Compliments of IBM

IBM Limited Edition

Agile For Dummies®

Learn:

- How agile teams work and the unique approaches they utilize
- What strategies and tooling help you scale your agile practice
- How a disciplined agile approach helps you deliver working solutions faster
- Ten agile adoption mistakes and how to avoid them

Scott W. Ambler
Matthew Holitza

Agile FOR DUMMIES®

IBM LIMITED EDITION

by Scott W. Ambler and
Matthew Holitza

WILEY

John Wiley & Sons, Inc.

Agile For Dummies®, IBM Limited Edition

Published by
John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030-5774
www.wiley.com

Published by John Wiley & Sons, Inc., Hoboken, NJ

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services, please contact our Business Development Department in the U.S. at 317-572-3205. For details on how to create a custom *For Dummies* book for your business or organization, contact info@dummies.biz. For information about licensing the *For Dummies* brand for products or services, contact BrandedRights&Licenses@Wiley.com.

ISBN: 978-1-118-30506-5 (pbk)

ISBN: 978-1-118-30554-6 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

WILEY

Table of Contents

Publisher's Acknowledgments

We're proud of this book and of the people who worked on it. For details on how to create a custom *For Dummies* book for your business or organization, contact `info@dummies.biz`. For details on licensing the *For Dummies* brand for products or services, contact `BrandedRights&Licenses@Wiley.com`.

Some of the people who helped bring this book to market include the following:

Acquisitions, Editorial, and Vertical Websites

Project Editor: Carrie A. Burchfield

Development Editor: Colleen Totz-Diamond

Editorial Manager: Rev Mengle

Senior Acquisitions Editor: Katie Feltman

Business Development Representative: Sue Blessing

Custom Publishing Project Specialist: Michael Sullivan

Composition Services

Senior Project Coordinator: Kristie Rees

Layout and Graphics: Jennifer Creasey, Lavonne Roberts, Christin Swinford

Proofreader: Jessica Kramer

Publishing and Editorial for Technology Dummies

Richard Swadley, Vice President and Executive Group Publisher

Andy Cummings, Vice President and Publisher

Mary Bednarek, Executive Director, Acquisitions

Mary C. Corder, Editorial Director

Publishing and Editorial for Consumer Dummies

Kathleen Nebenhaus, Vice President and Executive Publisher

Composition Services

Debbie Stailey, Director of Composition Services

Business Development

Lisa Coleman, Director, New Market and Brand Development

Introduction

Agile development principles have gone from something used only by cutting-edge teams to a mainstream approach used by teams large and small for things as varied as the following:

- Startup software development projects
- Enterprise-sized development efforts
- Complex, large-scale systems engineering initiatives (such as the electronics in the cars you drive and the airplanes you fly in)
- Legacy systems (which means systems that have been around for a while, such as mainframe)
- Embedded, real-time systems (such as pacemakers or life-support systems)
- High-compliance environments (such as healthcare, insurance, or banking)

About This Book

Welcome to *Agile For Dummies*, IBM Limited Edition. You've probably been hearing about agile for a long time, which isn't surprising. If you're not using agile methods already though, or if you've only been exposed to agile on small projects here and there, you may wonder how to get started with it. Can agile ever work in your environment? Relax. This book is here to help.

Foolish Assumptions

Many people and teams can benefit most from this book, but we took the liberty to assume the following:

- You're looking to pilot a project using agile. You're a project manager, a technical lead, or an aspiring product owner who wants to adopt agile practices but isn't sure where to start.

- You may have tried out some agile practices in an ad hoc manner, and you encountered some difficulties. Don't worry; many teams experience some missteps when first moving to agile.
- You've had some project success, and you're looking to grow the agile practice beyond your team. You're looking for ways to coordinate multiple teams with the same outcomes you experienced on your small team.
- You want to try agile, but your environment has complexities that need to be addressed. Maybe you have globally distributed teams or are subject to regulatory compliance mandates. You're wondering if agile practices can be effective in this environment.

But, no matter who you are, this book helps explain and reinforce the successful software development practices available today. There's great food for thought here, even if your current team or organization isn't ready to make the agile leap just yet.

Icons Used in This Book

Sometimes, information deserves special attention. The icons in this book identify such information for you. Here's a brief explanation for each icon so you'll recognize them when they turn up.

TIP

The Tip icon points to information that describes a special benefit of working with agile.

WARNING!

This icon identifies pitfalls and problems to avoid in your agile journey.

REMEMBER

The Remember icon presents you with tidbits that you won't want to forget after you finish the book.

TECHNICAL STUFF

This icon points out content that gets a little deeper into the weeds of agile development or explains agile jargon you may encounter. The info isn't crucial to your journey, so you can skip it if you like.

