but it has some problems and has to be patched a few times before settling down. Everybody remembers that and you vow that next time you won't take any shortcuts, thus reinforcing the belief that shortening the process produces lower quality.

Short Iterations, The Key to Agile Development

Feedback

We get feedback our whole lives. When we first come into the world, we need constant feedback in order to learn how to survive. Touching something hot is painful, going without food is painful, and falling down stairs is painful. In school we get feedback as to which subjects we are good at and which subjects we are not so good at. We can use this feedback to decide what to do more of, what to avoid and as an aid to improve, if we are able to.

Feedback is an important part of both learning and remembering the laws, rules, and processes that are all around you. You don't have to consciously think about them but you are constantly reacting to them and they are second nature to you. Because it is second nature, it is easier to go with the flow than it is to go against it. On your way to work you drive on the correct side of the road, you stop at stop signs, and you stay in your own lane. When you go to lunch, you pay for your food. After work you go to your house instead of going to somebody else's house.

It is easy to remember the laws, rules, and processes because you interact with them every day and you get feedback in the form of reminders and possibly even unpleasant consequences when you go astray. A great example of this is the rumble strip that has become commonplace on the side of highways. If for some reason you start to drift off the road, you'll get feedback in the form of a loud warning noise as your tires hit the rumble strip. If you are really violating the law, you'll hear an even louder sound accompanied by flashing lights.

An important part of absorbing all of these laws, rules, and processes is that they are clearly defined, easy to understand, and easy to remember. For instance, once you have mastered basic physical laws such as the fact that running at full speed into a door is painful, it is easy to understand the reasoning behind having an upper limit on the speed of vehicles on public roads. You may disagree with that limit, but you can generally guess what the limit is for a given stretch of road and it is posted at regular intervals to help you score your guesses. Sometimes there is even a referee.

Another process that you are part of is your software development process. With traditional development, the feedback is greatly delayed. For instance, it may be six months or a year between when a customer asks for something and you are able to demo it and get feedback on the result.

With Agile, the feedback loop is usually no longer than a month and often on a weekly basis. Agile is a simpler process which is more in keeping with natural human rhythms and capabilities. It provides a more people-oriented software development environment.

Accidentally Agile

In many ways, what you are doing now undoubtedly overlaps with the Agile practice of short iterations. If you do one or two major releases per year, you probably don't think of yourself as doing short iterations. But hold on, what about all of those release candidates you produce near the end of a release cycle? Sure, you probably released only a couple of them externally, and even then they were betas, not

“real” releases. And what about the three unplanned follow-on releases you did after that major release? How about hot fixes? Customer one-offs? Builds for QA? Demo builds?

If your customer is internal, you probably release once or twice per week or perhaps even twice per day. Before you start claiming to be agile, however, realize that frequent releases alone do not qualify as agile development.

You may think of all those unplanned and multiple candidate releases as exceptions or as impediments to producing high-quality software. Another way to look at it, however, is that change is a natural part of life and it is better to accept and embrace change as an opportunity and learn how to incorporate it into our plans rather than expect and hope that it can be reduced.

The funny thing is that most shops are unconsciously doing variations of Agile Development most of the time. They do an Agile release on a regular basis without even realizing it and then feel ashamed that they had to do it. Afterwards, they hope they won't have to do another one. Those feelings of guilt and shame get associated with the Agile release and it doesn't occur to anybody that they are missing what is right in front of them. I know that I've been looking the evidence right in the face for years and I never saw it.

I am speaking of maintenance releases, patches, and hotfixes. Forget for the moment that these are created out of necessity due to problems with “the big release” and that they are often done under intense pressure. Forget for the moment that sometimes you have to redo a hotfix with another hotfix. The need for the second hotfix is often found by the intense scrutiny that is in place during the aftermath of the first hotfix and due to the fact that the hotfix probably didn't get the full spin cycle of the normal QA process.

Think for a moment about how you later felt about that maintenance release. Didn't you think something like “too bad we had to do it, but we really pulled together and I'm really proud of that release?” Wouldn't you say that the type of release that spends the most time in production is actually a maintenance release or patch release?

A lot of the time, those maintenance releases aren't just maintenance releases. Usually, there are a smattering of small enhancements thrown in for good measure to appease a customer or two.

So, let's review. Your maintenance release has the highest quality, was produced in a short timeframe in comparison to your major releases, and it has a fair amount of new features in it. Guess what? This is very close to Agile development. Surprise! But then of course, you revert back to your old habits.

I believe the reason that most shops drift in and out of Agile development is that it fits better with our natural capabilities as people and it is hard to resist it. The challenge is to sustain it. In order to sustain it, you have to consciously intend to do it. Without conscious intent it is hard to notice the difference between being on the path and off the path unless you stay on the path for a while. The other problem is that while everybody in an organization might from time to time all be synchronized into an Agile

rhythm by pure chance, the synchronizing signal of the project plan, habits, and entrenched process interferes with the natural rhythm. Thus, the natural rhythm doesn't get a chance to take hold.

Two of the main ingredients of Agile are short iterations and one-piece flow. What this amounts to is doing a small amount of high-value work in a short period of time and making sure that you've got the requirements and design right, the code is written, that code has the right tests written, and they all pass.

With this in mind, let's take a look at what actually happens in the traditional day to day activities of software development over a span of a year. Let's say that for a particular organization, they have a planned release cycle of six months and they also do maintenance work on at least one previous release. Further, the maintenance work takes up 4 months of effort. Also, in reality that six months planned release cycle stretches out to 8 months. In summary, that's 5 months of new development work, 1 month of planned testing on that development, 2 months of schedule slip, and 4 months of maintenance work.

Let's take a closer look at the testing and schedule slip. In many cases, half of that schedule slip is scope creep, and the other half of it is unplanned testing and fixing. Scope creep is usually smallish stuff. If we look at what has now become 2 months of testing, that's really 2 months of testing, finding problems, fixing, and then going around again and again. During this phase, the bugs that are found and fixed also get new tests so that they won't reoccur. All of this is done in a very tight loop. This resembles a badly implemented Agile process. Agile teams do something similar to this, but they do it intentionally and as a result the actual process is much smoother and much shorter.

Let's total up all of the time spent being unintentionally Agile: 1 month of planned testing, 1 month of unplanned testing, 1 month of feature creep, and 4 months of maintenance work. That's seven months out of twelve doing Agile badly!

There's a big difference between doing parts or all of Agile unintentionally and doing it intentionally. When you are doing it intentionally, you can take advantage of the knowledge base that has grown up around Agile and tweak what you are doing to take advantage of what other people have learned. It is also much easier to get the full benefits on a regular basis when you are doing Agile intentionally.

Unintentionally Agile

In the early days of AccuRev, before we had any customers, we were unintentionally doing Agile development and we got most of the benefits. As the product owner, I maintained a backlog in a simple text file which was constantly updated, though we didn't use the terms product owner or backlog. We had two week iterations, though we used the terms "phase" and "stage". Each phase was a self-hosting milestone, and every stage was an iteration. AccuRev self-hosted right from the first iteration.

While we had done a fair amount of up-front conceptual design, it was mostly used to direct our path as we implemented. We used incremental design and YAGNI (though we didn't call it that), and within the same iteration we created the tests for the

functionality that we had implemented in that iteration. While we didn't do continuous integration, we did do nightly builds as well as full builds and fully automated test runs multiple times per day. We were entirely collocated and we frequently did pair programming.

We also had two very early customers that used the product for free a year before it was officially released. They took updates from us every two weeks and gave us immediate feedback.

Initially we had planned to offer a GUI, build management, issue tracking, replication, and change packages in the first version. Thankfully, the combination of YAGNI, self-hosting, backlog, and frequent releases kept us focused on the highest business value. It turned out that the highest business value was having a fully-functioning Stream-based SCM system that was cross-platform and worked via TCP/IP.

If we had used traditional development, then in the same timeframe we produced AccuRev 1.0 we would have had a little bit of all of the features we left out and nothing to sell for a long time. We would also have missed out on the discovery that took place during that time via the self-hosting and frequent releases. Unfortunately, as we grew we changed to do things “the way real companies do them” and we lost many of the benefits of our unintentionally Agile ways without even realizing it. The only concepts of Agile that we kept between then and our rediscovery of Agile were automated testing and YAGNI.

Short Iterations Reschedule Traditional Development Tasks

Consider the various tasks that take place during a release. Basically, there is a preparation phase consisting of things like gathering and elaborating requirements, creating specifications and designs, planning, etc. That's followed by coding, integrating, test writing, and then the test period.



Figure: traditional development during a release.

Short iterations are primarily a rescheduling of the many tasks involved in a single traditional release. Each task—writing the spec, doing the design, writing the code, writing the automated tests, etc.—for any particular change still takes the same amount of time. The difference is that instead of these tasks being spread out over the full timeframe of a release, these tasks now occur within an iteration which may be anywhere from a week to a month in length.

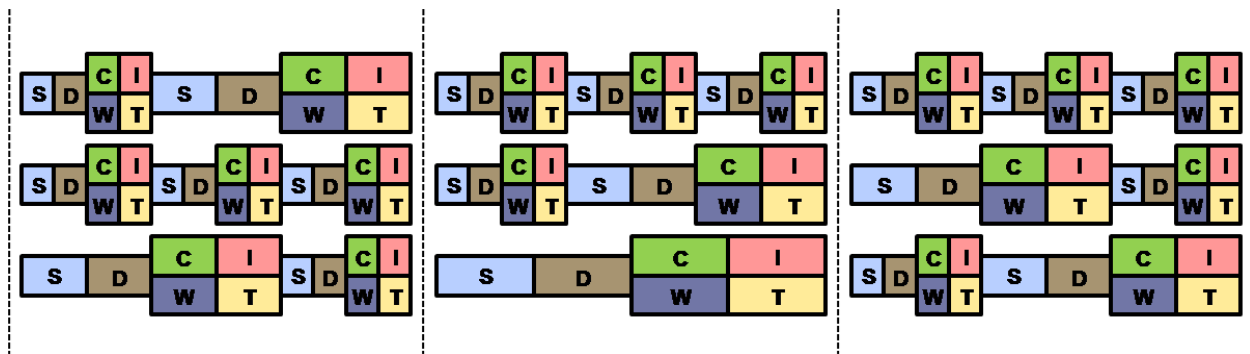


Figure: rescheduling of activities.

Say Goodbye to Feature Creep

During the long march from the start of a project to shipping it, the pressure to add more work items to the release grows and grows. After all, what are one or two more items among 500? Surely there will be time to fit them in!

How soon we forget our arch-nemesis: feature creep. Before you know it, you've been seduced into adding an extra 30 work items to the plan and the code freeze date is upon you. Somehow you have ended up doing lots of small new items but you still haven't done some of the must-have items from the original plan. But that's OK; you still have a long test period at the end, so you can get at least some of those items done then. You would like to have a high-quality release as close to on-time as possible. So, painful as it may be, you cut some of those must-have items from the plan and target a new follow-on release for them right after the main release.

Feature creep is a function of the time between releases. The more features you add, the longer it takes to get the release done, the more opportunity you have for feature creep. This reinforces the perception that releases take a long time which reinforces the bad habit of feature creep.

With short iterations, the development cycle is usually on the order of a week to a month. If a feature that was not previously considered important is escalated in priority, it only has to wait until the next iteration to get attention. In order to get into the iteration, something else will have to be taken out. This process serves to reduce the need for and the effects of feature creep.

Final Qualification of a Release

Let's take a look at an important aspect of traditional development, the final qualification of a release. The activities that take place from code-freeze to release involve a rapidly repeating mix of test case writing, test automation, testing, and coding. This coding is often called "rework." Typically, there are many problems found during the first round of testing and for a while the number of issues will go up and down with no discernable pattern until finally it declines to an acceptable level and the product ships. The whole time there is a struggle between the desire for high quality and the desire to ship the product.

In an Agile project, all of the test writing for each work item is redistributed to whichever iteration the work item will be done in. Thus, there is no test writing period during the end game of a release, which significantly shortens the time between code freeze and release. Also, since a work item is not considered done until all of its tests pass, the only problems found after code freeze are those that can only be found after code freeze. As a result, the number of problems found after code freeze is very small and the number of test/fix cycles is very low.

Remote Feedback

Perhaps your team has recently expanded, added distributed team members, or started doing outsourcing with a team half way around the world only to find that code churn has not only gotten worse but now it is also harder to coordinate with others to track down and correct the problem. One of the biggest problems with distributed development is communication. It is especially difficult to accurately gauge progress, spot the cause of problems, and determine the result of corrective action. These are the exact same trouble spots as with collocated teams, they are just amplified in proportion to the amount of physical separation.

One of the useful side-effects of short iterations is that you can get a true picture of actual status and progress more quickly from teams that are far away. Let's say you have just added an offshore development team that you haven't worked with before. In a traditional project, you may get frequent status reports saying that everything is going well, but how can you really tell? With the short iterations of an Agile project, you can expect to get working software on a regular basis. If you don't, or if it has less functionality than promised, that's another indication of trouble. In a more positive light, getting working code on a regular basis that is demonstrated to have the promised functionality provides confidence that you can trust the remote team.

Dealing with Uncertainty

Software Development is Inherently Unpredictable

It is impossible to know exactly how many customers or users you will have, exactly what features will bring in the most revenue, which implementation technologies will work the best for you, the effective useful life of various implementation technologies, or which of many possible features and combinations of features will best satisfy your users. You'll be working on tasks that haven't been done before, solving problems for new circumstances, and using new technologies.

To account for all of this you must use creativity, make guesses, and use trial and error. All of these techniques are essentially variations of predicting the future. As a result, software projects are highly unpredictable. More often than not they are either late, over budget, missing originally planned contents, much lower in quality than originally planned or a mix of all of the above.

One of the perceptions of Agile development is that because it is hard to know exactly what will happen with a software project, you might as well give up on predictability, schedules, and traditional project planning. In return, you will get the highest priority changes on a regular basis, so things aren't all that bad. After all, big releases are rarely delivered with the originally planned content on the originally planned schedule.

This sounds wonderful to developers. Finally! Somebody has come out and said it. The crazy deadlines and pressure are off. The inherent unpredictability of software development has been exposed! Vive la revolution! But hold on a moment. There are other professions which have similar problems and they have evolved effective techniques to cope with unpredictability.

Sales are Inherently Unpredictable

Another profession that has similar problems with predictability is sales. I know what you are thinking. What good are salespeople for? Good software doesn't need salespeople because good software sells itself and the only thing sales people are good at is wining and dining management in order to push unwanted products onto unsuspecting users and running off with their cash.

I used to think that as well until I experienced the benefits of a good sales organization first hand. It turns out that sales organizations are remarkably good at dealing with unpredictability. Sales people can't predict exactly which deals will come in during a quarter, nor exactly what the amount of any particular deal will be, but they are expected to hit their quotas and they are expected to show results on a regular basis. CFOs are perfectly happy with how their sales force goes about their process. They may not always be happy with the actual results, but they approve of the process. It is surprising in contrast that not only is the predictability of software projects worse than sales projections, it is tolerated more.

Successful businesses rely on sales forces to predict the future with a surprisingly high degree of accuracy. On the surface this may seem impossible. It would seem that developers have much more control over their own destiny. Whether somebody will be interested in buying a product on a given

date or how much they will be interested in buying seems completely random and unpredictable. Similarly, whether there is a sufficiently well performing algorithm to solve a problem and exactly how long it will take to find it, implement it, and verify it also seems completely random and unpredictable.

I'm not suggesting that sales and engineering are identical or that development should follow the example of what sales does exactly. I'm only using sales as an example of how another profession successfully deals with the problem of unpredictability. How the unpredictability manifests is not exactly the same. Good unpredictability happens in sales, sometimes an order will come in out of the blue and require very little effort to close. Salespeople call those "bluebirds." In reality, those don't happen very often. There is a similar phenomenon in software development. That's called "finishing ahead of time." Also rare! In any case, unpredictable is unpredictable.

Predicting the Future

Predicting the future is hard. The more information you gather and the more variables you consider, the better your prediction will be, but the more time your prediction will take. Consider the problem of predicting the weather. Modern forecasting software is pretty good, but the further out you look, the longer it takes for the forecast to run and the more likely it is for the forecast to finish after the day you are forecasting to have already come and gone.

Sometimes it seems like technology and user needs change almost as frequently as the weather, making prediction equally difficult. The more possibilities you take into account when you design your software, the bigger and more complex your design will become, the longer it will take you to produce the design. The bigger and more complex your design becomes, the more likely it is that you will have difficult design problems to solve. Solving difficult design problems can take a long time. Once you've finished this big and complex design, it will take a proportionately long time to implement that design.

When you put all of this together it is obvious that the more accurately you try to predict the future, the more time and effort it will take. The question is, how do you decide when your design is good enough? If you take it too far, you'll either never finish your design, or you'll never ship your software. But if you don't spend enough time on design, you may end up with unhappy users or software that needs to be rewritten.

Sales Techniques for Managing Unpredictability

Sales organizations rely on four techniques to manage unpredictability that are directly applicable to software development: using a sales pipeline, dividing time up into short intervals, forecasting, and process improvement. Let's take a look at each of these in turn and see how it relates to software development.

Sales Pipeline

Here is what a raw sales database might look like without doing forecasting. It is just a list of people that sales has been in contact with. If the contact has expressed an interest in a certain amount of product,

then the potential amount of the sale (opportunity) is listed. From this list it is pretty much impossible to predict sales.

Account	Opportunity
World Wide Widgets	unknown
Polyphonic	unknown
Zoom	\$ 20,000.00
Amazing Movies	unknown
Boxes and more	\$ 2,000.00
Shoesource	\$ 10,000.00
Just In Time Gizmos	\$ 1,000.00
Acme Inc.	unknown
Local Express	\$ 4,000.00
PDA Software	\$ 30,000.00
Ziffy	\$ 50,000.00
FlipFlap	\$ 40,000.00
Circuit Town	unknown
HappySoft	\$ 40,000.00
FinTransCo	\$ 50,000.00

In order to provide more information, each opportunity is associated with the current point to which it has advanced in what is called the sales pipeline. The pipeline is just a series of stages. The input to the pipeline is people that have either contacted or been contacted by the organization. They may be contacting the organization out of curiosity, real interest, or they may just have the wrong number or e-mail address. The output of the pipeline is orders for the company's product. Here's an example set of pipeline stages: suspect, qualified, interested, technical win, business win, in purchasing, and received purchase order.



Each of these pipeline stages acts as a filter. You assign a probability to each opportunity based on the stage that it is in. The further along an opportunity is in the pipeline, the more likely it is to result in a sale.

The Software Development Pipeline

The sales pipeline should seem very familiar to you. It is analogous to the typical steps in the software development process. There are set process stages that each work item goes through such as requirements gathering, requirements specification, design, development, integration, etc. Perhaps that sounds more like waterfall development than Agile development. Agile doesn't have development stages, right?

But when was the last time you saw code spontaneously pop into existence without being written, or code being written without any goal in mind, even if it was as vague as "make it better?" All changes to software go through a cascade of stages. Even if it is a fuzzy idea, change starts with a concept for what needs to be done. From this concept comes a plan. The plan may exist only as a flash of insight in the mind of the developer and they may not even be conscious of it, but there is always a plan. From the plan comes the code. Lastly, the proof of the original concept comes from trying the result. Thus, even in the least rigorous of environments, working on a single issue, there is at least a 6 step process that is followed.

All of the stages described may be intermingled, have no clear start or finish, and be repeated multiple times, but they are there. Here is an example set of stages for software development.

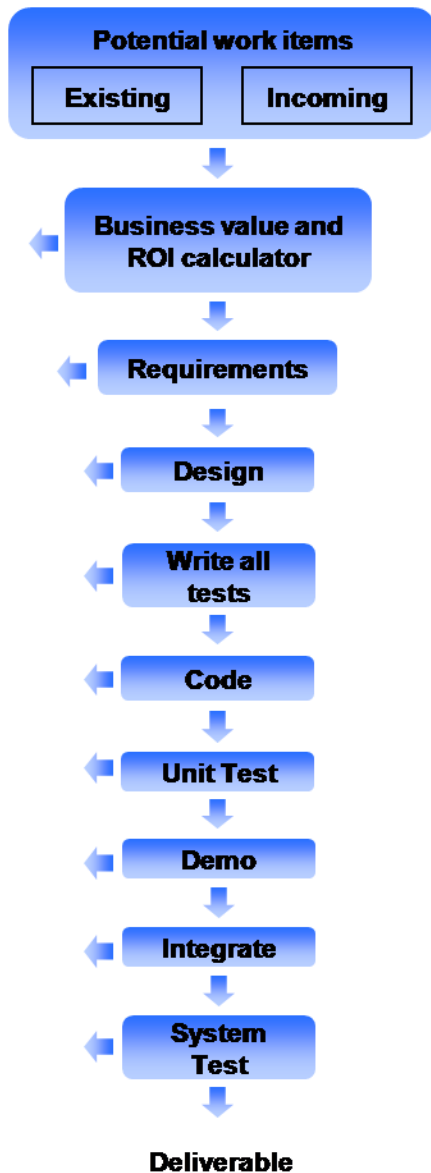


Figure: an example software development cascade.

The value of the cascade, when applied to a single work item, is in producing a well defined series of steps which when followed will detect and filter out problems. For any particular development team, those steps may vary a great deal, but by refining the steps and applying them consistently, you have a higher likelihood of producing consistent results.

Each of the stages of development also serves as a predictor. For instance, you know that if a task is in the design phase and it is taking much more time than expected or there is a problem which is proving

to be a tough nut to crack, there is a good chance that all further phases will be difficult. It isn't guaranteed, but it is a good leading indicator.

Sales Quarters

Sales divides the year up into short periods of time called quarters. Each quarter consists of three months and there are four quarters in the year. The quarters are also part of the pipeline. The overall pipeline consists of opportunities that are distributed across the quarters and also associated with a particular pipeline stage.

The combination of pipeline stages and quarters also serves to surface problems. If an opportunity is stuck at one stage in the pipeline, it gives you an early heads up that there may be a problem. If an opportunity has been forecast for a particular quarter, but a problem arises, it can be moved to another quarter further out. On the other hand, if it starts to move through the stages more rapidly than predicted, it is often moved to a quarter closer to the current date.

Short Iterations

Agile development uses short iterations in much the same way that sales uses quarters. Each release is broken up into short iterations, typically a month or less. The high value work is planned for earlier iterations, and the lower value work is planned for later iterations.

By breaking the release up into short iterations, there is a much lower chance that any one of them is delayed. If there is a problem in an iteration, none of the functionality leading up to the problem iteration is "held hostage" until the cause of the delay is removed. You can release at any time.

If a work item has been planned for a particular iteration, but a problem arises or its value drops, it can be easily moved to a later iteration. On the other hand, if its value rises it can be moved to an earlier iteration.

This is completely in line with customer expectations. Which would you rather do, tell a customer that all of the changes you promised will be late and you're not exactly sure how late or how reliable your estimate is, or tell them that 95% of what you planned will in fact be delivered on time and 5% is completely unknown? I think that's a no-brainer.

Sales Forecast

Sales uses the pipeline to produce a sales forecast. For each opportunity, the probability is multiplied by the deal size to produce a weighted amount. You add up all of the weighted amounts, and that produces your sales forecast.

Salesperson	Account	Opportunity	Stage	Quarter	Weighted Opportunity
joe	World Wide Widgets	\$ 5,000.00	10%	Q1	\$ 500.00
Joe	Zoom	\$ 20,000.00	50%	Q1	\$ 10,000.00
libby	Amazing Movies	\$ 5,000.00	90%	Q1	\$ 4,500.00
libby	Boxes and more	\$ 2,000.00	100%	Q1	\$ 2,000.00
		\$ 32,000.00			\$ 17,000.00
joe	Just In Time Gizmos	\$ 1,000.00	25%	Q2	\$ 250.00
susan	Acme Inc.	\$ 10,000.00	25%	Q2	\$ 2,500.00
frank	Local Express	\$ 4,000.00	50%	Q2	\$ 2,000.00
Tim	PDA Software	\$ 30,000.00	25%	Q2	\$ 7,500.00
		\$ 45,000.00			\$ 12,250.00
Libby	Ziffy	\$ 50,000.00	10%	Q3	\$ 5,000.00
Joe	FlipFlap	\$ 40,000.00	10%	Q3	\$ 4,000.00
Susan	Circuit Town	\$ 30,000.00	10%	Q3	\$ 3,000.00
Frank	HappySoft	\$ 40,000.00	10%	Q3	\$ 4,000.00
		\$160,000.00			\$ 16,000.00

Figure: Sales forecast example

The sales forecast is a surprisingly accurate and effective tool which is based on two underlying assumptions. First, it is impossible to reliably predict the outcome of any particular opportunity in the pipeline. Second, it is possible to organize opportunities in such a way that you can make useful predictions on your pipeline as a whole and act on that information in order to further increase the accuracy of your predictions.

It gives you a heads up on problems early so that you can react. Not getting enough leads? Focus on marketing. Not doing enough demos? Focus on sales. Not getting enough technical wins? Focus on product. Not getting enough business wins? Focus on sales tools and messaging for management. The forecast also helps in predicting cash flow in order to help with financial planning.

The sales forecast does rely heavily on four key factors for accuracy: the accuracy of the sales model itself, the experience of the salespeople, the number of salespeople, and the number of opportunities in the pipeline. If you have just created a sales model, it is unlikely to be very accurate, but it doesn't matter as long as you continuously refine the model based on your experience.

Some salespeople are very good at keeping their sales data up to date and as accurate as possible, and others are not as good. The more that the salespeople believe in the model and work with it, the better data you will have as input into the model.

Of course, the more salespeople you have, and the more deals you have in the pipeline, the less sensitive the model is to problems with individual salespeople or individual deals and thus the more effective it is at forecasting actual sales.

Sales Process

The sales pipeline and forecast are all part of the sales process. In most sales organizations, at least the good ones, the process is encapsulated in a document and possibly a PowerPoint presentation. Both documents are kept short. As parts of the process are shown to work well or not so well, the process is updated and the sales process documents are updated to reflect the changes. Changes are communicated to the whole team in real time and at the beginning of each quarter there is a sales “kickoff” which reviews the process and highlights all changes since the last quarter. It is very clear to all participants what the process is and how it works. As a result, what would appear on the surface to be a completely unpredictable process becomes surprisingly predictable. Of course, there are always surprises, but they don’t spoil the overall effect.

The Business Side of Agile Development

What sales does and what Agile teams do are *exactly* the same! They are both working with unpredictable quantities. Developers in an Agile team may not be able to predict exactly what will be the result of any particular release and sometimes not even the result of an iteration, but they can be held accountable to produce a regular flow of high business value software on a regular basis and to be able to demonstrate that the software is actually done on a regular basis as well.

So, if the organization is comfortable with the sales process, they should be equally comfortable with Agile development. In fact, anybody involved in the business side of software development should be very uncomfortable with anything other than Agile development and be actively working to figure out how to get their development process to be more like sales.

Art vs. Manufacturing

One objection to thinking of software development as predictable is that software development is a creative process. Developers often refer to what they do as “craftsmanship.” Developers want creative freedom and associate Process (“the P word”) with bureaucracy. They believe that software automates a well-defined problem but that the process of software development is a creative endeavor and cannot benefit from the same kind of thinking that goes into designing software.

On the other side, the business prefers a well-defined process, closer to a manufacturing approach with the predictability of an assembly line. This creates a constant friction where neither side gets what it wants. Each side believes that the other “just doesn’t get it.”

Creativity vs Infrastructure

Another way to increase overall predictability is to make a clear separation between creativity and infrastructure. After spending many years involved in thinking about how to improve the process of software development, experience has convinced me that a great deal of the process of developing software can be categorized as infrastructure. This infrastructure supports the creative parts of the software development process, but can be clearly delineated from it and dealt with independently.

The parts of the software development process that I associate with creativity are things like deciding which markets to go after; deciding which features to do; inventing new algorithms; deciding which algorithms to use; and writing user stories, documentation, tests, and the code itself. The parts of the software development process that I associate with infrastructure are things like scheduling and running builds, getting code in and out of your SCM system, tracking project management information, assigning issues, running automated tests, and the exact process that you use to orchestrate your software development activities.

Agile Provides a Manufacturing Platform for Creative Activities

By using short iterations and one-piece-flow and drawing a clear line between creativity and infrastructure, you can create a very stable and predictable manufacturing platform that forms the foundation of your creative activities.

Only work that is deemed ready for the iteration goes into the iteration, minimizing rework. Once work is in the current iteration it is completed, tested, and integrated. If there is a problem during “assembly,” the work is removed. Work is started and taken all the way to a deployment-ready state as fast as possible so that problems are found as fast as possible and either dealt with or removed right away, keeping the work constantly flowing. The iterations provide regular and predictable milestones for gauging real project progress and status. With Agile, both development and management get what they want.

The Software Development Lifecycle

All Phases of Software Development are Variations on a Theme

It is easy to think of specification, estimation, planning, design, implementation, and customer feedback as completely separate and unrelated activities. However, a useful way to think about them is as variations on a theme where the theme is producing a product which provides the best return on investment as measured by the ratio of revenues to cost. Each of the phases of the development life-cycle can be thought of as a combination of discovery, prediction, and design. The prediction comes into play when you make a decision. When you decide that you will schedule a particular requirement to be implemented in your product, you are predicting that the end result is something that will provide value to the customer. You have no way of being sure until the customer has actually used it as part of their normal course of business.

What does it matter if in fact it is true that all of these activities are variations on a theme? I believe it provides a conceptual framework that can help us make better decisions about the products we create and the way we go about creating them.

Requirements Gathering is Not Design

Requirements gathering is not the process of finding out what customers need, it is the process of understanding the customer. The problem with producing elaborate requirements is not just that requirements change over time or that people change their minds, it is that users can't tell you what they want because they don't know. Customers cannot tell you ahead of time what they will actually spend money on. At best, requirements are an approximation of what customers need and a best guess of what will provide them value at the time that they actually use what you implement for them.

Gathering requirements from customers and the market in general is much like removing compiler errors. For any particular module that you are working on, the first error is the most accurate and accuracy goes downhill rapidly from there. After you fix the first error, the results from your compiler or IDE will probably be very different. This is the same with customer requirements. For any particular functional area, the first thing customers ask for is most likely the most accurate. Once they see the new version which incorporates that feedback, what they want next is probably very different than what they would have asked for next before they saw the new version.

The most you can hope for is to gather information and ideas that are useful in producing a successful product. If you just build whatever customers ask for, then you are ceding design to them. Customers cannot design for you. If they can then they are not just customers they are also designers. It may be that in your situation your customer is also a designer, but then you should explicitly think of them that way. Otherwise, you run the risk of making the mistake of blending the process of gathering requirements and creating a design.

When you have multiple customers, it is especially true that the customer cannot tell you what they

need. It is rare that you will be able to do everything that all customers asked for, let alone everything that any one customer asked for and thus you will have to predict which combination of requests that your customers asked for will produce the best result.

Specification is Design

When you write a Marketing Requirements Document, Product Requirements Document, Specification Document, or whatever combination of documents you create along these lines and whatever you call them, let's call that "specification." A specification is a design. It is a design because you are taking all of the feedback from the customer and saying, at a high level, "this is what will provide value to the customer." You are deciding and predicting that this specification rather than any other specification is the way to go forward. It is the same as deciding that this line of code vs that line of code is the right way to go.

Specification is the First Approximation

When working with software, even if you have clear requirements/specifications that you have validated with users, it is not always clear how you will implement them. The specification is an approximation in two ways. First, it is only an approximation of what users actually want. Second, it is only an approximation of the final result. This same statement holds true at every stage of the game, from specification to design, to plans, to estimates, to the code itself. From initial customer interaction to final implementation you are building approximation on top of approximation.

I'm not advocating against specification, estimation, or design. It is useful and even necessary to produce these, and you will often hit on solutions or eliminate problems much more quickly and much more economically than jumping right to implementation, but in the end they are still only an approximation of reality, an interlocking web of educated guesses.

Not until you actually attempt to create something real will you find out how close to reality all of these approximations are and the real costs and difficulties involved. The longer you take to turn these approximations into reality, the more likely it is that they will be inaccurate and the larger those inaccuracies will be.

Estimation is Design

Estimation, which is also another word for approximation, is part of design. If somebody gives you a requirement do you just shout out a random number as the estimate? The accuracy of your estimate depends on the amount of design that you put into it. You may not think of estimation as design, but it is. Let's say that somebody says "I need a module which will calculate the sum of an array." If you have such a module at hand, you will give an estimate that involves integrating the existing module into the overall product. Deciding to use that module is a design decision. If you don't have such a module at hand, you will give an answer based on your prior experience doing the exact same thing. The decision to use an approach similar to a past experience is also a design decision. It is part of creating the final design.

On the other hand, if you are given a more complicated requirement, then you will probably break it down into sub-tasks and estimate those. That forces you to think about what it will really take to implement the requirement. The process of deciding what the subtasks are is actually design. The more design you do, the more accurate you can be. The accuracy of an estimation is on a spectrum from “wild guess” to delivery of product, from complete prediction to statement of fact, from “I believe it will take 10 days” to “it took 103 hours.” The other aspect here is how much is research/design and how much is counting. When well-known tasks are what is at hand, then it is a matter of counting. If you know you need to change 20 dialogs and each change will take 10 minutes then you know the task will take 200 minutes. That’s simple counting. If you need to produce something that has never been done before, then it is a research project and you won’t have accuracy until you have removed all of the uncertainty.

Planning is Design

The specific set of features that go into a product is in fact a design. By selecting that set, you are deciding (predicting) that this is the optimal set of features to achieve a chosen goal. While it is true that you may just be choosing a set of work items that sum up to the available time for a release, that is still design, but perhaps not the optimal design.

Design is Design

Obviously, the activities that we associate with “the design process” are part of product design. The design that you have prior to implementation may be a fully fleshed out design document with lots of illustrative figures, it may be in the form of UML, or it might be entirely in your head. As with requirements and estimation, the design is also an approximation. The design that you do prior to implementation is only an approximation because during implementation you have still more design to do and you will discover information that you didn’t have during your initial design.

Implementation is Design

During implementation you are doing two kinds of design: micro-design and re-design. Micro-design is all of the little decisions you make while coding. Will you use a vector or a doubly-linked list? Will you specify an index or let the RDBMS do it for you on the fly? Re-design happens when you find out that once it came time to implement something, an unexpected problem arose. For instance, the design specifies the use of a particular third-party library, but in reality that library doesn’t provide the functionality that it claimed to provide. Another example is that the implementation works fine, but doesn’t provide the expected performance and no amount of tweaking seems to be doing the trick.

Final Validation of the Design

Finally, when you deliver new features to customers, you discover how close your approximations were. You discover what customers like and don’t like, what they use and what they don’t use, what they will pay for and what they will not pay for, what provides value and what does not provide value. This information can then be used to determine what you will put more effort into and what you will discontinue. The tighter you can make the loop from discovery to delivery, the faster you can produce the product that your customers really value.

Software Development is a Search Algorithm

How we think about something can have a profound effect on how we do that thing. Thinking of the process of software development as a search algorithm rather than as the process of building software can lead to making better decisions about the products we create and the way we go about creating them.

The Search For Value

The end goal of designing software is to produce something which provides a benefit that is large enough for one or more customers to at least cover the cost of its production. Of course, the higher the value of that benefit, the more that folks will pay for it. On the other side of the equation you want to provide that benefit at the lowest cost. There is a point of diminishing returns both in looking for the highest possible value and in looking for the lowest possible cost.

As much as we might want this process to be deterministic, it isn't. Just because you produce something doesn't mean people will use it, and just because they use it doesn't mean they will pay you what you want for it. The only way to truly find out is to produce something and see how much value can be realized from it, for instance by charging money for it.

