DZone Refcardz

CONTENTS INCLUDE:

Agile Adoption: Reducing Cost

By Gemba Systems

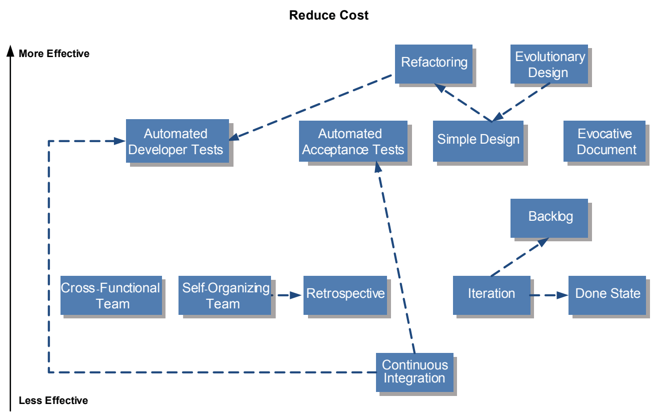
ABOUT THIS AGILE ADOPTION REFCARD

Faster, better, cheaper. That's what we must do to survive. The Time to Market Refcard (a companion in this series) addresses faster, the Quality Refcard addresses better, and this Refcard addresses cheaper. This is about building the system for less.

Some of the costs of software development are associated with man hours needed to build the system, others with cost of maintenance over time, and yet others include hardware as well as software platform costs. Practices that educe any or all of these costs without sacrificing quality reduce the overall cost of the system.

Then there is the Pareto principle – a.k.a. the 80/20 rule. This rule suggests that roughly 20% of the software system is used 80% of the time. This is also backed up by research that is even more dramatic [figure with usage]. Practices that help the team build only what is needed in a prioritized manner reduce the cost and still deliver the most important business value to the customer (the part she uses).

Figure 1 Practices that help reduce the cost of building software.



You will be able to use this Refcard to get 50,000 ft view of what will be involved to reduce the cost of developing your systems.

FOUR STRATEGIES TO REDUCE COST

Software development is complex and often very complicated. It is HARD. This is not some new revelation, in fact Fred Brooks in the well known paper, "No Silver Bullet.", states:

The essence of a software entity is a construct of interlocking concepts ...

I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation.

There are four major strategies that can help you reduce the cost of building and maintaining your software

Maintain the Theory of the Code

One way to look at software development is 'theory building'. That is, programs are theories – models of the world mapped onto software – in the head of the individuals of the development team. Great teams have a shared understanding of how the software system represents the world. Therefore they know where to modify the code when a requirement change occurs, they know exactly where to go hunting for a bug that has been found, and they communicate well with each other about the world and the software.

Conversely, a team that does not have a shared 'theory' make communication mistakes all the time. The customer may say something that the business analyst misunderstands because she has a different world view. She may, in turn, have a different understanding than the developers, so the software ends up addressing a different problem or, after several trials, errors and frustrations, the right problem but very awkwardly. Software where the theory of the team does not match, or even worse, the theory is now lost because the original software team is long-gone, is very expensive to maintain.

Building a shared theory of the world-to-software-mapping is a human process that is best done face-to-face by trial and error and with significant time.

Build Less

It has been shown that we build many more features than are actually used. In fact, we can see in Figure 2, only about 20% of functionality we build is used often or always. More than 60% of all functionality built in software is rarely or never used!

One way to reduce the cost of software is to find a way not to build the unused functionality. There are several Agile practices that help you get to that point.

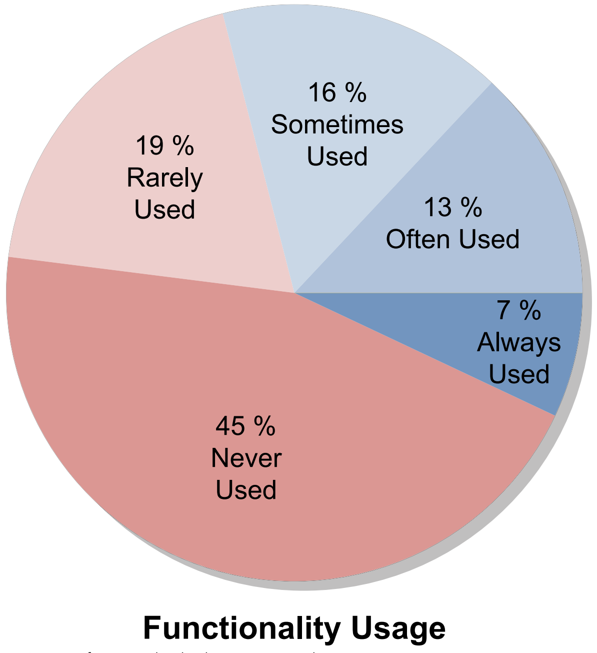


Figure 2: Most functionality built is never used.

