Get More Refcardz! Visit refcardz.com

DZone Refcardz

Agile Adoption: Decreasing time to market

By Gemba Systems

ABOUT AGILE ADOPTION

There are a myriad of Agile practices out there. Which ones are right for you and your team? What are the business values you want out of adopting Agile and what is your organization's context? This Refcard is focused on helping you evaluate and choose the practices for your team or organization when getting to market faster is of prime importance. Instead of focusing on entire methods such as Scrum and XP, we will talk about the practices that are the building blocks of these methods such as iterations and automated developer tests. We will answer two basic questions:

- *What Agile practices should you consider to improve Time to Market?*
- *How should you go about choosing from those practices given your organization and context?*

WHAT AGILE PRACTICES IMPROVE TIME TO MARKET?

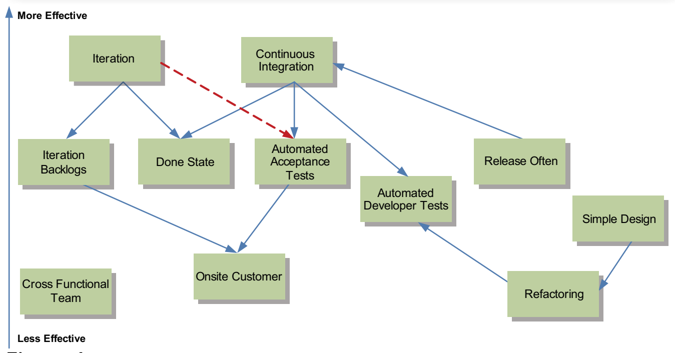
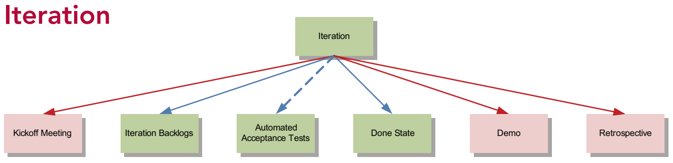


Figure 1- These are the Agile practices that improve time to market. The most effective practices are near the top of the diagram. Therefore iteration is more effective than Onsite Customer for improving time to market. The arrows indicate dependencies. Continuous Integration depends on Automated Developer Tests for it to be effective.

You will be ale to use this Refcard to get a 50,000 ft view of what will be involved in your team's delivery speed.

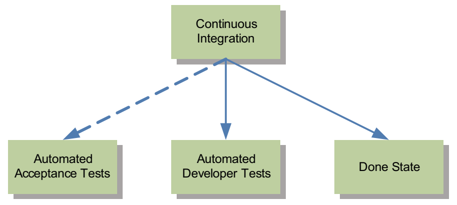
Iteration



*Practices in pink are ones that don't directly address time to market but are needed to support practices that do (hence a dependency). They are not described in this Refcard but can be found in the external references.

Definition	An iteration is a time-boxed event that is anywhere between 1 to 4 weeks long. The development staff works throughout this period – without interruption – to build an agreed upon set of requirements that are accepted by the customer and meet an agreed upon "done state".
Time to Market	To get the most of an iteration and reduce your time to market, an iteration needs to work from an iteration backlog and reach a solid done state at its completion. Such an iteration reduces time to market because every time-boxed iteration is a potential release. There is little "work in progress" between iterations and defects are found early and often for cheaper and faster removal.
	Any software team that is building software where they are not 100% sure of their outcome is a candidate for performing iterations. Without iterations the majority of learning (from mistakes) only happens at the end and course-corrections are difficult if not impossible.

Continuous Integration



Definition	Continuous integration (CI) is an advanced version of the "daily build" practice that has been around for years. The development team members perform CI by frequently integrating their code into the entire system and running all available automated tests. The system is integrated and tested several times a day.
Time to Market	Continuous integration reduces the total time it takes to build a software system by catching errors early and often. Errors caught early cost significantly less to fix when caught later. It leverages both automated acceptance tests and automated developer tests to give frequent feedback to the team and to pay a much smaller price for fixing a defect as shown in Figure 2.
	A team that has control or can get access to their build environment will be able to perform continuous integration. Teams that have agreed on a done state can benefit from continuous integration as a supporting practice to catch any failed automatic tests early.

