CONTENTS INCLUDE:

Agile Adoption:
Improving Software Quality

By Gemba Systems

ABOUT IMPROVING SOFTWARE QUALITY

Faster, better, cheaper. That's what we must do to survive. The Time to Market Refcard addresses faster, the Reduce Cost Refcard addresses cheaper, and this Refcard addresses better. This is about improving the quality of your software; that means reducing bugs and improving design.

The vast majority of software projects suffer from a steady degradation of design quality and it becomes more and more difficult to maintain the software with the same level of quality. As the software ages it calcifies and becomes harder and harder to maintain. In some cases it becomes too expensive to maintain and so it the software is put to rest and rewritten. In others, the software is released with a steadily increasing number of defects. Both of these common situations are deeply unsatisfying, but there is another way.

Many of the practices from the Agile world stop the degradation of software quality and turn the trend around. It is not unheard of for teams to have maintained a zero-defect status for months and years. Design and architecture have become malleable; they now emerge and transform over time. In fact, Gartner now recommends an emergent approach to enterprise architecture (http://www.gartner.com/it/page.jsp?id=1124112).

Figure 1 Practices that help improve the quality that your software development team(s) builds.

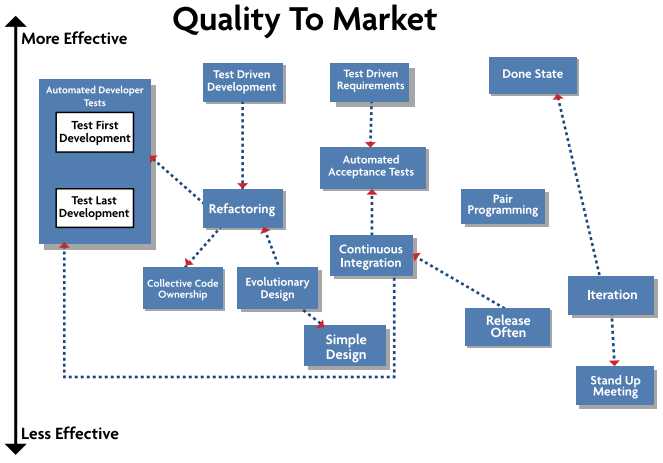


Figure 1: Improve Quality Practices

You will be able to use this refcard to get a 50,000 ft view of what will be involved to incrementally improve the quality of your software.

STRATEGIES FOR IMPROVING QUALITY

The Agile community has been fertile ground for quality improvements in software development.

There are four major strategies that can help you improve the quality of your software:

Reduce Defects

Reducing defects is the first thing that comes to mind when examining the quality of software. A low defect count is often synonymous with high quality software. Defects are also the most visible sign of quality problems.

Improve Design

Design is the model that a development team builds and maintains. High quality design makes for an application that is easy to understand and change as new requirements are discovered. Traditionally, the team has one shot to get the design right and then it degrades over time as it is patched over time. Agile practices, however, give an alternative; using practices like test driven development and refactoring teams are now able to continuously improve the design of their system.

Theory Building

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' makes communication mistakes all the time. The customer may say something that the business analyst misunderstands because she has a different worldview. 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, degrades in quality as design changes are made that don't fit with the theory, or even just as bad, cut & paste work is done because the theory is not understood. 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 with ample 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 that most functionality we do build is never used. So, one very effective way to improve the quality of our software is to build less of it. It makes it easier to understand, gives us more time to focus on the important parts that are actually used, and almost always has fewer defects.