Chapter 1

Getting the ABCs of Agile

In This Chapter

- Understanding where software development has been
- Dissecting the Agile Manifesto
- Defining agile today

If you're reading this book, you've seen software being made. Regardless of your role on the project, you know it's not a perfect process. You know it's hard to do well. Software development doesn't face problems for lack of trying or for lack of brain power. People in the software business tend to be some very bright, hardworking people. They don't plan to deliver software over budget, past deadline, and with defects (or without features people need). So what's been at the root of all these issues?

Agile is an attempt to make the process of software development better and more effective, and it's seen increasing popularity and success. In this chapter, you discover how agile is an incremental, iterative approach to delivering high-quality software with frequent deliveries to ensure value throughout the process. It places a high value on individuals, collaboration, and the ability to respond to change.

Looking Back at Software Development Approaches

To understand how agile has been successful, take a moment to look back at some of the software development approaches that have gone before it. As software development has evolved over the last 70-plus years, it has had several dominant models or methodologies. Each had reasons for coming into being, and really no model is used as is; models are almost always tailored to suite their unique needs. Each model has its benefits and drawbacks.

A *model* or *methodology* is just a fancy word for a process or approach to creating software, usually with specific steps or phases used to manage it. The common thinking is that it's better to have an approach in mind than to have none at all.

Agile itself is just a newer, best-of-breed collection of methodologies used to develop and maintain software.

Code-and-Fix/Big Bang development

The original approach to software development was *Code-and-Fix development,* where you wrote some code and then fixed things that were incorrect as you found them (or when others found them for you). Software was delivered all at once, in a "big bang" — hence the term, *Big Bang* — and software developers waited to find out what they may have done wrong, both in the form of outright errors and in the form of not meeting user needs or expectations.

This approach is a challenging way to deliver software even on the smallest, simplest scale. As the amount of code grew and became more complicated, this method was obviously too risky and expensive an approach to software development. Something better was needed.

Waterfall

To overcome the problems with the Big Bang/Code-and-Fix model (see the preceding section), software development began to take on specific stages:

1. Requirements.
2. Design.
3. Development.
4. Integration.
5. Testing.
6. Deployment.

This kind of sequential, stage-based approach became popular in the mid-1950s. Not until 1970 did this model become known as the Waterfall model — described as a waterfall because after finishing any one phase and moving on to the next, moving backward to make changes or corrections was very difficult (water going the wrong way up a waterfall is pretty hard).

Waterfall presented a step forward from Code-and-Fix and is still used in many industries to this day. Despite wide adoption and continued use, however, the model has problems:

- **Schedule risk:** Unless the system being designed is already well understood, Waterfall has a higher-than-normal risk of not meeting schedule. Business needs may change during a long project, and developers may be asked to squeeze in "one more thing," which can have an incremental impact on test and deployment teams. These changes in scope add up — an effect known as *scope creep.*
- **Limited flexibility:** The Waterfall model locks down requirements very early in the process, producing little wiggle room to add late discoveries or needed changes throughout the process. This is why scope creep (see the preceding bullet) is problematic with this model.

  In addition, with testing held until the end of the process, defects in the form of code errors are discovered long after the developers have written the code, making it not only harder for the developer to find and fix but also can potentially trigger the need for major design changes toward the end of the project.
- **Reduced customer involvement:** Involvement with customers in a Waterfall approach is limited and can cause companies to miss the market need. Studies show that customers either always or often use the capabilities of a typical system only 20 percent of the time.

The Spiral model

By the mid-1980s, developers were experimenting with alternatives to Waterfall (see the preceding section). Iterative and incremental approaches became popular:

- An *incremental approach* regularly delivers working code in small chunks.
- An *iterative approach* plans on learning from feedback on the deliveries and sets aside time to use this feedback to make improvements.

An incremental approach can be iterative because the small chunks result in feedback; feedback leads to changes in the chunks already delivered, and shapes future direction. This process happens much more rapidly with incremental delivery than waiting until the end of a long-release cycle. Besides, if you wait until the end of a long-release cycle, the users already have a long list of new features they need.

The Spiral model, introduced in 1988, was a landmark software development methodology. It used prototyping and incremental delivery process to manage project risk. It was designed to be especially effective for systems that had a high level of uncertainty around what exactly needed to be built. These kinds of projects struggle in the Waterfall approach (see the preceding section) because the detailed specifications created ahead of time ran aground in the project when the problem space was better understood.

The Spiral model — both incremental and iterative — delivered the final version of working software, and this archetype followed something closer to the Waterfall model. But it got its name because of the way the model conceptualized its incremental deliveries and the iterative work following a delivery.