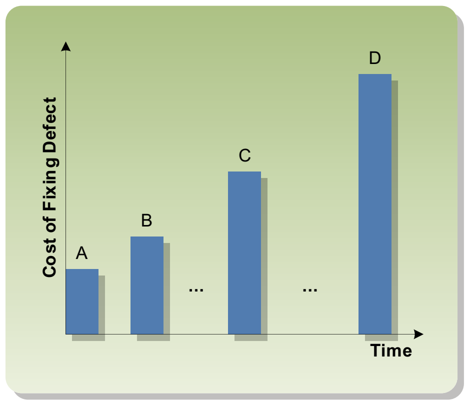
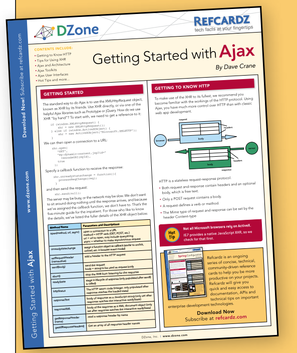


Figure 2- The cost of fixing a defect increases over time because of context switching, communication, and bugs being built on existing bugs.

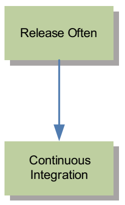

DZone REFCARDZ
Getting Started with Ajax
By Dave Crane

Get More Refcardz
(They're free!)

- Authoritative content
- Designed for developers
- Written by top experts
- Latest tools & technologies
- Hot tips & examples
- Bonus content online
- New issue every 1-2 weeks

Subscribe Now for FREE!
Refcardz.com

Release Often



Release your software to your end customers as often as you can without inconveniencing them.

Releasing often streamlines your development process and makes you deal with the pains of getting software good enough to go live. A team that releases often faces the pains and addresses the problems that make deployment difficult so that releasing is just another development task.

You are on a project where releasing often will enable you to produce revenue earlier. Having new features available frequently will not inconvenience your customer base. The quality of your releases is superb and your customers eagerly await your next release (instead of religiously keeping away from your 1.0 releases).

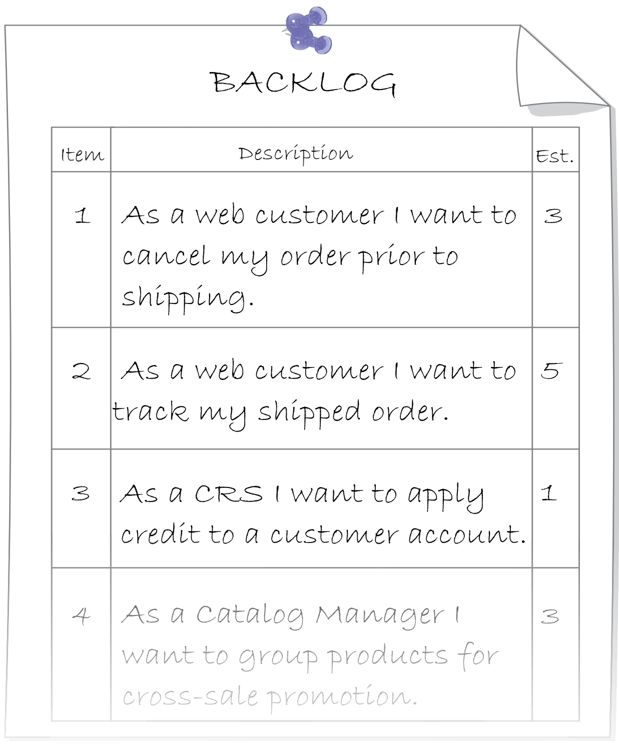
Done State

The done state is a definition agreed upon by the entire team of what constitutes the completion of a requirement. The closer the done state is to deployable software, the better it is because it forces the team to resolve all hidden issues.

A done state that is close to deployment enables the team to be confident in its work. The psychological effect of this confidence is a development team that gives good estimates, delivers regularly, and is confident in releasing its software. An executive decision can be made to release what has been built at anytime.

A team that should consider using a done state is one that has the necessary expertise and resources to build a requirement from end-to-end and perform all of the necessary build and deployment tasks.