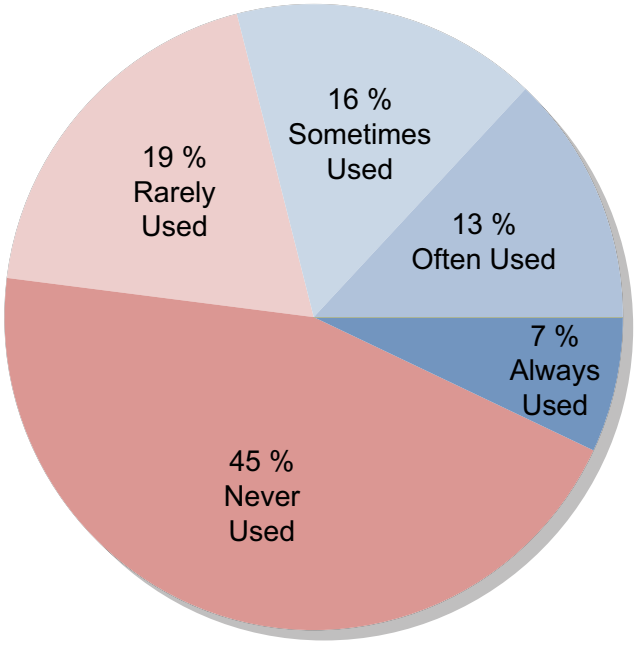


Figure 2: Functionality Usage, only about 20% of functionality we build is used often or always. More than 60% of all functionality built in software is rarely or never used! One way to improve the quality of software is to write less code which makes it easier to understand and maintain. There are several Agile practices that help you get to that point.

The four strategies above: maintain the theory of the code, build less, building less and improving the design are not independent.

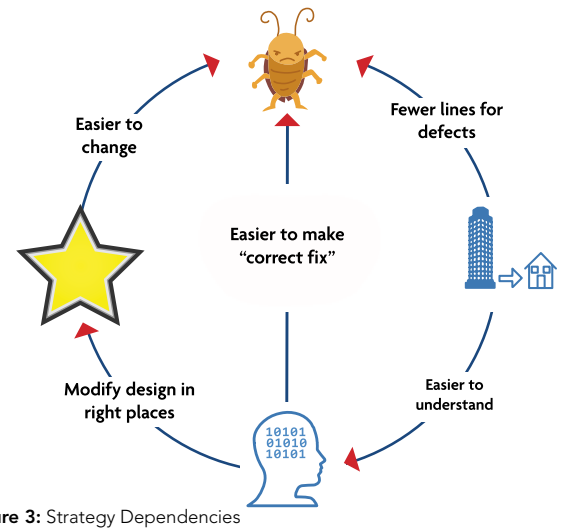
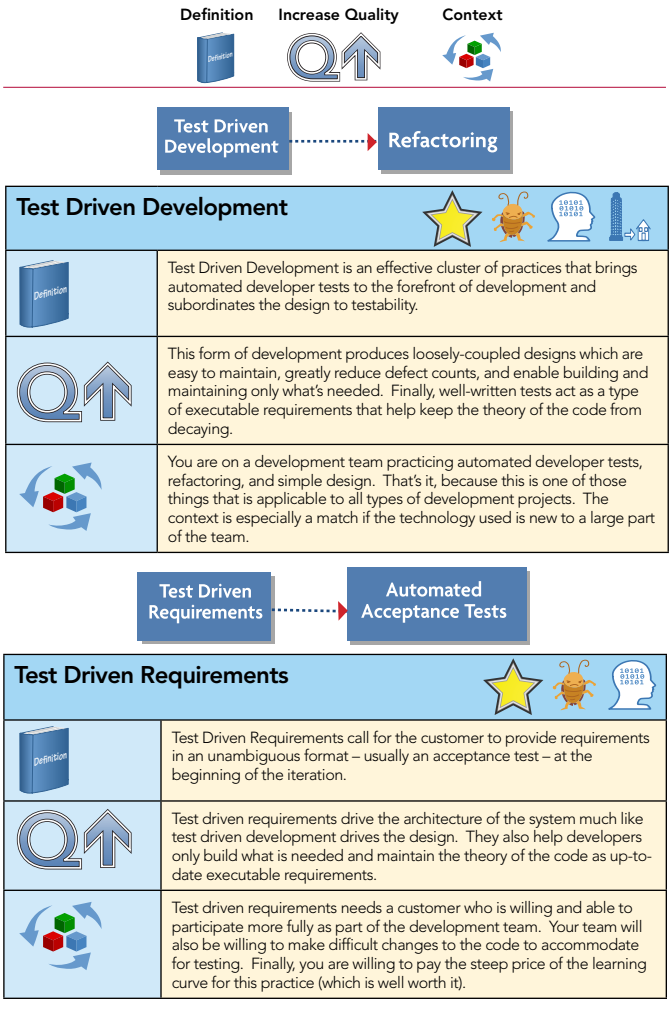