In the 1990s, more lightweight approaches gained popularity in an effort to come up with an effective alternative to Waterfall. RAD, or Rapid Application Development, relied on building prototypes to allow requirements to emerge and elicit frequent feedback. The Scrum and XP (Extreme Programming) methodologies took root, both placing a heavy focus on short iterations to allow frequent delivery of software. In general, to serve business needs and improve software project success rates.

Introducing the Agile Manifesto

In February of 2001, a group of developers interested in advancing lightweight development methodologies got together to talk about their views and to find common ground, and agile was born. The developers who created agile understood the importance of creating a model in which each iteration in the development cycle "learned" from the previous iteration. The result was a methodology that was more flexible, efficient, and team-oriented than any of the previous models.

All the agile methods look to the Agile Manifesto and 12 core principles for guidance. The adherence to the guidance provided by the manifesto and principles is what makes a software development team agile, not a specific process, tool, label.

The Manifesto

The *Manifesto for Agile Software Development* is a compact 68 words (now that's lightweight!) that stresses four values.

> **Manifesto for Agile Software Development***
>
> We are uncovering better ways of developing
> software by doing it and helping others do it.
> Through this work we have come to value:
>
> **Individuals and interactions** over processes and tools
> **Working software** over comprehensive documentation
> **Customer collaboration** over contract negotiation
> **Responding to change** over following a plan
>
> That is, while there is value in the items on
> the right, we value the items on the left more.

** Agile Manifesto Copyright 2001: Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas*
This declaration may be freely copied in any form, but only in its entirety through this notice.

No boxes with arrows, no swim-lane diagrams here. The Agile Manifesto is straightforward, but don't let the brevity fool you. This is powerful stuff. The following sections break down the components of the manifesto.

Individuals and interactions over processes and tools

Recognizing that software is made by people, not processes or tools, agile places a higher premium on people working together effectively. Processes and tools can aid in that but can't replace it.

Working software over comprehensive documentation

Valuing working software over comprehensive documentation stands in stark opposition to the Waterfall model. A highly detailed, accurate, and comprehensive specification document is of no value if it doesn't result in working software that meets users' needs. Working software may involve documentation, but agile only uses it in service to creating working software, not as an end (almost) unto itself.

Customer collaboration over contract negotiation

While agile isn't ignoring the reality of contracts, it values active collaboration throughout the software development process as a better way to deliver value instead of a carefully worded contract. A contract is no proxy for actual communication when you're doing something as challenging as creating software.

Responding to change over following a plan

Except for the most incredibly simple systems, it's massively difficult to think of every feature, every piece of data, and every possible use case for software. That means, in a collaborative process with the customer, a lot is discovered during the process of developing software. Also, the world changes pretty fast: Business needs and priorities can shift in the months or even years it can take for a large system to be fully built. Agile values the ability to change in response to new discoveries and needs over sticking to a plan created before everything was known.

The 12 principles that drive the Agile Manifesto

The people who wrote the Agile Manifesto later assembled 12 principles that inform and reinforce the manifesto. These further illuminate the things agile values in software development.

The Agile Manifesto follows these principles:

1. The highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity — the art of maximizing the amount of work not done — is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective and then tunes and adjusts its behavior accordingly.

The Agile Manifesto and the principles behind it are published at `http://www.agilemanifesto.org`.

Redefining Today's Agile

Since the time when the Agile Manifesto was drafted, agile has grown in popularity, and its use has been extended to increasingly larger organizations and more complex projects.

Growing popularity

Agile is a widely accepted and adopted approach to software development. Hundreds of books on agile exist, covering everything from how to manage agile development to how to apply it in specific industries to how to apply it with specific programming languages. You can attend agile training courses and be an agile coach. Practitioners old and new are blogging about their challenges, discoveries, and successes.

As businesses gain greater competitive advantage by being able to move and change faster, agile approaches are being used to develop many kinds of systems, including web-based applications, mobile applications, business intelligence (BI) systems, life-critical systems, and embedded software. Agile approaches are adopted by varied organizations, including financial companies, retailers, healthcare organizations, manufacturers, and government agencies — including defense.

Growing scalability

As the advantages of agile become clear and the number of success stories grows, more and more teams have been attempting to scale agile practices to ever larger and more complex software development projects. Teams are finding success with hybrid approaches that stay true to core agile principles and extend them beyond the software development stage to the entire software life cycle. Chapter 6 covers the scaling of agile practices.

Chapter 2

Understanding Agile Roles

In This Chapter

- Discovering the primary roles on agile teams
- Taking a look at additional team positions

On a disciplined agile project, any given person can be in one or more roles, and he can also change his role(s) over time. Roles aren't positions, nor are they meant to be. Teams practicing agile adapt to styles that suit their needs and may vary in their exact execution. However, all tend to have the same kinds of roles and very similar processes at their core. Agile deemphasizes specialized roles and considers all team members equal — everyone works to deliver a solution regardless of their job. With the exception of stakeholder, everyone's effectively in the role of team member. In this chapter, you explore the roles, arming you with a basis for understanding any style you may experience.