Iteration Backlog



BACKLOG

Item	Description	Est.
1	As a web customer I want to cancel my order prior to shipping.	3
2	As a web customer I want to track my shipped order.	5
3	As a CRS I want to apply credit to a customer account.	1
4	As a Catalog Manager I want to group products for cross-sale promotion.	3

A backlog is a prioritized list of requirements. There are two common flavors of backlogs, one for the current iteration and one for the product. The product backlog contains all of the requirements prioritized by value to the customer. The iteration backlog is a list of requirements that a team has committed to building for an iteration.

Properly prioritized backlogs that are used to set the goals for every iteration ensure that the team is always working on the most important requirements. When paired with iterations that produce working, tested software, backlogs give a development team the option to release at the end of any iteration having always worked on the most important issues.

An expert on business value is needed to be part of the team to prioritize the backlog. If your team has such a person or someone that can coordinate with the business stakeholders to do so then use a product backlog. If you are using iterations then use an iteration backlog to set clear goals for the iterations and a release backlog to maintain long-term goals.

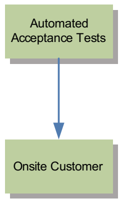
Automated Developer Tests

Automated developer tests are a set of tests that are written and maintained by developers to reduce the cost of finding and fixing defects—thereby improving code quality—and to enable the change of the design as requirements are addressed incrementally.

Automated developer tests reduce the time to market by actually reducing the development time. This is accomplished by reducing a developer's time in debugging loops by catching errors in the safety-net of tests.

You are on a development team that has decided to adopt iterations and simple design and will need to evolve your design as new requirements are taken into consideration. Or you are on a distributed team. The lack of both face-to-face communication and constant feedback is causing an increase in bugs and a slowdown in development.

Automated Acceptance Tests



Automated acceptance tests are tests written at the beginning of the iteration that answer the question: "what will this requirement look like when it is done?". This means that you start with failing tests at the beginning of each iteration and a requirement is only done when that test passes.

This practice builds a regression suite of tests in an incremental manner and catches errors, miscommunications, and ambiguities very early on. This, in turn, reduces the amount of work that is thrown away and enables faster development as you receive early feedback when a requirement is no longer satisfied.

You are on a development project with an onsite customer who is willing and able to participate more fully as part of the development team. Your team is also willing to make difficult changes to any existing code. You are willing to pay the price of a steep learning curve.

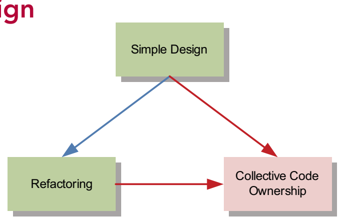
Onsite Customer

The onsite customer role in an Agile development team is a representative of the users of the system who understands the business domain of the software. The customer owns the backlog, is responsible for writing and clarifying requirements, and responsible for checking that the software meets the requirements specified.

The role of customer helps improve time to market by supporting the developers by giving them clear requirements, providing clarifications and verifying that the software does really meet the needs of the user base. The customer provides early feedback to the development team so they never spend more than an iteration down a blind alley. Finally, having a customer who correctly prioritizes a backlog allows the team to deliver the most important items first when time is of the essence.

The practice of onsite customer works best when the development team can be co-located with one or more domain experts. The person fulfilling the customer role is crucial to the success of the team and therefore will need sufficient time and resources to do the job.

Simple Design



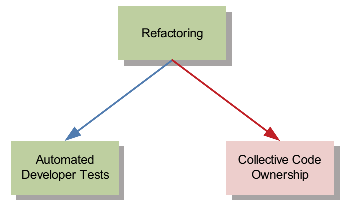
*Practices in pink are ones that don't directly address time to market but are needed to support practices that do (hence a dependency). They are not described in this Refcard but can be found in the external references.

If a decision between coding a design for today's requirements and a general design to accommodate for tomorrow's requirements needs to be made, the former is a simple design. Simple design meets the requirements for the current iteration and no more.

Simple design improves time to market because you build less code to meet the requirements and you maintain less code afterwards. Simple designs are easier to build, understand, and maintain.

Simple design should only be used when your team also is writing automated developer tests and refactoring. A simple design is fine as long as you can change it to meet future requirements.