Pay Less for Bug Fixes

Typically, anywhere between 60%-90% of software cost goes into the maintenance phase. That is, most of our money goes into keeping the software alive and useful for our clients after the initial build. Practices that help us reduce the cost of software maintenance will significantly affect the overall cost of the software product or system.

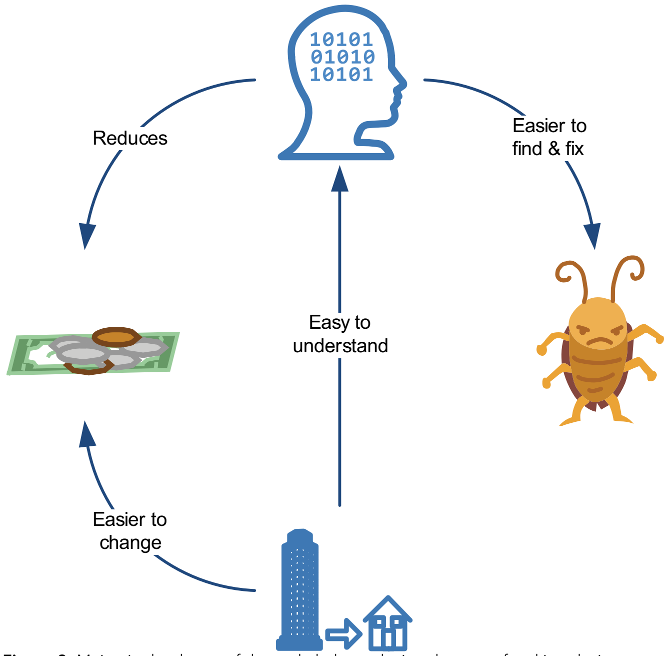


Figure 3: Maintain the theory of the code helps reducing the cost of making design changes and fixing bugs. Building less enables better understanding and helps to understand the theory of the code for a change because there is less to change

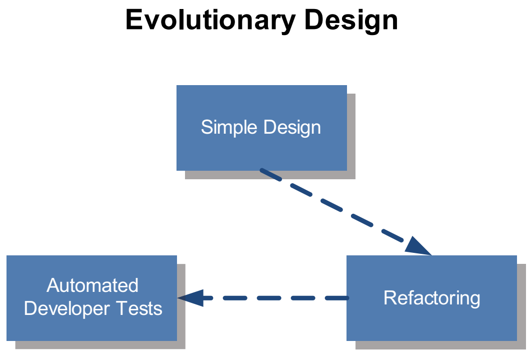
Pay Less for Changes

The only thing constant in today's software market is change. If we can embrace change, plan for it, and reduce its cost when it eventually happens we can make significant savings. One of the strongest points of Agile development is that its practices enable you to roll with the punches and change your software as the business world changes.

The four strategies above: maintain the theory of the code, build less, paying less for maintenance and being able to react to change are not independent.

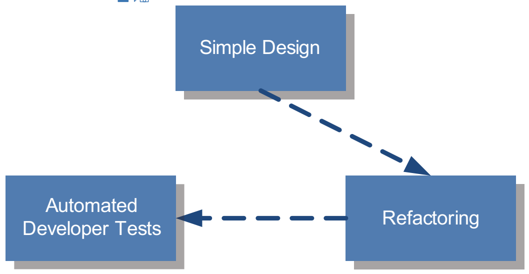
THE PRACTICES

Evolutionary Design



Definition	Evolutionary design is the simple design practice done continuously. Start off with a simple design and change that design only when a new requirement cannot be met by the existing design.
$	Evolutionary design reduces the cost by focusing on always building less. This, in turn, directly affects the cost of change drastically.
	You are on a development team practicing automated developer tests, refactoring, and simple design. That's it, because this is one of those things that is applicable to all types of development projects. The context is especially a match if the technology used technologies is new to a large part of the team.

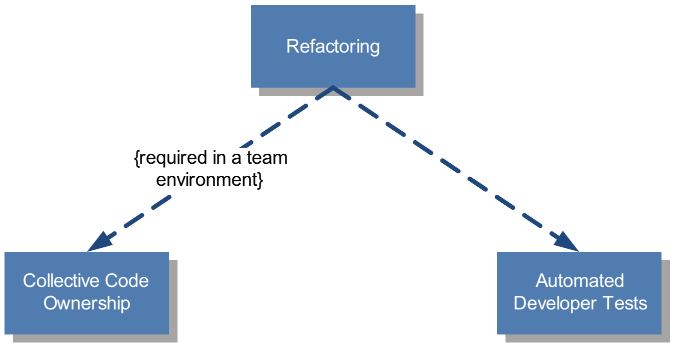
Simple Design



Definition	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 reduces cost 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


Refactoring
{required in a team environment}
Collective Code Ownership
Automated Developer Tests

Definition	The practice of Refactoring code changes the structure (i.e., the design) of the code while maintaining its behavioe.
$	Costs are reduced because continuous refactoring keeps the design from degrading over time, ensuring that the code is easy to understand, maintain, and change.
🔄	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. Or you may have just completed a task (with its tests of course) and want to change the design for a cleaner solution before checking in your code to the source repository.