Being a Stakeholder

A *stakeholder* is someone who's financially impacted by the outcome of the solution and is clearly more than an end-user. A stakeholder may be one of the following:

- A direct or indirect user
- A manager of users
- A senior manager
- An operations or IT staff member
- The "gold owner" who funds the project
- An auditor(s)

- Your program/portfolio manager
- A developer(s) working on other systems that integrate or interact with the one under development
- A maintenance professional(s) potentially affected by the development and/or deployment of a software project

Representing Stakeholders: The Product Owner

The *product owner* is the team member who speaks as the "one voice of the customer." This person represents the needs and desires of the stakeholder community to the agile delivery team. He clarifies any details regarding the solution and is also responsible for maintaining a prioritized list of work items that the team will implement to deliver the solution. While the product owner may not be able to answer all questions, it's his responsibility to track down the answer in a timely manner so the team can stay focused on its tasks. Each agile team, or subteam in the case of large projects organized into a team of teams, has a single product owner.

The product owner has the following additional roles:

- Communicates the project status and represents the work of the agile team to key stakeholders
- Develops strategy and direction for the project and sets long- and short-term goals
- Understands and conveys the customers' and other business stakeholders' needs to the development team
- Gathers, prioritizes, and manages product requirements
- Directs the product's budget and profitability
- Chooses the release date for completed functionality
- Answers questions and makes decisions with the development team
- Accepts or rejects completed work during the sprint
- Presents the team's accomplishments at the end of each sprint

Being a Team Member

The role of *team member* focuses on producing the actual solution for stakeholders. Team members perform testing, analysis, architecture, design, programming, planning, estimation, and many more activities as appropriate throughout the project.

REMEMBER

Not every team member has every single skill (at least not yet), but they have a subset of them and strive to gain more skills over time. Team members identify, estimate, sign-up for, and perform tasks and track their completion status.

Assuming the Team Lead

The *team lead* guides the team in performing management activities instead of taking on these responsibilities herself. She's a servant-leader to the team, upholding the conditions that allow the team's success. This person is also an agile coach who helps keep the team focused on delivering work items and fulfilling its iteration goals and commitments to the product owner. The team lead facilitates communication, empowers the team to self-optimize its processes, ensures that the team has the resources it needs, and manages issue resolution in a timely manner.

TIP

While an experienced team lead brings skills to a new team, this person can't be a true coach without mentoring. So for teams new to agile, you may have a part-time experienced coach working with the team for a few iterations.

Acting As the Architecture Owner

Architecture is a key source of project risk, and someone has to be responsible for ensuring the team mitigates this risk. The *architecture owner* is the person who owns the architecture decisions for the team and who facilitates the creation and evolution of the overall solution design.

You may not have to formally designate a team member as an architecture owner on small teams, because the person in the role of team lead often is the architecture owner, too.

Stepping Up As an Agile Mentor

A mentor is a great idea for any area in which you want to develop new expertise. The *agile mentor,* sometimes called an *agile coach,* implements agile projects and shares that experience with a project team. He provides valuable feedback and advice to new project teams and to project teams that want to perform at a higher level. On an agile project, the agile mentor is

- A coach only and isn't part of the team
- Often from outside the organization and objective in guidance without personal or political considerations
- Experienced in implementing agile techniques and running agile projects in different situations

Looking at Agile Secondary Roles

Your project may include the need to add some or all the following roles:

- **Domain expert:** Someone with deep business/domain knowledge beyond that of the product owner.
- **Specialist:** Although most agile team members are generalizing specialists, sometimes, particularly at scale, specialists such as business analysts or even project/program managers are required.
- **Technical expert:** Technical experts are brought in as needed to help the team overcome a difficult problem and to transfer their skills to one or more developers on the team.
- **Independent tester:** Some agile teams are supported by an independent test team working in parallel that validates work throughout the life cycle.
- **Integrator:** For complex environments, your team may require one or more people in the role of integrator responsible for building the entire system from its various subsystems.

Chapter 3

Getting Started with Agile

In This Chapter

- Looking at agile planning practices
- Managing and tracking your progress
- Reflecting for future improvement

In this chapter, you explore how the agile team organizes the software development process. Everything the stakeholders want in their software is broken down into small chunks, ranked, worked on in priority order over short iterations (typically one to four weeks), reviewed for approval, and delivered to production. This process repeats until the prioritized list is finished, called a *release.* An agile team expects re-prioritization, additions to the list, and subtractions from the list throughout the process but embraces them as a means to deliver the most value and the best possible solution.