Refactoring



*Practices in pink are ones that don't directly address time to market but are needed to support practices that do (hence a dependency). They are not described in this Refcard but can be found in the external references.

Definition	The practice of Refactoring code changes the structure (i.e., the design) of the code while maintaining its behavior. Collective code ownership is needed because a refactoring frequently affects other parts of the system. Automated developer tests are needed to verify that the behavior of the system has not changed after the design change introduced by the refactoring.
Time to Market	Refactoring improves time to market by supporting practices like Simple Design which, in turn, allow you to only write the software for the features that are needed now.
	You are on a development team that is practicing automated developer tests. You are currently working on a requirement that is not well-supported by the current design.

Cross-Functional Team

Definition	The team that utilizes the Cross Functional Team practice is one that has the necessary expertise among its members to take a requirement from its initial concept to a fully deployed and tested piece of software within one iteration. A requirement can be taken off of the backlog, elaborated and developed, tested, deployed.
Time to Market	Cross-functional teams primarily affect time to market by enabling true iterative and incremental development. Resource bottlenecks are resolved and teams can build features end-to-end.
	There is a hardening cycle at the end of each release indicating unresolved integration issues. Building a slice of functionality end-to-end in your system finds errors early and requires diverse expertise of many different people.

HOW TO ADOPT AGILE PRACTICES SUCCESSFULLY

To successfully adopt Agile practices let's start by answering the question "which ones first?" Once we have a general idea of how to choose the first practices there are other considerations.

Become "Well-Oiled" First

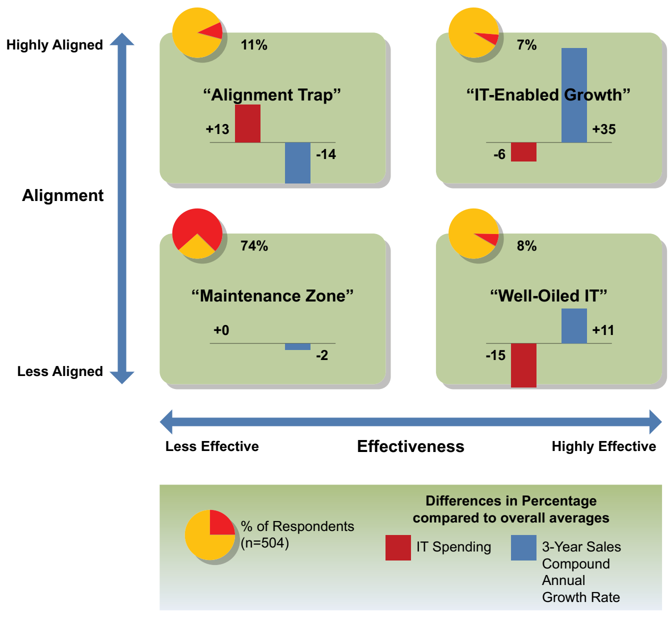
One way to look at software development is to see it as problem solving for business. When considering a problem to solve there are two fundamental actions that must be taken:

- Solving the right problem. This is IT/Business alignment.
- Solving the problem right. This is technical expertise.

Intuitively it would seem that we must focus on solving the right problem first because, no matter how well we execute our solution to the problem, if it is the wrong problem then our solution is worthless. This, unfortunately, is the wrong way to go. Research shows in Figure 3, that focusing on alignment first is actually more costly and less effective than doing nothing. It also shows that being "well-oiled", that is focusing on technical ability first, is much more effective and a good stepping-stone to reaching the state where both issues are addressed.

This is supported anecdotally by increasing reports of failed Agile projects that do not deliver on promised results. They adopt many of the soft practices such as Iteration, but steer away from the technically difficult practices such as Automated Developer Tests, Refactoring, and Done State. They never reach the "well-oiled" state.

So the lesson here is make sure that on your journey to adopt Agile practices that improve time to market (or any other business value for that matter), your team will need to become "well-oiled" to see significant, sustained improvement. And that means you should plan on adopting the difficult technical practices for sustainability.



Note: Based on 504 responses from 452 companies
Source: Bain Analysis

Figure 3- The Alignment Trap (from Avoiding the Alignment Trap in Information Technology, Shpilberg, D. et al, MIT Sloan Management Review, Fall 2007.)