The Max For the Minimum

In other words, producing software is a classic min/max search. You are searching for business opportunities with maximum value and implementations which minimize cost. You can think of it as a huge graph of opportunities and below each opportunity node is another graph of possible implementations. Complicating this is the fact that you would like the features to be close to each other from a market perspective in order to maximize the leverage of your sales and marketing departments and you would also like to have as much common implementation as possible in order to maximize the leverage of your existing implementation and engineering resources. For both you would also like to leverage your core competencies.

The faster you can traverse this graph, the faster you will unlock business value. Unfortunately, you only have a finite amount of time and resources, so you can't run a search by producing all possible product variations and seeing how they sell. Plus, the search space is infinite. So, you have to make predictions. You look for ways to eliminate certain paths and to get more information about other paths without actually implementing them. Whatever your methods are for figuring out what to produce, they are all based around searching the solution space as fast and efficiently as possible.

Searching For Value in All The Right Places

You may think that only parts of the development process are parts of this search, but doing so will limit the effectiveness of your searching. For instance, if you arbitrarily decide that a particular feature is what your customer wants and you then spend a huge amount of time talking to your customer base about the requirements for that feature, you are limiting your search to the various ways to do that feature, but what if the perfect implementation of that feature is only worth \$10? You are then

searching within a low value set of possibilities. Before you dive in too deep, spend a fair amount of time at a high level so as to maximize the value of searching at the next level down.

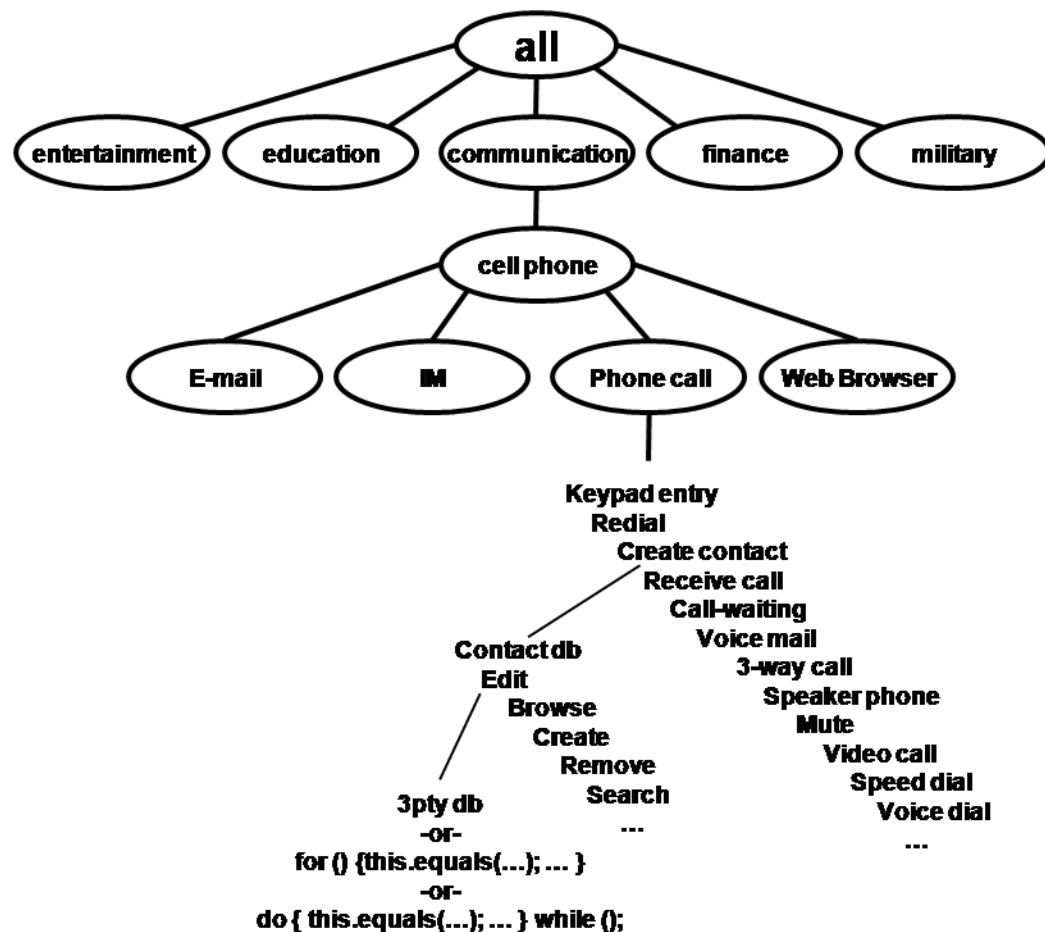


Figure: Multiple Levels of Search

The entire development lifecycle is part of your search. The first level of searching is to find potential high value opportunities from which to create new functionality. But don't fall in love with a particular opportunity too quickly because implementation may cost more than the value to the customer. There are usually a variety of ways to implement something, so don't fall in love with a particular design too quickly because there may be a cheaper implementation. And of course in order for the customer to determine whether they want to pay you or not they need to get their hands on it or at least see it in action. So, you are searching for opportunities and then within the opportunities you are searching for implementations and then you need to circle back with the customer to check the results.

Responding Rapidly is Better Than Providing Complete Feature Sets

When we design a new feature, we general think in terms of "feature sets." That is, a set of capabilities which together provide the new feature. For instance, if you were adding a calculator feature, the feature set would include things like plus, minus, multiplication, division, memory, clear, equals, etc. We

know that we probably won't get it exactly right the first time, but we also know that it will be a fairly long time until the next release so we better make the best guess that we can. Not only will it be a fairly long time until the next release, there is a good chance that we won't be able to add more features in this area in the next release, if ever. To increase the chances of including the set of features that the customer really needs, we usually end up including many more features than the customer really does need.

Let's say you have the capacity to produce 10 new features per month and you release every seven months. With a month of overhead, that's 60 new features every seven months. Marketing wants to add features for 4 new functional areas. Each functional area has a feature set composed of 15 interdependent features.

The response from the market is that three of the new functional areas are worthwhile, and of the 45 features in those areas, 25 of them are exactly what people wanted. That means you got a 41% hit rate. Not too bad. Marketing has now lost interest in the area that got no market interest, and has new ideas about what to do next in the three areas that were worthwhile. This time around they ask for two new functional areas and 30 new features in the three existing areas for a total of 60 new features.

The response from the market is that one of the new functional areas is worthwhile, and of the 15 features in that area, 10 of them are exactly what people wanted. Of the 30 new features for the existing functional areas, 25 of them are exactly what people wanted. Overall, that's 35 hits out of 60 or a hit rate of 58%. That's better than the previous release but still a lot of wasted effort. The average hit rate for these two releases is just 50%, which means that half of all of your effort was wasted.

The hit rate is generally proportional to the distance that a new feature is from existing functionality. If it is a new feature in an existing functional area, it is probably going to be successful. If it is a feature based on a feature based on a feature in a new functional area, it is much less likely to be successful.

One more thing to notice in the example is that customers using the new features in the initial release had to wait seven months to get the features that comprised the actual set of features required to provide the full feature set.

Minimally Useful Feature Sets

With Agile development, we can use an alternative method for producing feature sets which is faster and produces less waste. Instead of always trying to provide a full feature set the first time around, which almost never succeeds anyway, provide instead minimally useful feature sets. That is, the minimal set of features required to provide new value. For instance, in the calculator example it may be that many users asked for the ability to sum a list of values. The minimal set of features required for that is just addition and equals. From a traditional perspective, this list of features is incomplete.

The idea of producing an incomplete feature set is counterintuitive. There are a number of circumstances working together to produce the illusion that complete feature sets are the better approach. In traditional development, it takes a long time to produce a new release. So, if you delivered a minimal feature set in a traditional development context, it would be a long time before you could

deliver an update to create a more complete feature set. However, one of the primary advantages of Agile development is the ability to respond rapidly.

By taking advantage of this fact and using short iterations, you can produce a release which contains useful subsets of new functional areas. With the same capacity you can for instance produce 2 new features in 5 new functional areas.

By comparison, the Agile team may have delivered less of what the customer needed initially, but they can now turn around exactly what is needed next in the very next iteration. Possessing the capability to deliver rapidly is more valuable than trying to deliver full feature sets which don't actually meet the customer need.

Based on the success or failure of those features you can then decide what to do in the next release. Releasing your product with these features is not the only way to get feedback, but it is the most effective way. Using short iterations, you can get feedback much more frequently than with traditional development. Using this tighter feedback loop, you will logically converge on the highest value solution much faster.

Getting Customer Feedback Faster

There are many techniques for finding out what customers want, producing something which customers can provide feedback on as well as techniques for getting customer feedback more easily. For getting customer feedback faster you can use web-survey tools like [surveymonkey.com](https://www.surveymonkey.com), and web-enabled issue tracking systems for customers to submit enhancement requests and indicate desired prioritization. You can also encourage customers to produce user stories and use cases.

For producing something that a customer can provide feedback on there is: mock-ups, prototypes, short iterations to provide a sub-set of the full feature set, and web demos.

Fitting Work into Short Iterations

An essential part of Agile is breaking your release up into short iterations. A common objection to this is: “We’ve got lots of big changes that are each more than a month of effort, there’s no way we can fit it all into a one month iteration, Agile won’t work for us”. In any release that you do, there is a mix of tasks of different sizes. Just because you normally do hundreds or thousands of tasks in a release doesn’t make your software too big for Agile.

Divide and Conquer

When doing traditional development, you have a timeframe for major releases that seems to be “about right” for your circumstances. Too short and you’ll spend too much time on overhead, too long and you’ll miss opportunities. As a result there is no motivation to break the work up into small iterations, let alone break individual tasks into smaller tasks. Instead, you are in the habit of having long release cycles that can accommodate multiple large development tasks. You haven’t had to build your divide and conquer muscles. Certainly, many projects use project milestones, but those are not the same as iterations. Although you may not be in the habit of breaking up large tasks, it is easy to start doing it and over time it will become second nature.

For a traditional project, when planning a release, we simply ask the question “what should we do in this release given the release timeframe and the number of people we have for this project?” For the sake of simplicity, let’s look at a project which has a two month development timeframe and 7 developers and use a month as the iteration size. Let’s call it 280 person days of effort. So, we put 280 person days of work into the release as in the big block at the top of the following figure.

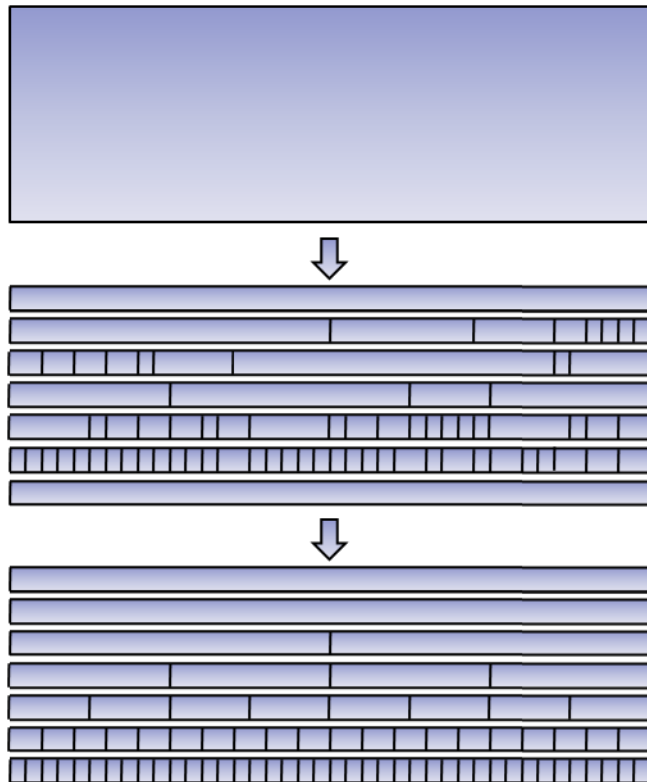


Figure: breaking “the release” down into smaller parts.

In the middle of the figure we can see the work that was completed by the seven developers in more detail. From the perspective of trying to fit things into iterations, the work is fairly arbitrarily laid out. It may be due to dependencies, or it may actually be entirely arbitrary.

We will get back to the complication of dependencies in a moment. In the meantime, let’s assume that there are no dependencies. At the bottom of the figure, the work for the release has been arranged by size. At the top there are a couple of two month tasks. There are also two one month tasks, etc.

If you were to arrange the tasks in your own projects in a similar fashion, what percentage of tasks could be done in less than a month of schedule time? In the example, it is about 70%. This is actually fairly typical.

Now consider the 30% of your tasks which don’t fit into a one month iteration. How many of them could be broken down into logical sub-tasks which could be done in less than a month of schedule time? By sub-tasks I mean tasks which are the “atomic unit” of smallest work. That is, a task which can’t be broken down any further. If you did, it would result in unfinished work. By that I mean code that would produce the wrong result. For instance, you might be able to change code in a way that doesn’t affect current behavior, doesn’t add any new functionality, but does get you further down the path of a higher level goal.

Breaking Components into Pieces

In the early days of AccuRev, circa 2002, we planned to implement a feature called

components. We did lots of design and wrote lots of code. We observed during development that components had many more special cases and performance issues than we had anticipated. At that time we believed that we needed to get it all working at once, so we didn't even consider releasing just parts of the functionality. Eventually, we decided we couldn't afford to delay the rest of the functionality, so we shelved components.

Sometimes even a piece of a bigger whole has value to the user. For instance, a part of defining components is to say which files and directories are included and which are not. Defining include/exclude rules is relatively straightforward. We eventually realized that the include/exclude part of the components functionality when separated out did not have any of the obscure special cases or performance problems of components as a whole and would be useful to users, so we refactored the components code to support stand-alone include/exclude rules and released it.

Include/exclude functionality has now become one of the most used features in AccuRev and it is hard to imagine life without it. Another benefit of breaking this functionality out is that it gave us a base of experience that enabled us to better understand what we needed to do for the next increment of functionality towards components, as well as provide a stable base of code to build on.

Consider a typical release for your organization. After breaking tasks down into sub-tasks, what percentage of work can now be fit into one month iterations? Probably 80% or more. Of course, for some changes it is not immediately obvious how to break them down into sub-tasks. We'll talk more about breaking tasks down in a moment. In the meantime, let's take a look at a technique which will give you some breathing room as your organization starts getting used to the idea of breaking large projects down into smaller tasks.

Using Multiple Tracks of Development

Your development tasks can be put into two categories: those that easily break down into small sub-tasks, and those that don't. Now you can do your development on two tracks instead of just one. The main track is work that can be completed in less than a month. The secondary track is for all other tasks. Once a secondary task has been completed, it gets merged into the primary track at the start of the next monthly cycle. That way, most of your work is done with the same rhythm and you start to get into the habit of breaking tasks down. The more you break tasks down, the more opportunities you'll see to do it and the more skills you will build up to do it.

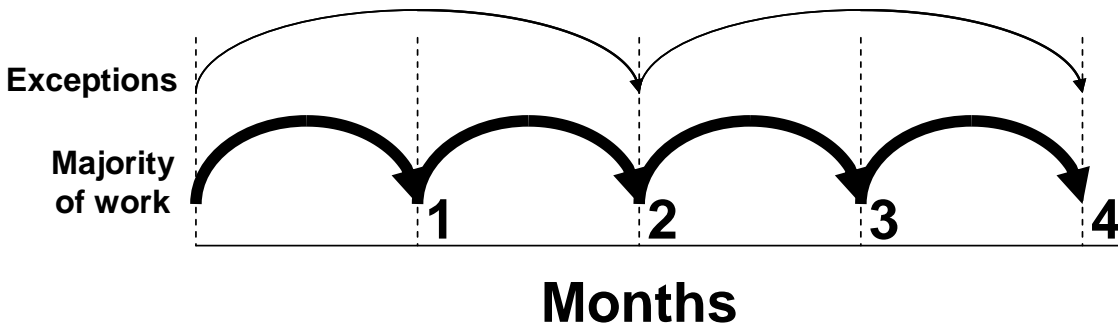


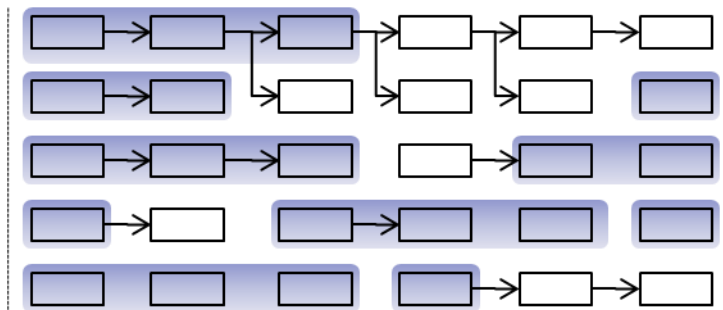
Figure: putting different kinds of work into different tracks.

After a few iterations, I think you will find that more and more tasks will be able to be done on the main track and fewer and fewer will need to be done on the secondary track.

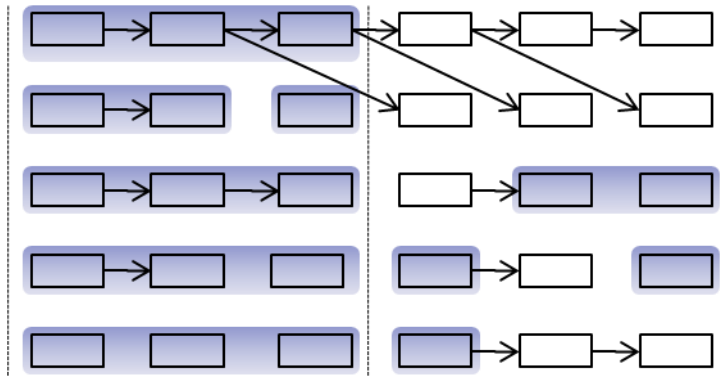
Breaking Functionality Down to the Essentials

Beyond the straightforward steps suggested in the previous section, the next level is to break feature sets down to their essentials. One of the reasons that a feature set will take a long time to implement is because we want to implement it entirely and “do the right thing.” But most fully implemented feature sets are not actually fully implemented because they do not represent what the user actually needs. While we may feel that a feature set is fully implemented, it is the customer’s perspective that matters.

The first principal is that in the first version of any particular feature we will only ship the absolute minimum required for a customer to realize value. The following diagram shows planned work which will take two months and be done by five people. The highlighted tasks are grouped together into minimally useful feature sets.



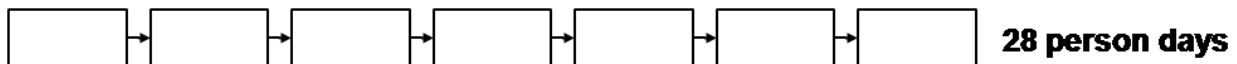
As is, it would seem that there is one set of tasks which prevents us from breaking this work up into two one month iterations. However, if we commit to doing just the minimally useful feature sets in the first iteration, then we can rearrange the work as depicted below.



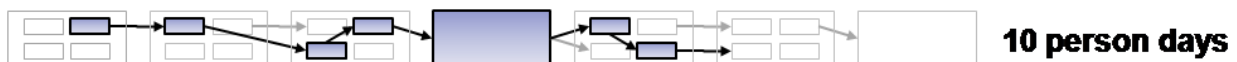
Breaking Features Down Even Further

What do we do if the minimally useful feature set takes more than a single iteration to implement? Consider a feature set which has been planned to take 28 person days. For simplicity, let's say that it has been planned as a series of 7 tasks each of which will take 4 days to implement and each of which is dependent on the previous tasks. Based on this information, it may seem that by definition this cannot be broken down further.

In traditional development, there's no reason to go any further. After all, if you are used to a 100+ working days between releases, why bother breaking down something that is only 28 days in duration? You only have a problem if it is 100+ days which is rare on any project. But if you are targeting a one month iteration time, which is 20-25 working days, then 28 person days of effort is too long.



After closer consideration, we decide that the last two tasks can be done in the next release. Now we are down to 20 person days. That would seem to be as far as we can go. But that's only if we are taking things at face value. Can the high-level tasks be broken down into sub-tasks which each represent usable features? For illustrative purposes, let's take a look at an example where it can and that each task breaks down into one day sub-tasks. In that case, it probably isn't always the case that a whole task depends on a whole task but rather that a sub-task depends on one or more other subtasks. In the illustration, the sub-tasks are broken out with their dependencies and the sub-tasks that correspond to the absolute minimum features required for a customer to realize value are highlighted.



In this example, by looking at things at a lower level than usual and with the perspective of doing the minimum required to realize customer value, we have reduced the initial effort down to 10 person days which will easily fit within a one month iteration.

Scaling Agile

Agile Exposes Scaling Problems

One of the first questions that seems to come up in any discussion of Agile is "How Well Does Agile Scale?" Sometimes this is asked explicitly, but more often there is an underlying assumption that Agile does not or can not scale very well. When I was first exposed to Agile, my first impression was that Agile didn't scale beyond a small team, say 1-12 people.

I used to try to tackle the question head on, but then I realized that there's something else going on here. Let's take a step back. What do we currently assume about traditional development? I think we come to the discussion thinking that just because there are teams of 100 engineers, 500 engineers, and even 2,000 engineers doing traditional development, that traditional development is a proven quantity when it comes to scaling. Let's first ask the question: "How well does traditional development scale?"

To answer that question we first have to define "scaling." I think a good working definition is that as you add new resources, there is a linear increase in effective output with little or no decrease in efficiency. For software, output and efficiency translate to new functionality and productivity. That would mean that if you have a 50 person engineering team and you add 50 more people you get twice the output. But when was the last time you saw that? Have you ever seen it? In my experience, after talking to hundreds of development organizations doing traditional development, productivity falls and thus output does not increase linearly with additional resources. In many cases, output actually decreases as more resources are added.

What do you actually do to "scale" your development organization? Do you have meticulously updated Gantt charts and estimates? Do you schedule lots of meetings? Do you spend a lot of time making sure that requirements and designs are just right? Do you reserve 30% of the development schedule for testing (and fixing)?

In my experience, after talking to hundreds of development organizations, the answer to the question "How well does traditional development scale" is "not very well." We've just suffered along with this problem, in large part because while the pain was there, the root causes were difficult to trace, let alone address.

The main point here is that if you are in the process of rolling out Agile to a large organization; don't be discouraged when you run into scaling problems. The problems were always there, they just weren't as obvious. Now, instead of wondering why things aren't coming together as expected at the end of the project, you may find out right away that your organization doesn't really have a good way to coordinate API changes for APIs that are used by multiple teams. As Agile exposes problems like this, you can take steps to solve the problems and thus create a more scalable development organization that just happens to be doing Agile development.

Process Perspective

There is no Bug

In the movie “The Matrix,” when Neo goes to visit the Oracle, he has a conversation with one of the potentials who is bending a spoon with his mind.

Boy: “Do not try and bend the spoon. That’s impossible. Instead... only try to realize the truth.”

Neo: “What truth?”

Boy: “There is no spoon.”

Neo: “There is no spoon?”

Boy: “Then you’ll see, that it is not the spoon that bends, it is only yourself.”

-The Matrix

Often, getting a bug to bend to your will (go away), is about as easy as bending a spoon with your mind. Consider the problem of coloring maps such that no two contiguous regions are the same color. Let’s say that in your experience, you’ve only ever needed 3 colors. You need to color maps using software, so you write it with the assumption that you only ever need 3 colors. You may encounter some bugs which end up putting the same color on two contiguous regions in a map that can be colored with just 3 colors. Eventually you will find and fix those bugs. But if you run your program on a map that requires 4 colors, the four color theorem says that no amount of bug fixing your program that uses 3 colors will ever work. You must realize that your assumption that 3 colors will always work is wrong.

Another problem with bugs is that sometimes fixing the bug doesn’t actually fix the root problem. You think you know what the bug is and you make a change so that the software now behaves as desired under the given circumstances. However, unbeknownst to you there are still 5 other ways that the problem can manifest itself and you didn’t fix any of those. Plus, you’ve inadvertently introduced a sixth problem. You don’t know about these 6 problems yet, but they are there. You’ve only fixed the symptom, not the cause. You have not “bent the spoon.”

Even when you find and fix the root cause of that particular bug, you will write more software using your existing process and it will produce new bugs to replace the old bugs. The perception of progress was an illusion. The spoon is still unbent. If anything, the repetition of the pattern has only served to reinforce the underlying process problem.

It is tempting to think of a bug as something external that creeps into your software after it is written, something that is separate that can be isolated and removed. But really, bugs are introduced during the process of developing software. They are not separate and isolated, they are introduced by flaws in the process itself.

Only when you realize that “there is no bug”, that the problem lies within, can you change your behavior such that no bug is found in the software that you deliver. I’m not saying that real software with zero bugs is an attainable goal, only that you consider a change of perspective in order to “bend the spoon.”

The problems lie within. Only when we acknowledge this and accept it can we hope to affect real change. Agile development is an excellent platform for seamlessly incorporating the required introspection and continual improvement. To paraphrase what the boy in The Matrix says, “Only try to realize the truth, there is no bug. It is not the bug that bends, it is only yourself.”

Think Of Your Development Process as Software

What do you do with software? You improve it. You add new functionality, you increase its performance and usability and you remove bugs. Also, software itself is a document: source code. It is a description of how to perform a set of tasks. You improve the software by changing the source code. Let’s call the combination of your team, tools, and techniques: “the process” and the source code for the process “the process document.”

Not only can you think of your development process as software, you can think of your whole development organization and the people in it as a combination of hardware and software with various communication links. I don’t recommend that you take this advice literally and think this way on a regular basis, or treat people as interchangeable cogs in a machine! However, by thinking about it this way you can leverage your technical design skills to think about how to organize and optimize your development organization and development process. You can leverage well-known design patterns. For instance, consider the communication aspect.

In the world of information transfer there is bandwidth, latency, and connectivity. The best environment for communication is high bandwidth with low latency that is always connected. The worst environment for communication is low bandwidth with high latency that is infrequently connected. This gives a well-known context for discussing human interaction.

On one end of the spectrum is self-communication. If you are responsible for two interdependent modules, then when you make a change to one, you instantly know you need to make a change to the other. You won’t misunderstand yourself or need to have a back and forth conversation to really understand. You just know. On the other end of the spectrum you have two people from different cultures revising a document via e-mail who live exactly half way around the world from each other. In between you have pair-programming, collocation, people working together but sitting on different floors of the same building, folks with a great deal of physical separation in the same time zone, a different time zone, etc.

Then there are different forms of information interchange such as video link, phone, e-mail, IM, wiki, document, etc. This way of thinking can guide your decisions about where to seat people, the value of having high bandwidth links, etc.

Creating a Robust and Scalable Process

When you are creating new software, what do you think about? Don’t you think about how it will scale to meet your customers’ needs as they grow with high availability? Even if you don’t achieve that on the first try, aren’t you still thinking about it and striving to achieve it? You know that to do it, you will need to make the right technology and architectural choices. Over time, you may need to change some of

those choices to keep pace with competitors. Even if your current needs are modest, software development architecture has evolved technologies and patterns to allow software to scale from a single user to hundreds or even thousands of users on multiple platforms at multiple locations with 99.999% uptime.

If you consider your development organization in the same way, how would you apply the same thinking? What technologies are you using? What is the architecture of your organization? Will it scale from its current size to double its size? Will it scale seamlessly to include new teams in new locations? What will happen if you acquire a company? When creating software, you want to design it so that it is flexible and adapts to new circumstances. The same should be true for your development organization.

Another way to look at it is how a particular process would fare if each of the resources available were available at seemingly random times for random periods of time. This exactly describes the world of Open Source Software (OSS). At any given time you have no idea who will be contributing on what or how valuable the contribution will be. You don't know where the contribution will come from and in many cases you don't even know who the contributor actually is. This is an extreme example of a development situation. Even though your situation is probably not as extreme as this, by using techniques from the OSS world you will be better positioned to handle unexpected events when they inevitably occur.

Problems are an inevitable and regular part of life. Examples include illness, job change, human error, flight delays, system failure, unanticipated market changes, and natural disasters. The ability to cope with these problems is one measure of the robustness of the organization.

Part of robustness is that as things scale up or down, the impact is minimal. Tools and processes should be selected and designed to work well together whether they are used by 1 person or 10,000. At all times, it should always feel like each individual is part of a team which is no bigger than 12 people. If practices are not scaleable from 1 to 10,000 then people can develop bad habits that resist scaling. If you develop habits that exist in a scaleable framework, then it is more likely that scaling can and will happen as and when needed.

A Simple Process

One of the things that became clear to me as part of spending so much of my career involved in the process side of development is the impact that process can have, both positively and negatively, on a development project. Unfortunately, "the P word" has some of the same connotations as the word bureaucracy and the phrase "red tape." I think that a lot of this perception comes from the large amount of manual work that is generally associated with process, as well as the rigidity of the tools and scripts that are often used to automate the process. The result is a large amount of manual process combined with automated process which is difficult to change once implemented.

People prefer to do things based on their memory, which isn't very good. The simpler that your process is, the less your process requires you to use reference materials, and the closer it is to what people expect and/or are already used to, the more likely it is that they will do the right thing.

A Single Process

Let's say there is a problem reported in the field which requires a hotfix. Well, if you are doing traditional software development, you can't exactly ask the customer to wait until you finish your current release. And if you were to put out a fix for them using your main development process, that would probably also take too long. So, you have a hotfix development process. By definition, this is a development process which is not used for regular development and it is not used very often (so one would hope).

As a result, you have two development processes: one process for regular development and one process for hotfixes. That means that when you most need for things to go smoothly you are going to use the process which is the least practiced and probably also something that only a handful of folks know how to do or are allowed to do. What's wrong with this picture?

The solution is to get the path from deciding to make a change and being able to deliver that same change as short as reasonably possible. If the "overhead" from start to finish for a hotfix is 4 hours, then any development task should take you no more than the overhead plus however long it takes you to write the code and the associated tests. All development tasks should follow this same streamlined process, not by cutting out truly necessary steps, but by ruthlessly removing steps that add no real value and automating as many of the remaining steps as possible.

Once you have a sufficiently small overhead, then you really only need one development process which you use for both regular development and hotfixes! Since it is the same process, everybody is practiced in your hotfix process and everybody has experience with doing it. This will reduce risk and increase confidence in the results.

In practice, there will probably be at least two differences. It is unlikely that anyone needing a hotfix will want to take the risk associated with taking any changes other than the hotfix, no matter how ready for release you say those other changes are. So, you will need to develop the fix against the release that the customer is using. You will probably also deliver the fix using a slightly different path than the usual. However, the closer you get to just these two differences, the better off you will be.

There is a shortcut you can take on the way to getting to a single process. You can refactor your regular process such that you act like every development task is a hotfix, and then do the rest of the process in a second stage. For instance, if you normally do a subset of your full test cycle for hotfixes, then always do that subset first during regular development.

For some software projects, getting the full test cycle down to a short time frame will be impossible or impractical. In this case, refactoring to have the first stage be the same as the hotfix process is still recommended and keeps you focused on keeping the second stage as small as possible.

One Virtual Site

Distributed development might seem to throw a monkey wrench into the idea of having a single simple process. In a distributed environment it is easy to experience "out of sight, out of mind." There's really

very few differences between teams separated by a set of stairs or an ocean. It is only a matter of degree. Apart from cultural differences, the biggest difference is the length of time between interactions.

When making decisions about your development process, always consider what would work best in the worst case. For example, which works best in the worst case, a whiteboard with sticky notes on it or a modern issue-tracking system? In the best case of a handful of developers all working together in an open area on a small project, the whiteboard will work fine. But if even one member of the team works in a different building, then the whiteboard will no longer work. On the other hand, the issue-tracking system will work where a whiteboard will work and it will also work even if a member of the team is halfway around the world.

Make as many of your development processes the same as possible at all sites. This simplifies the interactions between sites. For instance, if the same build mechanism is used everywhere, then the quality associated with the fact that a configuration builds at a particular site can be transferred to another site as opposed to having to be reestablished using a different build mechanism.

Another way to think of this is “make your process portable.” Just as you consider how to write your software so that it will work in a variety of environments, apply the same kind of thinking to your process.

A Shared Process Model

You already have a process, but it probably isn’t written down. And if it is written down, it is probably either woefully out of date or way too thick for anybody to actually read or follow on a regular basis. In any case, it is probably not quickly consumable. As a result, people have to absorb the process through osmosis. Peter joins and learns from Frank but Frank has a different perspective on what “the process” is than Joe. Joe interacts with Peter and thinks “Peter just doesn’t get it, what a noob.” But Joe hasn’t interacted with Frank before and doesn’t realize the problem is there. Without a shared perspective, it is hard to work as a unified team. We are only human and it is hard to remember something as complicated as a development process, especially when it takes years to gain the experience of running through it multiple times.

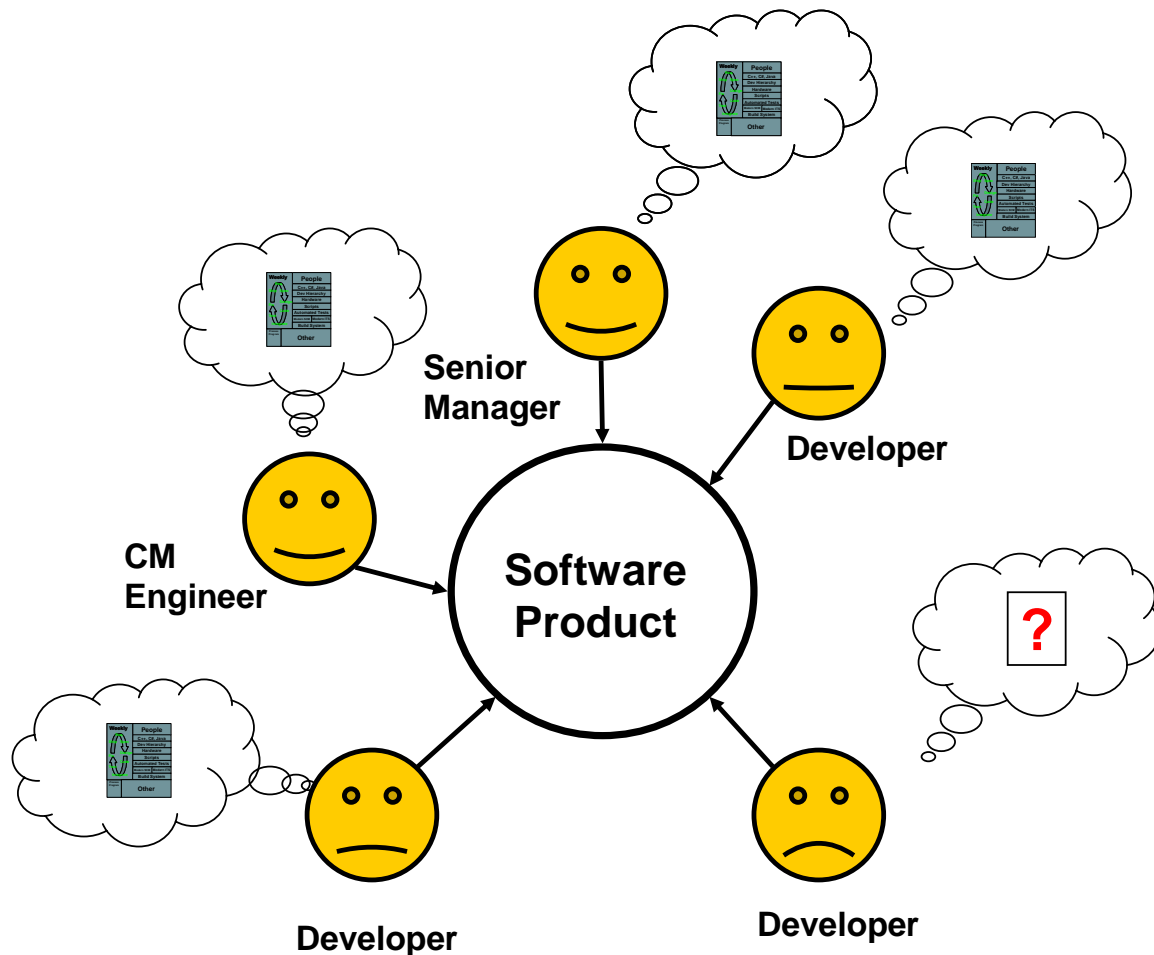


Figure: different mental models of development process, no “gold standard.”