Agile Planning

Teams following agile software development methods typically divide their release schedule into a series of fixed-length development iterations of two to four weeks — shorter is generally better than longer. Planning involves scheduling the work to be done during an iteration or release and assigning individual work items to members of the team.

To be effective and to reflect the team's position and direction, plans need to be accessible to everyone on the team and to change dynamically over the course of the iteration. Automated planning tools are available to facilitate this process, or you can have at it the old-fashioned way with whiteboards, index cards, or sticky notes.

During an agile project, planning occurs at three levels:

- **Release planning:** Release plans contain a release schedule for a specific set of features. The product owner creates a release plan at the start of each release.
- **Iteration planning:** Team members gather at the beginning of the iteration (referred to as a *sprint* in the Scrum methodology) to identify the work to be done during that iteration. This is referred to as *self-organization.*
- **Daily planning:** Development teams begin each day with standup meetings to plan the day. These meetings are generally 5 to 15 minutes long.

Attending the Daily Coordination Meeting

On agile projects, you make plans throughout the entire project daily. Agile development teams start each workday with a 15-minute (or less) daily coordination meeting to note completed items, to identify impediments, or roadblocks, requiring team lead involvement, and to plan their day.

In the daily coordination meeting, often called a daily standup meeting, each development team member makes the following three statements:

- Yesterday, I completed *[state items completed]*.
- Today, I'm going to take on *[state task].*
- My impediments are *[state impediments, if any].*

Daily coordination meetings can actually be quite fun when using the right tools. For example, the Taskboard view in Rational Team Concert (RTC) is a capability that arranges all work items as cards on a board with multiple columns. For more info on the Taskboard view, see Chapter 8.

Creating User Stories

When stakeholders realize the need for a new software system, feature set, or application, the agile process begins with the product owner defining what the software will do and what services it provides to its users. Instead of following the more traditional process of product managers and business analysts writing lengthy requirements or specifications, agile takes a lightweight approach of writing down brief descriptions of the pieces and parts that are needed. These become work items and are captured in the form of *user stories*. A *user story* is a simple description of a product requirement in terms of what that requirement must accomplish for whom. Your user story needs to have, at a minimum, the following parts:

- **Title:** <*a name for the user story*>
- **As a** <*user or persona*>
- **I want to** <*take this action*>
- **So that** <*I get this benefit*>

The story should also include validation steps — steps to take to know that the working requirement for the user story is correct. That step is worded as follows:

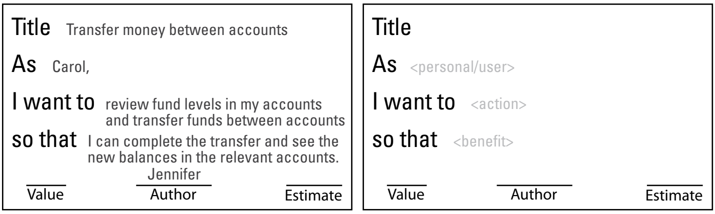
- **When I** <*take this action*>, **this happens** <*description of action*>

User stories may also include the following:

- **An ID:** A number to differentiate this user story from other user stories.
- **The value and effort estimate:** *Value* is how beneficial a user story is to the organization creating that product. *Effort* is the ease or difficulty in creating that user story.
- **The person who created the user story:** Anyone on the project team can create a user story.

  For user stories that are too large to be completed in a single iteration or sprint, some teams use Epics. *Epics* are basically a higher-level story that's fulfilled by a group of related user stories.

Figure 3-1 shows a typical user story card, back and front. The front has the main description of the user story. The back shows how you confirm that the requirement works correctly after the development team has created the requirement.



Title Transfer money between accounts

As Carol,

I want to review fund levels in my accounts and transfer funds between accounts

so that I can complete the transfer and see the new balances in the relevant accounts.

Value

Jennifer
Author

Estimate

Title

As <personal/user>

I want to <action>

so that <benefit>

Value

Author

Estimate

Figure 3-1: Card-based user story example.

The product owner gathers and manages the user stories. However, the development team and other stakeholders also will be involved in creating and decomposing user stories.

User stories aren't the only way to describe product requirements. You could simply make a list of requirements without any given structure. However, because user stories include a lot of useful information in a simple, compact format, they tend to be very effective at conveying exactly what a requirement needs to do.

Estimating Your Work

When the product owner sets the scope of an iteration, she needs to know that the scope is the right size — that there isn't too much work to get done in the iteration. Like any other developers, agile team members estimate their work. Unlike other developers, agile team members estimate in something called *points*. Points represent a size-based, complexity-based approach to estimation. Points are assigned in whole numbers (1, 2, 3, and so on with no fractions or decimals) and represent relative sizes and complexity of work items. Small and simple tasks are one point tasks, slightly larger/more complex tasks are two point tasks, and so on.