Minimize What You Build

Statistics show that most of what software development teams build is not used. In Figure 4 we see that only 7% of functionality is always used. And 45% is never used. This is a sad state of affairs, and an excellent opportunity. One of the easiest ways to speed up is to do less. If you have less to build, then not only do you spend less time writing and testing software, but you also reduce the complexity of the entire application. And by reducing the complexity of the application it takes less time to maintain because you have a simpler design, fewer dependencies, and fewer physical lines of code that your developers must understand and maintain.

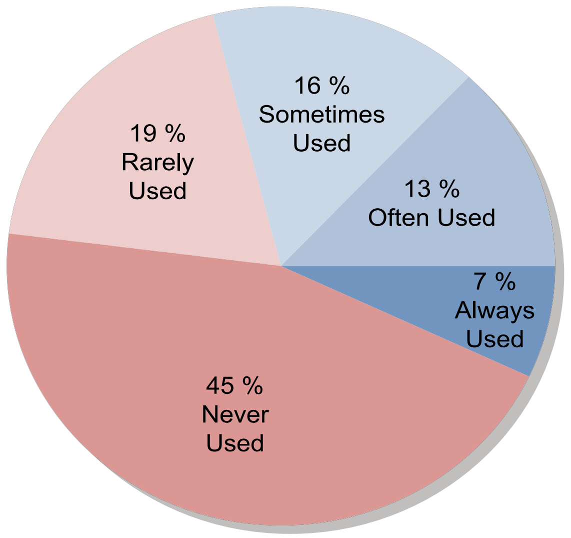


Figure 4- Most functionality built is not used.

Cross-Functional Team

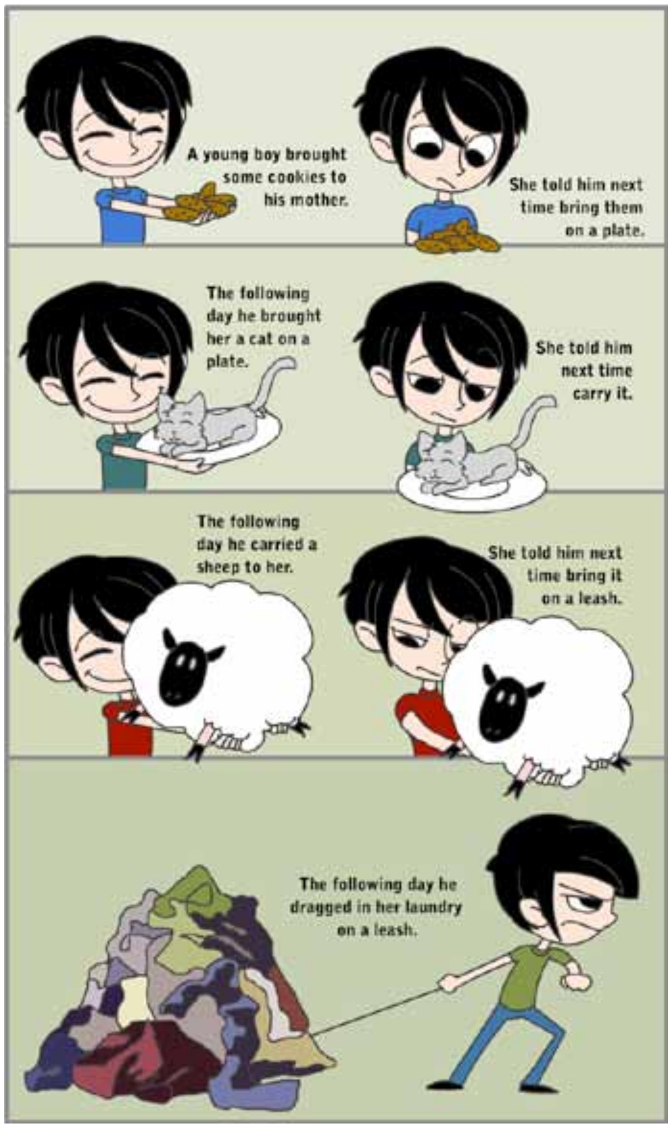

A young boy brought some cookies to his mother.
She told him next time bring them on a plate.
The following day he brought her a cat on a plate.
She told him next time carry it.
The following day he carried a sheep to her.
She told him next time bring it on a leash.
The following day he dragged in her laundry on a leash.