When everybody has a different mental model of the process, there is no shared model which serves as the basis for discussion.

Create and Maintain a Process Document

By using a process document, you create a shared understanding of your process. I realize that almost everybody hates “Process” and “Process documents.” But almost everybody hates bloated software too. I’m not advocating heavy process and process documents that will cause bruises when accidentally dropped on your toe. As with great software, I’m advocating for simplicity and elegance. Ideally, a process document should be no more than 1-3 pages in length. It should be just detailed enough to serve as a reminder for old hands and so that new hires have enough information to get started. There may be details which are important but not necessary if you have been through 3-4 iterations. For instance, the exact details on how to get access to the issue tracking system and what queries to run is important, but only for new hires. See Appendix A for more information about creating a process document including an example process document.

If you don't have a process document, it is akin to not having the source code for a production system. Imagine if you had to re-write your software from scratch for every release. You'd certainly learn a lot from the exercise, but many improvements would be forgotten and later "discovered" all over again.

The process document serves the same purpose as a good legal contract (yes there is such a thing): to act as a reference when a question arises. With a contract, when all is going according to the contract, there is no need to consult the contract. Conversely, if there is a question or dispute over what was agreed on, the parties involved do not have to use their memory about what was agreed upon, there is a contract. A process document should serve the same purpose and be consulted as rarely during normal operations.

Continuous Process Improvement

Unfortunately, Agile development is not a silver bullet. By itself, it will not solve all your software development problems. Here are some of the issues that Agile development does not directly address:

- technical problems
- choice of which technologies to use
- personality conflicts
- mismatched skill sets
- insufficient management skills
- poor listening skills
- lack of leadership
- poor communication skills
- underutilized resources
- over utilized resources.

However, there are two aspects of Agile development which can be applied to all software development problems: frequent feedback, and reduced complexity. The frequent feedback provided by Agile development provides a better problem solving environment than traditional development. The environment surfaces more problems, surfaces them faster, and provides more opportunities for addressing problems and gauging the effectiveness of those solutions.

Because problems are found and fixed faster, there is less chance of the quality of a project being poor for long stretches of time. When there are lots of tests that don't pass, it is difficult to get accurate feedback on new code. In contrast, code written on a stable base is more likely to be stable itself because there will be accurate and timely feedback on the results of the changes.

Management of people and the logistics of software development projects are very difficult. There are many people that attempt it, but few who truly excel at it. One of the benefits of Agile development is the way that it reduces the complexity of the problems involved. As a result, the methodology itself simplifies the management of people and logistics, enabling you to better leverage the people and skills you already have.

Some might say that simplifying the problems is “cheating,” that it isn’t really helping advance the ability of people to handle complex problems. If you can split a problem into a series of simpler problems, why wouldn’t you? Does your customer pay you extra because you solved a complex problem or do they pay you when you deliver the value that they need?

Policies and Procedures

In any development project, problems arise. They may be new problems or old problems. At some point, it will be decided to make a change in response to a problem. It may be that the change is to add a new policy or a new procedure. Whatever the change is, it will likely require additional effort. The effort is justified by the fact that the problem is addressed. It may be addressed by looking for the problem and fixing it or it may be addressed by removing the root cause of the problem.

When it does in fact require additional effort and does not remove the root cause of the problem, you have now added to the overhead of all future projects. This overhead contributes to the complexity of future projects and can lead to additional problems and additional overhead.

The increased overhead and problems associated with any particular policy or procedure is difficult to gauge in a traditional project. There just aren’t enough opportunities for feedback and adjustment. You may not know for a long time whether a process change had a positive or negative effect. The time between cause and effect may be too long to make any connection at all. In an Agile project, the practices of short iterations and one piece flow provide constant feedback and frequent opportunities for adjustment.

Frequent Feedback

By using short iterations, problems are exposed almost as soon as they occur and are evident to everyone involved. At the same time, the environment is extremely conducive to solving the problem quickly and getting feedback right away that the problem really is solved or that additional action is required.

The powerful combination of clearly defined process and reinforcement through constant feedback are absent in most software development organizations. This is surprising considering that two of the main ingredients in software itself are rules and feedback aka input/output.

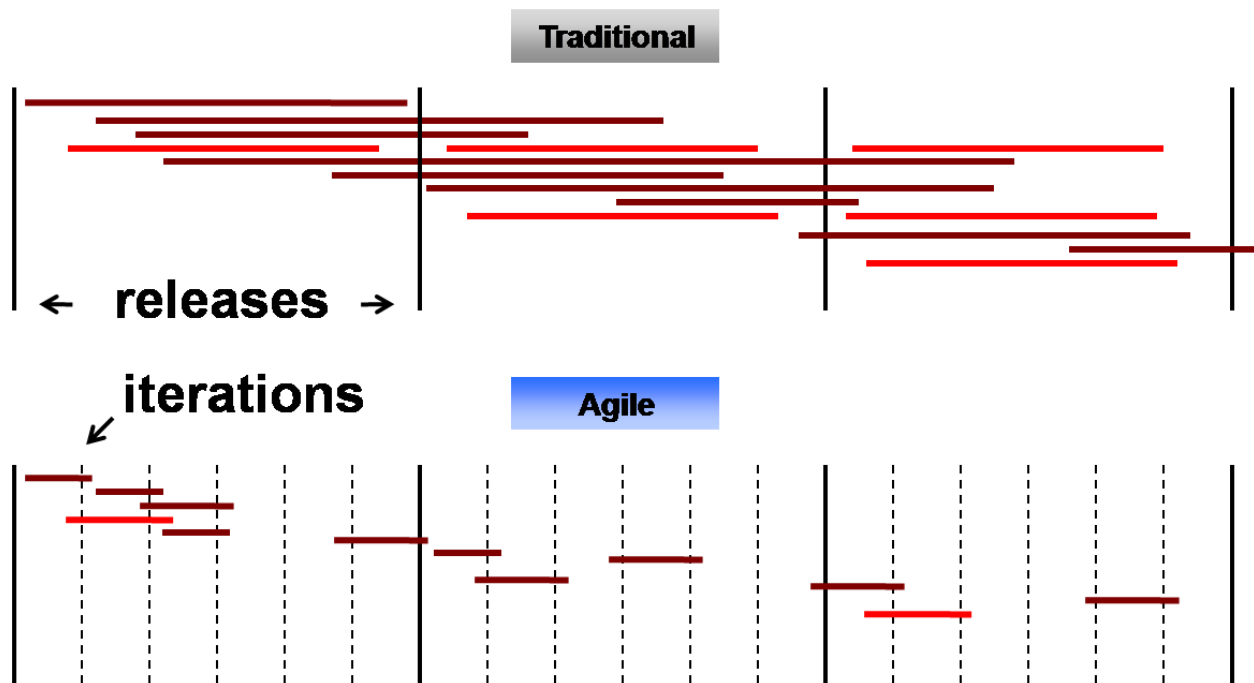


Figure: finding and fixing problems faster.

In the figure above, the lines represent problems. The start of the line is when a problem is introduced, and the end of the line is when it is resolved. The length of the line is thus the length of time that the problem is having an adverse effect on your project. Notice that some of the lines disappear for a time and then come back. Those represent problems that are not detected during the release process.

It is much harder for problems to hide out in an Agile project because the “moment of truth” that normally only happens during the release process happens every iteration. There are many more opportunities to find and fix problems. As a result, problems are caught earlier and have less impact on your project.

Experiment and see what works for your particular situation, but always be on the lookout for opportunities for process improvement. Don’t do things by rote, demand that all activities provide real value.

Your Process Has Bugs

One of the advantages of short iterations is that problems are found and fixed faster. That means that when you are trying to debug a problem, there is less likelihood that other problems will interfere with your ability to debug the problem that you are working on. But there is also the opportunity to experience a much larger benefit. At the end of the release, or even during the release process, there is usually some sort of discovery process, whether it is formal or ad hoc. Some organizations call these “post mortems” or “retrospectives”. The idea is taking a look at what went right and what went wrong and taking appropriate corrective actions. Even if there is no formal process, it is often the case that people will make suggestions for future improvement and some of them will be acted upon.

Sometimes there are unpleasant consequences such as people getting fired, people getting demoted, or people deciding that they just don't want to be a part of the process any more and leaving of their own accord. Projects get cancelled or have their budgets slashed, customers are unhappy with the results and demand changes. I'm sure you can think of other consequences as well.

In any case, the unpleasant consequences also represent changes. All of these changes can be split into two types: short-term improvements and long-term improvements. For instance, if one of the changes is that an incompetent team member was fired, then that person will no longer be causing problems. But what if the new hire has similar problems?

Discovering that you've made a bad hire can be painful and awkward. I'm sure you can think of a couple of examples that you've witnessed yourself. Discovering that there is a problem, giving the person a chance to address whatever the problem is, and replacing the person with a good hire are all simplified with Agile development. As an added bonus, you may even find that somebody you would have traditionally labeled as a "bad hire" will turn out to be a good hire.

The Process is the Problem

I once worked with a developer that I'll call Peter. Peter was a hardworking developer, and prided himself on writing code with good performance. But Peter was not actually very good at validating that his code performed well. Within one release, he was responsible for three different pieces of significant functionality, all of which were intimately involved in the performance of the product. All three had very poor performance, but this wasn't discovered until after the release.

As a result, we took two corrective actions. First, we raised the issue with Peter and he promised to do better next time. Second, we vowed to do performance testing of Peter's changes sooner for the next release.

But neither action corrected the problem. The exact same problems happened again in the next release, but this time with two different features. At this point, Peter quit and we had to find a replacement.

As it turns out, we lucked out and Peter's replacement did a much better job of producing high-performance code. At this point it was now three major releases (over a span of 3 years) before the customer could say "good job, no performance regressions in this release!"

In retrospect, I believe there is a better than 50/50 chance that in an Agile environment, Peter would have quickly adapted and become a highly valued member of the team instead of quitting.

One of the principles of Lean is to apply root cause analysis and remove the root cause of problems. In the case of a poor hire, take a look at your hiring practices. Are there changes you can make to hire better team members?

Shift Into Hyperdrive

The two practices of short iterations and root cause analysis can have a profound impact on your project. Let's take a look at what happens in an Agile environment.

The Self-Correcting Feedback Loop

We have a developer that had a tendency to write code much too quickly and ended up creating a bunch of bugs, not all of which were caught before release. Perhaps more analysis would have revealed the root cause, but for whatever reason it was not entirely clear what the root cause was. As soon as we moved to Agile development, it was immediately obvious. Now that tests are written prior to each work item being considered done, the time between cause and effect is on the order of days instead of months (or years). In this case, the person who pointed out that there was a problem was also the root cause of the problem. They were also the person that discovered the root cause and then took corrective action on their own.

In this example, the customer saw value 8 times faster in the case of short iterations paired with root cause analysis; one release of three months versus two releases in approximately two years. If we had implemented Agile earlier, it is very likely that Peter would have adapted as well. In that case, Peter would have increased his value to the team and we would not have had to recruit a new team member. In addition, over the course of the three years, there would have been a potential for an extra two years of customer value added because the problem was found and corrected sooner.

The reason we use software in the first place is to reduce manual labor and increase productivity. If your process is poor, then it is a drain on your productivity, just like any other piece of software that has quality, usability, or performance problems. Why not treat it as the same problem and work on improving it? If there are problems, as in the example of Peter, think of the problem as a bug in your software and fix it! By fixing the bug, your development process will run that much smoother and the productivity, quality, and responsiveness of your team will always be increasing.

Of course, change takes time, but the quick feedback of the short iterations helps people to notice problems on their own and establish new habits. This is reinforced by team members giving each other frequent feedback: negative when they are regressing and positive when they are headed in the right direction.

Automate Everything

Repetitive tasks should be automated and computers should be used for what they are good at: mindlessly performing repetitive tasks. People absolutely should not do repetitive tasks because there is too much chance for human error. Repetition is the core competency of computers, not people. Our core competency is creativity.

Software development is a process of many steps which often involves a large number of manual processes. However, if our fundamental reason for existing as software developers is to automate manual processes, shouldn't we be practicing what we preach? Shouldn't we be using automation more and more instead of less and less? Of course it makes sense that if we are going to be doing something new, there may not yet be off the shelf automation that fits the bill, but when there is, shouldn't we use it? The main reason the software industry exists in the first place is a fundamental belief that people will pay for the productivity gains created by automation.

The more automated a process is, the more robust it is. Automation reduces or eliminates human error and reduces the learning curve for new team members. Of course, automation can also have the effect of amplifying human error. If somebody initiates the wrong automated process, it can do the wrong things faster than humanly possible which is why it is important to have excellent disaster recovery procedures in place that are also fine-grained enough to recover from small mistakes.

There is very little value in repeating something manually if it is possible to automate it, especially if doing it manually is hard to do, takes a long time, is tedious, or is all of the above. I learned the principle of "automate everything" very early in my professional career.

My Life as a Build Tool

My first job out of college was at the Open Software Foundation (OSF) in 1990. I was the human equivalent of Cruise Control. When I started at OSF the nightly build was typed in every night. I don't mean that an automated script was kicked off by hand every night, I mean that 40-50 separate command sequences were typed in from memory every night. Essentially, it was a script that was hand typed every night. To make matters worse, we built on 9 platforms every night which meant kicking off the build on 9 different machines.

It was expected that by having somebody dedicated to it, the role of the nightly build engineer could expand. The value to the company was making sure that people were aware of build problems as soon as possible. The organization was willing to pay me to run the nightly build, hand-parse the results, go and tap people on the shoulder all day and then do it all again the next day. Although the exact failures and people that had made changes was different every day, the process was the same every day.

Realizing that my real value was in automating this task and not performing the task itself, I proved it by fully automating it and putting myself out of a job. Since there were many other problems that needed solving, some known and some as yet unknown, I was able to get a new job as a tool smith.

Always Act Like This is the Release

I think of this principal as a major guiding principle of software development. If every practice is done as though it is part of the final act of releasing the product, then you will automatically have fewer

practices. Fewer practices means less chance of problems falling through the cracks until the last minute and a higher level of maturity for things that have been done multiple times.

For instance, instead of having one process for developers to build during development and another build process for release, use exactly the same one. That way, when it comes time to release, the path is well worn and the chances are better that you will be able to completely reproduce the release if and when the time comes.

Customer Interaction

Product Management

One reason that companies fail is that their software isn't really differentiated from their competition. Just adding all of the bells and whistles that your customers ask for is not a way to differentiate. Your competition is being asked for exactly the same things.

While the appeal of rapidly responding to customer requests is very tempting, there are a few cautions in order. Customers don't always know exactly what they want and don't always express their high-level requirements. Often, they will ask for something because they have established a manual process to work around the fact that you are missing a whole feature set and there is one feature that would really help their manual process which wouldn't even be needed if you had the feature set that they really need. By responding rapidly to customer requests as though they were well thought out requirements, you run the risk of creating Frankenstein's monster: lots of bits and pieces that don't really work well together and don't form a very intelligent solution.

Also consider that unless you have a very big team or a very small stream of requests from users, it is unlikely that your development organization has the bandwidth to keep up with all of the requests from users. And of course, the more you provide, the wider your market appeal will be and the more customers you'll have and the more requests you'll get.

Each of the requests that you implement has both a cost and a benefit associated with it. If there is not enough benefit to justify the cost, it is debatable if you should even consider fulfilling the request. Remember though that the benefit may come in the form of customer satisfaction which can lead to more customers. In any case, you should do at least a rudimentary cost/benefit analysis to at least rank a particular request among all of the other requests you could do. Those with the best ROI should bubble to the top.

Even doing an ROI analysis is not really enough. What you really want to do is to add new capabilities to your product or new products to your product line with a good ROI while maintaining the integrity of those products. That is, you want to maintain the quality, usability, and performance of your products. Doing this right requires some amount of product management. A full treatment of product management is outside the scope of this book, but there are a variety of sources of information on product management, some of which are in the bibliography.

The Business Value of Software Development

Software development is not about developing software, it is about solving problems that have value to people. This is called business value. Business value is directly connected to how the value of a software product is measured. How this is measured depends on the environment in which the product operates. In a pure software company, that success is often measured by the profit generated by that product. But there are other factors to consider as well, some of them intangible, such as contribution towards the strategic objectives of the business, customer referenceability, and market reputation.

Other things to think of when considering business value include who benefits from the requirement at the customer, (end user, buyer, etc), how frequently does the need arise, and how much value would the customer receive from satisfying that need? Also, does it satisfy a market need which is currently underserved? Are there current sales opportunities which need this requirement, and how large are they?

Tighter Requirements

How many times have you heard or thought something similar to: “The last time we didn’t get the requirements right, we better spend more time getting it right and get more details.” And what about the follow-on which probably isn’t said aloud: “Let’s make sure it is clear to everybody that we did the right thing and that nobody can point the finger at us and say that we didn’t do our job. Let’s make the requirements look as professional and complete as possible.”

Over time, these two sentiments produce a lot of boilerplate and CYA-style artifacts which only provide value after the fact when the finger-pointing starts. You can refer to the requirements and say “but see, we did what was asked of us.” So the value that is produced is that you can absolve yourself of blame. But nobody will pay you for that absolution and in fact you may lose opportunities because you didn’t satisfy the customer.

Agile projects do use requirements, but only the absolute minimum required: no more and no less. The use of short iterations means that you will find and implement your customers' true requirements much faster so there is much less need for speculation and prediction. Requirements that turn out to be for things that customers don't actually need can be abandoned instead of elaborately detailed prior to the first customer feedback on working software.

For those that have auditing requirements, consider that if you have enough documentation for somebody to maintain your software then you should have an auditable “paper trail.” Conversely, if you don’t have an auditable paper trail, how can you expect to maintain your software or easily add a new team member?

Requirements

Regardless of what you call them, everything starts from requirements. Requirements state what a product is supposed to do. Documentation is sometimes substituted for requirements, but documentation and requirements serve two different purposes. Requirements are a concise description of behavior and operation which are intended to be used by end users, developers, QA, doc writers, and product managers. Requirements are also used to understand and communicate with prospects and customers about intended functionality. The purpose of documentation is to facilitate the user education process.

It has also been suggested that test suites are a substitute for requirements. If your test suites can be read and understood by business people and end users, then that’s fine. Otherwise, you will have a much easier time communicating with your users via human readable requirements. I’m not suggesting

that your requirements need to be long documents filled with boilerplate, just that they be usable by non-programmers and non-testers. It has also been suggested that code is the embodiment of requirements. But it is not easy for a new person or somebody doing maintenance to deduce requirements from code. Yes, it is possible, but it is very error prone. A couple of simple sentences about what the software is supposed to do in English is a much better way of expressing intent. It is also a good double-check.

A requirement should be as brief and to the point as possible. It should be from the point of view of the user and use the user's language. Ideally, it should be stored as a work item in an issue tracking system and the scope should be only as large as can be implemented in a week. If the scope is larger, then it should be broken up into sub-requirements. It should be written with the perspective that it will be all that is needed for somebody new who needs to maintain the associated code to understand the problem at hand (but not the design or implementation). Imagine that you need to write a test that a requirement is satisfied and the requirement and code were written ten years ago. What would you need from the requirement?

The clearer that a requirement is and the higher its business value, the easier it is to evaluate it for inclusion into the product. Here are the most important factors for a requirement:

- Make it as clear and simple to understand as possible, but no simpler!
- Create a very clear use case for the requirement which explains exactly why users need this requirement rather than some other requirement.
- Provide clear evidence for the business value of the requirement.
- Explain how the requirement fits into the overall product backlog and business strategy.

User Stories

Agile refers to requirements as "user stories". Originally, these were written on 3x5 cards or post-it notes. Now that Agile is being adopted by larger and more distributed teams, it is becoming more common for user stories to be stored in issue tracking systems and Agile project management systems due to their ability to easily edit, search, categorize, re-organize, report, track, manage, distribute, share, collaborate and do backups. Further, automated systems make it possible to access the user stories from any computer that has a connection to the internet.

On first glance, user stories seem very similar to traditional requirements. They do share many similarities. The end goal of both is to describe something which has value in the market and that people will pay for. The description should be detailed enough that it can be designed and built (assuming it can be implemented such that it is usable).

Where requirements and user stories differ is in their creation, format, scope, level of detail, and lifecycle. Ok, so the goal is the same but pretty much everything else is different. The simple reason is that user stories are designed to fit within the framework of Agile, and traditional requirements

documents are designed to fit within the framework of traditional development. Since the frameworks are different, it is not surprising that the implementation for describing user needs is different.

For a detailed description of how to write and use user stories, I recommend Mike Cohn's book on user stories. See the bibliography for details.

The important points are as follows. Your requirements should be as brief and to the point as possible. Ideally, they should be implemented via work items in an issue tracking system. They should be from the point of view of the user and use the user's language. If you are using a domain language (see the Bibliography for information about Domain Driven Design), that's even better. The scope should be only as large as can be implemented in a week. If the scope is larger, then it should be broken up into sub-stories. It should be written with the perspective that it will be all that is needed for somebody new who needs to maintain the associated code to understand the problem at hand (but not the design or implementation).

Use Cases

For each requirement, the use case must be well understood. For instance, if a customer says that they want a better command line interface, it is important to understand where the requirement comes from instead of just entering a request to "improve command line interface". It may well be that they use the command line because the GUI is too slow and the IDE plug-in that they use does not have enough functionality to support their day-to-day usage of your software. The high level user need may in fact be to increase the functionality and performance of their IDE plug-in so that they can stay in that environment and not have to use the command line.

On the surface, use cases may seem very similar to user requests, but there are very important differences between user requests and use cases.

Typically, user requests:

- Are proposed solutions to unstated problems.
- Are far too low-level and encapsulate only what the user asked for.
- Do not describe the goal which the user wishes to achieve.
- Are full of references to existing product issues.
- Are not abstracted from the specific product.
- Use low-level product terminology rather than domain terms.
- Are specific to the user making the request rather than being genericized to cover a broad base of user situations.
- Put too many restraints on the potential solution set.

Design and Architecture

Software Design Basics

Designing software requires a myriad of skills and knowledge to do it well including: domain knowledge, market knowledge, technical aptitude, up-to-date knowledge of the technology relevant to the domain, and the ability to spot patterns and trends. If you have all of these, you have the basics required for the knowledge-based decisions that you must make as you design your software. But designing software also involves dealing with the unknown.

Future-Proof Architectures, Major Releases and Large Feature Sets

A common sentiment when using traditional development is that because it is so hard to redo architecture, because it takes so long to make changes, because it takes so long to produce a release, we had better get the architecture right the first time. We had better take into account as many possibilities and eventualities that we can think of. This then sets us up for another problem. Since we are going to spend a long implementing this big architecture, then of course we better plan to do a lot of features for the release. All of those features mean we need to take more variables into account and have an even more thoroughly thought out architecture. In a traditional project, the interplay between architecture, release timeframe, and feature set create a vicious cycle which tends to increase the scope of all three.

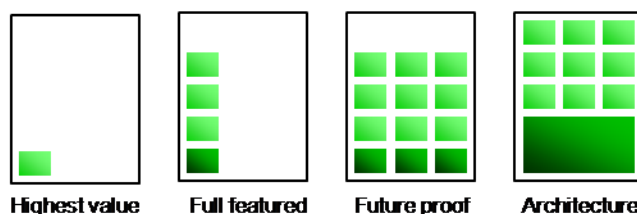


Figure: from simple to over-engineered.

In the figure above, the first box on the left contains something that the customer really needs and will clearly provide a great deal of value. It is simple by itself and requires little or no “architecture.” In a traditional project, we will make sure that we provide something “fully featured” for two reasons. We aren’t sure we are producing something the customer really wants, so we will put in some more features that in that functional area to increase our chances of satisfying the customer. Now that we’ve added more features, we are starting to have to add some architecture in support of the features. Not only that, but we want to make sure that this feature set is “future proof.” We don’t want to have to come back and change this code since it may be a long time before we have the opportunity. So we add in more code to take care of every possible problem and eventuality we can think of. That produces even more code which needs even more architecture.

Compared to the original high-value need which we are pretty sure the customer needs and is very simple, we have produced a lot of code which has a large “surface area.” We may end up producing ten times the amount of code than if we had just done the simple thing first. All of that code is new and immature. The more code you have, the more potential you have for introducing bugs, the more code

you have to maintain, the more code people have to learn in the future, the more complex your architecture will become.

It isn't just the code that is a problem. For all of that code you need to produce requirements, specifications, designs, and tests. Producing this takes effort.

In an Agile project, the practices of short iterations and iterative design reinforce each other to keep the amount of architecture that is required as small and simple as possible yet no less than necessary. The product backlog and short iterations keep the feature set focused on those features which will produce the highest customer and market value instead of trying to build a product which takes into account all possible future requirements.

Building Architectural Bridges to the Future

One of the principles of Agile, mostly related to design and architecture, is "The Simplest Thing That Could Possibly Work." This is sometimes interpreted as a license to "slap something together." But that is not the intention. A better way to express it would probably be something like "The Simplest Solution That Could Possibly Satisfy Your Requirements." For instance, if you have a requirement to create the back end for a web site like amazon.com, then while a perl/cgi solution on a single core machine could possibly "work," it doesn't work from the point of view of high availability, fast response time, or reliability.

From Oversimplification to Rube Goldberg

On the one hand, there is a wide spectrum of complexity of construction ranging from doing nothing to Rube Goldberg level complexity. On the other hand, there is the set of solutions that work, meaning that they meet all of the requirements. TSTTCPW refers to the solution which works and which is lowest in complexity.

Part of being simple means simple to read, maintain, use, design, understand, and implement balanced against the time it takes to get the job done. Spending too much time to create the ultimate in simplicity starts to get you into a different kind of trouble.

As somebody that struggles with this principle on a regular basis, I was happy to stumble upon an example of this principle which can be captured in a picture and kept in mind as I am working on a new design. Perhaps you will find it to be useful food for thought as well.

A Bridge Too Far

There's a construction project that you've probably heard of which is affectionately called the "Big Dig." Part of this project was the construction of the "Leonard P. Zakim Bunker Hill Bridge" aka the "Zakim bridge." This part suspension bridge, part cantilever bridge is an enormous one of a kind architectural marvel. It supports five lanes of traffic in either direction for a total of ten lanes. It was built at a cost of approximately \$11M per lane.

Running parallel the Zakim (on the left in the photo) is another bridge, the Leverett Circle Connector Bridge. It serves a total of four lanes of traffic. It was built at a cost of approximately \$5M per lane.

Part of the requirements for the Zakim bridge were clearly “create a stunning new Boston landmark.” On the other hand, the Leverett Bridge is a very simple but also very strong bridge. It could have been made even more simply, but not without a safety risk and/or a shorter lifespan. In other words, it is “The Simplest Thing That Could Possibly Work.”



The more possibilities you take into account when you design your software, the bigger and more complex your design will become and the longer it will take you to produce the design. The bigger and more complex your design becomes, the more likely it is that you will have difficult design problems to solve. Solving difficult design problems can take a long time. Once you've finished this big and complex design, it will take a proportionately long time to implement that design and it will create a large body of code. The more code you have, the more chances you have that the code will have a problem and the more tests that will be required to validate the quality and the more tests you will have to maintain.

The “Faberge Egg” Widget

There was a developer that worked for me once, I’ll call him George . He wrote a lot of really good code. But one day he decided he wanted to make his mark on things. George believed that the way to do it was to create a beautiful widget. He wanted it to be so beautiful and so useful that it could be used generically and would be something that we could sell separately on its own merits.

The widget was part of a Java/Swing user interface for an issue tracking system. It was responsible for the data model used by a form, a query editor, and a handful of other objects. So, it did need to be fairly general purpose, but it didn't need to be so generic and so functional that it would be something that people would want to buy separate from the application.

George referred to this widget as his “Faberge Egg.” It was an apt name for the widget. It was overly ornate, intricately detailed, and supported a very wide range of functionality that we had no immediate use for and still don’t to this day.



I was very clear with George that I only wanted a “Cadbury Egg” and tried hard to convince him that he could provide much more value to the company in other ways and the company would reward him for that, not the Faberge Egg. I like folks to have the freedom to grow and explore. Even though we were under deadline pressure, I gave George some wiggle room while at the same time trying to guide him towards the simplest design that would work. Unfortunately, after weeks of work and many conversations, George’s desire to produce a masterpiece of a widget prevailed.

Over time, that widget has been the source of more than its share of problems, both in bugs and in making it more complicated for other developers to maintain it and extend it. It has been partially refactored several times, but it seems there has never been the time to really do the full job required.

The same functionality that the widget provides was needed in our new web interface. Instead of reusing the code as we have done for other functionality, we decided to start from scratch. Doing just what was needed for the job at hand took only two days and that code has been very reliable right from the start.

Refactoring

Refactoring is the process of improving the maintainability of code without changing its results. This practice originated with Agile Development but has since become a common coding practice as evidenced by its inclusion as a feature in Eclipse and other IDEs.

Refactoring has the advantage that over time it makes code easier to understand and maintain. This can contribute to the maturity of the code. It has the disadvantage that it requires changing the code and any change has the potential to reduce the maturity and stability of the product.

Serendipity

I think there is a very important part of design which is often either overlooked or just plain denied because people don't want to admit that it is true. Motivation, skill, and experience all play important roles in the design process, but just because you have these doesn't mean that you are going to be able to design, let alone build what you have set out to do. It almost always requires trial-and-error, serendipity, creative inspiration, and research. Agile development provides more opportunity to take advantage of serendipity.

Planning

Helmuth von Moltke (the Elder) said, “No battle plan survives contact with the enemy.” In the case of software development, the “enemy” is reality. What we think we need to do to satisfy customers and how we think we need to implement it generally changes as we implement and as we deploy to customers.

Agile is designed to take advantage of the fact that planning, design, and development have a learning component. As you discover new information during an iteration, you can easily adjust your plans to take what you have learned into account. If you discover that you need to change tactics in the middle of your overall strategy, or you have a change of strategy, it is simple to change tactics with short iterations. Just change your plans for future iterations. As soon as your current iteration is done, you can begin the next iteration which implements your new tactics.

As Dwight D. Eisenhower said, “In preparing for battle I have always found that plans are useless, but planning is indispensable”. Planning (short and long, tactical and strategic) is still important in the Agile world. The difference is that Agile is specifically designed to anticipate and embrace the inevitable need to change tactics, and thus plans on a regular basis.

Planning an Agile project is pretty much the same as with a traditional project. There is really only one difference. With traditional development, if you have a six-month cycle, you plan it as one big project. In Agile, you would plan to do the exact same work, but structure it into six one-month iterations where the work of each iteration is shippable at the end of each iteration and then you ship at the end of the sixth iteration. If you have multiple years of work to do, it is the same process. You break it up into iterations and you can release at the same intervals as you do now. As a side benefit, the release interval is now independent of the work because you have the option of doing a release at the end of every iteration.


By breaking a big release up into iterations, you can still do planning. But instead of a single end date, you have a series of milestone dates. As you learn more about what the market needs, you can adjust the planning of your future iterations, move functionality from one iteration to another, and give an early heads up when problems are discovered. A heads up doesn’t mean just letting people know that you aren’t going to make it, it can also be a call to action that says the project is going to need more resources than expected to make the desired date and/or keep the planned content, so if those things are important, more resources can be provided or alternative plans can be devised.

Product Backlog

The use of a product backlog and the role of a product owner is most closely associated with Scrum. The use of a product backlog vastly simplifies project planning, keeps you focused on your target market, and helps to prevent feature creep. Once a requirement has been created, the next step is to get it into the product backlog. A product backlog is simply a list of high-level product requirements, prioritized by business value as determined by the product owner. As new requirements are gathered or submitted, they are inserted into backlog at the position that best represents their business value with respect to

requirements that have already been entered. As requirements are met, they are removed from the product backlog.

Each backlog item has a high-level estimate associated with it. This estimate may have come from the product owner's interaction with the engineering team, or it may be the product owner's best guess. In either case, the estimate in the backlog should be considered to be a very rough guess and never used with customers or prospects or used for engineering planning purposes. The estimate in the product backlog is primarily used by the product owner to do high level planning in preparation for more detailed planning.



Issue	Short Description	Est Time
9622	estimation widget (Pert)	2
3527	ranking of issues	20
6252	change field values in dispatch issue at promote time	5
9281	support for user avatars	5
9289	visual ITS to replace most ITS query and update operations	10
2085	generic issue links	5
2084	sub-tasks	5
9623	critical path calculation	10
9702	load leveling (via rebalancing suggestions)	20
9624	network diagram of issues	10
9288	queues	5
5234	AccuWork: have type for streams	5
9704	release objects	10
9703	iteration support	10
9626	dashboard engine	20
9625	dashboard (gui)	20
9701	ability to print dashboard	5
9700	web interface underlying infrastructure	20
9699	embedded ITS integration	20

Figure: work item ranking, aka “backlog”.

Long term planning is accomplished by ranking everything that you hope to accomplish within the planning timeframe, adding estimates, and then breaking the backlog up into iterations. Of course over time you will re-plan, but you're not coming into a new iteration and doing the planning from scratch. This gives management more comfort that you know where you're going — instead of just seeing where you end up.

Estimation

Keep in mind that when doing Agile development, the need for high accuracy is greatly diminished. If you underestimate some things, the worst that happens is that you have 10% or more work for an iteration than you anticipated. Since an iteration is short, the “penalty” for estimation errors is small.

Also, since you are using a backlog, the things that get pushed are going to be the lower value work items.

Break tasks down into smaller tasks as much as possible as described in the section on fitting work into short iterations. This increases the number of estimates and increases your chance of overall accuracy. It can also flush out areas that are not as well understood as you may have originally thought.

While there are many different methods for doing estimation, picking one and using it consistently is much better than using none at all. Related to that, using something simple is much better than using something complicated. It will take longer to create the complicated estimate and even in the unlikely event that it is more accurate, if you have more than just a few things which need estimates, it is likely that the errors will average out. By using something simple you have a higher chance of it being used and people getting better at using it.

One simple method is the PERT method. It is widely used, and it is very simple. Take the least possible amount of time, add 3 times the expected, add in the most time it could possibly take, and divide the sum by 5. That's it.

Using Story Points For Estimation Instead of Units of Time

In my experience, the best unit to use for estimates is story points. Two different people with two different skill sets or levels of ability in an area may take different amounts of time to perform a particular task. Estimating in hours mixes together the scope of the work that needs to be done with the speed at which a particular individual can do that work.

On the other hand, story points are a relative measure of the scope of a user story. Story points separates out the “what” from the “who.” For instance, if you have one individual that is stronger with .Net than with Java, they will estimate a Java story as taking more hours than somebody that is stronger with Java. But they will probably both agree that something that is twice as easy to implement will take half as long to do.

To use story points, you need to create a relative scale of scope. A simple approach is to find a simple and straightforward story that you use to represent a single story point. Then think of stories that are 2, 3, 5, and 8 times larger in scope. You should have a couple of examples for each story point value to take into account that some stories have more test than coding, more documentation than test, etc.