Points are kind of like t-shirt sizes. There are small, medium, large, extra large, and potentially other sizes (extra small and extra extra-large). These sizes are relative — no regulation dictates how much larger medium is compared to small. Sizes vary a bit from manufacturer to manufacturer. T-shirt sizing succeeds not because of high precision — it's pretty imprecise, actually — but through its general accuracy.

Planning Poker

As you refine your requirements, you need to refine your estimates as well. *Planning Poker* is a technique to determine user story size and to build consensus with the development team members. Planning poker is a popular and straightforward approach to estimating story size.

To play planning poker, you need a deck of cards with point values on them. There are free online planning poker tools and mobile apps, or you can make your own with index cards and markers. The numbers on the cards are usually from the Fibonacci sequence.

Only the development team plays estimation poker. The team lead and product owner don't get a deck and don't provide estimates. However, the team lead can act as a facilitator, and the product owner reads the user stories and provides details on user stories as needed.

In practice, one team's three-point size estimate for a work item may correlate to another team's two-point estimate for an identical work item. Teams need only agree on what size and complexity corresponds to what point count and remain internally consistent in their use.

You may be tempted to equate points to hours. Don't do it! An agile team gains consistency by using point values that don't vary based on the ability of the person doing the work. A five-point story has the same size and complexity on a given team regardless of who does the work. The team still accommodates faster and slower team members in the number of points assigned in a single iteration, but the value delivered to the product owner and stakeholders (measured by size and complexity) remains consistent. If you want to track effort and hours, keep it separate from points.

Tracking Velocity

At the end of each iteration, the agile team looks at the requirements it has finished and adds up the number of story points associated with those requirements. The total number of completed story points is the team's *velocity,* or work output, for that iteration. After the first few iterations, you'll start to see a trend and will be able to calculate the average velocity.

The average velocity is the total number of story points completed, divided by the total number of iterations completed. For example, if the development team's velocity was . . .

Iteration 1 = 15 points

Iteration 2 = 19 points

Iteration 3 = 21 points

Iteration 4 = 25 points

. . . your total number of story points completed is 80. Your average velocity is 20 to 80 story points divided by four iterations. After you've run an iteration and know the team's velocity, you can start forecasting the remaining time on your project.

Measuring Progress with Burndown Reports

Burndown reports track the number of points completed and are used for monitoring single iterations, releases, and the entire project backlog. They get their name from the concept that the iteration or project backlog gets "burned down," completed, and cleared away. Burndown reports show progress, reflecting both the value delivered (in points) and the team's velocity. See Figure 3-2 for a simple project burndown report, with iterations along the X-axis and the points in the entire product backlog on the Y-axis.

You may also find that you need to re-estimate the backlog. As the team progresses through a project, it discovers more about that project, its technology, and the business concerns the project addresses. Greater understanding leads to better estimates, so the points associated with some work items in the backlog can become more accurate over time.

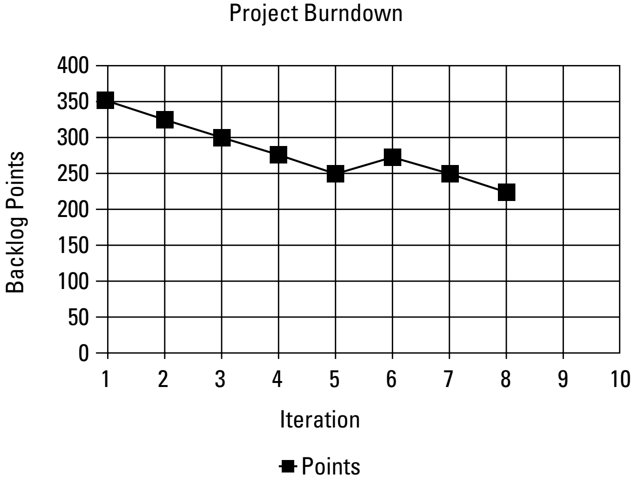


Figure 3-2: A project burndown report.

Test-Driven Development

Testing occurs throughout the agile life cycle. While an independent tester may or may not be a member of the cross-functional team, developers are expected to test their code. Some of you are saying, "Hold on, developers can't test their own work. They don't want to. They're not good at it!" But trust me; it's not as bad as you think. In fact, it's not bad at all.

Agile teams use two common practices to handle their testing:

- Test-Driven Development (TDD)
- Automated unit tests

When used together, they're a powerful force.

Performing TDD means that before the developer writes a piece of code, she first writes a small test that validates the code she's about to write. She runs the test to make sure it fails and then writes the code that makes the test pass. This may seem odd, but with practice it's much more efficient than writing a lot of code, running it, and going back later to figure out everywhere it's broken (a process known as *debugging*). This process puts the developer in a testing mindset while writing code, which leads to higher-quality code.