Figure 5- Context matters. Choose Agile practices that fit your context.

The practices are all described within context. So, for example, the context for the Release Often practice indicates that your customers should be willing to install and run frequent releases and that the quality of your current builds are exceptional. If this is not the case, if your current releases go through a 'stabilization phase' and your customers have learned never to take a 1.0 release, then do not adopt Release Often, you will end up hurting your relationship with your customers.

Learning is the Bottleneck

Here is a hypothetical situation that we have presented to many experienced software development teams:

> *Suppose I was your client and I asked you and your team to build a software system for me. Your team proceeds to build the software system. It takes you a full year – 12 months – to deliver working, tested software.*
>
> *I then thank the team and take the software and throw it out. I then ask you and your team to rebuild the system. You have the same team. The same requirements. The same tools and software. Basically – nothing has changed – it is exactly the same environment.*
>
> *How long will it take you and your team to rebuild the system again?*

When we present this hypothetical situation to development practitioners – many of them with 20+ years experience in building software – they typically respond with anywhere between 20% to 70% of the time. That is, rebuilding a system that originally takes one year to build takes only 2.5 to 8.5 months to build. It is a huge difference!

So, what is the problem? What was different? The team has learned. They learned about each other as a team and have gelled over the year. They learned about the true requirements – not just those written down. They also learned to use the toolset, they experienced the idiosyncrasies that come up during all software development, and basically they worked through all the unknowns until they built and delivered a successful software solution. Learning is THE bottleneck of software engineering.

The learning that occurs makes up a significant percentage of the time spent on the work. That's the main reason that Agile practices work so well – they are all about recognizing and responding to change. Agile practices, from continuous integration to iterations, all consist of cycles that help the team learn fast. By cycling in every possible practice, Agile teams accelerate learning, addressing the bottleneck of software engineering. Call it "scientific method," "continuous improvement" or "inspect and adapt", to truly benefit from these practices you and your team(s) must learn well and learn often.

Know What You Don't Know

Since learning is the bottleneck, it makes sense to talk a bit about how we actually learn. The Dreyfus Model of Skill Acquisition, is a useful model of learning. It is not the only model of learning, but it is consistent, has been effective, and works well for our purposes. This model states that there are levels that one goes through as they learn a skill and that your level for different skills can and will be different. Depending on the level you are at, you have different needs and abilities. An understanding of this model is not crucial to learning a skill; after all, we've been learning long before this model existed. However, being aware of this model can help us and our team(s) learn effectively.

So let's take a closer look at the different skill levels in the Dreyfus Model:

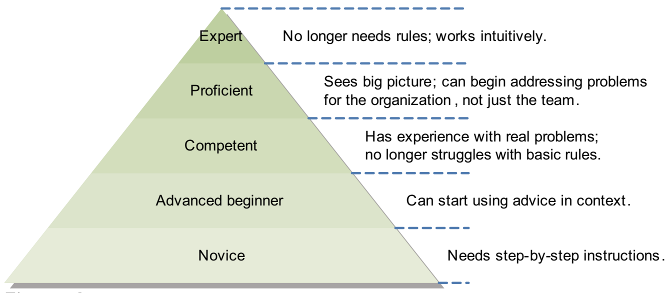

Expert
No longer needs rules; works intuitively.
Proficient
Sees big picture; can begin addressing problems for the organization, not just the team.
Competent
Has experience with real problems; no longer struggles with basic rules.
Advanced beginner
Can start using advice in context.
Novice
Needs step-by-step instructions.

Figure 6– The Drefyus Model for skill acquisition. One starts as a novice and through experience and learning advances towards expertise.

How can the Dreyfus Model help in an organization that is adopting agile methods? First, we must realize that this model is per skill, so we are not competent in everything. Secondly, if agile is new to us, which it probably is, then we are novices or advanced beginners; we need to search for rules and not break them until we have enough experience under our belts. Moreover, since everything really does depend on context, and we are not qualified to deal with context as novices and advanced beginners, we had better get access to some people who are experts or at least proficient to help guide us in choosing the right agile practices for our particular context. Finally, we'd better find it in ourselves to be humble and know what we don't know to keep from derailing the possible benefits of this new method. And we need to be patient with ourselves and with our colleagues. Learning new skills will take time, and that is OK.