Story points are primarily used for planning, not for implementation. Story points are used to help determine the contents of an iteration by calculating a velocity.

Velocity

In Agile, the velocity of a team is simply the number of story points associated with stories that are finished in an iteration. For instance, if the team completed 8 stories that were each 5 points in an iteration, then their velocity for that iteration was 40 story points. In a stable team, a team that is

comprised of the same individuals working full time as part of that team, the velocity is a good measure of the overall throughput of the team.

Knowing your velocity helps with planning. For example, if you know that the velocity of your team is 40 points, then you know you can expect 40 story points for each iteration. The team decides which stories to take based on the backlog which is maintained by the product owner.

Estimation Using Planning Poker

Planning Poker is an estimation technique first described by James Grenning in 2002 in a paper by the same name. When I first heard about Planning Poker, I couldn't help but chuckle. It seemed a bit silly to mix software development and poker. After reading Mike Cohn's excellent book "Agile Estimating and Planning" I had a good grasp of the mechanics of Planning Poker, but it felt like just a variant of the Delphi method and thus nothing new.

Planning Poker may be one of those things that is best understood by doing it. In 2008, we had an outside Scrum trainer come in and do some Scrum training for us. That was just the shot in the arm that we needed to supercharge Agile adoption at AccuRev. After doing some exercises using Planning Poker we decided to try Planning Poker on a real project. The results were terrific and after a single session we were hooked. Planning Poker is now a standard part of our Agile process.

The Basics

In order to play Planning Poker, each participant will need a deck of Planning Poker cards. These are easy to obtain, just Google for "Planning Poker cards" or make your own. You will need the following cards: ½, 1, 2, 3, 5, 8, 13, 20 and a card that indicates "I've had enough." The numbers on the cards represent estimates. You can use story points, ideal days, or some other measure, but I am assuming (and advocating) story points. Typical decks contain cards for much larger numbers than 20, but I think you'll find that 20 and higher are rarely used once you've been using Planning Poker for a year or two.

The whole team gets together for a set amount of time to do story estimation. An hour is usually a good amount of time, but this is completely up to you. When I say "the whole team" I am assuming that you are using small cross-functional teams of 5-9 people (see the section on whole teams for more information.)

Estimation is done on a story-by-story basis in the same order as the backlog, starting with the first story that needs estimating. Somebody reads and describes the story. This is often the product owner, but could be anybody. Some teams do story point estimation without the product owner and just skip stories which they are unable to do without the product owner's involvement. After the story has been read, participants discuss the story for a bit and then decide on an estimate by picking one of their cards and laying it face down in front of themselves. Once everybody is ready, all of the estimates are shown. If the estimates are all the same, you are done.

Usually, some of the estimates are particularly low or particularly high. A low estimate can indicate that the estimator left something out, or possibly that they know a way to reuse existing work or have some other good information that other folks weren't aware of. A high estimate may indicate many things, but most commonly it means that the estimator is thinking of something that other folks may not be aware of. For instance, somebody may point out that there is no test framework or tooling for a particular technology and that to adequately test the story the team will need to do a lot of setup work.

In any case, after a round of discussion, everybody chooses a new estimate. This is repeated until the team comes to consensus on the estimate. The estimate is given to the story and then you move on to the next story. You end when you hit the end of the meeting time, you run out of stories to estimate, or somebody plays the "I've had enough" card.

The Benefits

One of the biggest benefits of Planning Poker is the sense of team that it creates. The whole team is participating in the estimation. This creates a greater sense of team ownership and team responsibility for each story. Another major benefit of Planning Poker is that you leverage the collective wisdom of the whole team. However, this really only works well when you are using whole teams in the Agile sense of the term.

Whole Teams

Planning Poker reminds everybody that the estimate includes all of the work that needs to be accomplished in order for the story to be considered done. It includes development, testing, documentation, and anything else required for the story to be considered done. It is a reminder that you are part of a whole team and that nobody on the team gets credit for a job well done until the whole story crosses the finish line.

User Stories Simplify Estimation

Because user stories are a simple and easy to understand description of the work, user stories allow you to focus on estimating rather than spending lots of time discussing what a particular enhancement request or requirement really is. To maximize the benefits of Planning Poker, you need to be good at creating and using user stories.

Planning Poker Reduces Both Risk And Waste

There are three particularly valuable things that can happen during a Planning Poker session. They may also happen during any flavor of estimation meeting, but seem to be more likely when using Planning Poker.

Big Stories Can Hide Problems

I recommend that you never use a story size greater than 13. Most stories that are estimated at 8 points or above can be split into smaller stories. You should always be looking for opportunities to break larger stories into smaller stories. If you have an 8 point or larger user story, it is probably actually two or more

stories in disguise. The bigger the story, the more likely your estimates are wrong and the more likely that you have many smaller stories masquerading as one large story.

For instance, you may find that a story that originally looked like a 20 point story was really two 8 point stories, one 5 point story and 2 three point stories for a total of 27 story points. Better to break that huge story down into its five constituent stories and estimate them each individually.

But remember, a story is only a story if it provides value to the user. Splitting a story up into “As a user I want the backend for X” and “As a user I want all of the UI for X” is not the right way to go. If you can’t create smaller stories that still provide user value, then the story is already as small as you currently know how to make it.

If a Story Feels Like a Research Project, It Probably Is

You may realize while discussing a story that the story contains a big unknown, something that feels like it won’t be resolved during the implementation of that story. In that case it is best to split the story up into a research story and the story itself. For instance, “As a developer I want to know how to XYZ.” That way, if you never do figure out how to do the unknown part, you haven’t invested any effort into the overall story. This technique should only be used as a last resort, but it is much better to do this than to know going in that there is a big unknown and have to pull the whole story out near the end of the iteration.

If You Don’t Have Enough Information, Don’t Try to Make it Up

Lastly, you may decide that you just don’t have enough information to estimate a story. In that case, you should have a mechanism for informing the product owner such as marking the story “need more info.” There’s no point spending time on estimation if there is insufficient information to do so. You’ll just be glossing over the problem and producing a false sense of security.

Planning Poker Fosters Good Habits

Breaking out research stories and kicking stories back to the product owner may seem like procrastinating, but it tends to build good habits. It keeps the team from committing to work that includes research projects, clearly defining some stories as research stories, and keeping the product owner on his or her toes.

In summary, if you are mostly working on small user stories that have end user value, you are reducing the chance that you are putting something into the product that never gets used and you are also reducing the chance of starting work on something that never gets finished or has to be discontinued part of the way through. The smaller your stories, the smaller your risk and the less effort you’ve wasted when you run into problems.

Quality

There's a saying that "you can't test quality into a product." That may be true, but it is very difficult to assess the level of quality of a product without testing it. One of the problems with quality is the way that it is talked about. There are all different categories of testing such as regression, black-box, white-box, exploratory, automated, manual, etc. Each of these categories serves a useful purpose, but what is the overall purpose of testing?

Some people argue that testing is how you verify that the product works. Other people argue that the purpose of testing is to uncover what does not work. My view is that the purpose of testing is to show that the software works. By "works" I include that it correctly handles bad input, bad data, and diabolical usage. If you try to use the product in a way that was not intended and it crashes or gives no result at all, then you have found a way in which it does not work.

Quality Assurance and testing are not the same thing. You can implement all of the QA you want: good process, code reviews, coding standards, coding for testability, and whatever else you want to assure quality short of actual testing, but without testing, you have no idea how the product will actually perform. Let's say you produce a product with absolutely no test plan or you have a great test plan but you don't execute it. I suppose that it is possible that the quality that the customer experiences will be exactly the same as if you had a test plan and executed it. The difference is that your knowledge of the quality of the product will come from your customer after you have already shipped your product. The purpose of testing is to derive knowledge about the quality of the product prior to shipping it so that you can make good decisions about what to do now that you have that knowledge.

Product quality is an ongoing effort. No matter how much testing you do, users will find ways to use your product that find holes in your test plan that you didn't think of. For any change, whether it is a bug fix or a new feature, you don't really know the impact until customers start using it. That means that the maturation of a change doesn't really start until the change ships. If you have a long release cycle, you will have changes that are made early that are basically gathering dust until you ship. You can't really say that they are tested until you test the whole product as it will ship, so again, true maturation doesn't start until you ship.

The Value of QA

The value of QA is often assumed to be increasing the quality of the product or at least keeping the quality at the same level that it was at previously. But those values are actually provided indirectly or in combination with other actions. That is, making changes based on the information provided by QA. QA does not (or at least should not) direct these efforts, business value should. Perfection cannot be the goal because perfection is not achievable. The goal should be to add the most business value to the product by leveraging the information provided by QA.

The direct value that QA provides is in the area of quality assessment, verification of completion, risk assessment, and exposure assessment. However, this value can only be realized if it is acted upon.

Separation of Development and QA

A common pattern in traditional development is the separation of development and QA. Of course everybody always says the classics:

“QA is everybody’s job”

“You can’t test quality into a product”

“QA should be involved right from the start”

But, what inevitably happens in practice is that QA occurs at the end. One contributor to this problem is that “during development” the product isn’t stable enough to test and you want to get the most bang for the buck out of your testing because it is so expensive to do and it takes so long to qualify a release. The natural result from these circumstances is the separation of development and QA which is constantly reinforced. Using traditional development patterns, all efforts to unify development and QA are inevitably defeated. Nobody is consciously trying to make it happen, it just happens.

Sometimes the separation of development and QA is intentional. The separation can feel “more honest.” Sometimes development doesn’t want QA to know what is going on because they are sure that they can catch up and fix things and then deliver something to QA which is good. But in order to get the benefits of QA without the help from the official QA folks, development ends up having to do QA themselves which reduces the time spent on development. In any case, traditional development practices tend to separate development and QA.

Agile practices do just the opposite; they tend to bring development and QA closer together. The writing of tests for a particular change is done either before the coding starts or during the coding, not in a separate phase at the end of all coding. Tests are run constantly using the practice of Continuous Integration. QA doesn’t have to wait until the traditional code-freeze to start the bulk of their work. This creates a positive feedback loop where QA gets to show its value on a regular basis, development spends less time fixing things at the last minute, QA learns more about the product, and development and QA become even closer.

The Role of QA in Agile Development

An interesting question in Agile Development is “how do you incorporate a QA person?” I guess that all depends on what you see as the role of QA. For instance, if have your QA person involved from start to finish, then you really wouldn't have to change a thing. For instance, a QA person can:

- help to determine if your stories are well defined
- add stories related to testability
- add stories related to usability
- make sure that people are creating good unit tests

- write unit tests
- create tests cases before or at the same time that code is being written
- increase your code coverage by adding more automated tests
- automate tests cases before or at the same time that code is being written
- organize usability testing
- do exploratory testing on early builds
- verify the completion of stories as they are completed
- organize demos
- do demos of new functionality as part of verifying that work items are complete

If QA is involved from start to finish in this manner, then you significantly reduce the need for a well-defined QA stage.

Maintaining the Balance Between Development and QA

At a high level, you should work to keep the ratio of development to QA balanced. The ratio will vary from project to project, so I'm not commenting on that, but once you have a ratio there are several ways to maintain it. In an Agile project, you may find at the end of an iteration that QA is falling behind. This indicates that the ratio itself is off and needs to be adjusted. In a project with long iterations, you might not find out that the ratio is out of whack until it is "too late."

In any case, once you find that you are short of QA resources, there are a couple of options. The first option of course is to get more QA resources. Failing that, you can use development resources. It may also be that you have a short-term need for more QA and in that case using development resources might be exactly the right short-term solution.

I think a common rule of thumb, which especially applies in the context of compliance, is that different people do the development and testing of a particular requirement. That is, if Sue is responsible for issue 12345, then Sue doesn't do the testing of it.

As to the exact use of development resources, I advocate that developers take on a sub-set of QA activities, thus allowing for better leveraging of the existing QA resources. For instance, creating automated unit tests or the general generation of automated test cases.

You can of course set whatever policy makes sense for your organization. Having people only write test cases for other people's code is always a good idea. Beyond that, I would assume that QA would take what the developers had done as a starting point, supplementing with additional tests, reviewing what the developers had done, running the official test cycle according to the test plan and reporting on the results.

End of Iteration Touch-up

With short iterations, it is likely that there will still be some touch-up work to be done on the previous iteration when the next one starts. Perhaps QA has not finished evaluating the iteration, or perhaps somebody wants to do a demo but runs into a bug that needs to be fixed first.

Going With the Flow

For AccuWorkflow we were doing variable length iterations where the iterations were between 2 days and a week, so we ran into this hand-off period on a regular basis. The way we handled it was to create a branch for the “hand-off” period.

As an iteration ended, we created a new branch based on the existing development stream. The new branch then represented a stable version of the “previous iteration.” If we needed to do a little follow-on work, for instance to get things ready for a demo, we used another configuration called “demo” based on the “previous iteration” branch. To prepare for the demo we made changes in the demo branch without affecting the previous iteration or the current one. Once we finished the demo we then merged any changes into the current branch.

Having branches representing multiple iterations also made it possible to easily find out what had changed between iterations which aided in tracking down problems.

My experience is that with automation, branching, and the supporting process, the goals of determining the quality of an iteration and creating stable baselines can be achieved in parallel. The first step is automation. Without a high degree of automated testing, I'm not sure it makes sense to try to do this in parallel. However, with continuous integration which includes full builds and full test suite runs, there should be few if any problems at the end of the iteration.

Some might say that their automated build and test takes a very long time. That's a whole separate topic. In the meantime, let's assume that the full automated build and test cycle takes on the order of hours.

By branching, you can start the next iteration without disturbing the previous one. If QA finds any problems, development can fix them on the previous iteration branch. At some point that branch will be declared "done." Those changes are then merged into the current iteration. At that point, there is very little if any difference between where things are and where they would have been if development had started from the "done" baseline.

The advantage here is that development did not have to stop, and you still have a stable baseline. My experience is that once a team gets into a rhythm, the amount of time spent on "touch-up" is minimal.

This also applies to iterations that become releases. The "previous iteration" branch becomes work towards the release and then hopefully soon thereafter the release itself. Meanwhile, work continues towards the next release.

Requirements Based Code Coverage Testing

Just doing code coverage is not enough. The best code coverage is decision-based code coverage. However, it is not enough because if the code is wrong, you aren't actually testing what the product is supposed to be doing. For instance, if a function is supposed to work one way if the value supplied is more than 10 and another way if the value is more than 20, but the code only has the case of 10, you'll get 100% code coverage if you test 10 and 11, but you'll only have 66% of Requirements coverage. Code coverage is only the same as requirements testing if the code is perfect.

Writing Tests Early

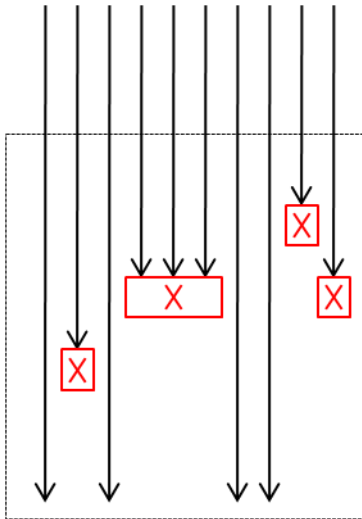
Tests and requirements are very closely related. In order to write either one, you need to know the answer to the question "what is this functionality supposed to do?" If the person responsible for writing the tests is struggling with writing them, it is a good early warning that it is not clear what the software is supposed to do. If it is not clear what the software is supposed to do, it is unlikely that anybody will be able to write the software to satisfy the end user. Thus, it makes sense to write the tests first to reduce the chance of writing code that does not reflect the needs of the end user.

While writing tests first is good, it may not always be practical. However, any tests that are going to be written prior to release should be written in the same iteration that the code is written. This has two benefits. First, it helps to establish the natural rhythm of the iterations by fine tuning the ideal amount of testing required within an iteration. If test writing is left to the end, just before the release process, the time dedicated to testing is likely to be compressed. The second benefit of writing tests early is to help with the early detection of problems.

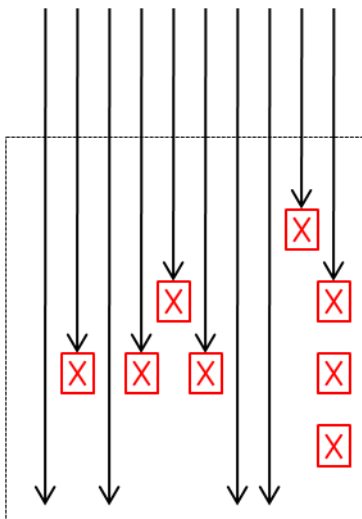
The False Economy of Big-Bang Testing

There are several rationales for putting off testing until the end: a belief that you get more bang for the buck and that there are many potential problems that can only be found when all of the work has been integrated together. That is, with a single test cycle you'll find more problems than if you ran the same test cycle multiple times during the release. This is actually a very bad idea for many reasons. First, it delays the gathering of important information about product quality and project status to the very end of the project when there is less time to deal with problems, less time for changing plans, and fewer options for mitigating risk.

Second, it is rarely true that you will run significantly fewer test cycles. There is a simple reason for this, many problems can't be found and many changes cannot be verified until blocking problems are resolved.



In this figure, the tests find four problems. However, because of the problems found, it is impossible to gain knowledge of the full product until the initial problems are fixed.



Once the initial problems are fixed, you need to run another test cycle. The next test cycle serves two purposes: to verify the fixes and to find any problems that were blocked by the first set of problems. Unfortunately, with the large number of problems and fixes that are typical at this stage of development, it is highly likely that new problems will be introduced. This cycle has to be repeated many times until things finally stabilize.

While it is true that any tests that require all of the work to be done and integrated can only be done when all work has been done and integrated, that's really not a good reason to wait. Even if tests can't be run, they can still be written. At the very least, the necessary test plan can be created. This will very frequently uncover requirement and design problems. Isn't it better to have this information as soon as possible?

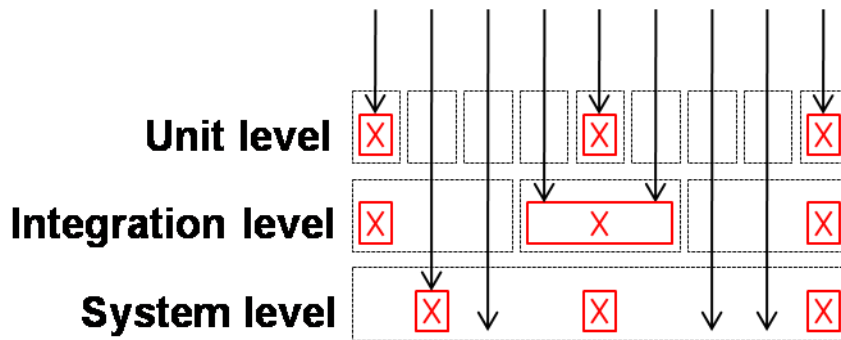
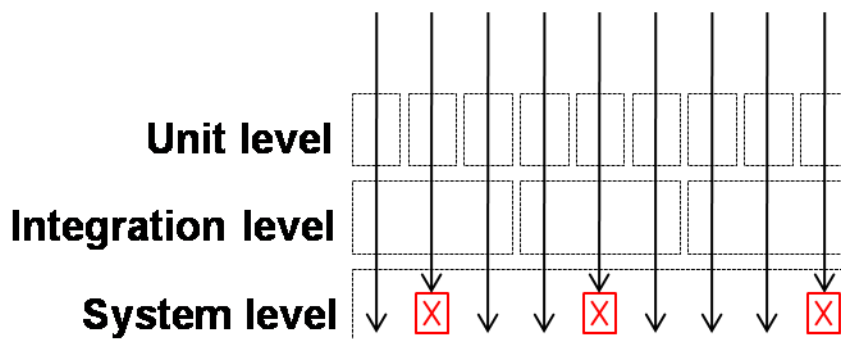


Figure: testing late in the cycle.

In the figure above, testing was done late in the cycle. Problems at the unit level blocked the discovery of problems at the integration level and problems at the integration level blocked discovery at the system level. In this example, you would need to go through at least three test and fix cycles to discover all problems, fix all problems, and verify all fixes. And that's just in the simple case that there is only one level of blocking at each level!



If instead you run unit tests and integration tests as often and as soon as you can, you will uncover unit and integration problems sooner and have the opportunity to fix them sooner. Then, the only problems you will find and need to fix at system test time will be the problems that can only be found at system test time. Isn't it better to find and fix problems as early as possible and minimize the problems that you have to deal with at the end of the cycle?

The practice of Continuous Integration means it is much more likely that you will have your whole product integrated most of the time. Thus, you will be able to run your system tests on a regular basis all during development.

Sometimes we may think "what does it matter at what point during development that I fix a bug? I can write the test now and find problems or write it later and find problems. What's the difference?" When there are problems with requirements, specification, design, test, or code, there is a downstream effect both for the task at hand and other tasks as well. A problem with a requirement can lead to problems with specification and on to the design, tests, and code. Designs which must incorporate multiple requirements where one of the requirements is wrong may end up with a design which poorly serves all

of the requirements involved. New code which contains a bug may influence the code that is written later in a way that adversely affects it.

For instance, let's say that somebody else has written code with a bug in it. I now write code based on the buggy code and assume that that code is working correctly. Some of my code makes an accommodation for a certain behavior caused by the bug. I don't know there is a bug so I just assume that is the correct behavior. Now there is code which is not necessary and may pass the behavior of the bug on to whoever depends on my code, etc, etc.

The longer that it takes to detect a problem, the more things that will be affected by that problem, the more accommodations that may be created, and the more work that will be required to take corrective action. That it is another benefit of one piece flow. By doing all stages of development for any particular task prior to moving on to the next task, you find problems as quickly as possible and reduce the amount of work done on an unknown state.

Automated Testing

Manual testing is useful when you are exploring a new area or have a test case which resists automation, but it is not a sustainable strategy for the bulk of your testing needs. Each test that you do requires setup, correct execution, and comparison of results with a known good result. Doing this manually is time consuming and error prone. If you have a test plan with 10,000 individual steps and each step takes an average of 60 seconds to setup and execute, another 30 seconds to validate the results and another 30 seconds to record the results, that's a full person week of effort. The chance of people executing those 10,000 steps, examining all of the results and recording the results without error is very low. Producing a useful report of the results of the testing will take still more time.

By automating as much regular testing as possible you get the following benefits: reduced cycle time, increased accuracy of results, automatically generated results, and the ability for anybody to run the test suite at any time as often as needed. You also get a cost savings because an automated test suite costs less to run than having a person do it. Even if you invested in running the full test suite manually every time it was needed, if you run it 20 times during the development of a release, that's a huge investment of effort.

Considering that a major enabler of short iterations is frequently running through the entire test plan, it is imperative that you get as close to full automation as possible to ease the transition to Agile development.

When choosing how to allocate resources, it can be difficult to do an effective cost benefit analysis in a short period of time. One technique that I use is to look at things from the perspective of "what would I do if I was the only person on the team?"

Many Hands Make Light Work, But I've Only Got Two

In 1992, I convinced my manager at the OSF that it would be more productive for me to spend my time improving the tools that the fourteen Release Engineers used than for

me to be one of those Release Engineers. The transition gave me a lot of energy and excitement. I really threw myself into the job and made many more changes to ODE (the OSF Development Environment) than had ever been made before.

I had no idea what I had taken on. Now that I was making many more changes to ODE, there was also much more testing to be done. There was just one little problem. There were no automated tests for ODE and there were no documented test cases either. There was a tsunami gathering just out of sight.

The test plan consisted entirely of “round up as many volunteers as you can and have each volunteer test on one of the nine platforms.” The testing that each volunteer did was entirely up to their discretion. It was different volunteers every time, and there was no record of what they did, only a “seems fine to me” or “I found these problems.” Once the number of problems got down to an acceptable level, we’d ship the new version.

The first couple of releases had lots of problems that were discovered after the release. This was my first inkling that the tsunami was approaching. It was clear that I needed to try a different approach. My first attempt was to document a standard set of test cases. At first this seemed to work really well. Testers commented that it was much easier and took much less time. I felt like I had gotten a consistent set of results from a consistent set of tests. But a test/fix cycle requires running the same tests over and over again until things stabilize. Following the same steps over and over again can become pretty mind-numbing. Pretty soon I couldn’t get the volunteers I needed and I was starting to suspect that people were skipping steps. I had built levies, but I had nobody to help me maintain them.

I also discovered another problem. As bug reports came in from production use, and as I thought of test cases that were missing, the list of test cases mushroomed. Since QA was ultimately my responsibility, I had to pick up the testing slack. But if I did all of the testing, then I would end up spending much more of my time testing than coding. Now I could see the tsunami bearing down on me quite clearly.

I remembered how I had gotten the job of tool smith in the first place, by automating myself out of my previous job of manually kicking off builds on all platforms. Test cases are basically a set of steps to take and the expected results. Automating test cases is basically coding, and that was much more fun than manual testing. A couple of inspired weekends later I had automated all of the test cases and added many more new ones as well. I had built an army of levy builders. I was saved.

Automating tests as code is written may mean that test cases and the automation of those test cases will need to be rewritten or updated as behaviors or user interfaces change over time. There is an argument that can be made that it is more efficient to wait until things have stabilized before creating tests cases and automating them.

Let's break this argument down and take a look at the individual parts. There are two parts to an automated test case: the test case itself and the automation of that test case. Once you have decided to create a test case, it is difficult to argue that running that test case manually over and over is cheaper than automating it. So really, it isn't a question of when to automate a test case but rather it is a question of when to create a test case.

Let's consider some scenarios. In the first scenario, we have a classic waterfall project where all requirements are specified up front. From these requirements, all or most of the test cases can be created as soon as the requirements are ready. What is left is the automation of those test cases. It may be possible to execute some test cases prior to code freeze, but when test execution starts is orthogonal to this discussion. During the regular course of development, all of the test cases will be run multiple times. Not just the new test cases, but also all of the existing test cases. This is very expensive. If a test case needs to be changed prior to ship, then it needs to be changed. If you've invested in creating test cases, then you've already bought into the idea of updating test cases and their expected results as needed.

In contrast, consider the case where as soon as you create a test case you also automate it. Also, you are using short iterations. Let's say that you've broken the work up into 12 iterations. That means there is the possibility in iterations 2-12 that you need to rewrite test cases and automated tests for some of the work done in iteration 1. The same is the case for the rest of the iterations as well. So, the cost with this approach is on the rewriting side and not on the manual execution side.

So the question here is, which is more expensive? The cost of manual execution of an ever growing set of test cases or the cost of rewriting whatever test cases and test automation is affected by changes?

Keep in mind that there is another factor to consider in the case of the short iterations. When you are doing short iterations, you can take advantage of the fact that there will be requirements that you would have planned on doing using the traditional method that you may have cancelled during the development cycle and that you never would have even gotten to using Agile. That means that using the iterative approach, you will end up creating fewer test cases that you end up throwing away.

In my experience, it is always cheaper to change test cases and test automation than it is to rely on manual testing. This is especially true in an iterative project where you are running the tests more often and getting the benefits from that and where you are doing less work that never gets used.

Avoid "Random" Testing

There seems to be a belief that there is value in having people "play with" a release to test its quality and to look for bugs. This is not to be confused with manual testing or exploratory testing. Manual testing is testing that is not automated but that is following written test cases. It is just that the test execution is done by a person. Exploratory testing is generally done by experienced QA people who are trying to learn about new functionality in preparation for creating a test plan and writing test cases (whether those test cases are then automated or not).

The practice of “playing with” or “banging on” a release that is not following documented test cases and is not exploratory testing is random testing. An example would be somebody sitting down with the goal of “seeing if I can find any bugs.” This kind of testing is a complete waste of time. First, any random testing that is done is actually very likely to overlap with an existing test case. Second, without a plan, it is more likely that the random testing that does not overlap with an existing test case is testing a low value area of the product.

Lastly, even if the random testing is guided in some way, such as giving each person a list of areas to play with, the exact testing that ends up being done is irreproducible.

Mustgo Soup

When I was a kid, my mother used to make “mustgo” soup. It was soup made from leftovers. It was always delicious and my sister and I would always ask her to make the exact same soup the next night. She would always explain that she couldn’t because she hadn’t kept track of the exact ingredients and amounts.

Without an exact list of the steps followed and the expected results, the testing that is done is irreproducible. The only value from “banging on” a release is whatever bugs were found. Considering the amount of effort that is generally invested in such testing, the cost is very high, especially if the people helping out are not professional testers. It is unlikely that people that are not trained testers will find nearly as many problems per dollar invested.

The other problem is that once you fix those bugs, even if you create tests to verify the fixes, you don’t have the level of quality that you might think you have. You might assume that the banging on measured a certain level of quality and the resulting quality is that same level plus the increase represented by the bug fixes. The actual level of quality is the level of quality measured by the increased level of coverage provided by the new tests you added. The reason is that those bug fixes may have caused other problems that are not covered by your test suite.

Any value that comes out of random testing has a high cost associated with it. You will always be better off taking those resources and investing in a planned increase in your test cases.

Iteration Reviews, aka Demos

These are often referred to as sprint reviews or demos. A review should demonstrate the functionality, but it is not the same as a demo. The word “demo” is often thought of as a sales demo. The problem with a sales demo is that you are trying to sell. In a review, you have two purposes, neither one of which is selling. The first purpose is to demonstrate that you have done the work to completion, the second is to solicit and record feedback.

Demonstrating new functionality should be a standard part of the completion of all work. It is best if the demo is done by somebody other than the author of the work. Demonstrations provide a test of the functionality, show progress towards goals, and build confidence in the work. The audience for a demonstration should include all key stakeholders including a customer or ideally multiple customers.

Demos are a much more visible, trustworthy, and confidence building way of showing progress. Anybody can show on a chart that you have made 80% progress towards a goal, but if you can't demo anything, how do you know that it isn't actually only 40% of the way towards the goal?

Demos partially serve the purpose of usability testing. If you have somebody, such as a QA person, do the demonstration, they are in effect testing the functionality, testing the usability of the functionality, and reviewing the requirements all at the same time.

When I started writing software, I enjoyed the thrill of showing people something they hadn't seen before. Even today, one of the main reasons I enjoy working in the software industry is the thrill of demoing new software. When you demonstrate new software, you become a magician, conjuring feats of computation that dazzle the imagination. The audience starts out skeptical, wondering if you are just a two-bit side-show act. You slowly build up to the main event and then, when you're lucky, they gasp in amazement as you show them something they'll no longer be able to live without.

The Magic of Demos

One of my favorite demos was many years ago when I was showing an early version of a product to some folks for feedback. As part of the demonstration I interrupted the power to the laptop (with the battery already removed), and showed them that the software continued as though nothing had happened when the power was restored.

Even though we had told them that the software wouldn't ship for at least six more months, they called us the next day to place an order anyway. For them, the value outweighed the risk. We decided to accept the order. That early exposure to a real customer changed the way we thought about things. Even though we considered the product to be pre-release, we made sure that every new feature worked as it was developed instead of waiting until the end game of the official release. As a result, the end game was much smoother than we had expected.

Usability

An important part of quality is usability. There are usability issues which are not strictly bugs, but usability is really the true measure of goodness, not low defects. You can have very few bugs and still have unhappy customers. Think of it this way: why is it that users don't like bugs? Basically, they don't like bugs because they prevent them from doing what they need to do. In some cases, it prevents them permanently, in other cases they find a workaround. If they find a workaround, then the problem soon becomes one of loss of productivity. Essentially, people hate bugs because they hinder productivity. Thus, the real problem is loss of productivity. So, what else affects productivity? A user interface that is

difficult to use affects productivity negatively. New features which enhance productivity affects productivity positively. Usability issues can often have a greater effect on productivity than bugs.

Users don't always report usability issues (or bugs for that matter). It is worth while to do usability testing on a regular basis to discover usability problems (and bugs!).

A better way to think of this is: what are the typical paths through the product? What are the typical tasks that users want to accomplish and how do they accomplish them, including writing things down or repeating something multiple times. This is not how you think they should do it but how they actually do it. Bugs found along those paths are interesting, but fixing the usability issues are actually more important. It is sort of like spending time on the appearance of your front door vs the appearance of your roof.

Integration

How frequently have you merged your code with changes from the mainline, only to find that the result doesn't build, or it builds but it doesn't work? Monthly? Weekly? Daily? Hourly? Or worse, how often have you made changes that broke the build, requiring you to quickly correct the problem while getting flames from your team members?

If everybody in development is making changes against the mainline, introducing a bad change affects everybody until it is detected and corrected. Waiting for someone to fix a change that is impacting everybody is bad enough when the change was committed locally, but it becomes severe when the committer won't even be awake for another eight hours.

Big-Bang Integration

Integration is tough enough when you are just integrating your work with the work of other folks in your small team, or the whole effort is being done by a small team, but when you are part of a large team there is also something called “Big-Bang” integration. That’s the integration of the work that multiple teams have been working on for long periods of time. In a typical project, this integration is done in a phase toward the end of the project. During integration, many problems are discovered for the first time which leads to delays and/or changes in scope.

Continuous Integration

A practice that has emerged to address the problems of integration is called Continuous Integration. The basic idea is that if integrating changes together and getting build and test results on a regular basis is a good idea, integrating and getting build and test results on every change is even better.

With Continuous Integration, all work from all teams is integrated into a single codeline as frequently as possible. Every check-in automatically triggers a build and usually a subsequent run of the test suite. This provides instant feedback about problems to all interested parties and helps to keep the code base free of build and test failures. It also reduces the integration headaches just prior to release.

I’m a big fan of continuous integration. I’ve used it and recommended it for many years with great success. But it has a dark side as well. The larger and/or more complex the project, the higher the chance that it devolves into what I call “Continuous Noise.”

For any meaningful project, a full integration build and test will take time. Even if you somehow get that time down to 10 minutes, a lot can happen in 10 minutes, such as other people checking in their changes and invalidating your results (whether you realize it or not). It is more likely that this full cycle will take on the order of an hour or more. While you can certainly do something else during that hour, it is inconvenient to wait and necessitates task switching which is a well-known productivity killer.

In this case, you get notified every 10 minutes or so (depending on how much building and testing is going on) that the build and/or test suite is still failing. It may be for a different reason every time, but it doesn’t matter. It is tough to make progress when the mainline is unstable most of the time. This problem is not caused by CI, but it is exposed by CI.

Continuous Integration does not actually reduce the amount of integration required. It doesn't really solve a scaling problem either. There are only two benefits of Continuous Integration. The first is that it forces you to break work down into small, manageable pieces. The second is that it spreads the integration work out over the entire development timeframe instead of just a short period at the end when you have less time to find and fix all of the issues discovered during integration.

The real question is, what is a good way to structure this integration so that it will scale smoothly as you add more people to the equation? A good starting place is to look around for a pattern to follow. What are some similar situations? I have found that everything your organization needs to do in order to produce the best possible development organization can be entirely derived from the patterns and practices at the individual level. This approach makes it much easier to understand and much more likely that it will be successfully followed.

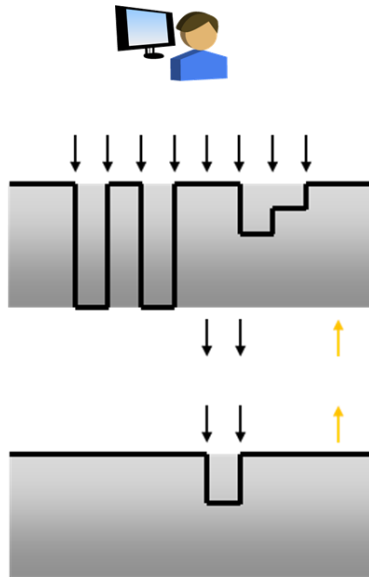
Self Integrity

When you as an individual work on a change, you often change several files at the same time. A change in one file often requires a corresponding change in another file. The reason that a single individual works on those files is because the changes are tightly coupled and don't lend themselves to a multi-person effort.