When a developer executes a small test against code she's written, it's called a *unit test.* When these tests are run in batches all at once (automated), they become very powerful. Agile teams write a lot of unit tests, automate them, and run them frequently against the code they write as individuals and against their combined code that makes up the entire application. Running automated unit tests frequently against the code reveals problems quickly so they can be addressed quickly. This approach finds defects long before they'd ever reach a traditional test cycle, which means higher-quality applications.

Continuous Integration and Deployment

Continuous integration (CI) is the practice of regularly integrating and testing your solution to incorporate changes made to its definition. Changes include updating the source code, changing a database schema, or updating a configuration file. Ideally, when one or more changes are checked into your configuration management system, the solution should be rebuilt (recompiled), retested, and any code or schema analysis performed on it. Failing that, you should strive to do so at least once if not several times a day.

Continuous deployment (CD) enhances CI by automatically deploying successful builds. For example, when the build is successful on a developer's workstation, she may automatically deploy her changes to the project integration environment, which would invoke the CI system there. A successful integration in that environment could trigger an automatic deployment into another environment and so on.

On a developer's workstation, the integration job could run at specific times, perhaps once an hour, or better every time that she checks in something that is part of the build. This whole process of continuously integrating a developer's code with the rest of a team's code in and then running automated test regressions in an integration environment is a critical part of agile done right. CI ensures high-quality working software at all times, and CD ensures that the software is running in the right place.

Presenting Results at the Iteration Review

The *iteration review,* or *sprint review* in Scrum, is a meeting to review and demonstrate the user stories that the development team completed during the iteration. The iteration review is open to anyone interested in reviewing the iteration's accomplishments. This means that all stakeholders get a chance to see progress on the product and provide feedback.

Preparation for the iteration review meeting should not take more than a few minutes. Even though the iteration review might sound formal, the essence of showcasing in agile is informality. The meeting needs to be prepared and organized, but that doesn't require a lot of flashy materials. Instead, the iteration review focuses on demonstrating what the development team has done.

Collecting Feedback in the Iteration Review Meeting

Gather iteration review feedback informally. The product owner or team lead can take notes on behalf of the development team, as team members often are engaged in the presentation and resulting conversation. New user stories may come out of the iteration review. The new user stories can be new features altogether or changes to the existing code.

In the first couple of iteration reviews, the team lead may need to remind stakeholders about agile practices. Some people hear the word *demonstration* and immediately expect fancy slides and printouts. The team lead has a responsibility to manage these expectations and uphold agile values and practices.

The product owner needs to add any new user stories to the product backlog and rank those stories by priority. The product owner also adds stories that were scheduled for the current iteration, but not completed, back into the product backlog, and ranks those stories again based on the most recent priorities. The product owner needs to complete updates to the product backlog in time for the next iteration planning meeting.

Learning and Improving at the Iteration Retrospective

After the iteration review is over (see the preceding section), the iteration retrospective begins. The *iteration retrospective* is a meeting where the team lead, the product owner, and the development team discuss how the iteration went and what they can do to improve the next iteration. If the team wants to, other stakeholders can attend as well. If the team regularly interacts with outside stakeholders, and it should, then those stakeholders' insights can be valuable.

TIP

You may want to take a break between the iteration review and the iteration retrospective. The team needs to come into the retrospective ready to inspect its processes and present ideas for adaptation.

The goal of the iteration retrospective is to continuously improve your processes. Improving and customizing processes according to the needs of each individual team increases team morale, improves efficiency, and increases *velocity* — work output.

Agile approaches quickly reveal problems within projects. Data from the iteration backlog shows exactly where the development team has been slowed down, so the product owner should revisit the backlog to prepare for the next iteration. Have priorities shifted? Have important new issues appeared? The product owner has to actively manage the backlog in preparation for the next iteration.

Chapter 4

Choosing an Agile Approach

In This Chapter

- ▶ Understanding various agile approaches
- ▶ Looking at the strengths and weakness of each approach

You can use several agile methods as a base to tailor a strategy to meet the unique needs of your situation. This chapter discusses these varied approaches.

Scrum: Organizing Construction

Scrum is the most popular approach to agile software development. With this approach, any adjustments the development team makes to any aspect of the project is based on experience, not theory. Scrum provides four deliverables:

- **Product backlog:** The full list of requirements that define the product
- **Sprint backlog:** The list of requirements and associated tasks in a given sprint (Scrum calls iterations *sprints*)
- **Burndown charts:** Visual representations of the progress within a sprint and within the project as a whole.
- **Shippable functionality:** The usable product that meets the customer's business goals

Five practices, covered in detail in Chapter 3, are key to Scrum. They are release planning, sprint planning, the daily scrum meeting, the sprint review meeting, and the sprint retrospective.