Choosing a Practice to Adopt

Choosing a practice comes down to finding the highest value practice that will fit into your context. Figure 1 will guide you in determining which practices are most effective in decreasing your time to market and will also give you an understanding of the dependencies. The other parts in this section discuss other ideas that can help you refine your choices. Armed with this information:

Small steps and failing fast are the most effective methods to release quickly. Weed out defects early because the earlier you find them, the less they will cost to fix as shown in Figure 2 and you won't be building on a crumbling foundation. This is why Continuous Integration and Iteration lead the practices that most positively affect time to market. They are both, however, dependent on several practices to be effective, so consider starting with Automated Developer Tests and the Iteration trio – Iteration, Iteration Backlog, and Done State.

Next Steps

This refcard is a quick introduction to Agile practices that can help you improve your time to market and an introduction of how you to choose the practices for your organizational context. It is only a starting point. If you choose to embark on an Agile adoption initiative, your next step is to educate yourself and get as much help as you can afford. Books and user groups are a beginning. If you can, find an expert to join your team(s). Remember, if you are new to Agile, then you are a novice or advanced beginner and are not capable of making an informed decision about tailoring practices to your context.

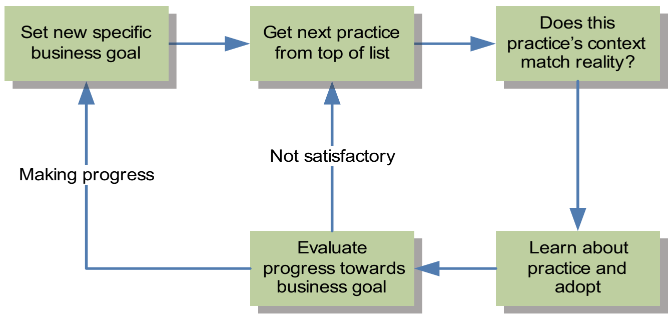

Set new specific business goal
Get next practice from top of list
Does this practice's context match reality?
Making progress
Not satisfactory
Evaluate progress towards business goal
Learn about practice and adopt

Figure 7- Steps for Choosing and Implementing Practices

REFERENCE TABLE

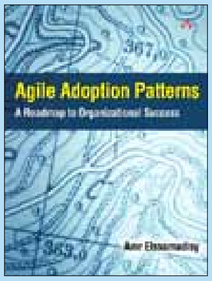
	Iteration	Continuous Integration	Release Often	Done State	Iteration Backlog	Automated Developer Tests	Automated Acceptance Tests	Onsite Customer	Simple Design	Cross-Functional Team
Astels, David. 2003. Test-driven development: a practical guide. Upper Saddle River, NJ: Prentice Hall.						X				
Beck, Kent. 2003. Test-driven development by example. Boston, MA: Pearson Education.						X				
Beck, K. and Andres, C., Extreme Programming Explained: Embrace Change (second edition), Boston: Addison-Wesley, 2005	X		X	X				X	X	X
Cockburn, A., Agile Software Development: The Cooperative Game (2nd Edition), Addison-Wesley Professional, 2006.										X
Cohn, M., Agile Estimating and Planning, Prentice Hall, 2005.					X					
Duvall, Paul, Matyas, Steve, and Glover, Andrew. (2006). Continuous integration: Improving Software Quality and Reducing Risk. Boston: Addison-Wesley.		X								
Elssamadisy, A., Agile Adoption Patterns: A Roadmap to Organizational Success, Boston: Pearson Education, 2008	X	X	X	X	X	X	X	X	X	X
Feathers, Michael. 2005. Working effectively with legacy code. Upper Saddle River, NJ: Prentice Hall.						X				
Jeffries, Ron. "Running Tested Features." http://www.xprogramming.com/xpmag/jatRtsMetric.htm							X			
Jeffries, Ron. 2004. Extreme programming adventures in c#. Redmond, WA: Microsoft Press.						X				
Kerievsky, Joshua. "Don't Just Break Software, Make Software." http://www.industriallogic.com/papers/storytest.pdf							X			