As an individual developer, there are two things that you do to shield yourself and others from instability. Although you make frequent changes to your workspace which means its stability goes up and down rapidly, you only check-in your changes when you feel like you've done enough testing of those changes that you've sufficiently reduced the risk of disrupting everybody else that is depending on the mainline. When you check in your changes, you are essentially making an assertion that your change has reached a higher level of maturity.

Conversely, you only update your workspace when you are at a point that you feel you are ready to absorb other people's changes. Because other people only check-in when they feel the changes are ready and you only update when you feel you are ready, you are mostly shielded from the constant change that is going on all around you.

These two simple practices act as buffers which shield other people from the chaos of your workspace while you prepare to check-in and shield you from problems that other people may have introduced since you last updated your workspace.

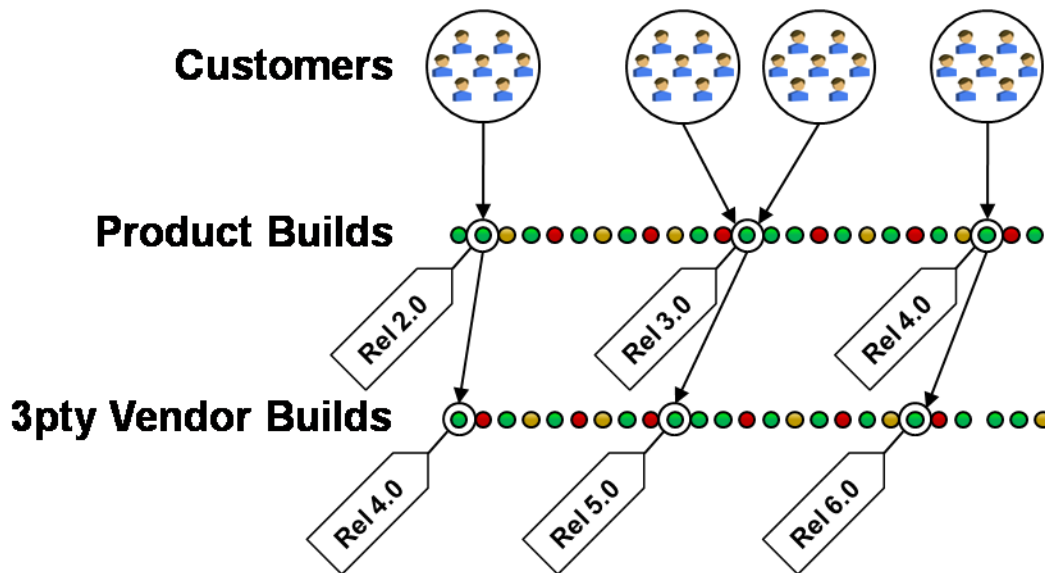


Individual developer integration pattern.

While it may seem like a bit of a trivial case, you can think of this process as self-integration. This is the basis of Multi-Stage Continuous Integration: if individual isolation is a good idea, then isolation for features, teams, team integration, staging, QA and release is an even better idea.

Moving From Known Good to Known Good

This integration pattern is also found in the interactions of customers and software suppliers and the interactions of software producers and third party software suppliers. Your customers don't want random builds that you create during development and you don't want random builds from third parties that you depend on. The reasons are simple and obvious. The level of quality of interim builds is unknown and the feature set is unknown. Your customers want something that has a high level of quality that has been verified to have that level of quality. Likewise, you want the same from your suppliers. Your suppliers include things like third party software libraries or third party off the shelf software that your application depends on such as databases and web servers.



Consumers at every level only take stable builds, aka releases.

As with your own software, each third party that you rely on produces hundreds if not thousands of versions of their software, but they only release a small subset of them. If you took each of these as they were produced, it would be incredibly disruptive and you would have a hard time making progress on your own work. Instead, you move from a known good build to a known good build, their externally released versions. Your customers do the same.

This simple principle should be applied throughout your development process. Think of each individual developer as both a consumer and producer of product versions. Also think of them as a third party. Think of each team as well as each stage in your development process this way. That is, as a developer think of your teammates as customers of the work that you produce. Think of yourself as a customer of the work that they do. You want to move from known good version to known good version of everything you depend on.

It's All for One and One for All

The next level of coupling is at the team level. There are many reasons why a set of changes are tightly coupled, for instance there may be a large feature that can be worked on by more than one person. As a team works on a feature, each individual needs to integrate their changes with the changes made by the other people on their team. For the same reasons that an individual works in temporary isolation, it makes sense for teams to work in temporary isolation. When a team is in the process of integrating the work of its team members, it does not need to be disrupted by the changes from other teams and conversely, it would be better for the team not to disrupt other teams until they have integrated their own work. But just as is the case with the individual, there should be continuous integration at the team level, but then also between the team and the mainline.

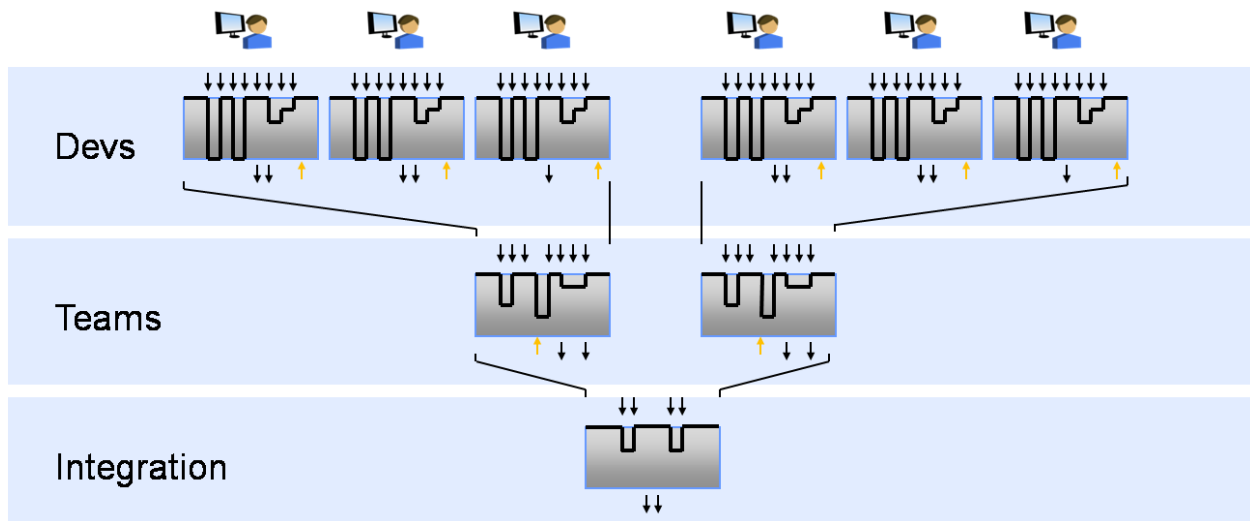
Multi-Stage Continuous Integration

So, how can we take advantage of the fact that some changes are at an individual level and others are at a team level while still practicing Continuous Integration? By implementing Multi-Stage Continuous Integration. Multi-Stage CI takes advantage of a basic unifying pattern of software development: software moves in stages from a state of immaturity to a state of maturity, and the work is broken down into logical units performed by interdependent teams that integrate the different parts together over time. What changes from shop to shop is the number of stages, the number and size of teams, and the structure of the team interdependencies

For Multi-Stage CI, each team gets its own branch. I know, you cringe at the thought of per-team branching and merging, but that's probably because you are thinking of branches that contain long-lived changes. We're not going to do that here.

There are two phases that the team goes through, and the idea is to go through each of them as rapidly as is practical. The first phase is the same as before. Each developer works on their own task. As they make changes, CI is done against that team's branch. If it succeeds, great. If it does not succeed, then that developer (possibly with help from her teammates) fixes the branch. When there is a problem, only that team is affected, not the whole development effort. This is similar to how stopping the line works in a modern lean manufacturing facility. If somebody on the line pulls the "stop the line" cord, it only affects a segment of the line, not the whole line.

On a frequent basis, the team will decide to go to the second phase: integration with the mainline. In this phase, the team does the same thing that an individual would do in the case of mainline development. The team's branch must have all changes from the mainline merged in (the equivalent of a workspace update), there must be a successful build and all tests must pass. Keep in mind that integrating with the mainline will be easier than usual because only pre-integrated features will be in it, not features-in process. Then, the team's changes are merged into the mainline which will trigger a build and test cycle on the mainline. If that passes, then the team goes back to the first phase where individual developers work on their own tasks. Otherwise, the team works on getting the mainline working again, just as though they were an individual working on mainline.



Multiple Stages of Continuous Integration

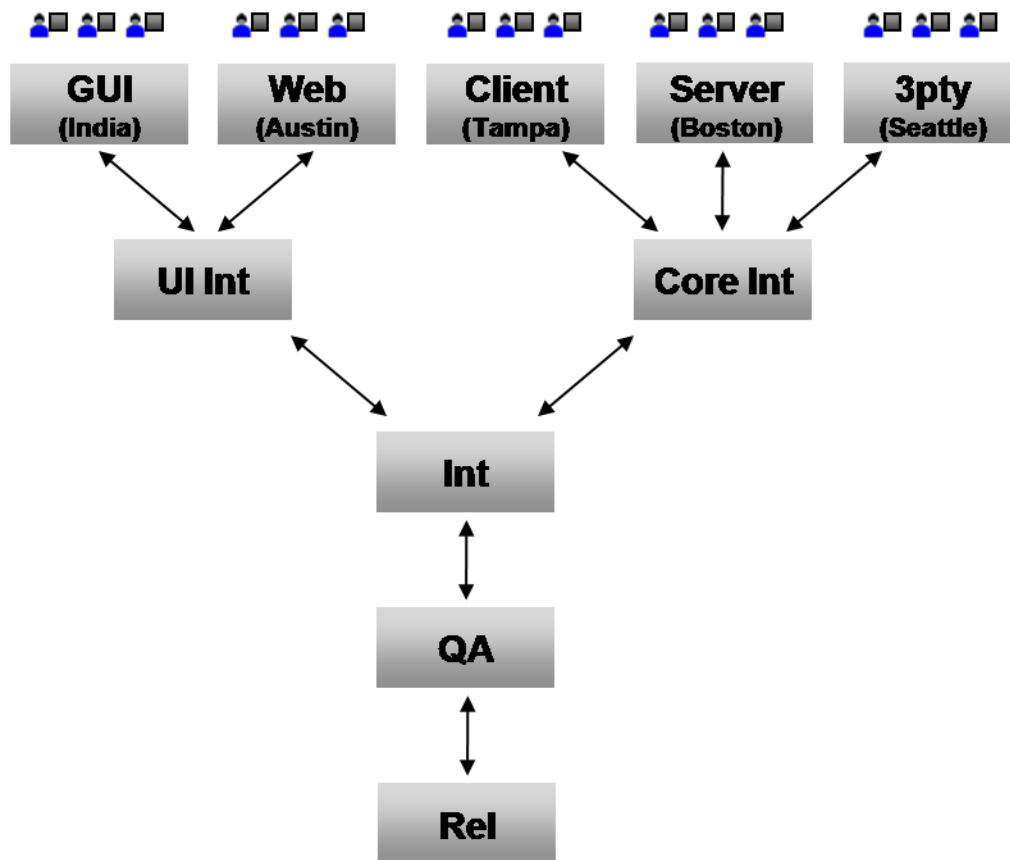
This diagram shows a hierarchy of branches with changes flowing from top to bottom and in some cases back towards the top. Each box graphs the stability of a given branch in the hierarchy over time. At the top are individual users. They are making changes all day long. Sometimes their work areas build, sometimes they don't. Sometimes the tests pass, sometimes they don't. Their version of the software is going from stable to unstable on a very frequent basis, changing on the order of every few minutes. Hopefully, users only propagate their changes to the next level in the development hierarchy when the software builds for them and an appropriate amount of testing has been done. That happens on the order of once per hour or so, but ideally it happens no less than once per day.

Then, just as individuals check-in their changes when they are fully integrated, the team leader will integrate with the next level and when the integration build and test are done they will merge the team's changes to the next level. Thus, team members see each other's changes as needed, but only team member's changes. They see other team's changes only when the team is ready for them. This happens on the order of several times per week and perhaps even daily.

Changes propagate as rapidly as possible, stopping only when there is a problem. Ideally, changes make it to the main integration area just as frequently as when doing mainline development. The difference is that fewer problems make it all the way to the main integration area. Multi-Stage CI allows for a high degree of integration to occur in parallel while vastly reducing the scope of integration problems. It takes individual best practices that we take for granted and applies them at the team level.

Distributed Integration

All of the reasons that make continuous integration a good idea are amplified by distributed development. Integration is a form of communication. Integrating distributed teams is just as important as integrating teams that are collocated. If you think of your teams as all being part of one giant collocated team, and organize in the same manner as described in the section on Multi-Stage Continuous Integration, it will be much easier to coordinate with your remote teams.



Thinking of distributed teams in terms of function rather than location.

Multi Stage Frequent Integration at Litle and Co.

At Litle, the biggest problem they ran into as they have grown was integration of newly developed code into the existing codebase. To address this, in 2007 they added the practice of Multi-Stage Frequent Integration to their implementation of XP. They do frequent integration instead of continuous integration because a full build/test cycle takes 5 hours.

Prior to implementing Multi-Stage Frequent Integration, they would have to manually pore over all build and test failures to determine which change caused which failures. This was done by very senior developers that were familiar with all aspects of the system to be able to understand complex interactions between unrelated components of the system.

Using Multi-Stage Frequent Integration, each project works against their own team branch, merging changes from the mainline into their team branch every day or two, and doing their own build/test/cycle with just that team's changes.

Thus, any failure must be a result of one of their changes. When the build/test cycle is successful, the team then merges their changes into the mainline. As a result, any team-

specific problems are identified and removed prior to mainline integration and the only problems that arise in the mainline are those that are due to interactions between changes from multiple teams. The isolation also simplifies the job of figuring out what changes may have caused a problem because they only have to look at changes made by a single team, not the changes made by all developers.

Adopting Multi-Stage CI

Getting to Multi-Stage CI takes time, but is well worth the investment. The first step is to implement Continuous Integration somewhere in your project. It really doesn't matter where. I recommend reading the book "Continuous Integration" by Paul Duvall, Steve Matyas, and Andrew Glover. The next step is to implement team or feature based CI. Once you have that working, consider automating the process. For instance, you can set things up such that once CI passes for a stage, it automatically merges the changes to the next level in the hierarchy. This keeps changes moving quickly and paves the way for easily adding additional development stages.

I've seen Multi-Stage Continuous Integration successfully implemented in many shops and every time the developers say something like: "I never realized how many problems were the result of doing mainline development until they disappeared."

Releasing Your Software

Sink or Swim Maturation Process

One of the issues we are mostly unaware of by doing major releases is the need to give each individual change its own natural maturation cycle. All changes take different amounts of time and effort to mature, but this is often ignored. We do try to guess how long it will take to produce a change, but that's about as far as we take it. Let's say we estimate (guess) that something will take a week, but in fact it takes two weeks. Then, during testing, a problem is found and it takes another week of effort to fix it. Let's say that happens three more times. So the actual maturation of this change took 5 weeks and 5 touch points. That is, it took 5 rounds of feedback to get it just right (with the last feedback being just validation) and we originally planned for 1 week and 1 touch point.

How this example actually plays out has a lot to do with when during the development cycle you start working on it. It may be that the first three weeks of effort happen prior to release and the last two happen after release because the problems were found by a customer. Or, it may be that it was the first thing that was worked on and since it was done so early during development that it gets exercised enough to find all of the problems prior to release.

The result of this is that we expect major releases will have problems that are addressed in follow-up releases which concentrate solely on these kinds of problems. We are conditioned to expect and accept this. This is the standard process for maturing software. Because of this, we resist the idea of doing frequent releases because we believe that those releases will all contain mostly immature changes.

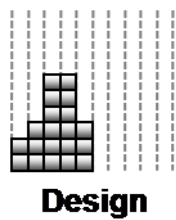


Figure: traditional maturation of features as a group.

Take for example the figure above. Within each section, there are small boxes representing individual features. The left side of the graph represents a low amount of maturity, the far right represents a high degree of maturity, features which won't need further work in the future.

This graph represents the maturity of the features at the end of the requirements and design process. Some features are well described, well understood, and have been implemented in one form or another by the same team in the past. Other features are described rather vaguely and are poorly understood, though they have impressive looking documents describing them. As we look at later stages of development, we will look at the same twenty features, ignoring any other features that are added along the way.

The maturity level of the features represents their actual maturity, not the maturity level as measured by us. This is an important distinction. If we can't describe a feature or we can't build it or we can't test it, then it is easy to see that it is immature. But true maturity is measured by customer use.

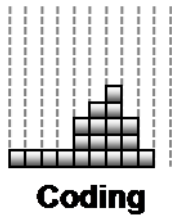


Figure: maturity after coding

During coding, various design problems are discovered and fixed, bringing the maturity of the features to a level that the test/fix process should take them the rest of the way.

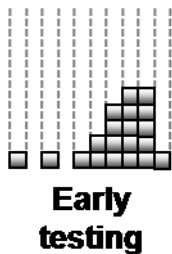


Figure: maturity after first round of testing and fixing.

During testing, it will be discovered that some features need more work. Other features are so problematic that they have to be commented out, disabled, or otherwise removed from the release.

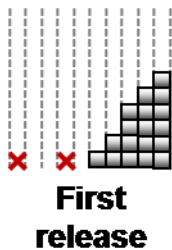


Figure: maturity after first release.

After the first release, customers will provide feedback which results in fixing bugs and enhancing features. It is quite possible that as a result of this feedback a feature or two are removed or scaled back because they just aren't ready for use and can't be brought to an acceptable level of maturity within the foreseeable future.

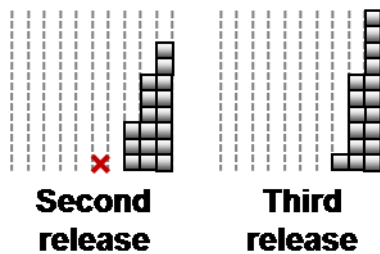


Figure: maturity after multiple releases.

After a couple more releases, the features will finally be at a high level of maturity. Notice however, that different features matured at different rates and many features were actually at a high level of maturity early on.

Different customers are willing to take different levels of risk. Some will trust that the first version of a major release will have high enough quality for their needs and will adopt it. Others will wait for the first or second patch version and rely on the fact that other people have verified that it is okay either by using it and finding very few problems or problems that are fixed in a subsequent release. This is the traditional maturation process.

The maturation of each individual change needs to be thought of as a process in its own right and at least conceptually decoupled from the overall development cycle. That is, if a change reaches maturity prior to a particular release, it is fine to include it in that release, otherwise it is best to defer it. It isn't always easy or possible to do this, but it is a better way to think about product maturity and a good goal to work towards. Several aspects of Agile development make it easier to take this approach.

First of all, features are produced iteratively. In the first iteration, the bare minimum requirements are gathered and the bare minimum design is created. The coding and testing that is done is done as though this is all that will ever be done. That is, while the requirements and design may be minimal, the code and associated tests should be very thorough. The set of features that are done for a major feature are the absolute minimum required. The feature set for the major feature then evolves from iteration to iteration until it is considered mature. In each iteration, it only moves forward the absolute minimum required to get to the next level of useful functionality. This maximizes the effect of customer input and minimizes the amount of guessing that is required. Thus, instead of trying to mature a huge group of large features all at once, features are matured iteratively at their own pace.

With Agile, tests are written early so problems are found early. If it seems like the work that has been planned for an iteration is not at the desired level of maturity, then you can insert another iteration which concentrates on getting the features to the right level. The effect of this may be that you delay the release, but you have that information early and made that decision early.

Because requirement gathering, design, test writing, coding, and testing are so intertwined instead of being separated out in time, much less gets overlooked. In a traditional project, problems tend to get found all at the end of the cycle and there's much more chance of problems getting overlooked.

With short iterations, it is much easier to defer or back out changes that clearly need a longer maturation cycle. Changes that are identified as needing longer maturation cycles can be moved to a parallel iteration which is longer than the main iteration. When maturation is done on a per-change basis, much less of the maturation takes place as a result of customer usage.

Customers Don't Want Frequent Releases

A very common objection to Agile Development is that customers don't want frequent releases. First, Agile Development does not require frequent releases. The benefit of realizing ROI faster comes from frequent releases, but that is just one of the primary benefits. But let's look at the objection that customers don't want frequent releases.

Guess what? It's true. Customers don't want frequent releases. But it doesn't matter! Let's take a look at the example where you only have a single customer, for instance your application is for in-house use. Now the choice is up to them which release to take and when to take it. Keep in mind that a requirement for frequent releases is that the quality of those releases needs to be up to your regular standards. Releasing frequently does not give you a license to release poor quality software! And whether that single internal customer takes that release or not, they can also take a look at it and give you their feedback.

In the case of multiple customers, the important point to remember is that there are multiple customers! You can't think of all of your customers as being a single entity. Your customers don't take all of your releases when you release them, no matter how frequently or infrequently you produce new releases. At AccuRev, we've had customers that stayed back on 2 year old and even 3 year old releases, despite the fact that bugs they had reported had been fixed and features they had asked for had been added.

Basically, the frequency of releases is completely disconnected from the frequency of customer uptake of releases. The exception is when the customer has an urgent need. In that case, you had better be able to attend to that need quickly. Since that is the case anyway, again there is no connection between the frequency of regular releases and customer uptake of releases.

Customers take new releases when there is a good reason to do so. Usually, that is when the delta between where they are and the current release is big enough to warrant the disruption and they have a place in their schedule to put it. As the provider, this window is pretty much unpredictable for you. When that time comes, the customer will take your most recent release or perhaps the one before that under the assumption that more is known about the state of the older release. If you have more than one customer, the chances that your entire customer base is ready to take a release when you release it is pretty low.

When you have multiple customers, those that have a good reason to take the release will do so, and those that do not will not. One of the advantages of frequent releases is that there is a high likelihood

that one of your customers will move to it fairly soon after the release, thus giving you an opportunity to gather feedback right away.

If you are just not ready to release frequently, that's ok. Instead, use short iterations and just go on to the next iteration instead of releasing it. You can still get most of the feedback related benefits of Agile Development by having folks take a look at a demo of the new iteration and avoid the risk of disrupting your customer base with an iteration that is not yet ready for release.

Size of Customer Base

An important consideration when considering frequent releases is how many customers you have. If you have a single customer, for instance an in-house customer, the application of the practice of frequent releases is a bit different. I'll assume that you have multiple customers for the moment and then come back and address the case of a single customer near the end.

Overhead Associated with Producing and Consuming a Release

Another factor to consider with frequent releases is the overhead associated with producing a release, and what is the overhead associated with consuming (upgrading to) a release. The release overhead will influence how frequently you can produce releases and the upgrade overhead will influence how frequently any given customer can upgrade to a particular release. To get the most out of frequent releases, you must work hard to reduce the overhead in both cases.

Supported Versions

When you are releasing infrequently, the number of releases that you have to support in the field is probably pretty low: one or two major releases plus the patch train for each of those major releases. For the sake of argument, let's say it is two releases in a one year cycle. Supporting two releases seems pretty straightforward and the overhead associated with that is entirely manageable.

If you increase your frequency to once per month, doesn't that mean you've moved from two releases to twelve releases? Supporting twelve releases compared to two does seem pretty unrealistic! But let's take a closer look at this situation. Your patch train is unlikely to be a single release after your major release. It is more likely that there is a patch release every 1-2 months at the very least. And what is the policy on that patch train? If you have a problem with a previous patch, upgrade to the latest patch. The majority of the companies that I've talked to (which is in the hundreds, across a wide variety of industries) are already doing frequent releases, it is just on the patch side rather than on the major release side.

So the question is, how does breaking up the release of the major release into multiple releases fit into this equation? There are at least four ways of doing this as described below. The method you use depends on your exact circumstances. As a provider, you should carefully consider the needs of your customer and give the customer as much control over the decision of upgrading as possible.

Customer Burnout From High Frequency of Releases

Unless you force your customers to upgrade, which I don't recommend, then customers will take your releases according to their own schedule, no matter how frequently you release. So, there's no more chance of burnout if you release frequently than if you don't. When you have multiple customers, there is a good likelihood that at least one of them will upgrade to your latest release, so you will get feedback on those releases, it may just be different sets of customers each time.

Frequent Releases for a Single Customer

When you have a single customer, getting uptake of your frequent releases is much harder since there is only one customer to take those releases. However, there is still opportunity here because even with a single customer there is more opportunity for them to take your release when you are releasing more frequently. Consider an example. Your customer has 3 windows during the year during which they can do an upgrade. You produce a major release on the average of every 6 months. That means that while they could have taken 3 releases from you, you can only take advantage of two upgrade windows. Worse, if you miss a window, then they may not take that release in the next window because now the next release has something they really want and they would prefer to wait for that release. So, you may end up only being able to take advantage of a single release window that year. But if you had monthly releases, then it is much more likely that you will be able to take advantage of your customer's upgrade windows.

It is Better to Find Customer Reported Problems as Soon as Possible

Let's say you develop your software in 30-day iterations. At the end of every iteration all functionality introduced during that iteration had all of its tests written during that iteration instead of at the end of the one year cycle. All of those tests pass. You take testing very seriously; you've got very high test coverage numbers, you are using decision based coverage instead of line coverage, and you fix every bug you find.

Let's say that you could release every month but instead you choose to release every year. Now at the end of this year you release your product. I guarantee that your customers will find problems. Let's say for the sake of simplicity that they find exactly 12 and each one is linked to functionality introduced in one of the 12 iterations. This means that for the bug in functionality in iteration 1, you waited 11 months to find the issue that a customer would have found right away.

I'm not saying that you should forgo testing or rely on your customers to be part of your QA department. I'm only saying that despite your best efforts, it is inevitable that there will be issues that you only find after you release, so keep on testing, don't stop that. But release as often as you can.

Also in this example, your customers would still be exposed to the same number of bugs, just not all at the same time.

In addition to keeping your testing standards high, it is better to find problems that you are likely to only

find by releasing to customers as soon as you possibly can. Therefore, release as often as you reasonably can; which may be once a week, once a month, or once a year, but always strive to release often.

Beta

Why do we do beta releases, and what are the characteristics of a beta? The point of a beta is to try to find immature changes and fix them prior to anybody using the product in production. Typically, a beta is expected to have lower quality than a GA release, it has not gone through a full QA cycle, it is expected to have functionality that is not quite done yet, there is no guarantee that any particular feature or behavior will be in the GA release, and users are discouraged from using it in production. In fact, you may be required to sign a document that says you will not use a beta in production.

As a result of all of this qualification, the feedback from a beta is often superficial. However, the expectation is still that beta is verifying that the product is all but ready for production use. But actually, that kind of feedback won't really come until it is put into production use and most betas won't accomplish that goal.

Since everybody expects and accepts that a beta will probably be lower quality than GA, there is no motivation to make it GA quality. In fact, you may hear or say things like "don't worry about that bug now, we'll fix it after we ship the beta" or the beta may start prior to the end of development. After all, it's only beta, you are not really shipping yet. The whole idea of a beta reinforces and perpetuates bad habits.

Once customers starting using the new release in production and find problems, folks in your company may ask "why didn't we find this during (insert any phase of development here from requirements gathering to beta)?" And of course this may lead to calls for more discipline and a move to more traditional development treatments for these problems which will only exacerbate the problems.

Another problem with a beta is that with a long release cycle there are many changes that you want customers to take a look at and the opportunity for customers to look at the changes only comes up once per release cycle.

So why do we do a Beta? Because we believe that if we skipped it, customers would find lots of problems after the release and we'll have to do lots of fixes. Since it takes the organization a long time to produce a release with a lot of changes, we'd like to find the problems prior to GA. We want to make sure that the software is mature enough for general use. If that's really our goal, then let's take a look at an alternative, the limited availability release.

Limited Availability Releases

Right after customers start using the first version of a major release they find problems. That's because your customers have different environments than your QA environment. Even if you did a beta and they tried it, it is very unlikely that they used it in their production environment.

By definition, problems that can only be found in the customer's production environment will only be found when they put your release into production. So, you need to be prepared for this no matter what.

One way to mitigate this risk is to do a limited availability release. Find out who would adopt your LA release first and then limit your initial rollout to those folks. Let them know that there is no difference between what they are getting and what everybody else will get except you just aren't making it available to everybody. And since it is a limited release, you will be able to respond more quickly if they run into a problem.

Let's say you have 20 customers and 10 of them plan to deploy your new release. If all ten deploy and all ten run into a problem and they are all different, then you need the troubleshooting and development resources to rapidly respond to 10 different problems. From the perspective of any one of these customers, it doesn't matter how many people adopted the release, only that you respond quickly to solve their problem. If you do a limited availability release, you accomplish two goals: you are limiting your exposure and you are maximizing your ability to respond to any problems found in the release.

An easy way to implement this is to have people sign up for the release and then provide access to those that sign up until you hit whatever limit you have set for the release. Choose a period of time to go by for the limited release, say two weeks or a month, and then repeat with a wider audience. Once you are comfortable that you are ready, make the release GA.

You have now accomplished the same goals of a beta with better feedback and at low risk. When doing limited availability releases, there is no longer a need for a beta.

Moving to Frequent Releases Takes Time

I'm not advocating releasing frequently no matter what, I'm advocating releasing as frequently as you can. For some that might mean once every two years, and for others it might mean once per week. The frequency depends on your exact circumstances. Simply moving to frequent releases before you are ready is a recipe for disaster. At first it might mean simply changing the release cycle from a year to nine months.

The important point is: if you could release more frequently, but you don't, ask yourself why not. Releasing frequently is a goal, and on the way to that goal you'll find impediments which must be dealt with in order to reach that goal.

Release Options

There are many different ways to provide frequent releases.

Release Option 1: Support all Versions in the Field

This option is really only usable if your customers upgrade frequently enough that you generally have few versions in the field. Using this option, you release as often as you can and support whatever version that a customer is on when they encounter a problem that requires a fix.

Release Option 2: Always Upgrade to the Latest Release

In this option, there is only one release train. If you have a problem with the release you are on, the only way to get it addressed is to upgrade to the latest release. To use this option, you've really got to have

your act together. Your software must be incredibly easy to use and you must be able to find and fix problems almost instantaneously. Re-training must be either non-existent or minimal. It is also good if you have a history of having very few problems in production.

The upgrade must also be incredibly easy. Preferably it is either a single-click to install, or it is fully automatic. In some cases, such as virus definitions, it makes sense to have an automatic update capability. In other cases, taking an update is disruptive and the customer would prefer to wait until they really need to do it.

Release Option 3: One Release For All

This option is primarily a Software as a Service (SaaS) release method, for instance Litle & Co, Gmail, Mapquest, Salesforce.com, and RallyDev . In this case the application is hosted by the provider and when they upgrade the application, all users are affected. There is only ever one supported version and that is the current version. As with the previous option, you've really got to have your act together to support this option.

Release Option 4: Dual Release Trains

The Dual Release Train option is the closest to a traditional model, but still allows for frequent releases. In this model you have the "conservative" train which is your "hardened" release family and the "frequent release" train. Each train has a separate policy. The policy of the conservative train is that releases come out infrequently and do get patches when needed. The policy of the frequent release train is the same as option 3 above: always upgrade to the latest release.

There is one significant difference with this option as compared with the traditional release pattern. All releases except the patches come from the same source: the frequent release train. Thus, the size of all releases are small. For example, instead of having release A in a six month period, you have A, B, C, D, E, and F. The frequent release train consists of all six releases. The conservative train consists of A with a patch or two and then moves to F. Customers can opt to jump from train to train, realizing that they have different policies.

This option is not the same as the stable/unstable model where the unstable "early release" is not yet fully tested. All of the releases are stable. The frequent releases are not like traditional patches, they are like traditional major releases, except their content is smaller. They are all well tested, stable, and of high quality. If an organization can't do this, then it should not change its frequency of releases until it can. Agile is not an excuse for poor quality, it is a way to produce significantly higher quality.

We The People

Whole Team

This is one of the harder practices to implement, but also one of the most important and most effective, especially when combined with collocation of the team. The idea of this practice is that for a given project, that project is implemented using a team that is composed of the same members for the entire iteration and that those team members do not have assignments outside of that project. 100% of the work required to complete the project is done by members of the team. Anything the project needs is done by the team members. For instance, if you have a “QA team” that does all QA and the QA for the project is done by the QA team, then you do not have a whole team. With a whole team, you have the QA resources you need dedicated to the team for the duration of the iteration. It is even better if the team stays together for the duration of the project.

A whole team is a cross-functional team comprised of between 5-9 people. By cross-functional I mean the team as a whole contains all of the skills needed to accomplish the team’s goals, not that each team member is capable of doing any task. If you have more than 9 people, then consider splitting people up into multiple whole teams.

The whole team practice encourages everyone on the team to think in terms of the outcome of the project rather than optimizing the work done in their particular area of expertise. For instance, if you have a GUI team, a database team, and a QA team and you have three different projects, the people in the various teams will feel affinity to their functional area rather than affinity for any particular project. The customer doesn’t care if your GUI team is particularly good at what they do, they care if the product that they use meets their needs. By having whole teams, it is more likely that people will be focused on customer value rather than functional area efficiency.

The existence of real whole teams is a good indication that Agile development is alive and well in your organization. Accomplishing and sustaining it indicates that the value of Agile is understood at many levels of management, that you have balanced your resources and eliminated multi-tasking, and that customer value is paramount.

Members of a team feel more ownership of the project which provides benefits for the individual members, the team, and the organization as a whole. Team members have a greater sense of responsibility and pride of ownership. Instead of people saying “well, I finished the development, I don’t know what is taking the tester so long” they are more likely to say “what problem are you running into and how can I help?”

A Better Work Environment

A new member of an Agile team that had previously expressed interest in getting a cube along a window, when asked if he preferred being part of a whole team without a window or being a shared resource with a window said that he absolutely preferred being a part of a whole team. “In the past, when there was a demo of a product that I worked on, I didn’t feel a sense of pride because I didn’t feel like I was part of any team”

Collocation

Delays in the communication of important information can be a major impediment in an Agile project. If you suspect that you are experiencing problems due to communication delays, collocation may provide some assistance. For a very large project, it may be impractical to collocate everybody, but in that case you can collocate your whole teams.

Great care needs to be taken when using collocation. Collocation can greatly aid communication, but if it is relied on to make up for other problems, it can keep those problems hidden. It can also lead to problems when the structure of the team changes. For instance, if a team is completely collocated and has unknowingly built up habits based on the implicit assumption that the whole team will always be collocated, it makes it difficult to add new team members or new teams that are not collocated. For instance, if the team relies on a single shared whiteboard for project status, but a new team member is working from home for an extended period or perhaps permanently, then the whiteboard alone is no longer sufficient.

When using collocation for a project which has more than twelve people on it, consider collocation at the whole team level. For instance, if you have 100 people on the project, split the project up into 8-10 whole teams where each team can function as though it had its own project and can do all or most of its own design, development, QA, and documentation. Then you can collocate whole teams without having to collocate the whole project.

Transitioning to Agile Development

Focus on the Goal

The goal is to experience some or all of the benefits listed at the beginning of the book, not to follow the book by rote. It would be nice if everything in this book was exactly right for you and would work if followed exactly for all circumstances. The only thing that will really work is focusing on the goals that you want to achieve. The intention of the book is to impart a mindset which you can then leverage in pursuit of your goals. If something in the book just doesn't make sense for your circumstances, then don't do it.