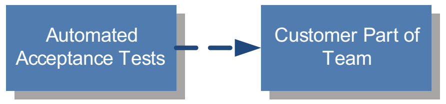
Automated Developer Tests

Definition	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 cost of software development by creating a safety-net of tests that catch bugs early and enabling the incremental change of design. Beware, however, that automated developer tests take time to build and require discipline.
🔄	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.

Evocative Document

Definition	Evocative documents are documents that evoke memories, conversations, and situations that are shared by those who wrote the document. They are more meaningful and representative of a team's understanding of the system than traditional documents.
$	Evocative documents help by accurately representing the team's internal model of the software and allowing that model to be handed down from master to apprentice. The better understanding of the system over time also reduces the maintenance cost of the system over time because appropriate changes reduce the deterioration of the software.
🔄	Current documentation isn't working – as a document is passed from one person to another much of the context and value is lost, and as a result, the maintenance team's understanding of the codebase constantly deteriorates. This is resulting in the calcification of your software system.

Automated Acceptance Tests



Definition	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 therefore enables building less. The tests also catch bugs and act as a safety-net during change.
🔄	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.

The remaining practices also help reduce the cost of software development. Because of the limited size of the refcard, we will only summarize them below.

Backlog	A backlog is a prioritized list of requirements that enable a team to build less by making sure they always work on the most important items first and help the team understand the theory of the code when used as an evocative document that shows a larger picture of the system.
Iteration	An iteration is a time-box where the team builds what is on the backlog and is a potential release and therefore enables building less.
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 early and thus reduces cost.
Cross-functional Team	The cross-functional team 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.

Self-organizing Team	A self-organizing team is in charge of its own fate. Management gives the team goals to achieve and the team members are responsible for driving towards those goals and achieving them. A self-organizing team recognizes and responds to changes in their environment and in their knowledge as they learn. A self-organizing team is frequently a cross functional team as well.
Retrospective	The Retrospective is a meeting held at the end of a major cycle - iteration or release - to gather and analyze data about that cycle and decide on future actions to improve the team's environment and process. A retrospective is about evaluating the people, their interactions, and the tools they use.
Continuous Integration	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.
User Story	A user story is an evocative document for requirements. A user story is a very high level description of the requirement to be built –it usually fits on a 3 x 5 index card – and is a "promise for a conversation" later between the person carrying out the Customer Part of Team practice and the implementers.

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

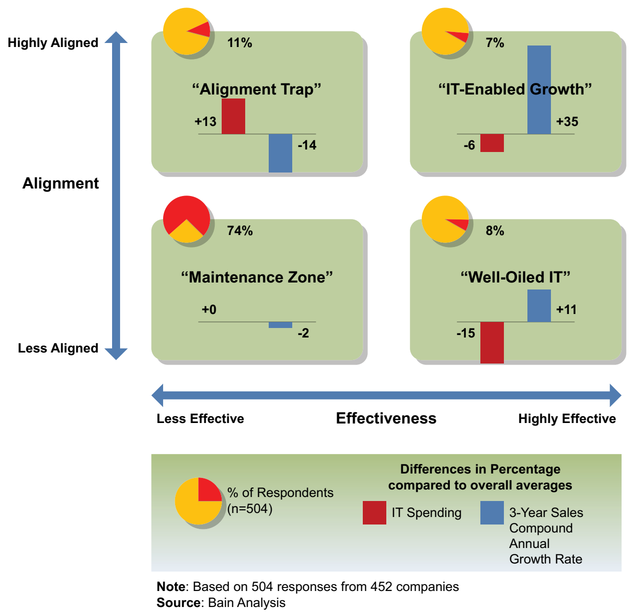


Figure 4: The Alignment Trap (from Avoiding the Alignment Trap in Information Technology, Shpilberg, D. et al, MIT Sloan Management Review, Fall 20078.)

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.