Larman, C., Agile and Iterative Development: A Manager's Guide, Boston: Addison-Wesley, 2004	X			X						
Larman, C., and Vodde, B., Scaling Lean and Agile Development, Boston: Addison-Wesley, 2009										X
Massol, Vincent. 2004. Junit in action. Greenwich, CT: Manning Publications.						X				
Meszaros, XUnit Test Patterns: Refactoring Test Code, Boston: Addison-Wesley, 2007.						X				
Mugridge, R., and W. Cunningham. 2005. Fit for Developing Software: Framework for Integrated Tests. Upper Saddle River, NJ: Pearson Education.							X			
Poppendieck, M., and Poppendieck, T., Implementing Lean Software Development, Addison-Wesley Professional, 2006.										X
Rainsberger, J.B. 2004. Junit recipes: Practical methods for programmer testing. Greenwich, CT: Manning Publications.						X				
Schwaber, K., and Beedle, M., Agile Software Development with Scrum, Upper Saddle River, New Jersey: Prentice Hall, 2001.	X		X	X	X			X		

ABOUT GEMBA SYSTEMS

Gemba Systems is comprised of a group of seasoned practitioners who are experts at Lean & Agile Development as well as crafting effective learning experiences. Whether the method is Scrum, Extreme Programming, Lean Development or others - Gemba Systems helps individuals and teams to learn and adopt better product development practices. Gemba Systems has taught better development techniques - including lean thinking, Scrum and Agile Methods - to thousands of developers in dozens of companies around the globe. To learn more visit http://us.gembasystems.com/

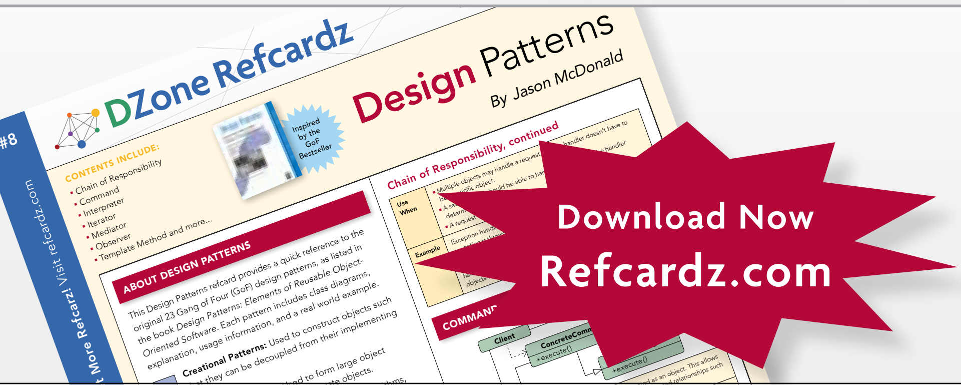
RECOMMENDED BOOK



Agile Adoption Patterns will help you whether you're planning your first agile project, trying to improve your next project, or evangelizing agility throughout your organization. This actionable advice is designed to work with any agile method, from XP and Scrum to Crystal Clear and Lean. The practical insights will make you more effective in any agile project role: as leader, developer, architect, or customer.

BUY NOW
books.dzone.com/books/agile-adoption-patterns

Professional Cheat Sheets You Can Trust



Download Now
Refcardz.com

"Exactly what busy developers need: simple, short, and to the point."
James Ward, Adobe Systems

Upcoming Titles
RichFaces
Agile Software Development
BIRT
JSF 2.0
Adobe AIR
BPM&BPMN
Flex 3 Components

Most Popular
Spring Configuration
jQuery Selectors
Windows Powershell
Dependency Injection with EJB 3
Netbeans IDE JavaEditor
Getting Started with Eclipse
Very First Steps in Flex

DZone

DZone communities deliver over 4 million pages each month to more than 2 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-48-6
ISBN-10: 1-934238-48-1
50795
9 781934 238486
$7.95

Copyright © 2009 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher. Reference: Agile Adoption Patterns, Amr Elssamadisy, Addison-Wesley, July 2008

Version 1.0