Stage 1: Getting Ready

Flying Blind

I was at Sugarloaf one weekend when I was just starting to snowboard. They had very little cover and very few trails open. But then Saturday night, they got 33" of powder. The next morning, a friend and I came to a trail that was closed. It looked like a great trail; endless powder with no tracks. The problem was that it had been rocks and grass the day before and there was no base underneath, so it was just the same as riding on rocks and grass. It was not a pleasant experience.

In retrospect, we were lucky. We saw sparks flying from each other's snowboards many times on the way down as our edges hit rocks. We were flying blind and could have easily fallen and hit those rocks with something a bit more fragile than our edges.

Adopting Agile without understanding it and without creating a proper ecosystem for it is destined for a similar fate. I've visited far too many companies that attempted to adopt Agile after no more than a single visit from an Agile coach. In these circumstances, usually only a few people are on board initially and there are many skeptics because there was insufficient effort invested in preparing for success.

Adopting Agile development requires breaking down mental barriers and building up new skill sets. There is nothing particularly hard about actually doing any of the Agile practices, but many of them are counterintuitive and do take a bit of practice to get used to. Don't underestimate the amount of effort required. It is at least on the order of taking a team which is very used to writing in C++ and moving to Java. There's nothing particularly difficult about such a transition, but there are many subtleties which must be learned and it takes a while to build up the same base of experience.

Self Education

Before getting too far along, make sure that you have done your homework. Read other books on Agile, find other folks in your organization that have done Agile development before. Go to conferences, join

the local Agile user group, become a certified Scrum Master, do whatever you do to find people that you can lean on when you need it.

Scope

Determine the best scope of the adoption. As with most things, it is best to think big, but start small. Is there a small project with no more than 12 people that is amenable to piloting something new? There are two advantages in starting small: minimizing disruption and leveraging what the pilot group learns about doing Agile in your environment when transitioning other groups.

Scouting

Agile development has certain perceptions related to it. One of the most prevalent perceptions is that it is “for small groups.” That was certainly my perception when I first started hearing about it. Another perception is that small iterations aren’t a good thing because customers don’t want frequent releases, there’s more overhead involved, the quality will be lower that way, and it makes marketing’s life more difficult.

If you just advocate Agile without knowing the landscape, you run the risk of alienating the people whose support you need in order to go Agile. Find out how receptive your organization is to going Agile. Think about who is in a position to help or hinder its adoption. Those are the key stakeholders. You will need to find out where they stand, what they like about the idea of going Agile, and what their objections are. This information will come in handy later in the adoption process.

Prepare The Organization

Once you have a basic lay of the land, see what you can do to raise people’s awareness and understanding of the advantages and potential pitfalls of Agile. Do a presentation for folks that are interested, invite in somebody from the Agile community to do a presentation or workshop. Recommend books and websites that you found helpful.

Transition Development and QA Together

The most important component for reducing the rework period that comes from long iterations is improving your testing process. For many if not most organizations, this is the hardest goal to achieve in practice.

If you don’t have any automation at all, it is a good bet that there is an ingrained belief that automated testing is either a bad idea, doesn’t work as well as manual testing, is too expensive, or that the tools are too expensive. As a result, it may be that there are no QA automation experts in the building and possibly nobody with scripting skills in the QA group. The best course of action in this case is to concentrate on introducing the idea of QA automation.

If it is clear that there is a bias against automated testing that is too strong to overcome any time soon, another tactic is to have the development organization champion automation with an eye towards handing it over to QA once the idea catches on. A good place to start is with unit tests. It should be clear from the start that your goal is to have QA own test automation. Developers will write good tests, but

they are too optimistic by nature. Developers start from the assumption that “it will work.” QA people start from the assumption that “it doesn’t work and I’ll prove it to you.” Pessimism is a good trait for a person creating tests.

Keep Your Healthy Skepticism

Think carefully about the value of each practice that you plan to adopt and make absolutely sure that it is appropriate for your exact circumstances before you adopt it. You shouldn’t be adopting practices simply for the sake of saying that you are adopting Agile practices. Every practice you adopt should be because now that you know about it you simply can’t imagine getting by without it.

Don’t Throw Out the Baby With the Bath Water

I’ve seen many first-time Agile projects fail because they threw out everything they already knew about developing software. Most of the individual steps of developing software with an Agile methodology are the same as traditional software development. You still have to talk to customers, decide what you are going to do, write code, write tests, do testing, etc. Agile development is simply a different framework for those steps. You have a business to run, and you don’t really need to introduce large and sudden risk factors. Before you decide to chuck everything that you know and start from scratch, spend some time developing your knowledge and understanding of Agile. Look closely at your existing practices and see which ones will fit well within Agile, and which ones may cause problems. Create a game plan and start out gradually.

Don’t make lots of changes all at once. Keep doing whatever you have been doing until you specifically decide to adopt a new practice.

Create a Process Document

The exercise of creating a process document will give you a much better idea of exactly where you are today and help you to determine next steps. This process is described in the following chapter.

Stage 2: Establishing a Natural Rhythm

The next stage is to make some early and measurable progress to encourage adoption. As it is, you have actually already experienced some success. You’ve planted the idea of process improvement, you’ve got a map of the areas of general consensus and contention, and you’ve provided a forum for people to learn more about the current process, uncover misunderstandings and correct them.

One of the areas of consensus you’ve uncovered is very likely a problem which everybody agrees should change, there is a consensus or near consensus on how to improve it, enough pain to easily instigate a change, and a sufficiently small scope that the change can be made in a relatively short amount of time. That should be your first area of attack. It doesn’t really matter what it is or if it has anything at all to do with Agile. At this point, you just want to establish the idea that process improvement is worthwhile, doable, and worth doing on a regular basis.

Once you have successfully made this first change, update the process document to reflect the change to the actual process. Make sure that people understand that this is a living document and needs to be continually reviewed and updated just as you would with requirements, code, comments, and tests. It is a document which everybody can read to learn more about the details. It can be used for educational purposes when adding a new member to the team.

Short Iterations

The main point of this stage of adoption is to discover the natural rhythm of your team and the project that you are working on. It may be a month, it may be a week, or it may be some other timeframe between 1-6 weeks.

The ideal iteration time is 1 week, but any time up to a month is fine. If you are doing Agile development for the first time, getting the iteration time down to one month is a good initial goal, but not required.

For your first iteration, start with a goal of a one month iteration time. It is better to make it a goal rather than a hard requirement. Making it a hard requirement will encourage rushing to finish or taking shortcuts at the end which will defeat the purpose of establishing a rhythm. It will take time to get used to short iterations.

Don't plan to do a release from the first iteration. Whichever iteration you do plan to do a release from, keep your release process the same for now. The purpose of the first couple of iterations using Agile is to identify obstacles in order to start removing them.

Create The Initial Backlog

You don't need to have a product owner or ScrumMaster or make any changes to how you plan a release at this point. Just do whatever you would normally do to plan the contents for a release, but focus on the work that has the highest business value and what you can reasonably expect to accomplish in your first iteration.

Once you have a list of work items for the iteration, turn it into a backlog. The backlog will help you to resist the temptation to add more items than you can do in a single iteration. To create a backlog, order the list from highest business value to lowest business value. To determine what to work on for the first iteration, make a conservative estimate of how much work your team can take on. Create a backlog of the highest priority items, adding estimates to them as you go in order to determine the cutoff point.

There are many tools you can use to manage your backlog. The minimum requirements are that you can easily create an ordered list and easily change the position of items in the list. Two simple tools that are frequently used for this purpose are 3x5 cards and spreadsheets. They have the advantage of simplicity and flexibility, but do not offer the capabilities of software tools to share and backup information. Currently, there are only a handful of software tools that provide the ability to create an ordered list of work and easily manipulate it, but more are becoming available all of the time.

Define “Done”

You should have a clear definition of what “done” means for a work item. At a high level, any definition of done should include whatever you believe is necessary for that work item in isolation to be considered shippable. Think about your entire release cycle, not just what it takes to get something ready to check-in. Do you eventually do a code review? Then that should be a pre-requisite for “done.” Other candidates are: documented, tests written and all pass, demoed by a QA person to a third party (that is, not just the developer and the QA person themselves).

Transitioning to Agile QA

You may still have some tests that resist automation. However, all of the tests that you run should be run from a test plan and the manual component of your test plan should be as short as possible.

The first place to start improving your testing process is to only consider a work item ready for release when there are automated tests for it and they all pass. While these tests may include tests contributed by developers, they should contain all of the tests that would be written by QA as part of the normal release process. Until you have fully automated your test suite, you should continue to run your regular release process whenever you do a release.

In parallel you should start fully automating your test suite. This will take time. As you automate your test suite, this is a good time to consider what your test suite should contain regardless of what your test plan currently specifies.

To determine which tests you should automate, which may involve the automation of existing manual tests cases or the addition of entirely new tests, create and calculate a test exposure graph. This will serve as your guide for prioritizing which tests to create first.

Recognizing Success

In a traditional project, it is well known that the software doesn’t need to be ready to ship until just prior to release. Test case writing and test case execution are mostly done at the end of the cycle and problems hide until that time. Also contributing to the problem is that in a traditional project, bugs during development are found and reported, but not always fixed right away. As the code freeze date gets closer, the amount of work required to button things up and get the software ready for release increases. If you were to graph the amount of effort it would require at any point in time to stop development, write all necessary tests, find and fix all problems, and then ship the software it might look something like this.

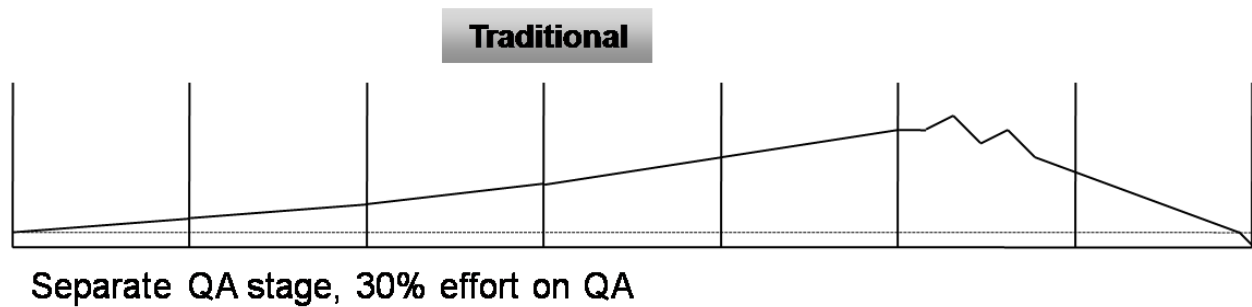
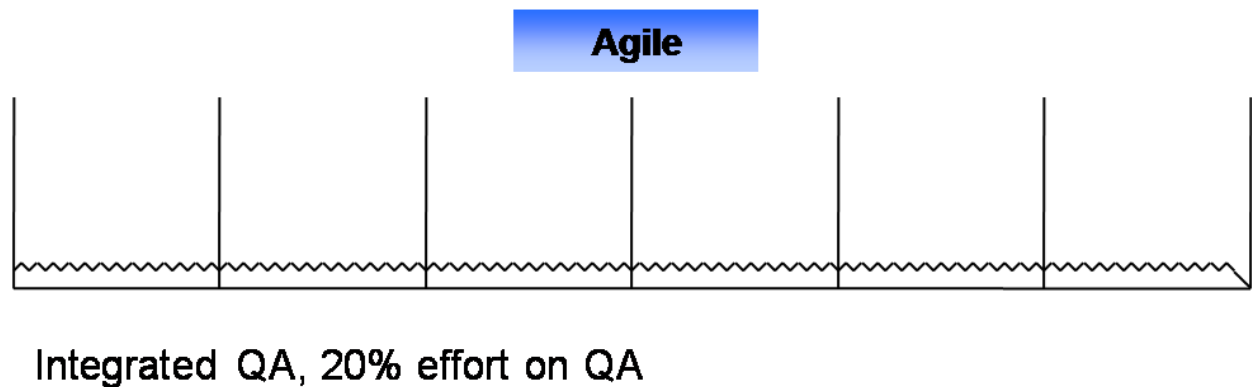


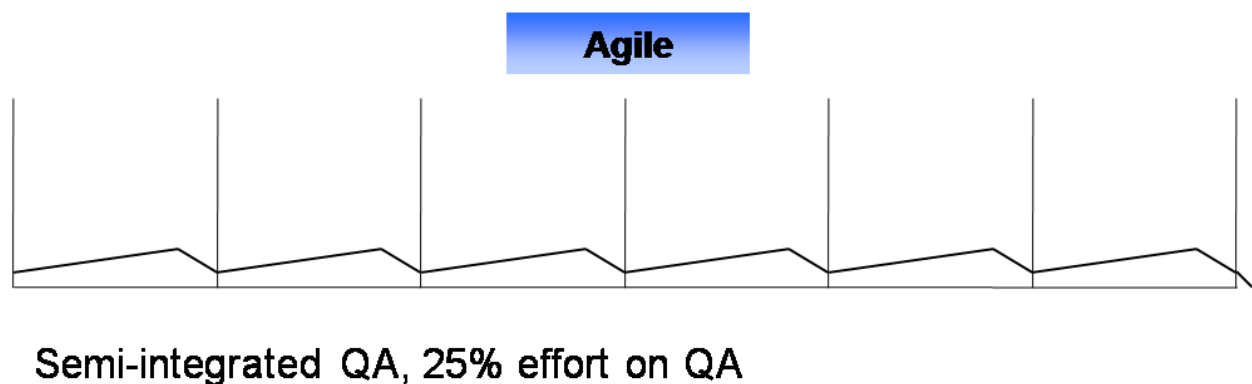
Figure: ready for release signature of a traditional project.

In a mature Agile project, development and QA are fully integrated. Coding and test writing are done together. Ideally, tests are written prior to the start of coding, but in any work items are not considered done until the tests are written and passing. The ready for release signature in this case looks like this:



If the practice of short iterations is used, but testing is still done all together just before the release, then the signature looks like the same as it does in the traditional development example.

Even if QA is not fully integrated, getting all tests written and passing prior to the end of the iteration is a good start.



These graphs are meant to illustrate the signature of various possible development scenarios in order to more easily differentiate between traditional development, successful Agile development, and

unsuccessful attempts at Agile development. The intent is to help teams that are new to Agile to recognize success.

Update the Backlog and Plan for Future Iterations in Parallel

Don't try to do requirements gathering and long-term planning at the beginning of each iteration. The only thing you should be doing with the backlog at the beginning of an iteration is planning the contents of the iteration from a pre-existing backlog. Gathering requirements, updating the backlog for future iterations, and planning for the future should all be done on an ongoing basis in parallel with other activities.

Iteration Retrospective

Before you get too giddy with the success of your first iteration, be sure to schedule a retrospective. This is exactly same as the traditional "post-mortem" but doesn't have the same air of death about it. The purpose of the retrospective is to take a look back at the iteration and look for opportunities to improve your next iteration. What went wrong? Why did it go wrong? What went well? What ideas do people have for improving the next iteration? Most likely, the duration of your first iteration overshot your goal. This should be the focus of the retrospective. If in fact you did make the goal of 1 month, what could you do to reduce the time to 2 weeks? What could you do to have a higher ratio of value-producing activities such as adding more new functionality? You should not change your goal of 1 month to 2 weeks, even if you made the goal. At this point, it is more important to establish a rhythm rather than reduce the time of the interval.

At the end of the retrospective, decide as a team which changes you want to make in the next iteration. Don't try to change everything at once, concentrate on the top 3-5 ideas. Make sure that somebody is taking notes. The notes will be invaluable in your next retrospective for detecting trends.

Celebrating Your First Iteration

Once you have successfully completed an iteration and held the retrospective, be sure to celebrate your success! Let everyone know that you achieved your first goal. Give a demonstration of all of the work the team has done to all of the key stakeholders in the project. This will provide evidence that the process changes are working. Make sure to highlight all of the "invisible" work that has occurred behind the scenes such as "all of the code in this iteration has been fully code reviewed" or "there are tests for all of the changes and they all pass." And don't forget to do something fun to celebrate your success. This will reinforce that an important milestone has been achieved and reward the team members for their success.

Repeat

Breaking down entrenched habits and ingrained beliefs while simultaneously establishing new habits and beliefs takes time and patience. The most important things at this stage are to establish a rhythm via the short intervals of the iterations and to find and remove obstacles. Because you should already be experiencing positive results, there is no need to rush to the next stage and endanger the progress that you've made. Remember that there are lots of people watching the success or failure of this process and

support for a new way of doing things is still shaky at best. That includes you and everybody else on the team! I recommend you simply repeat the process with another 2-3 iterations, finding and removing obstacles as you go.

Keep Your Process Document Up to Date

As you successfully implement process changes, keep your process document up to date. Don't update it prior to successfully implementing a change. Remember that the process document describes what you are really doing. A good time to update it is right after each retrospective.

Stage 3: Your First Release

Release

At some point in this process, you will do a release of your software. Resist the urge to make changes to your release process! It will be incredibly tempting, you will see opportunities for improvement everywhere, but just don't do it. This is where the most risk to your new venture lives. Go into the process knowing that all of your hard work in the iterations leading up to the release will pay off in the form of fewer problems during the release process and take that as your consolation for leaving the process untouched.

Release Retrospective

Ok, you know you wanted to make changes to your release process, now is the time to talk about it. A release retrospective is exactly the same as an iteration perspective, except in scope. In a release retrospective, the scope is everything that happened between the last release and now including the release process itself. Make sure to reserve more time for this meeting. Again, choose at most 3-5 changes for the next release.

It is important that your release process be as simple as possible. Everything from "code freeze" to the actual deployment of the software should be examined with a very critical eye. Anything that can be removed or automated and done during each iteration should be a candidate for change. The more that you can fold into each iteration itself, the shorter the release process will be and the more energy you can put into creating customer value.

Celebrating Your First Agile Release

Congratulations! You've done it. Don't forget to celebrate. You've got the basics down and could continue on indefinitely, but now is the time to leverage your initial investment in Agile and get even more benefits with just a little more investment of effort.

Stage 4: Continuous Process Improvement

There is always room for improvement. Every obstacle you remove will improve your overall understanding of your environment and open up new opportunities for improvement that you may not have noticed before. Sometimes, after making a series of small apparently unrelated changes, a whole new class of opportunities for improvement will open up to you.

For each iteration, document your goals for process improvement. This should be a mix of improvements that are specific to your circumstances as well as adding practices from the process improvement sections.

Manage Resource Imbalances

Resource imbalances manifest as a backlog of work in one or more types of activity. For instance, you may have a QA backlog. There are six options for dealing with a QA backlog: do less QA than needed and thus shift the burden of finding problems that you could have found onto your customers, increase your QA capacity, decrease your development capacity, have development idle, have development help with QA or allow the backlog to grow. The larger your testing backlog, the longer it will take to ship it and the greater your opportunity cost.

The imbalance may be in either direction. After you transition to Agile development, you may find that you have more QA resources than are needed. In that case, you have the option of having QA take on some of the work currently done by developers. See the section on the role of QA in an Agile project for more on that topic.

Wherever there is an imbalance you have the same six options as described above. For example, you may end up with project plans that are never used, developers idle because the design isn't ready yet, etc. To the extent that some of the resources are actually the same people you can use that fact to manage this problem.

In any case, at this point it is very likely that you have resource imbalances. It is very important that you keep an eye on this and take steps to even things out.

Reinvest in the Engine

By now, you should be able to make a good case that process improvements have led to productivity improvements and that productivity improvements have led to benefits to the organization. From here on out, always be looking for opportunities to reinvest in the engine and remove impediments. This includes:

- Career advancement activities
- Hiring additional resources
- Addressing any HR problems
- Upgrading developer machines, build farms, and any other hardware that will remove impediments
- Acquiring any software that will remove impediments
- Improving the working environment (as defined by those that work in it)

Recommended Order of Practice Adoption

There are many areas of software development that work together to contribute to the agility of an engineering group. For each area, there are practices which build on each other. Each practice represents a positive step forward. This section lists each practice in its own area. By looking at these lists you can assess where you currently are and what your next step should be. This should help you structure your process improvement efforts.

Process/Planning

Use a process document – as described above.

Use a backlog – as described above.

Short iterations – as described above.

Define done – as described above.

Iteration retrospective – as described above

Release retrospective – as described above.

Reduce Your Iteration Interval – keep making improvements and working to get the iteration time down to 1 month. If it makes sense for your circumstances, keep working to get the iteration time down to 1 week.

Stop the line – stopping the line is a practice from Lean Manufacturing. When it was first instituted this literally meant pulling a cord that stopped the entire assembly line whenever a problem was detected. Over time this evolved to stopping just the part of the line where the problem existed. There are two parts to stopping the line: stopping work to prevent the introduction of more problems, and performing a root cause analysis to find the root cause of the problem and remove it so as to permanently fix the problem.

Whenever a problem crops up, stop the line. That is, anybody involved in the cause of the problem should stop whatever else they are doing (unless of course it is something critical) and do an on-the-spot retrospective. Once the cause is found, decide on a fix and implement it right away.

If something crops up and you are not sure if it is really a problem or not, stop the line anyway. At first, this will happen with alarming regularity, but over time it will occur less and less frequently as the root causes of problems are found and removed.

Root cause analysis – this is the same as stop the line with the additional requirement that not only is the cause found and fix, the problem that allowed the problem to occur in the first place is found and documented for fixing at a convenient time.

Root cause removal – now, when a root cause analysis is done, the fix is implemented right away.

Build

One-step build – whatever your current process is for doing a build, it should be possible to do a full build with a single command. This does not mean running one script multiple times with different parameters, it means running a single command with no parameters.

Everybody using the same build process – this is a fundamental step towards continuous integration.

Automatic results summary – the results of the build are parsed and summarized automatically. There is no need for manual reading and summarization of the build results.

Nightly build – this should be straightforward. Simply set up a regularly scheduled task which runs your one step build every night and then follow it up by running the automatic results summary. It should start at a well-known time based on a well-known branch or label and the build should finish prior to the time that developers come in to work in the morning.

Continuous Build - builds are initiated by changes in the SCM system.

Test/QA

Sanity test – create automated sanity tests. Sanity tests are very simple tests to make sure that the software has at least enough functionality to go on to the smoke test. For instance, with a car this would consist of turning on the car, making sure it can go forwards and reverse, turn, and that the brakes and lights work.

Smoke test – these are the absolute minimum tests required to make sure that all core functions can perform their most basic function.

One-step test – set up your automated test suite so that it can be run in its entirety with a single command which takes no parameters.

Automatic results summary – the results of the testing are parsed and summarized automatically. There is no need for manual reading and summarization of the test results.

Automatic test initiation – all automated testing is initiated by a successful build.

Integration

One step integration – this runs your one step build and if the build succeeds, it runs the full test suite immediately afterwards.

Multi-stage integration – if there are multiple teams working in parallel or there is work on multiple major features in parallel, implement multi-stage integration.

Do all build and test practices – keep implementing all build and test practices until you are doing them all.

Deployment test – whenever both the build and test succeed, automatically generate your deployment artifacts, whatever they may be. Automatically run a deployment and run tests to verify that the deployment succeeded.

Releasing

Be sure to consult the section on “Tuning the Frequency of Your Release” for details.

Same as always – there are many hidden issues in your current process. Just going to shorter iterations will expose enough to begin with, don’t change your release frequency at the beginning of your adoption.

Using Agile for maintenance – this is a natural starting point. Maintenance releases already share many characteristics of Agile development and are a good proving ground.

Reduce your release timeframe by 1 iteration – this will start the process of discovering your natural release cycle. Different organizations have different natural release cycles. If your release cycle is more than 6 months, you might consider reducing by two iterations to start.

Further reduce your release timeframe – continue the process of reducing your release timeframe every release until you reach your natural release cycle.

Single release cycle - move from having major and minor releases to just a single release cycle. At this point, your release timeframe should be short enough that customers will not have to wait a long time for an important piece of functionality and your quality should be high enough that the need for hotfixes and service packs is much lower than before.

Shipping at will – this capability is reached when you have your process set up such that you have a version of the software which is ready at all times to begin the deployment process.

Going Agile When Nobody Else Will

If you are interested in going Agile, but you are surrounded by folks that are not, don’t despair. You can still experience significant benefit from Agile on your own. Also, after reading this section you won't be able to say that somebody is preventing you from going Agile. This section is primarily directed at developers, but other folks may benefit from a similar approach.

The basic idea is to adapt the various Agile practices for a team of one. Let’s start by taking a look at scrum master, user stories, one piece flow, product owner, backlog, daily standup, iteration reviews, retrospectives, unit tests, and refactoring.

Scrum Master of One

Since there is only one person in your team, you are automatically the Scrum Master. Congratulations, but try not to abuse your new found authority over your team. Remember, the Scrum Master is a facilitator, not a boss! The key thing that the Scrum Master does is to facilitate Agile. The Scrum Master

understands how Agile works, makes sure that all of the meetings are happening, and also looks for and removes impediments.

If you are going to be an effective Scrum Master, you need to understand Agile development. In addition to reading this book, if you can get your company to sponsor you to take a Certified Scrum Master course, that's a great way to get started. If not, consider sponsoring yourself. It is expensive and will probably require you to expend two vacation days, but I assure you it will pay off in the end.

Reverse Engineered User Stories

The next step is to start working with User Stories. A user story is a simple statement in plain English (or whatever language you are primarily using) that describes the work that needs to be done from the perspective of the end user. They are generally of the form "As a user, I want X" where the "X" describes a goal that can be achieved with the software to provide value to the user. For instance, "As a user I want to be able to purchase anything that I am looking at with a single click." The user story doesn't describe the nitty-gritty details.

I realize that since you are not the product manager (or business analyst), you don't get to determine what you are working on or how it is described. The way that you can benefit from user stories is to reverse-engineer whatever description you have been given of the work you need to do into user stories. The benefit to you is that you will either have a better high-level view of what you are looking to accomplish, or you will immediately start to see holes in the description of the work you have been given. User stories give you a framework for thinking about your work.

Once you start using user stories, you will start having more questions for your manager or the product manager or business analyst. At first, folks may see your questions as annoying, but over time I believe they will see your questions as reflecting your interest in their success and the success of the company.

Divide And Conquer

Now that all of your work is described in terms of user stories, think about the size of those stories. Can any of the stories be further broken down? For instance, if you had a story that was "As a user I want a simple calculator function" you could break that down into "As a user I want to be able to add a number to an existing value," "As a user I want to be able to subtract a number from an existing value," etc. Breaking your work down into user stories has many benefits from an Agile perspective. We'll look at just a few of them here.

One Thing at a Time

Most Agile teams use iterations for organizing their work. As a team of one, that probably doesn't make much sense. Instead, as a team of one, try to do One Piece Flow. The idea of One Piece Flow is that you do all of the work required for a single user story before going on to the next one. Try to resist working on two stories at once.

Product Owner For a Day – Creating the Backlog

Ok, now you have a bunch of user stories that describe everything you've been assigned to do and they are as small as you can make them. It is time to put on your product owner hat. As product owner, you need to rank each of these stories relative to all the rest in terms of value. If you are not sure what the value of the stories is, ask around to see if you can gather enough information to make a reasonable guess. Perhaps you know one of the actual end users. Ask them what they think about the stories you have.

Over time, you'll get better and better at ranking your user stories. No matter what you do, having some sort of ranking is much better than none. Rank all of your stories from most value to least value and start working on the top story. This ranking is called a backlog.

The benefit of having a backlog of small user stories and working on just one story at a time comes into play when somebody taps you on the shoulder and asks you to do something other than what you are currently doing. When you are interrupted, you now have some options. You can point to your backlog and say "where do you think this new request fits in this backlog? Is it really more important than this task that I am currently working on?" If you are lucky, the person interrupting you will decide that their request has a lower priority than whatever you are currently working on. At the very least, the backlog provides a framework for discussing the relative priority of the new request.

If the new request does seem to be higher priority than what you are working on, you have one more tactic you can try. Since your stories are small, you can say that you only have one task in progress and how much work remains on the current story and suggest that you finish what you are doing prior to starting the new task. Over time, this tactic should work more and more often as you get a reputation for delivering higher quality work and being able to quickly change gears. The backlog examination exercise should also slowly train other folks on the value of user stories and the backlog. Your goal should be to have folks give you new requests and for them to say "let's see where this belongs in your backlog."

Even if you have to stop whatever you are currently working on and start working on a new task, it is much easier to change course if you only ever have one story in progress at a time and it is small. Don't forget to treat this new request the same as any other task. Create one or more user stories for the request and order them relative to each other at the top of the backlog.

Once you get the hang of maintaining a backlog, consider making your backlog visible to others. One way to do this would be to print out your backlog in a large font along with an indication of which stories you are currently working on and post it near you in a place that passersby can clearly see. You may even find that the number of interruptions goes down as people can see for themselves that you are clearly working on the most important thing and leave you alone to get it done.

Another benefit of having your backlog visible to others is that it facilitates the process of having more people contribute to ranking the backlog. If somebody walking by has information that you don't have

and they see that your backlog doesn't reflect the information that they have, you have a better chance that they will share that information with you to help provide an even better ranking.

Talking to Yourself

It may seem silly to do a "daily standup meeting" with yourself every morning. So don't stand up. But do go through the exercise of going through the three questions:

- What have you accomplished since the last meeting?
- What are you working on next?
- What impediments do you have?

This will help to keep you focused on your current story. If you find that you just aren't making the kind of progress you'd like to make, it is time to get some help! Let's face it, if you go work on something else in the meantime, you'll eventually have to come back to face whatever it is that is blocking you now. Yes, it is possible that you will solve the problem on your own, but why not at least find somebody to describe the problem to? Maybe the simple act of explaining the problem will provide a new insight. In any case, if you do end up working on something else, try to keep the total number of stories you have in progress to a minimum.

The Moment of Truth - Did it Work For You?

When you finish a story, now it is time to do an iteration review. Normally this would be done at the end of an iteration, but since you aren't doing iterations, you might as well do the iteration review now. It is simple. Find somebody to demo the story to. The idea here is to have an audience and get feedback. If you feel like you aren't ready to have somebody else see a demo of your work... then your story isn't really done, is it? Keep working on it until you feel ready for an audience.

In Retrospect, That Was a Good Idea!

Once you have finished your story and done the "iteration review," it is time for the retrospective. Find a quiet place that you can spend 30 minutes or so to reflect back on how things went, from the time that you had the story ready to the time that you finished the iteration review. What went well? What didn't go so well? What could you do better in the future? Write down your thoughts for future reference.

Unit Tests

If you aren't already doing unit tests, now is the time to consider it. Unit tests are something that you can decide to do on your own for your own sake. At first it may seem that it is slowing you down, but consider all of the rework you have to do when QA finds problems with your work. Why not invest that effort into writing unit tests? If you are using Java, you can use JUnit. If you are using .Net, you can use NUnit.

Refactoring

Unit tests go hand in hand with refactoring. Refactoring is simply the practice of rewriting code to make it simpler to understand and change. The more unit tests you write, the more you will refactor the code and the easier it will be to understand the code and add new functionality.

Growing Your Agile Team

See if you can get somebody in QA interested in what you are doing. You might look for their feedback on the ranking of your backlog or the quality of your unit tests. A QA person would also be a good person to invite to an iteration review. The more that QA sees the work you are doing as helping them, the more they will want to help make you successful. A close relationship with QA is an essential ingredient to Agile success.

As you practice these Agile techniques, I think you will find that you are producing higher quality results in less time as well as keeping focused on the work that the business cares the most about. You may also find that what you are doing gets noticed and appreciated. If you can get other folks to follow your example, you may soon find yourself part of a real Agile team. Good luck!

Process Improvement: Creating a Process Document

When it comes to the process of getting work done, it is like we are living in the days before the written word when knowledge was passed down through storytelling and things were always lost or embellished along the way. Your process document is the “source code” for your process and will form the basis of all of the changes to your process. What is this program composed of? How well does it perform?

What is Your Process Today?

The best place to start is to document the existing development process that the team uses today. I don't mean the process that they wish they were using or that they are supposed to be using, but rather the process that they are actually following. How exactly do they decide what changes will be made? If they call ad-hoc meetings, write ideas down on paper, and then whatever two particular individuals can agree on goes in, then that is the process used today. Write that down.

Whatever people actually do today, that is your actual process. It is important to go through this stage to gain an accurate understanding of how things work today. The mere process of documenting the existing process will often uncover a surprising amount of valuable information.

Is the process different for new development versus patches? Even if the process sometimes involves a developer building on their own machine and sending the product directly to a customer, write that down. What steps does that developer follow?

Use a low level of detail in the process document. It should be just enough so that a new person on the team can read the document in a few minutes and understand it enough to get started. Over time, you may find that people run into trouble in a particular part of the process. In the short term, you should update the document such that people better understand the problem area, but as soon as you can you need to consider changes to the process to eliminate the problem. Whenever you find yourself writing what amounts to a script, consider automating that step.

Here are some things to consider as you create your process document:

- The process itself, such as gather requirements, create design, etc.
- Whether you are using short or long iterations
- The length of your iterations
- The people on the team, their skill sets and interaction styles
- The physical location of all teams
- The hardware that the team uses for building, testing
- Scripts for managing, building, testing, deploying

- Source control system
- Issue tracking system (bugs, defects, enhancements, requirements)
- Build tools: make, continuous integration, build accelerators
- Languages
- Compilers
- Virtualization software
- Static analysis tools
- Leak detection software
- Automated tests
- Manual tests
- Test plan
- Customers and the market
- The steps required to deliver a release to customers
- Hotfix process
- Service pack process
- Hiring process
- Technology adoption process

After you have documented the current process as you understand it, circulate it among all of the team members for comment. You should ask for two kinds of comments: first, is there anything missing or inaccurate and if so, what changes do you suggest. Second, how do you wish the process was different and what suggestions do you have for improvement?

It is not important to do a large number of iterations on the process document. If you circulate the document for comment only once, that is a sufficient start. It is likely that there is sufficient noise or disagreement at this point that a 100% accurate picture of the process is unobtainable.

Process Improvement: Complimentary Practices

This section describes a variety of complimentary Agile practices which may aid in your adoption of Agile. They are not included in the main adoption section because they are not universally applicable. Whether you adopt them or not depends heavily on your specific circumstances.

Respecting Domain Knowledge

I've noticed a pattern in software development that very frequently there is a perception that code is code: if you hire somebody that has written code before, then they should be able to write code for you. It strikes me as being reasonably similar to hiring somebody that speaks English for a job that requires speaking English. Both a newscaster and a teacher talk as a primary part of their job, but they are certainly not interchangeable.

The same is true in software development. All tasks have a domain knowledge component. Sometimes it is large and sometimes it is small. The larger it is, the larger the risk of problems if someone without the requisite domain knowledge is assigned tasks in that domain.

The biggest risk associated with changes made by somebody who doesn't have the requisite amount of domain knowledge is that most domains have aspects which are counterintuitive. Code that looks "obviously" wrong can be completely correct and conversely code that seems right can be completely wrong. Changes to that code may seem to work because they pass all of the tests in the test suite.

On the other hand, sometimes people grow attached to code and feel that it is "their code" and that their value comes from their knowledge of and ability to update that code. The truth is, most value comes from doing new things. If you are known as easy to work with, knowledgeable, and skilled, people will want you on their team and you will get the rewards that come from contributing to success. It is good for domain knowledge to be shared as much as possible, while at the same time acknowledging that having domain knowledge is not the same as being a domain expert.

This doesn't mean that only domain experts should be changing code, only that it is useful to respect domain knowledge. Respecting domain knowledge means that if you get into some code that doesn't make sense to you, keep in mind that it may be a domain knowledge issue. At the very least, consider consulting with somebody that is more familiar with the code. Remember that just because you refactored the code and it still passes tests, it doesn't guarantee that everything is ok.