Know What You Don't Know

The Dreyfus Model of Skill Acquisition, is a useful way to look at how we learn skills – such as learning Agile practices necessary to reduce cost. 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:

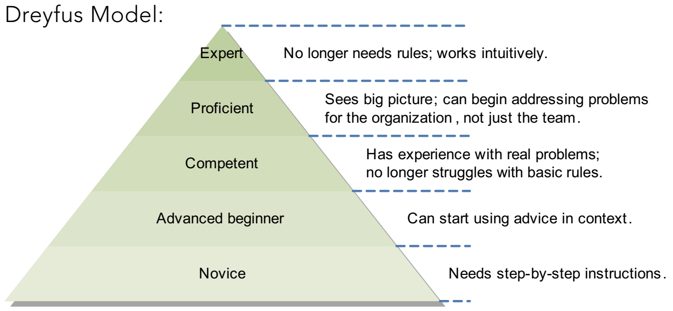

Expert
No longer needs rules; works intuitively.
Proficient
Sees big picture; can begin addressing problems for the organization , not just the team.
Competent
Has experience with real problems; no longer struggles with basic rules.
Advanced beginner
Can start using advice in context.
Novice
Needs step-by-step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

Figure 5: The Dreyfus Model for skill acquisition. One starts as a novice and through ecperience 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, How to Adopt Agile Practices Successfully?, discuss other ideas that can help you refine your choices. Armed with this information:

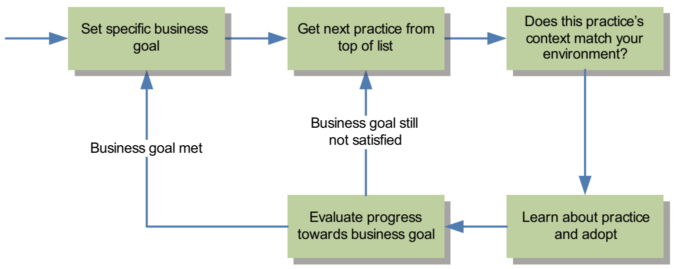

Set specific business goal
Get next practice from top of list
Does this practice's context match your environment?
Business goal met
Business goal still not satisfied
Evaluate progress towards business goal
Learn about practice and adopt

Figure 5: Steps for choosing and implementing practices.

WHAT NEXT?

This Refcard is a quick introduction to Agile practices that can help you reduce the cost of building and maintaining your software 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.

REFERENCES

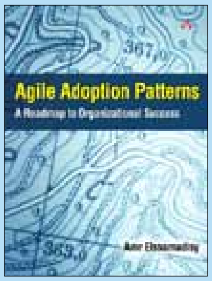
	Evolutionary Design	Simple Design	Refactoring	Automated Developer Tests	Evocative Document	Automated Acceptance Tests	Backlog	Iteration	Done State	Cross-Functional Team	Self-Organized Team	Retrospective	Continuous Integration	User Story
Astels, David. 2003. Test-driven development: a practical guide. Upper Saddle River, NJ: Prentice Hall.			x	x										
Avery, Christopher, Teamwork is an Individual Skill, San Francisco: Berrett-Koehler Publishers, Inc., 2001											x			
Bain, Scott L., 2008, Emergent Design, Boston, MA: Pearson Education	x	x	x	x										
Beck, Kent. 2003. Test-driven development by example. Boston, MA: Pearson Education.			x	x										
Beck, K. and Andres, C., Extreme Programming Explained: Embrace Change (second edition), Boston: Addison-Wesley, 2005	x	x	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							x
Crispin, L. and Gregory, J., Agile Testing: A Practical Guide for Testers and Agile Teams						x								
Derby, E., and Larson, D., Agile Retrospectives: Making Good Teams Great, Raliegh: Pragmatic Bookshelf, 2006.											x	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	x	x	x	x
Feathers, Michael. 2005. Working effectively with legacy code. Upper Saddle River, NJ: Prentice Hall.			x	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	x										
Kerth, N., Project Retropsectives: A Handbook for Team Reviews, NY: Dorset House Publishing Company, 2001.												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	x	x				
Massol, Vincent. 2004. Junit in action. Greenwich, CT: Manning Publications.			x	x										

	Evolutionary Design	Simple Design	Refactoring	Automated Developer Tests	Evocative Document	Automated Acceptance Tests	Backlog	Iteration	Done State	Cross-Functional Team	Self-Organized Team	Retrospective	Continuous Integration	User Story
Meszaros, XUnit Test Patterns: Refactoring Test Code, Boston: Addison-Wesley, 2007			x	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	x										
Schwaber, K., and Beedle, M., Agile Software Development with Scrum, Upper Saddle River, New Jersey: Prentice Hall, 2001.							x	x	x		x			
Senge, P., The Fifth Discipline: The Art and Practice of The Learning Organization, NY: Currency 2006.											x			
Surowiecki, J., The Wisdom of Crowds, NY: Anchor, 2005.												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

DZone, Inc.
2222 Sedwick Rd Suite 101
Durham, NC 27713
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-58-5
ISBN-10: 1-934238-58-9
50795
9 781934 238585
$7.95

Version 1.0