Figure 3: Strategy Dependencies

Maintaining the theory of the code makes it easier to modify the design because of a greater understanding of the existing design and also directly affects the number of defects and the difficulty in fixing those defects once found. Improving the design also makes it easier address defects by being inherently easier to change. And building less code makes it easier to understand and communicate the theory of the code and is directly related to the number of defects in the system.

THE PRACTICES



Done State

Done State

Definition	The Done State practice is a definition that a team agrees upon to precisely describe what must take place for a requirement to be considered complete.
Q↑	Defining and adhering to a done state directly affects the quality of the software by reducing defects. A properly defined done state is as close as possible to deployable software which means that defects are removed to achieve the done state. There is no partial credit with done state, either you are 100% done or you are 0% done; this mindset is crucial to successfully implementing this practice.
	You are on a development team performing iterations; this implies that you need specific, measurable goals for requirements to be met at the end. Alternatively, you may not be performing iterations and have a high rate of defects. You can agree on a done state to be met for each and every requirement and still gain the benefits of improved quality.

Automated Developer Tests
- Test First Development
- Test Last Development

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.
Q↑	Automated developer create a safety-net of tests that catch bugs early and enable the incremental improvement 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.

Automated Acceptance Tests

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.
Q↑	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. Finally, by making the codebase testable, you are implicitly reducing the coupling which often result in improved design.
	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.

Refactoring
- Collective Code Ownership
- Automated Developer Tests

Refactoring

Definition	The practice of Refactoring code changes the structure (i.e., the design) of the code while maintaining its behavior.
Q↑	Incremental improvement of design is the name of the game with refactoring; 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.

Pair Programming

Pair Programming

Definition	Two developers work together at the same computer to build a feature. One developer is the driver, and the other is the navigator; the driver is at the keyboard building the task-at-hand, and the navigator is thinking forward to design implications and reviewing the work being done. Pair programming is sometimes described as a continuous form of peer review.
Q↑	This practice improves the design and reduces the defects because two people working together to solve the same problem almost always do a better job even if they are mismatched in experience and talent. Also, because they build the software together, the theory of the code is communicated to more than one person.
	Your are on a development team where quality is near the top in business values or you are going through a period of adopting some of the more difficult practices such as test driven development. You have the ability to trade off some development speed for quality.

The remaining practices also help improve the quality software development. Because of the limited size of the Refcard, we will only summarize them below.

Continuous Integration	Continuous integration reduces the defects in a software system by catching errors early and often and enabling a stop-and-fix process. It leverages both automated acceptance tests and automated developer tests to give frequent feedback to the team and prompts removing these defects promptly.
Collective Code Ownership	Collective code ownership means that members of a development team have the right and responsibility to modify any part of the code. They get more exposure to the entire code base and are able to remove defects wherever they are found and incrementally modify the design of the system accordingly.
Evolutionary Design	Evolutionary design is the simple design practice (below) done continuously. Teams start off with a simple design and change that design only when a new requirement cannot be met by the existing design.
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 and forces regularly removing defects to reach the agreed upon done state.
Release Often	Releasing your software to your end customers as often as you can without inconveniencing them forces you to constantly have your software in releasable quality and allows you to build in smaller increments and get feedback before too much of an investment is made.
Simple Design	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. In fact, Gartner now recommends an emergent approach to enterprise architecture (http://www.gartner.com/it/page.jsp?id=1124112).
Stand Up Meeting	Stand up meetings are daily meetings for the team to synch-up and share progress and impediments daily. This helps keep the entire team aware of what is being done and where in the system.