Anticipating Unintended Consequences

The so-called "Law of Unintended Consequences" states that any human action will always have at least one consequence that was not intended. The bad part of this is when the unintended consequence causes a problem. Another way to look at this is that each change will have an effect; one of the effects is the reason for making the change in the first place. All of the others can be classified in two different ways: expected or not and reasonably predictable or not. When making a change, there may be effects that are expected even though they were not the reason for making the change. As long as they are considered to be beneficial, that's probably ok. If there is any reason for not wanting the effect, even if

some observers might consider it beneficial, then it is probably best to think of the effect as non-beneficial.

The other classification is whether or not an effect can be reasonably predicted. Some effects might not be immediately obvious. As with chess, the effect of moving a queen into a capture square of a pawn is immediately obvious. Generally, it means that the queen is going to be captured. However, it might also be the case that the queen was moved there in order to guarantee checkmate in five moves. Unless the other player is looking ahead at all possibilities five moves in advance, there is a very good chance that they will miss this. Thus, the worst part of the law is unintended effects which are harmful and cannot be reasonably expected to be predicted.

Some changes have a non-linear effect on a system. That is, the change was very small – perhaps a single character was changed, but the impact was very large, such as destruction of large amounts of data. Because the amount of change often has no connection to the potential impact, it is always best to keep the amount of change to the absolute minimum required.

Even if a change could be reasonably expected to be predicted, sometimes the effort to do so is not invested. In any case, any effects that were not determined prior to incorporating the change are considered to be unforeseen consequences.

There are some strategies for minimizing the risk of unforeseen consequences. The first is to make a reasonable effort to determine effects other than just the intended one. The second is to do an impact analysis and either verify that the current requirements and code coverage are sufficiently high to mitigate the risk or to increase coverage to a reasonably high level.

Self-Organizing, Self-Managing, Self-Directing Teams

The terms “self-organizing,” “self-managing,” and “self-directing” have been used to mean many things, but are general thought of as being in the same ballpark. It is an ill-defined area of Agile development. Some folks have taken it to mean that there is no manager, that the Scrummaster is the manager, etc. The idea of self-managing teams is not new to Agile, it is written about as its own topic (see bibliography).

Most of what you need to do to become Agile requires only re-arrangement of what you are already doing. For instance, managing a backlog is still a process of deciding which work to do and in which order to do it, it isn't something that boggles the mind and makes people say "some people might be able to do that, but I don't see how we could." Managing what work to do with a backlog is more like tuning the engine rather than moving from steam to internal combustion. Managing a backlog is straight-forward to explain, learn, and implement.

I am neither for nor against self-managing teams. I see self-management as orthogonal to the core ideas and primary benefits of Agile development. It may well be that self-management provides a significant benefit, but it is something that can be applied to Agile teams or traditional teams or teams that are doing things other than software development. In general, self-management is one of the least well

understood areas of Agile development. Since it is not well described or understood and potentially one of the more disruptive practices, I suggest that it should be considered optional. Also, if you decide to try it, I highly recommend you either find an expert on self-management itself or get a book that focuses on self-management. Within the Agile literature, you will be lucky to find more than a few pages of material devoted to self-management.

That said, let's take a closer look at "management" and think about which aspects it might make sense for a manager that works in an Agile environment to delegate and which aspects it might make sense to retain or expand upon. There are many aspects of management. Here is a suggested breakdown into two groups: those which fit well into Agile and those which are more challenging.

Group one

- Strategic management
- Product direction
- Conflict management
- Hiring and firing
- Disciplinary action
- Leadership
- Motivation
- Budget process
- Removing obstacles
- Reviews
- Compensation
- Interaction with upper management

Group two

- Team-level product implementation
- Team-level process management
- Team-level project management
- Management of one's day-to-day work. That is, completing one's assigned tasks.

Agile development removes the need to "look over people's shoulders" and allows teams to optimize the parts of the process that have the most bearing on their productivity.

“No-Brainer” Self-Management Practices in Agile

While there may be some confusion over the meaning of self-management in Agile, not to mention some contention as to whether or not it is a good idea, many of the Agile practices contribute to self-management without requiring the wholesale adoption of self-management. For simplicity, I will define “self-management” as anything that is done by a team member that would traditionally be done by a manager. The practices contribute to self-management either by reducing the amount of management required or by encouraging the delegation of management tasks to team members. Some practices do both. As a result, managers have more bandwidth for dealing with things which are a more leveraged use of their time.

The Agile practices related to self-management include:

- whole teams
- collocation
- short iterations
- using a scrum master
- using a product owner
- user stories
- maintaining a product backlog
- assignment of tasks by team members
- estimation of tasks by team members
- producing a burn down chart
- retrospectives

Simplifying and Redistributing the Management Load

Let’s start with a simple example: task assignment. Traditionally, tasks are assigned to engineers by their manager. In Agile, tasks are selected by the engineers themselves. It may take some time for everybody to get used to this and to work out the kinks, but the practice itself is not particularly earth-shattering. Certainly, somebody may find themselves struggling with a task they don’t fully understand, or one engineer may accomplish a task more slowly than whoever the manager might have chosen to do the task. But is that really all that different than what typically happens when the work is assigned by a manager? How likely is it that a manager has enough time to devote to the minutia required to optimize task assignment? Wouldn’t it be better for the manager to spend their time mentoring team members on how to pick tasks that are appropriate for them while at the same time encouraging engineers to stretch their capabilities?

Shared Code Ownership

One way of interpreting “shared code ownership” is that the code belongs to everybody and that anybody can do anything at any time for any reason. In practice, this can lead to problems. If anybody can change any code at any time, people may end up changing things that really shouldn’t be changed. A better way to interpret “shared code ownership” is one of mutual respect and responsibility. If you are working on code that you are unfamiliar with, consult with others that are more familiar with it first. Perhaps there are comments, documentation, tests, or wiki pages on the functionality.

There are a number of benefits to shared code ownership. The first benefit is in reducing the risk that a key member is no longer available, whether they are on vacation, no longer with the company, or unavailable for any other reason. Shared code ownership also helps to make sure that all code is easily understandable, changeable, testable, and maintainable. It discourages knowledge hoarding and obfuscation.

One of the assumptions that makes shared code ownership work well is that there are adequate unit tests and other test coverage that reduce the risk of making changes to code. Keep this in mind as you consider this practice. Two practices that compliment shared code ownership are **pair programming** and **meritocracy**.

Pair Programming

Pair programming is one of the main techniques in Extreme Programming. It may not be for everyone, but there are many folks that swear by it. When practiced by two people that work well together and enjoy the practice, it is the best way for two people to cooperate on a project. Pair programming is an ultra-high bandwidth communication channel with ultra-low latency.

Perl One, Knit Two

In the early days of AccuRev, my friend Ted and I used to pair program all the time. We would often go on-site to customers to work on Perl scripts and we would only be provided with a single machine. I was the domain expert and Ted was the Perl guru.

Most of the time I would describe what needed to be done next and Ted would code it up. While he was coding I would sometimes find that I hadn’t described what was needed well enough and we would discuss the requirement while looking at the code itself. At other times I would find that I just couldn’t describe what was needed and so I would drive. Since I was not as strong in Perl as Ted, it was immensely useful to have Ted there to ask how to do this or that and have the answer immediately available.

One company that uses pair programming for all coding is Litle & Co.

Two of a Kind Beats an Ace

The director of software development at Litle & Co, David Tarbox, says “at first I thought I would hate pair programming, but I quickly came to enjoy it and depend on it.” Some of the side benefits that they see in pair programming are better on-boarding of new developers due to the mentoring and training that is inherent in pairing. They also like the fact that all code was developed by two people which provides both redundancy for knowledge of all code and real time code review. It may be that one of the developers for a piece of code is on vacation or otherwise unavailable when somebody needs to talk to them, but it is rarely the case that both are.

The fact that pair programming has not been as widely adopted as some of the other Agile techniques leads me to believe that it is unlikely to become a mainstream practice. If the success of a methodology depends on a technique which won't be adopted by all team members, then it is not possible to consider it as an integral part of a methodology designed for mainstream adoption. Thus, if you have team members who wish to use pair programming, by all means allow them to overlay it on your process as long as it is transparent to the process and functioning of the team overall.

Consider that you probably already do pair programming on a regular basis without thinking of it as pair programming. How often do you enlist the aid of a co-worker when debugging a difficult problem? When somebody is sitting next to you asking questions and giving suggestions as you debug a problem, that's pair programming. And how about on-the-spot code reviews? If you ask somebody to look over a section of code with you that you feel is particularly tricky, that's pair programming.

When using pair programming on a large, distributed project, don't try to simulate the conditions of having two people working side by side. Pair programming works best when the two people involved are physically next to each other.

One more consideration when using pair programming is the impact of compliance. A key fraud prevention control is the separation of duties, also called segregation of duties. This is a well known financial, management, and legislative control which is now being applied to software development. If you are considering the practice of pair programming and you are subject to compliance, you need to be careful how you go about it.

Meritocracy

It is often useful to look at an extreme version of an issue to discover a robust solution. In this case, an extreme example of **shared code ownership** would be a code base that is maintained by volunteers and that anybody in the world is welcome to volunteer for. The volunteers may or may not provide a resume, they may use a pseudonym, and you might only know them via e-mail. How can you operate in such a situation? This example should sound familiar. It is the Open Source Software (OSS) development model as practiced by OSS development projects such as Apache.

You don't have to release your software under an open-source license to benefit from the open-source development process. There is a great deal of benefit from adopting practices from this model for any development project. The open-source development model has been used by very large distributed projects, including Apache and Linux, with great success.

Apache handles this problem via a system that they characterize as a “meritocracy.” Basically, you start out with read-only access to the system. You can propose changes via patches, and if you propose a change which is approved, your credibility increases in the area that you contributed. Over time, you can earn write access to more and more of the system and can eventually become one of the people doing the approvals.

Meritocracy allows developers to contribute outside of their traditional areas, to build trust in their capabilities, and to allow them to naturally gravitate to the areas where they are the most effective.

An efficient way to implement meritocracy is to have developers implement their ideas on their own branches. If they are successful in implementing their idea and the result is accepted, it is then a simple matter to incorporate the change into the development process.

In a global environment where you are incorporating new team members into an existing project, this is an excellent way to mitigate risk. If you are using an SCM system, you can leverage the SCM system itself to facilitate this process. Instead of mailing around patches, proposed changes can be made on branches and then merged into the project when approved.

Stand Up Meetings

Stand-up meetings are another way to reduce delays in communicating important information. They can also be used to eliminate time-wasting status and progress meetings.

Stand up meetings are most closely associated with Scrum and are called “Daily Scrum Meetings” within Scrum, but have become popular independent of Scrum which is a good indicator that people find the stand up meeting to be a useful practice in and of itself. This is a good indicator of suitability for mainstream use.

A stand-up meeting is simple to implement. There are just a handful of guidelines:

- Limit the time to fifteen minutes.
- Pick a regular time for the team to meet, preferably in the morning.
- Start on-time regardless of who is absent.
- Each person answers these three questions:
 - What have you accomplished since the last meeting?

- What are you working on next?
- What impediments do you have?
- All discussion and problem solving is deferred until the end of the stand-up meeting.
- Follow-up as needed in smaller groups

Although it is called a stand-up meeting and standing is encouraged, the time limit is the most important part and standing is optional.

The point of a stand-up meeting is to improve communication and to discover and resolve impediments, not to have a meeting just for the sake of having a meeting. If the team feels that other practices make the stand-up meeting redundant, then by all means reduce their frequency or even discontinue use until such time as it appears to be necessary again.

To help make this decision, let's take a look at the expense side of stand-up meetings. First, people have to get to it. And then they have to get back to their computers. Scrum discusses how to minimize this time, but practically speaking, there is more overhead than just the ideal 15 minute meeting. If you are at a larger company, somebody has to book the room and let people know where it is. Let's call the cost of the meeting 20 minutes per person. If you have 12 people in a stand-up meeting, that's 4 person hours per day. That's the equivalent of half of a person.

Now let's take a hard look at the stand-up meeting itself. One of the basic ideas of Agile (and Lean) is continual self improvement. If the value of the meeting exceeds the cost, then there's no problem with the meetings, especially if they are eliminating other meetings. If the stand-up meeting is the only remaining meeting, that seems like a good thing. However, continuous improvement means we're never satisfied. Now that you are down to just the one meeting, you should still ask the question: "is it providing more value than the cost? Is there a better way?"

What is the purpose of a stand-up meeting? To quickly find out if people are making progress or if something is blocking them. If it is more efficient to do that via e-mail, IM, an issue tracking system, or other means, then use those means.

Or perhaps the stand-up meeting is needed because otherwise folks wouldn't complete their work, or people wouldn't speak up when they run into an impediment. In that case the stand-up meeting may be acting as a crutch. For instance, perhaps somebody isn't completing their work because they don't like it, but the constant peer pressure of the standup meeting is goading them into completing their work anyway. So then the real problem is lack of job satisfaction or low moral or something along those lines. Until you fix that problem, the stand-up meeting is just acting as a band-aid.

The real measure of project status and health is having an increment of shippable value at the end of every iteration. A standup will expose problems that are on people's minds, but the forcing function of the increment of shippable value is where you will get the true picture of how things are going. A one

month iteration interval is good, but if you can get it down to 2 weeks or even 1 week, that may do far more to expose more deeply rooted problems than a standup will.

Using 3x5 Cards to Capture User Stories, Plan Iterations, and Manage Backlogs

One of the impediments to adopting Agile development is that the supporting infrastructure is still heavily oriented towards traditional development. Most software tools were designed to fit into a framework of traditional development practices and have not yet adapted beyond superficial changes to meet the challenge of agile development. With the exception of a few tools such as Rally Dev, Version One, and Mingle there is little support for things like backlogs. As a result, many people use Excel or 3x5 cards to manage backlogs.

The important thing to keep in mind when using 3x5 cards is that the main benefit is not the use of 3x5 cards themselves, it is the benefit of being able to easily capture and manage user stories and to easily plan iterations and manage backlogs.

The whole reason we are in the software business in the first place is exactly so that people don't end up using things like 3x5 cards to manage their business. Imagine if you suggested to a user of whatever software that you produce that one of the new features they wanted would be implemented using 3x5 cards? What if you also suggested that they pay for that feature?

However, if you find the use of a software tool too constraining for capturing user stories, planning your iterations and managing your backlog, then by all means try 3x5 cards. Many people find them to be a useful way to very quickly get a full picture of what needs to be done and to plan for the future.

Keep in mind that by themselves, 3x5 cards have a number of disadvantages. Many or most of these disadvantages can be countered by using 3x5 cards as an intermediate representation of information.

The Walls are Alive

One of the companies that I work with is a leading workforce management company with a 600 person development team that started towards Agile development in 2004. Pretty soon, they had walls covered with 3x5 sticky notes. The airflow of people walking by the notes caused them to gently move which created a gentle rustling. It was almost like taking a walk in the woods.

They then transitioned to Excel and used macros to turn the spreadsheets into printed 3x5 cards for planning and then transcribed the results back into Excel. Once they had things working the way they wanted they created a set of requirements for selecting an Agile project management tool that they were comfortable using to replace their homegrown system.

The advantage of using a tool in conjunction with or instead of 3x5 cards is that you then have the ability to edit, search, categorize, re-organize, report, track, manage, distribute, share, and do backups on the

information. You can perform these actions from your computer, from a computer on the other side of the building, from home, or from a remote site.

The Magic of Gathering

The use of 3x5 cards reminds me of using 3x5 cards to keep track of role-playing characters when I was a kid. The whole time I was using them I kept wishing that they were on my computer. And I did actually try that, but storing the info on tape cassette and trying to find the exact place on tape was just too much of a manual process. These days, role-playing games are on-line, in 3D, and globally distributed. Things change fast.

Outsourcing

It is rare to be able to find the talent that you need all in one place these days, whether you are using outsourcing or not. In the end this really comes down to finding the right skills to do the job that you need to do in a cost-effective manner.

When managed well, outsourcing has many advantages. It can reduce costs, provide a means for scaling up and scaling down capacity, and provide another source for finding the talents you need.

The trick is finding the right outsourcing team and the management on both sides which has experience with the special challenges that come with distance. When you have an external team, any process issues that you have are magnified until you fix them. Of course, the flip side of this is that it gives you an opportunity to find and remove or fine-tune practices that don't scale well. These process changes can then have benefits when applied to your internal teams. At a high level, the biggest area of focus should be keeping your internal and external teams in sync. That means keeping in constant contact about plans, progress, status, and expectations and being very candid in both directions.

Process Improvement: Infrastructure Considerations

Use Off The Shelf Automation

In the field of software development, there is a great deal of automation available for the process of software development itself. These include: requirements management tools, modeling tools, issue tracking tools, source control tools, build tools, continuous integration tools, compilers, static code analyzers, etc. Take a look at your software development process from top to bottom. For all of the steps which are manual or use homegrown scripts, consider whether or not there is an automated solution available.

Project Management, Defect Tracking, and Requests for Enhancement

Do you ever feel like things are out of control on your project, that you are adrift in a sea of conflicting priorities and requests? Do you suddenly find out at the last minute that you are the bottleneck and everybody is breathing down your neck asking you what is taking you so long to create the `moveStuff()` method but you had no idea that anybody even cared about `moveStuff()` or that you owned it? Do you ever find yourself in the exact opposite position, wondering why Sue and Bob didn't get their stuff done that you need and then your boss walks by while you are surfing the net waiting for Sue and Bob? And who is Bob anyway?

The solution is simple! All you need to do is get everybody to move to Project. Well, if you have somebody you can spare full-time to keep Project up to date of course. Oh and I almost forgot, you'll need to start using a requirements tool. But that's it really, other than integrating them all together over the weekend and of course that's assuming you've already got a CRM tool for workflow.

There is a simpler solution. It isn't perfect, and it doesn't solve all problems, but it can definitely provide the following benefits:

- reduce the chaos
- increase your vision into where you are and what's going on
- reduce the number of status and/or project management meetings
- reduce the need to provide the same information over and over again
- simplify collaboration both locally and for distributed teams

The answer is to reduce the amount of rework that you are already doing. Right now you are probably storing defects in a defect tracking system, enhancements (aka RFEs, requirements, etc) in a requirements management tool (usually Excel or Word but sometimes an actual RM tool), and if you are using a project management tool it is probably MS-Project. All three of these product areas evolved to provide different aspects of project management for different groups of people and as a result they have lots of overlap. Considering how hard it is to coordinate three different systems, why not consider standardizing on one system for most of the work? The only question is, which system?

If we are going to try to do most of our project management work in a single tool, we should first decide what the interesting activities are. I believe they are: recording enhancement requests and defects as they are gathered by marketing or reported by users, load balancing, estimated completion calculation, critical path determination, work assignment, workflow, and reporting.

First let's consider how well suited that Project is for doing most or all of these tasks. Project is good at taking a small static collection of large tasks and doing load balancing, estimated completion, and critical path determination. Thus, it is mostly used for the very narrow task of project management of enhancements.

Next let's consider requirements management. For whatever reason, most people use Excel or Word as their requirements management tool instead of a "real" requirements management tool. Excel and Word are just not appropriate for project management.

Lastly, there is defect tracking. A defect tracking system covers the assignment, tracking, workflow and reporting of defects. There is usually a higher volume of defects than enhancements, and they are usually smaller in scope and have a more complicated and often more time critical lifecycle. If it works well for defects, it should work equally well for enhancements.

Based on this analysis, it makes sense to extend the project management that you are already doing with a defect tracking system to include enhancements. A generic name for something that is either a requirement, enhancement, or defect is "work item." By using work items to track all work, it is easy to see where you are and what remains to be done. For instance, if you use the system to track what stage work items are in, you can easily run a query to see which work items have their code written but do not yet have any tests. Similarly, you can see which work items are done from a coding perspective and have tests but have not yet been verified as done by QA. This will give you a much more complete view of your overall project status and progress.

While Project has much more functionality for making sure that the work that needs to be done is well distributed and managing the critical path, it is a lot of effort to keep Project up to date and very few companies even try to keep Project up to date.

When all work items are tracked together and are kept in synch with development progress, you can use it for many of the tasks that are usually done using a project management system. The data in the ITS will also be more accurate and timely since it is kept up to date by the people doing the work. As an added bonus, since all information is readily available in the work item tracking system, you can eliminate those status meetings that exist for the sole purpose of updating project plans.

Overall, an issue tracking system serves as a facilitator. It simplifies the collection of defect reports and requests for enhancement from all sources. It isn't just the developers responsible for fixing the defects that find problems. Customers, developers working on dependent systems, and testers also find defects. Even if you have a policy of fixing defects as soon as they are found, it isn't always logistically possible to do so. For instance, if you are currently working on fixing a defect and in the process of doing so you find another one, you don't want to lose track of it. An issue tracking system coordinates the collection of

defect reports in a standard way and collects them in a well known location, insuring that important information about the functioning of your system is not lost.

An issue-tracking system also manages the progress of work through the development life cycle from recording, to triaging, to assignment, to test development, to completion, to test, to integration, to delivery. It simplifies the answering of customer questions such as “what is fixed in this release” and “what release will the fix appear in.” A defect tracking system also allows for the collection of metrics which aids in the spotting of trends.

I have heard from multiple sources that metrics collected from an ITS are worthless because developers will just game the system. That may be true in an unhealthy environment. However, in an environment where developers are actively participating in the improvement of the process, they will want this information in order to help to find and fix process problems, including the root cause of specific problems. Discovering trends in order to fix problems is a crucial part of root cause analysis.

For requirements and enhancements, the entire description does not need to be in the ITS’ fields directly. However, there should be at least a high-level description in the ITS in order to help tie requirements directly to code changes. If there are sub-requirements, these can be entered as sub-tasks.

Though it can be tempting to do so, there is no need to put every field imaginable into your ITS system. You should always strive to keep the total number of fields and the number of required fields to an absolute minimum. The less that people need to fill out, the less likely that using the ITS will be seen as a waste of time and the more likely it is that people will see it as an aid.

Tracking all work in an issue tracking system also simplifies compliance and auditing efforts. This may not be a very motivating reason, but if you are already using an ITS to track all work and status, you get the compliance support for free.

All Electronic Artifacts

All artifacts of your software development process should be fully electronic. Paper is hard to backup and store off site. If you want a truly robust software development process, all artifacts and documentation should be stored in electronic form, backed up, and stored in a secure offsite backup.

Paper is also difficult to search, duplicate, organize and distribute. Even if there is very useful information written down that somebody somewhere could use, if it is on paper the only people that can benefit from it are the people that already know it is there. Even if you are the author, if you are working from home, in a coffee shop, on an airplane, or at a customer site and the information you need is written on a slip of paper that is on a wall at work, it is going to be very difficult to access it.

To top it off, information stored on paper is difficult to sort, re-sort, re-arrange, graph, chart, and analyze. In today’s world the amount of useful technology that you can apply to data is truly staggering. The amount of useful technology that you can apply to paper is essentially the same as it was 3,000

years ago with the exception of turning it into digital data. And of course paper was a great leap forward from stone tablets which were much more difficult to use and distribute than paper.

Considering that the fundamental purpose of software development is the automation of manual labor, it would seem to me that the use of paper in the development of software should be avoided at all costs. It is sort of like saying to our customers “oh sure, automation is great if you like that kind of thing, but we prefer good old tried and true manual labor.”

Version Everything

Since everything is stored in electronic form, i.e. in files on disk, it should be a simple step to version everything. The advantage of versioning everything is pretty obvious: you get all of the advantages of version control for all of your documents.

People often think of a source control system as being just for software source files, probably due to the name and the early history of source control. A simple way to fix one of those problems is to just refer to source control as version control. Another reason for only versioning software source files is that they are text and older version control software was notoriously bad at versioning non-text files, also known as binary files. If your current version control system has problems with binary files, it is not a modern version control system.

Store all SDLC artifacts in your version control repository. Having all of your SDLC artifacts in one place simplifies coordination and reproducibility, enhances workflow, and provides more opportunities for safely checkpointing work.

The SDLC artifacts to consider include:

- Requirement docs
- Design documents
- User stories
- Test plans
- Test cases

Atomic Transactions

Imagine that somebody on a remote team is checking-in a large batch of changes over a slow and flaky connection. If you update your work area during this time and those changes affect you, there's a very good chance that post-update your work area either won't build or the resulting executables won't work properly. Of course, you can update your work area again, but how do you know when to stop? Just waiting for an empty update could take a while, and is no guarantee that you have a cohesive set of changes.

Most of the newer SCM systems available today feature atomic transactions. This feature has many benefits for collocated development, but should be considered a minimum requirement for global development.

Atomic transactions either complete or fail; there's no in-between. This has the added benefit of grouping operations on multiple objects into a logical unit. Now, if there is a long-running check-in in progress, you are protected. When you update your work area, it will be updated to a particular transaction level. Only the transactions in the system up to the time you start your update will be brought into your work area.

Infrastructure Support For One Virtual Site

Use the same tools for all teams and team members that are working on the same project to remove the need for information migration and translation. By using the same tools you will be using a shared culture and a shared means of project status communication. When you use different tools you are introducing a communication barrier which will reinforce and increase the distance between teams. The most essential tools to share are your SCM, issue tracking, build, wiki, and project management tools. If you aren't already, you should also consider using discussion forums instead of e-mail for discussing development topics. It is a much better way to keep related information together. Many wikis include forums, and some include the ability to link wiki documents, discussions, and work items.

Asynchronous Coordination

Another important characteristic of good software development tools is that they provide the ability to reduce the dependence on things happening at or near the same time. When things have to happen at or near the same time, it can cause interruptions and delays. It is much harder to coordinate actions so that they all occur at the same time anyway. Development tools can provide the same sort of convenience and continuity that we are used to in other areas. For instance, store-and-forward message queues, Digital Video Recorders, voice mail, and e-mail. These are all methods for removing the need for same-time synchronization and coordination. In software development, all of the tools mentioned in the previous section also facilitate asynchronous coordination.

All modern software development tools are well equipped to work efficiently over the internet. If you have a tool in your development stack that is not easy to use over the internet, consider replacing it. A good rule of thumb is to use tools which use TCP/IP natively and do not rely on shared network drives. Software tools that are accessible via the web are also useful. Not all development tools have fully-featured web interfaces, but when they do it makes it easier to link information together.

There are many advantages to having a modern tool stack. You can:

- “Smooth out” time zone differences
- Reduce the need for face-to-face interaction
- Centralize information and interaction
- Greatly simplify the integration of work
- Provide metrics on progress regardless of location

Process Improvement: Advanced Techniques

Decoupling Iteration Activities From the Iterations

There is a general software engineering principle that can be applied to Agile: decoupling. That is, separating two or more things which are currently coupled together but don't need to be. This is considered an advanced technique because when first learning Agile it is best to link certain activities to the iteration cadence. These activities are: iteration planning, having a shippable increment of work, the size of the largest story, iteration reviews, retrospectives, and releases. If you have been practicing Agile for a while, it is likely that you have already started to decouple some activities from the iterations.

Decoupling Iteration Meetings and Story Point Estimation

Many Agile teams have a weekly meeting to estimate story points for stories that have made it near the top of the backlog and don't yet have story point estimates. This decouples story point estimation from both the iterations and the iteration planning meeting itself. How far down into the backlog you go depends on what goal you are trying to achieve. If you just want to simplify iteration planning, you only need enough stories to cover one iteration worth of stories and perhaps a bit more just in case some stories aren't chosen for that iteration.

Decoupling Retrospectives

When just starting with Agile, it is much harder to use 1 week iterations than 4 week iterations. If you are doing 4 week iterations, you would naturally do a retrospective once every four weeks. If you move to 2 week iterations, you would naturally do a retrospective every two weeks.

But why not do retrospectives every week or at least every two weeks, regardless of whether you are doing 4 week iterations or two week iterations? Shouldn't the cadence of retrospectives match the cadence of their usefulness? In my experience, there is always something worth talking about every week of any project. There is always something to reflect on and improve. In any case, instead of just scheduling a retrospective at the same cadence as your iterations, schedule the retrospectives at a cadence that makes sense for your needs.

Decoupling Iteration Reviews

Another practice to consider decoupling from your iteration cadence is iteration reviews. Let's say you have 1 week iteration reviews, but it is logistically difficult to schedule iteration reviews involving multiple customers more often than once per month. Perhaps you normally have both internal and external stakeholders at the reviews. One solution would be to continue to have weekly reviews with just the internal stakeholders that are interested in being there every week, and have monthly reviews for external stakeholders. That may actually require a bit more work on your part, but if the value is there, why not consider it?

Simplification and Delegation

Another benefit of decoupling is that by breaking things apart it makes it simpler to tackle or delegate process improvement efforts because you can concentrate on smaller parts which are running at their

own cadence. In the example of the iteration reviews, by separating out the external stakeholder meeting, you can now focus on ways to simplify or delegate that part of the process. Perhaps the only problem is getting a web meeting solution in place. Now you can solve that problem without impacting the internal stakeholders.

Unfortunately, not everything can be decoupled. For instance, it would be unwise to release anything other than a completed iteration. Releases do depend on the iteration being complete, so that can't be decoupled.

Appendix A – Example Process Document

One of the best places to keep your process document is on an internal Wiki.

Process Document for Acme Widgets Inc.

This document covers the process used for developing our online calculator application [link]. Please note that the various tools that you will need have installation instructions on the various pages that this document links to.

Requirements Gathering

All requirements are gathered by the product manager and entered into our issue tracking system [link]. If you have a question about any requirement, please contact [link]. Please note that while we call these requirements, we are using the concept of user stories. But folks didn't want to change the terminology, so we still call them requirements (at least for now).

If you have a great idea for a product feature, please enter it into the issue tracking system and set the state of "suggestion." The product manager regularly reviews these and discusses them with the user community. Historically, many of these suggestions have been implemented.

Iteration Planning

The product manager determines the proposed content of each iteration. The next iteration plan can be accessed from the issue tracking main page. Please feel free to peruse this and put comments into the feedback section of each issue. You can volunteer for any issue at any time.

At the beginning of an iteration, there is a planning session. In the planning session, estimates are given for each proposed issue. The product manager and development team then work together to come up with a plan that will fit within the iteration. The team then works to implement the plan.

Test Plan

For each work item in the iteration plan, there must be a "test plan." The test plan is simply the test cases which are required to conform to our quality standard [link] . The test plan is kept in the test plan field of the work item. Development and QA work together to determine who will implement the various test cases. Test cases should be automated unless it is impractical to do so. Test cases may be implemented by developers or QA folks, but the completeness of the test plan is the responsibility of QA.

Development

Once the initial test plan is created, development may start. All of the test cases must be implemented and passing prior to check-in. The developer should update the test plan as they do the development to reflect anything that they learned during development. Once the initial test plan has been executed with zero failures and the code is checked-in, the developer then transitions the work item to "complete."

Integration

When the work is checked-in, the Continuous Integration system will automatically build and test the changes. If there are any problems, you will get an e-mail with the problems that you need to address. Fixing this takes priority over any other tasks.

Exploratory Testing and Verification

Once a work item is marked complete, then QA will do some exploratory testing. After the exploratory testing phase, QA updates the test plan. Development and QA then discuss the updated test plan and adjust it to create the “official” test plan. The content is the responsibility of QA. This plan may change again during the remainder of the iteration, but this is the first official version. Any test cases added since the initial version must now be implemented and pass. Implementing these and making code changes to get them to pass takes priority over any new work. Once this is done, QA transitions the work item to “verified.”

Iteration Completion

At the end of the iteration, all unstarted work is moved back to the backlog. All work in progress that has not yet been integrated is moved back to the backlog. The next steps for all other work is determined on a case-by-case basis. All work that is still unfinished must be brought to a “verified” status in order for the iteration to be closed.

Release Process

Once an iteration is closed, the next step is either to release the product or move to the next iteration. The release process is kept in a separate document here [\[link\]](#).

Appendix B – Case Studies

AccuRev's Transition to Agile

It is hard to imagine a company more devoted to traditional development than a process and tools vendor that makes its bread and butter selling to other companies that are devoted to process and tools! Converting AccuRev from a die-hard process and tools traditional software development company with a total aversion to Agile to a champion of Agile development took three years. Resistance and skepticism were understandably very high throughout the company, from upper management to the trenches.

In 2005, while doing an internal presentation on Agile methods, I classified them as a passing fad that was only for small undisciplined teams with simple projects. Nobody objected, and most people nodded their heads in agreement.

The first person I had to convince that Agile was a good idea was myself, and I wasn't looking to be convinced. As far as I was concerned, traditional development methods were just fine. Ok, so many companies complain about predictability, quality, visibility, and rework but that's just the cost of doing business, right? How can 3x5 cards and having one person sit on their hands while another person types be better? As I describe in the introduction, I eventually saw the light, but then I had my work cut out for me to transform the rest of the company.

The first stage was to convince the rest of the management team. The first convincing argument was that many of our most successful customers, including the ones with large distributed teams, were using Agile. The turning point came when we had a very tight deadline to finish a project for submission to the Jolt Awards and I used that project to pilot Agile internally. We produced a whole new product in just 7 months that was innovative enough to win the Jolt Product Excellence award.

The Development of AccuWorkflow at AccuRev

AccuWorkflow was conceived of in April of 2006. At around the same time we were thinking about what to enter for the 2007 Jolt Awards. We decided that AccuWorkflow would be a big part of that but then we also quickly realized that if we wanted to have a shot at the Jolt Award for 2007, we had to go from PowerPoint to shipping product in just 7 months. Using Agile development seemed like a good idea.

For the development of AccuWorkflow a couple of the aspects of practices were particularly useful: doing regular demos to stakeholders and meritocracy. In the first months of AccuWorkflow development, support for the project was shaky. The regular demos of significant new functionality helped to show that it could be done and to build excitement and support for the project. Not only were we able to get feedback internally, we also used the early demos to get feedback from prospective and current customers which further built support for the project internally.

About halfway into the project it turned out that the original estimates were too low for what we wanted to do. We had already trimmed and trimmed the project, so there was really nothing more that could be trimmed and still have something Jolt-worthy. In addition, it was felt that we needed at least

one more killer feature. We needed more resources. But the commitment of resources to projects was set, we couldn't get committed resources from other projects. So we used the volunteer aspect of the Meritocracy practice.

Anybody that was interested in helping out in their free time was welcome to take a look at the work that needed to be done and pitch in. The benefit to a volunteer was that they could do something that they considered fun, contribute to a good cause, and get experience in new areas. It worked. We got volunteers from all over the company and all departments. We even got contributions of project discretionary time from managers.

I had the partial use of multiple QA resources over time, so I had a "virtual QA person" which was different people at different times including developers doing some QA tasks.

For AccuWorkflow we were doing variable length iterations where the iterations were between 2 days and a week, so we ran into this hand-off period on a regular basis. The way we handled it was to create a branch for the "hand-off" period.

As an iteration ended, we created a new branch based on the existing development stream. The new branch then represented a stable version of the "previous iteration." If we needed to do a little follow-on work, for instance to get things ready for a demo, we used another configuration called "demo" based on the "previous iteration" branch. To prepare for the demo we made changes in the demo branch without affecting the previous iteration or the current one. Once we finished the demo we then merged any changes into the current branch.

Having branches representing multiple iterations also made it possible to easily find out what had changed between iterations which aided in tracking down problems.

Everybody at AccuRev contributed in one way or another to winning the award, either through working on the product, testing, marketing, or just the simple act of believing that we could win and offering encouragement and support.

Our hard work and the use of Agile development paid off. In December of 2006 we shipped the first version of AccuWorkflow. In January we were announced as a finalist and in March of 2007 we won the Jolt Product Excellence award for AccuRev 4.5 with AccuWorkflow. I am 100% sure that the primary reason that the version of the product we submitted for consideration was ready on time with high quality was Agile development. As I accepted the award I realized I was hooked and there was no going back.

Litle & Co.

Litle & Co. is a leading provider of card-not-present transaction processing, payment management and merchant services for businesses that sell directly to consumers through Internet and Multichannel Retail, Direct Response (including print, radio and television) and Online Services. They interface with the world's major card and alternative payment networks, such as Visa and Master Card, and PayPal, and must be available 24/7 with no downtime.

During the founding of the company in 2001, the initial team of six developers wanted to use Extreme Programming (XP), an Agile development methodology. Some of the developers had used it before and the rest had read about it and liked the ideas. When they talked to Tim Litle, the founder and Chairman, about using Extreme Programming (XP) they told him "you'll have to accept that we won't be able to tell you exactly what we will deliver in 9 months." Tim said, "That's fine with me, I've never gotten that anyway!" He agreed not only to use it, but to insist on it as the company grew and brought in additional management. He liked the idea that he could change his mind about prioritizing feature development for merchants without cancelling or shelving work in progress.

Six years later, in 2006, Litle & Co. landed at No. 1 on the Inc. 500 list with \$34.8 million in 2005 revenue and three-year growth of 5,629.1 percent. In 2007 Inc. magazine cited Litle & Co.'s 2006 revenue as \$60.2 million, representing a three-year growth rate of 897.6% over its 2003 revenue of \$6.0 million. How has Litle achieved these impressive results? One factor that they site is their use of Agile development.

Litle uses many of the XP practices including pair programming. The director of software development, David Tarbox, said "at first I thought I would hate pair programming, but I quickly came to enjoy it and depend on it." Some of the side benefits that they see in pair programming are better on-boarding of new developers due to the mentoring and training that is inherent in pairing. They also like the fact that all code was developed by two people which provides both redundancy for knowledge of all code and real time code review. It may be that one of the developers for a piece of code is on vacation or otherwise unavailable when somebody needs to talk to them, but it is rarely the case that both are.

They started with weekly iterations then moved to 2 weeks then 3 and are now monthly. They gravitated to a month because it was the easiest cadence for everybody to work with and it meant that the whole company was able to adopt that same cadence. Agile development is part of the psyche of the whole organization. It permeates every aspect of the organization.

One thing that other areas of the organization had to get used to was that they had to be very clear about what they wanted from the development organization because there is no slack time. If you want something and you expect to have it done in the next iteration (monthly cycle), you have to be able to state it very clearly or it won't even get started. It also means that development is very closely connected to the corporate strategy. For every task, developers ask "is this aligned with our overall direction?"

Agile has had a very positive effect on their hiring. Developers like the fact that Agile provides the additional challenge of solving business problems instead of just technical problems which requires thinking at a higher level. Developers at Litle report that they have a higher level of job satisfaction than in previous companies that were not using Agile because they see the results of their software development efforts installed into production every month. Also, they like the fact that there is much less work which amounts to “implement this spec.”

Litle’s software is hosted at multiple off-site datacenters and they upgrade their software every month like clockwork. In reality, they are doing ~7 week iterations which overlap such that they are releasing monthly. The first week of the development cycle is planning, the next four are development, and the final two are production acceptance testing (PAT) and deployment. Each iteration starts on the first Monday of the month. Some iterations are 4 weeks and others are 5 weeks, but there are a total of 12 iterations a year.

Although updating such a mission-critical application might seem risky to do every month, a lot of the perceived risk comes from the typical lack of fully-automated testing. Litle has a very large body of automated tests (over 30,000) and also has a test system which simulates the usage patterns and load of organizations like Visa and Master Card. Every change is subjected to the same sort of testing that typically takes traditional development organizations multiple person years of effort to achieve. As a result, the quality of their monthly upgrades is unusually high even compared to traditional product release cycles of six months to a year or more.

Their development teams are entirely collocated with no offshore or offsite development teams. For each project they assign a team consisting of multiple pair programming teams. Over time they have grown to a development team of 35 including QA and a codebase that contains 50,000 files including test cases and have had to add practices in order to scale XP. The biggest problem they ran into as they have grown was integration of newly developed code into the existing codebase. To address this, in 2007 they added the practice of Multi-Stage Frequent Integration to their implementation of XP. They do frequent integration instead of continuous integration because a full build/test cycle takes 5 hours.

Prior to implementing Multi-Stage Frequent Integration, they would have to manually pore over all build and test failures to determine which change caused which failures. This was done by very senior developers that were familiar with all aspects of the system to be able to understand complex interactions between unrelated components of the system.

Using Multi-Stage Frequent Integration, each project works against their own team branch, merging changes from the mainline into their team branch every day or two, and doing their own build/test/cycle with just that team’s changes.

Thus, any failure must be a result of one of their changes. When the build/test cycle is successful, the team then merges their changes into the mainline. As a result, any team-specific problems are identified and removed prior to mainline integration and the only problems that arise in the mainline are those that are due to interactions between changes from multiple teams. The isolation also simplifies the job

of figuring out what changes may have caused a problem because they only have to look at changes made by a single team, not the changes made by all developers.

Tests and code are written in parallel for each work item as that work item is started. They do not distinguish between enhancement requests and defects. All assigned work is tracked in their issue tracking system and all work that developers do is associated with a work item in the issue tracking system. This is done to automate compliance as they need to pass biannual SAS70 Type 2 audits and annual Payment Card Industry (PCI) audits.

The PCI audits go smoothly. According to Dave: “You have to do them yearly, we’ve been doing them for 5 years. We’ve never had a problem. If you’re doing development the right way, you’re not going to have a problem with audits, because all they’re checking is that you follow the best practices of software development. You must use source code control, track why changes are made, ensure that the work done aligns with the company’s goals, and that your app meets data security requirements. On the auditors’ first visit, they said passing it was going to be a big problem because of our size (they figured, ‘small’, meant insufficient), and that we would likely have to make a lot of changes to pass. When the audit was complete--the auditors were surprised. They said they had never seen anyone do this good of a job the first time out.”

Dave also sees their long track record of successful use of Agile as a competitive advantage: “Because we have our monthly releases, our sales people are able to work with our product people to adjust the company priorities for what makes the most sense for the business. Rather than having to pick everything we need to fit in for the year now and then get together again in a year, it becomes a regular monthly meeting where they look at the priorities together. So if there is a vertical we’re not in yet or there’s a vertical that we are trying to expand in or there’s something that makes sense for the business, if we decide as a company we should go after that business, we bid on it.”

The Iterative Design of a Transforming Lego Sports Car

In the lead-up to the debut of the movie “Transformers,” there was already plenty of Transformers merchandise available. My son, who was 2 ½ years old at the time was instantly entranced. While my wife was strolling him through the bookstore he grabbed a Transformers calendar without her noticing. When she got to the register he then used his charms to convince her to buy it.



The Challenge

This got me to thinking. Wouldn't it be fun to build my son a car out of Legos that could transform into a robot? After all, how hard could it be? I decided to give it a shot and I set myself the following design goals and constraints:

1. Build a Lego model that has two forms: a sports car and a robot
2. Folks looking at it agree that each form is clearly recognizable as that form
3. On casual inspection, it doesn't look like it transforms
4. 100% Lego, no other parts, and no glue
5. Easily held in one hand
6. Strong (parts don't fall off easily during use)
7. Robot form can stand on its own
8. The model can easily transform from one form to the other
9. Model is completely connected and transformation does not require the addition or removal of any parts
10. Fully articulated

Now, a year and several generations of Lego models later, I've finally created a Lego model which meets these deceptively simple goals.

As the project progressed, I wasn't thinking of it as an exercise in iterative design or a formal project, it was always just something I did for fun in my spare time. I didn't realize the link to iterative design until I started talking to people about the history of the project.

Transforming Lego Car, Iteration 1

The first attempt to build the transforming Lego car didn't go very well. I was able to create a reasonable looking car which had lots of interior space for the mechanics needed to transform, but creating the sliding parts and the joints proved to be more difficult than I had imagined. I had lots of different hinges, flat parts, special joints, and Technic parts, but all of the possibilities I came up with were either too big, too fragile, too loose, or didn't support articulation well enough. I was stumped. After a quick trip to the nearby Lego store to see if there was something I had missed, I reluctantly decided to shelve the project.

From a requirements perspective, I had satisfied requirements 2-6, which is half of the requirements. Of course, it was easy to satisfy "doesn't look like it transforms" because it didn't transform at all. :-) From an Agile perspective, the car was fully functional as a car in its own right, and I had discovered lots of valuable information that would help with later iterations of the car.



Little did I know that a hidden bias had kept me from seeing what was right in front of me. It wasn't until a couple of months later that serendipity broke through that bias and brought together the necessary ingredients for the first transforming version of the car.

At the beginning of the project, I had the exact requirements and they didn't change at all from the start of the project to the end. However, just having the right requirements did not mean that I could immediately design something that would meet the requirements. I had a Lego car that met many of the basic requirements, but still didn't transform. I was stuck because I didn't have the technology that would meet the rest of the requirements.

Part of iterative design is keeping your eyes open, always looking for ways to intelligently expand the solution space. If the information you need is not already at hand, trying variations of things you already know may not lead to a good solution.

Thinking Outside The Box

I had always eschewed the Bionicle Legos as "not really Lego," so I hadn't originally considered those. But then two things happened that breathed new life into my Lego project. My son's third birthday was coming up, and my wife suggested that we make it a Lego birthday. I was responsible for filling the gift boxes for the guests, and I decided to start by visiting the Lego store to see what they might have.

The Lego store had a wide variety of gifts for \$5 or less including Duplo dinosaurs, a pony and princess set, knights on horses, and most importantly (for the project) small Bionicle sets. I had never bought a

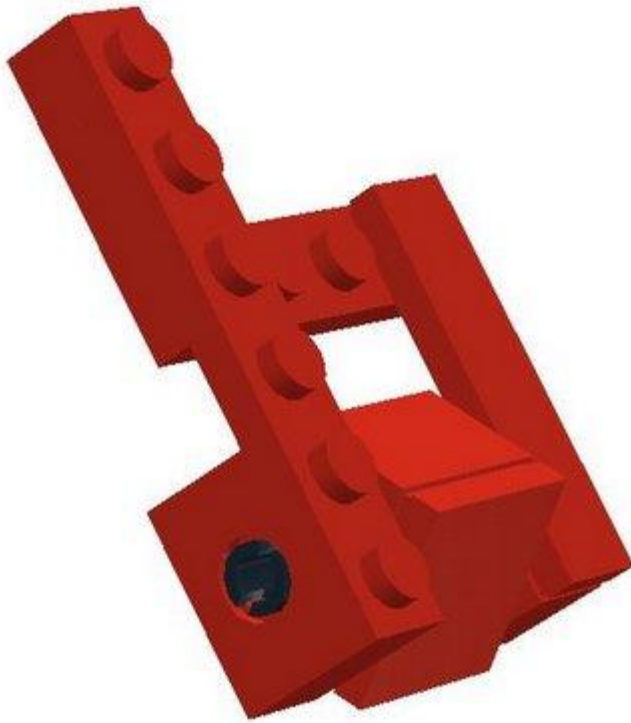
Bionicle set before, so I had never noticed that many of them had articulated joints. The joints were ball-and-socket parts which were small and had enough friction to make them poseable. They were perfect for the project.

There is another kind of Lego set called Exo-Force which has a completely different technology for doing articulated joints and also has a part which when combined with a Bionicle part provides a third technology for articulated joints.

Creating limbs which can fully fold up is a difficult proposition at best. But, I managed to create a couple of variations that allowed me to create the next version of the project. It didn't really meet the goals of looking like a sports car or hiding the fact that there was more to it than met the eye. It was quite boxy, had odd proportions, and some of the joints were exposed. It was also fairly fragile. But, it could transform!

Sometimes Flexibility is Constraining

Sometimes software that is too generic or too flexible creates another set of problems such as taking too many resources or being too difficult to configure for the task at hand. The knee and elbow joints of the robot form had the same problem. Because they were fully articulated they were just too darn big and difficult to hide. I knew that these joints didn't need to have the same degrees of freedom as the shoulder, hip, and ankle joints, but had not previously been able to create a hinge that could fold flat and have the full range of motion required. I repeated the exercise of determining all of the available hinge options and found a couple of new ones. The most promising one was a dual Technic hinge which was perfect for the knees.



For the arms, this hinge was too large. But then I realized that less resistance was required for the arms and a single hinge would do. Perfect! Now I had the robot skeleton, I just needed the car exterior.

Inspired by Elegant Design

Many months later, when looking for a new Lego set for my son, I saw a new car set (#4939) that was perfect for the project. I'm not an artist by any means, so one of the challenges has been finding Lego car designs to emulate. Previously they had always been either too big or too small. At one point I had been very excited about the Ferrari that comes with both red and yellow exterior pieces, but it was double the size I needed. Set #4939 seemed to be the perfect size. It didn't hurt that it was very cool looking and reasonably priced. I had the feeling that this was it, I was finally going to achieve the goal.

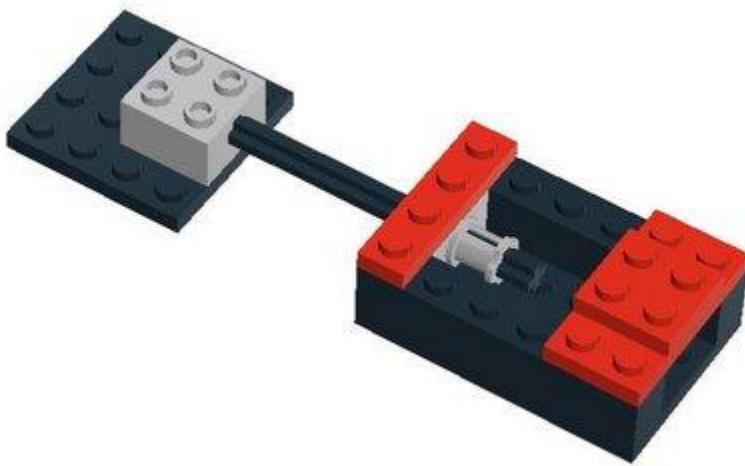
Using set #4939 as inspiration, I was able to layer a car exterior onto the robot skeleton without too much trouble. As a bonus, the head, which had previously never really worked out well, was just there. I was within reach of the goal. But still one problem remained. I couldn't cover the shoulder and hip joints without preventing transforming. I needed something that would allow the top and bottom parts to slide out and then lock in place. I had created parts that slid from place to place before for other Lego projects, but never something that could also lock in place.

Sliding Into Home

After trying many variations using small hinges to act as the locks, I had something that worked, but it was just too big. It was also very clumsy. Luckily, I was in denial. I forged ahead anyway. I wasn't sure what to do about it being clumsy, but I was determined to create more room in the model. The most

obvious step was to do what they do to create stretch limousines: take a regular limo and stretch it out. I made the whole model a bit longer, wider, and taller while keeping the same overall proportions. I also removed all unnecessary internal parts or replaced them with parts that would still do the job but took up less space.

It wasn't enough, and I couldn't stay in denial about the sliding part being clumsy forever. But while I was stretching things out I observed that some of the technic parts would slide around a little bit when I exerted too much pressure on them. Of course, the part that would give depended on the amount of friction. Whichever had less friction was what moved and the other one did not. From that came the solution: two bricks together to anchor the axle, and one brick that moved freely but with enough friction that it would stay wherever it was slid to.



The slider was the last major challenge. Once I had this part, it was just a matter of trial-and-error shifting things around until it all worked. There are two sliders; one for the lower body, and one for the upper body, the following picture shows the car with the upper and lower body pulled out as the first steps of transforming.



After the car has been extended, the legs are unfolded.



Lastly, the arms are extended. The head is also fully articulated.



There is still room for improvement, but at this point I think it is more a matter of aesthetics than mechanical design. I'm happy with the current design and hope that perhaps a Lego fan with more art skills than I will take the design to the next step. My other hope is that perhaps the amazing folks at Lego will start producing transforming Lego sets. And of course, if they were Transformer branded, that would be a wonderful (but not necessary) bonus.

Motivation, skill, and experience all play important roles in the design process, but they are no guarantee that you are going to be able to design, let alone build what you have set out to do. It almost always requires trial-and-error, serendipity, creative inspiration, research, and an open mind.

Developing a User Interface: Traditional Development vs Agile Development

AccuRev's first interface used Tcl/Tk. We quickly realized that it wasn't going to meet our long term needs, so we decided to switch to Java and Swing. Instead of using Agile development techniques, we set out to build the entire functionality in a single release. The first version of AccuRev's Java GUI took a year to build.

Within two months of the introduction of the Java GUI, our sales increased dramatically. It wasn't because of the functionality in general, and it wasn't because of the price (we substantially increased the price). It was simply because a new feature made AccuRev's value proposition immediately apparent whereas before it was dry and abstract. That feature, called the Stream Browser, was the first feature that we did in the Java GUI, and it was ready for use in the first month of development.

If I had it to do all over again I would do it using Agile. If we had released just that one feature, we could have increased both our price and our sales a year earlier. One could argue that it might not have had any effect at all, but the change in sales was so dramatic and feedback from customers was so clear, there is no doubt in my mind that introducing the Stream Browser as soon as it was ready would have advanced AccuRev's growth by a year.

Recently, we had the opportunity to apply this lesson. Until June of 2008, there was no web interface for AccuRev. Nothing. Nada. Although that may seem rather startling in this day and age, we have always been internet-enabled via TCP/IP. As it turns out, the business value of other functionality has always come ahead of putting effort into developing a web interface. But at the end of 2007, it was apparent that many of our strategic initiatives depended on a web-based interface.

This time, we are using Agile development to develop the interface. We are using one month iterations and a quarterly release cycle. The functionality is shippable after every iteration. As of this writing, we do not yet do monthly releases. However, we do self-host it for internal use and use the most recent iteration for doing customer and prospect demos which provides lots of great feedback.

For the first release, the functionality with the highest business value was enabling users to browse repositories, do code reviews via diff and annotate, and create a standard for durable URLs so that people could embed URLs in other tools such as issue tracking and wiki tools and not worry about the format changing or getting the wrong version of something in the future. Another important factor was that since a web interface was the only way to provide this functionality, it did not overlap with existing functionality and so it did provide new business value. As a result of this functionality, we have been able to get many new customers that were already relying on this sort of functionality using other tools and would not have purchased without it.

Appendix C – Mainstream Practices

Mainstream Practices

A good way to see what it takes to become mainstream is to take a look at what the current mainstream software development practices are. These are practices which most people would agree that most people do. It may not be that they do that exact practice, they may do something equivalent or further along the spectrum of what constitutes a good practice, but they at least go to that level.

A mainstream practice is not a practice which is described by policy or documented as the correct way of doing something. A mainstream practice is something which can be observed to be in effect and in actual usage. It is not something which is sometimes done but usually avoided, it is not something which people know they are supposed to do but work around. A mainstream practice is something that is in common usage.

For example, consider the integration of bug tracking and source control. There is a whole spectrum here from no use of source control or issue tracking to use of one, use of both without any integration, integration via typing a bug number into a comment, tight integration on a file/version level via scripting or built-in capability to do validation and other actions, tight integration via transactions (sometimes called change sets or change lists), and tight integration via change packages. Corresponding to this spectrum is the concept of tracking all work that goes into production. That is, just as all bugs get a bug #, all enhancements get similar treatment. A further refinement is when the same system is used for both defects and enhancements.

Whew! That seems like a lot of detail, but actually this is just the beginning. Integration between source control and issue tracking is just one link in the chain of “requirements traceability” which is a round-trip connection from customer request to requirement to developer task to changes to release and all the way back. There are only a very few organizations worldwide that do that last one, though many people talk about it. And when I say that they “do it” I mean that do it like you drive to work in the morning: naturally, reliably, and without effort. And there are absolutely zero that do it at low cost. But that’s ok. There was a time when source control and issue tracking were considered advanced concepts.

The point in this spectrum which is currently the mainstream adoption level is integration via typing a bug number. There is actually a growing segment of the industry which goes at least one more level, to transaction-level integration which is mostly companies which use commercial tools, but there are also many companies which have accomplished this with OSS and scripting.

So, if you are not currently at least this level, when most people believe that there is enough benefit to do it, you should strongly consider it. Staying at a lower level is sort of like saying “electricity and hot and cold running water aren’t for me, I much prefer candles and fetching water from the well.”

Current Mainstream Practices

While each of the following individual practices is considered a mainstream practice, that does not mean that it is mainstream to being doing all of the following practices. That is, in some of the areas that these

practices cover, any particular development organization may be operating at a lower or higher level than these practices, but for any particular practice, most development organizations operate at or above the level of these practices.

Basic flow – the common development activities are: talk to customers and/or do market analysis, document market needs via requirements, create a design, implement the design, write tests, execute the tests, do find/fix until ready to ship, ship.

Preparation

Requirements documented using Microsoft Word or Excel – while there are actually quite a few off-the-shelf requirements tools available, most people do not use them. The most common method for documenting requirements is via Word and Excel.

Recording of defects in Bugzilla – there are very few organizations which aren't using at least Bugzilla to track bugs and Bugzilla is definitely the most popular choice. And recording of defects is pretty much as far as it goes

Basic initial project planning – this is the simple act of picking a bunch of work to be done, estimating it, dividing it up among the folks that are available to do the work and determining a target ship date. Some teams use MS-Excel, some teams use MS-Project. This does not mean sophisticated project planning. The extent of re-planning is a simple “are all tasks done yet” with the occasional feature creep and last minute axing of important functionality at the last minute.

Basic design – prior to coding features or defect fixes that will take more than two days to do or are highly impactful, a design document is created, circulated, discussed, and updated.

Development

All source code is stored in a source control system – this is not to be confused with “all work is done in a source control system.” It is actually surprisingly common to see the source control system used for archiving of work done rather than as a tool for facilitating the software development process.

Source control and issue tracking integration via a hand-entered bug number in the check-in comment, without enforcement – this was discussed at length in the introduction.

Defects have a defined workflow that is defined in and enforced by the bug tracking system – even if it is as simple as Open, Assigned, Closed.

Mainline development – all developers on a project check-in to the mainline (aka the trunk).

Refactoring – while the term “refactoring” has come to mean just about any change to the software, the idea that code should be periodically cleaned up and simplified is pretty much universal at this point.

Nightly build – the entire product is built every night. This does not necessarily mean that tests are also run automatically or that there is an automatic report of the results, just that there is a nightly build.

Quality

Mostly manual testing – this one is the hardest to understand. The process of testing software is one of the most backwards parts of software development.

Unit tests – while the overall testing of software is still mostly manual, unit testing has caught on rapidly in a very short span of time.

Using defect find/fix rate trend to determine release candidacy – this is not actually a particularly good practice, but it is what most people do.

Separation of developer, test, and production environments – you might think this is so obvious it isn't even worth mentioning, but this principal is violated enough that it is worth mentioning that it is actually a mainstream practice. I mention it to emphasize that if you aren't doing this, you really need to.

Releasing

Basing official builds on well-defined configurations from source control – the official process for producing production builds includes the step of creating an official configuration in the SCM system and using that configuration to do the build.

Releasing only official builds – all builds that go into production are produced using the official process.

Major and minor releases - creating a major release every 6 to 12 months and minor and/or patch releases on a quarterly basis.

Recommended Reading

Business

First, Break All the Rules, Marcus Buckingham and Curt Coffman

If you think any person should be able to perform well in any role, you should read this book. This book makes the case that as an employee and as an employer you will get more return on investment by getting people to spend more time doing and improving upon what they are good at and less time doing and trying to improve what they struggle with.

The 17 Indisputable Laws of Teamwork, John C. Maxwell

An indispensable book on teamwork. Great for team members and team leaders alike.

Product Management

Crossing the Chasm, Geoffrey Moore

Why is it that so many promising technology companies with great ideas, great products, and great customer service fail? One reason is because they don't change their marketing and product development strategies as they transition through the various stages of the technology adoption life cycle.

Differentiate or Die, Jack Trout and Steve Rivkin

Another reason that companies fail is that they aren't really differentiated from their competition. Just adding all of the bells and whistles that your customers ask you for is not a way to differentiate. Your competition is being asked for exactly the same things.

Requirements, User Stories, and Use Cases

User Stories Applied, Mike Cohn

The Agile equivalent of requirements is user stories. This book is the definitive guide to creating and using them.

Writing Effective Use Cases, Alistair Cockburn

Use cases are typically associated with traditional development and requirements. However, Alistair is an Agile advocate and this book reflects his Agile attitude. Whether you decide to employ use cases or not, there is a lot to learn here about gathering information from users to effectively design highly usable software.

Managing Software Requirements, Leffingwell and Widrig

While this book doesn't mention the words "Agile" or "Lean", it does refer to iterative requirements gathering. The real value of this book is in helping to produce robust and useful requirements regardless of what methodology you are using. This is one of those "leave no stone unturned" books.

Lean

The Machine That Changed The World, James P. Womack, Daniel T. Jones, and Daniel Roos

This is the classic book describing the Toyota Production System.

Lean Six Sigma, Michael L. George

I read this book because I was interested in learning more about Six Sigma. However, it was the material on Lean that really stood out. The way that the problem and solutions for finished product sitting on a factory floor is explained immediately suggested parallels to finished features sitting in the source repository waiting for release.

The Toyota Way, Jeffrey K. Liker

Lean manufacturing, pioneered by The Toyota Production System, is one of the key influencers of Agile thinking. In this fascinating book, Mr. Liker not only describes Toyota's approach to Lean, he also explains how it is just one aspect of Toyota's total approach to producing high quality products in an unusually short amount of time.

Lean Software Development, Mary Poppendieck and Tom Poppendieck

These folks have done an excellent job of taking the lessons of Lean manufacturing, pioneered by the Toyota Production System, and translating them to software development.

Estimation

Agile Estimating and Planning, Mike Cohn

Many of the estimation techniques that were developed prior to the advent of Agile are difficult to apply within an Agile framework. As the name suggests, this book is laser focused on Agile and contains a complete repertoire of Agile estimation and planning techniques.

Estimating Software-Intensive Systems, Stuzke

More information about estimation than you would have ever imagined existed. An exhaustive and definitive work on the subject.

Software Development

Extreme Programming Explained – First and Second Edition, Kent Beck

Kent's book is a short and easy read that is well worth your time. A seminal book on Agile development. The second edition is practically a different book. It is worth reading both if you can find the first edition.

Software Teamwork: Taking Ownership for Success, Jim Brosseau

Independent of your chosen methodology, the people and the personal interactions are a big factor in the success of any software development project. On an Agile project, the tight feedback loop not only surfaces technical problems faster, it will also make people issues much more apparent. This book offers a great framework for looking at the interactions of individuals, groups, teams, and stakeholders as well as many practical approaches for identifying and addressing specific issues. This is a terrific book to have at your side as you take the Agile plunge.

Managing Virtual Teams, M. Katherine Brown, Brenda Huettner, Char James-Tanny

This book covers all of the bases for optimizing your distributed development, from blogs to meetings to wikis. If you are managing a team that has a distributed development component, leveraging the combined experience of the three authors will definitely save you time.

Patterns of Agile Practice Adoption, Amr Elssamadisy

This is a no-nonsense book for those that are looking to adopt some Agile practices (even if you aren't looking to go Agile) but aren't sure where to start. The book will help you to identify which problems you have that Agile practices can help solve and then gives straightforward advice on how to adopt those practices.

Agile Software Development (Second Edition), Alistair Cockburn

A comprehensive treatment of all aspects of Agile development. Each section is chock full of insightful and illustrative anecdotes, nearly 100 in all!

Continuous Integration, Paul M. Duvall

There's much more to Continuous Integration (CI) than just kicking off lots of builds and this book proves it. I've always been a big fan of CI, but even so I was skeptical that there could be enough material to fill a whole book (283 pages). There's really much more in this book than just CI. The author uses the context of CI to cover lots of software development best practices. Topics include: Continuous Build, Continuous Test, Continuous Inspection, Continuous Database Integration and many more. Release Engineers in particular will want to recommend this book as it eloquently and effectively covers many topics which are RE pet peeves such as "it works on my machine" and creating a consistent directory structure.

Domain Driven Design, Eric Evans

Domain Driven Design is a fairly new way of thinking about software design. It goes a long way to break down the communication barriers between users and developers. In terms of Agile development, it greatly facilitates refactoring, maintainability, and testability. This is a must read for developers, folks in QA, architects, and even product managers.

Agile Project Management, Jim Highsmith

This unique book abstracts Agile from software development and applies it to product development in general. By doing so, it removes the hidden "hardwiring" and provides unexpected value within software development projects.

Agile Project Management with Scrum, Ken Schwaber

If you are already familiar with Scrum, this is a treasure trove of examples of how it has been applied in the real world.

Agile Software Development with Scrum, Ken Schwaber

One of the most influential books on Agile development. A must read.

The Capability Maturity Model

While CMM is often associated with heavyweight process, there is still a great deal that can be learned and applied from the ideas of the CMM when reading with an Agile mindset.

Usability

The Design of Everyday Things, Donald A. Norman

This book doesn't even mention software, but it provides insights into usability and design which applies to anything that humans use. Once you read this book, you will literally look at the world completely differently.

Designing Visual Interfaces, Kevin Mullet and Darrell Sano

I think I may have bought this book by accident while buying a bunch of other books. If you happen to come across it in a bookstore, you'll immediately get the impression that it is dated and possibly obsolete (it was published in 1994). However, most of the material in the book is both timeless and hard to find in any other book. I'm so glad that I looked past the dated screenshots and you will be too.

The Humane Interface, Jef Raskin

One of the designers of the original Mac user interface, Jef Raskin was a true pioneer in the field of software usability. While you may find some of the material in this book a bit "out there," that's simply a side effect of pushing the envelope.

Information Dashboard Design, Stephen Few

I actually bought this for the information on dashboards but loved it for the wonderful information on visual usability. If it was up to me, I would remove the word Dashboard from the title because although it uses dashboards as examples, all of the information is generally applicable.

User Interface Design for Programmers, Joel Spolsky

I thought usability was a pretty straight-forward subject which I knew all about until I read this book. While the title says "User Interface," the focus is on usability in general. You don't need to be a GUI person to appreciate this book. Joel's insights cracked through my false perceptions as a programmer and gave me the basis for appreciating what usability is truly about. This book also gave me an appetite for obtaining as much information on usability as I could. As a bonus, this is a fun read.

Software Quality Assurance and Testing

A Practitioner's Guide to Software Test Design, Lee Copeland

If you are interested in applying advanced QA techniques, this is a good place to start.

Software Quality Engineering, Jeff Tian

For people that are really, really serious about QA and have a taste for formality.

Test-Driven Development, Kent Beck

An invaluable resource for learning how to write effective test cases early in the development process, even before you have any code to test!

Fit For Developing Software, Rick Mugridge and Ward Cunningham

Another great resource for writing tests early in the development process, specifically using the FIT toolset.

Software Configuration Management

Configuration Management Patterns, Berczuk and Appleton

If you are looking for ideas on how to leverage your SCM system as part of re-architecting your development process, this is the best place to start.

Index

3x5, 13, 26
99.999% uptime, 64
AccuRev, 14, 36
AccuWorkflow, 14
Agile Manifesto, 20
AJAX, 13
algorithm, 16
Apache, 146
assembly line, 131
Atomic Transactions, 153
Automated Testing, 97
automated tests, 92
automation, 153
backlog, 36, 85, 125
bad habits, 64
bandwidth, 75
baselines, 93
black-box, 90
branches, 146
budgets, 71
bugs, 101
Bugs, 70
bureaucracy, 47
C#, 29
Capability Maturity Model, 177
cash flow, 46
CFO, 40
chess, 141
Chinese Finger Puzzle, 31
CMM, 177
Code Churn, 33
code coverage, 92
Code Coverage, 94
code freeze, 38
coding standards, 90
collocated, 39
Collocation, 121
compliance, 92
confidence, 101
contract, 68
core competency, 72
cost, 97
cost/benefit analysis, 75
counterintuitive, 140
creativity, 72
critical path, 32
customer, 18, 35, 101
customer satisfaction, 26, 75
customers, 90
Customers, 75
deadlines, 40
debug, 70
defect reports, 152
demo, 93, 160
demonstration, 128
demos, 159
development resources, 92
discovery, 70
Documentation, 76
domain knowledge, 140
early warning, 94
Eclipse, 26
employee satisfaction, 26
Estimation, 176
expenses, 26
exploratory testing, 92
Extreme Programming, 19, 144, 176
feature creep, 38
filter, 43, 44
forecast, 45
Frankenstein's monster, 75
Gantt charts, 22
global, 146
habit, 32
habits, 35, 72
hotfixes, 35
Hydra, 33
Hyperdrive, 72
impact analysis, 141
Inc. 500, 27
intermingled, 44
Issue tracking system, 139
issue-tracking system, 66
Iteration Retrospective, 128
Java, 29
Jolt Awards, 14, 159
lean, 71, 107
Lean, 152, 175
Lean Manufacturing, 131
Linux, 13, 146

Little & Co, 27, 28
 long iterations, 32, 138
 mainline, 103
 mainstream, 18
 Mainstream, 16
 maintenance, 35
 Manifesto, 13
 manual labor, 72, 153
 manual process, 75
 marketing, 46
 maturity, 74, 83
 merged, 93
 Meritocracy, 145, 160
 metrics, 152
 Niagra method, 30
 objections, 123
 offshore, 39
 one month iteration, 125
 one-piece flow, 36
 Open Source Software, 64, 145
 opportunity, 46
 osmosis, 66
 outsourcing, 28, 39
 pair programming, 26
 Pair programming, 144
 Paper, 152
 patches, 35, 146
 PERT, 86
 phases, 45
 pipeline, 45
 post mortems, 70
 post-mortem, 128
 predictability, 40, 41
 predictors, 44
 process, 17
 Process, 63, 70
 process document, 63, 138
 process improvement, 124
 Product Management, 75, 175
 product owner, 36
 productivity, 26, 28, 72, 101
 profits, 26
 progress, 101
 purchase order, 42
 QA, 35
 QA resources, 92
 quality, 26
 Quality, 21
 quality quotient, 126
 Quicken, 15
 Refactoring, 83
 regression, 90
 Release, 73, 129
 release candidates, 34
 Release Retrospective, 129
 requirements, 75
 Requirements, 76, 94, 175
 requirements gathering, 43
 retrospectives, 70
 revenues, 28
 rework, 32, 38, 123
 rhythm, 58, 93, 94, 128
 risk, 140, 141, 146
 Robust, 63
 ROI, 15, 22, 23, 75
 root cause analysis, 71, 131
 sales, 40
 salespeople, 40
 Sarbanes-Oxley, 13
 Scaleable, 63
 SCM, 146, 153, 179
 script, 138
 scripting, 123
 Scrum, 19, 26, 177
 second nature, 34
 shippable, 126
 short iterations, 22, 34, 125
 shortcuts, 33
 Six Sigma, 176
 Software Configuration Management, 179
 source code, 63
 source control, 153
 Source control system, 139
 stability, 108
 stakeholders, 101, 128
 standardization, 17
 sticky notes, 66
 stone tablets, 153
 stop the line, 107
 sustainable pace, 24
 Sustainable Pace, 24
 test automation, 23

test plan, 90, 97
testability, 90
Test-Driven Development, 178
tests, 94
Tests, 94
Toyota, 176
transactions, 154
triaging, 152
Unintended Consequences, 140
unpredictable, 40
usability, 92, 101
Usability, 101, 178
usability testing, 102

User Stories, 175
variable length iterations, 93, 160
version control, 153
Virtualization, 139
Visibility, 22, 24
volunteer, 160
waterfall, 19, 20
Waterfall, 26
whiteboard, 66
white-box, 90
work item, 45, 126
work items, 32
YAGNI, 36