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. Then, once you've chosen the first practices that best fit your environment, you and your team(s) will need to be aware of the mindset you'll need to get the most out of the practices you choose.

Choosing a Practice to Adopt

Choosing a practice comes down to finding the highest value practice that will fit into your context. Figure 4 contains practices that help improve the quality that your software development team(s) builds. Figure 4 will also guide you in determining which practices are most effective in increasing the quality of your software 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:

Figure 4: Steps for Choosing and Implementing Practices

For improving quality, the set of practices that will give you the most value are those nearest the top of Figure 1. Four of the practices are independent: done state, automated developer tests, automated acceptance tests, and pair programming. Consider adopting pair programming as a support practice to the other three practices, then take them on one or two at a time. Next on your list (or maybe even concurrent) to consider should be done state and automated developer tests, and then finally automated acceptance test (probably the most difficult of this set to adopt correctly).

The Mindset

Be Disciplined, Confront Issues, Respond Positively to Pain

The practices involved in improving the quality to market are some of the most difficult to do from the body of Agile practices. Things will get harder before they get easier. The first rule is to expect the difficulty, be patient, and don't stop the practices just because they uncover significant problems; be disciplined in your practice. Once you start a practice give it a chance because you will slow down and confront frustrations before speeding up. For example, pair programming is frequently seen as a waste of resources and uncomfortable to many developers who are used to (and enjoy) working alone. Consider giving it a chance by agreeing as a team to practice pair programming for a couple of months before deciding whether it is worth adopting permanently.

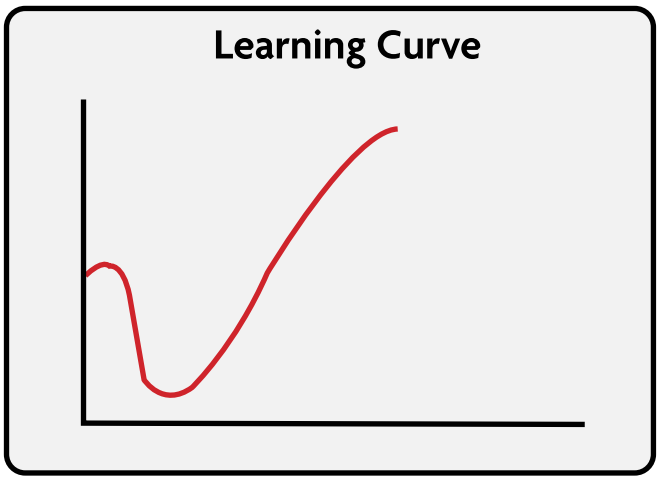


Figure 5: Learning Curve

Figure 5 The J-curve is what to expect when adopting new practices. First things will be hard and you will be less productive; stay with it and it will improve.

Confront issues when they come up instead of stopping a practice because it is 'too painful'. Deal with pain differently than you are used to; instead of discontinuing something painful, examine it and find the source. Often Agile practices will uncover problems that have always been there but have not been felt. Feeling the pain is a chance to correct a problem and improve towards your goal of increased quality. A good example of this happens when teams start adopting done states for the first time. There is no partial credit, either you are 100% done or 0% done. A team that adopts this for the first time frequently works on multiple features at a time and at the end of the iteration they have not fully completed any of the features. Therefore they are 0% done with all of their tasks. This is discouraging and painful and a common response is to stop doing the practice instead of examining the pain and looking for alternatives to correct the problems in the next iteration.

Get Good at Small Steps

Small steps are going to save your life with these practices because many are completely new ways of doing things that may slow you down and frustrate you as you are learning them. Take one practice, do it well, and do it regularly. You might consider pair programming along with any and all of the practices to make it easier and keep you on-track. How do you know you are doing a practice well? You get the value that you originally hoped to get – i.e. the quality of your software noticeably increases. You also have confronted pains and learned from them. If a practice is completely easy and comfortable from the get-go, or has not noticeably improved the quality of your work then you probably are not done yet.

Be Prepared to "Suspend Your Disbelief"

Much of what you will be doing will not make immediate sense. It will feel that you are doing things that are more trouble than they are worth. For example – writing your tests first, before writing your code in the automated developer tests practice is non-intuitive. What can you possibly gain by doing things backward? Those who have successfully adopted this practice have "suspended their disbelief" and done it anyway. After experientially learning the practice they then made their

judgments about its utility and usually kept doing it because they saw the value.

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 improve quality. 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:

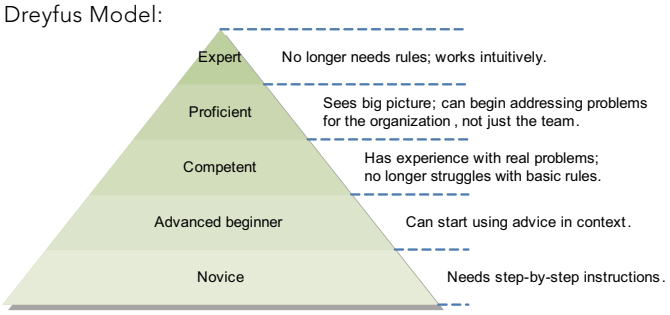


Figure 6: Dreyfus Model

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.

WHAT NEXT?

This Refcard is a quick introduction to Agile practices that can help you improve the quality of your software by reducing defects, improving design, sharing the theory of the code and building less. It includes an introduction of how 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								
Duvall, Paul, Matyas, Steve, and Glover, Andrew. (2006). Continuous Integration: Improving Software Quality and Reducing Risk. Boston: Addison-Wesley.			x	x	x			x					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

Fowler, Martin. 1999. Refactoring: Improving the Design of Existing Code. Boston: Addison-Wesley.	x			x		x				x				
Feathers, Michael. 2005. Working effectively with legacy code. Upper Saddle River, NJ: Prentice Hall.	x			x		x				x			x	
Jeffries, Ron. "Running Tested Features." http://www.xprogramming.com/xpmag/jatRtsMetric.htm.		x	x			x								
Jeffries, Ron. 2004. Extreme programming adventures in C#. Redmond, WA: Microsoft Press.	x			x						x	x	x	x	
Kerievsky, Joshua. "Don't Just Break Software, Make Software." http://www.industriallogic.com/papers/storytest.pdf .		x	x			x								
Larman, C., Agile and Iterative Development: A Manager's Guide, Boston: Addison-Wesley, 2004								x			x	x	x	
Larman, C., and Vodde, B., Scaling Lean and Agile Development, Boston: Addison-Wesley, 2009								x	x	x	x	x	x	
Martin, Robert C., Clean Code: A Handbook of Agile Software Craftsmanship, Upper Saddle River, NJ: Pearson Education. 2008.	x		x	x		x	x			x			x	
Massol, Vincent. Junit in action. Greenwich, CT: Manning Publications. 2004.				x										
Meszaros, XUnit Test Patterns: Refactoring Test Code, Boston: Addison-Wesley, 2007.				x		x								
Mugridge, R., and W. Cunningham. Fit for Developing Software: Framework for Integrated Tests. Upper Saddle River, NJ: Pearson Education. 2005.		x			x					x			x	
Poppendieck, M., and Poppendieck, T., Implementing Lean Software Development, Addison-Wesley Professional, 2006.								x	x	x	x	x	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		x

ABOUT THE AUTHORS

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.
140 Preston Executive Dr.
Suite 100
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-89-9
ISBN-10: 1-934238-89-9
50795
9 781934 238899
$7.95

Version 1.0