XP: Putting the Customer First

A popular approach to product development, specific to software, is *Extreme Programming* (XP). The focus of XP is customer satisfaction. XP teams achieve high customer satisfaction by developing features when the customer needs them. New requests are part of the development team's daily routine, and the team must deal with requests whenever they crop up. The team organizes itself around any problem that arises and solves it as efficiently as possible. The following are XP practices:

- **Coding standard:** Team members should follow established coding guidelines and standards.
- **Collective ownership:** Team members may view and edit other team members' code or any other project artifact. Collective ownership encourages transparency and accountability for work quality.
- **Continuous integration:** Team members should check in changes to their code frequently, integrating the system to ensure that their changes work, so the rest of the team is always working with the latest version of the system.
- **Test-Driven Development (TDD):** In TDD the first step is to quickly code a new test — basically just enough code for the test to fail. This test could either be high-level acceptance or a more detailed developer test. You then update your functional code to make it pass the new test, get your software running, and then iterate.
- **Customer tests:** Detailed requirements are captured just-in-time (JIT) in the form of acceptance tests (also called story tests).
- **Refactoring:** Refactoring is a small change to something, such as source code, your database schema, or user interface, to improve its design and make it easier to understand and modify. The act of refactoring enables you to evolve your work slowly over time.
- **Pair programming:** In this practice, two programmers work together on the same artifact at the same time. One programmer types the code while the other programmer looks at the bigger picture and provides real-time code review.

- **Planning game:** The purpose of the planning game is to guide the product into successful delivery. This includes high-level release planning to think through and monitor the big issues throughout the project as well as detailed JIT iteration/sprint planning.
- **Simple design:** Programmers should seek the simplest way to write their code while still implementing the appropriate functionality.
- **Small releases:** Frequent deployment of valuable, working software into production is encouraged. Frequent deployments build confidence in the team and trust from the customer.
- **Sustainable pace:** The team should be able to sustain an energized approach to work at a constant and gradually improving velocity.
- **Whole team:** Team members should collectively have all the skills required to deliver the solution. Stakeholders or their representatives should be available to answer questions and make decisions in a timely manner.

Lean Programming: Producing JIT

Lean has its origins in manufacturing. In the 1940s in Japan, a small company called Toyota wanted to produce cars for the Japanese market but couldn't afford the huge investment that mass production requires. The company studied supermarkets, noting how consumers buy just what they need, because they know there will always be a supply, and how the stores restock shelves only as they empty. From this observation, Toyota created a JIT process that it could translate to the factory floor.

The result was a significant reduction in inventory of parts and finished goods and a lower investment in the machines, people, and space. The JIT process gives workers the ability to make decisions about what is most important to do next. The workers take responsibility for the results. Toyota's success with JIT processes has helped change mass manufacturing approaches globally.

The seven principles of lean manufacturing can be applied to optimize the whole IT value stream. The lean software development principles are eliminate waste, build in quality, create knowledge, defer commitment, deliver quickly, respect people, and optimize the whole.

Kanban: Improving on Existing Systems

The *Kanban method* is a lean methodology, describing techniques for improving your approach to software development. Two Kanban principles critical to success are

- **Visualizing workflow:** Teams use a Kanban board (often a whiteboard, corkboard, or electronic board) that displays *kanbans* (indications of where in the process a piece of work is). The board is organized into columns, each one representing a stage in the process, a work buffer, or queue; and optional rows, indicating the allocation of capacity to classes of service. The board is updated by team members as work proceeds, and blocking issues are identified during daily meetings.
- **Limit work in progress (WIP):** Limiting WIP reduces average lead time, improving the quality of the work produced and increasing overall productivity of your team. Reducing lead time also increases your ability to deliver frequent functionality, which helps build trust with your stakeholders. To limit WIP, understand where your blocking issues are, address them quickly, and reduce queue and buffer sizes wherever you can.

Agile Modeling

Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and lightweight manner. AM was purposely designed to be a source of strategies that can be tailored into other base processes.

With an Agile Model Driven Development (AMDD) approach, you typically do just enough high-level modeling at the beginning of a project to understand the scope and potential architecture of the system. During construction iterations you do modeling as part of your iteration planning activities and then take a JIT model storming approach where you model for several minutes as a precursor to several hours of coding. AMDD recommends that practitioners take a test-driven approach to development although doesn't insist on it.

The Agile Modeling practices include the following:

- **Active stakeholder participation:** Stakeholders (or their representatives) provide info, make decisions, and are actively involved in the development process.
- **Architecture envisioning:** This practice involves high-level architectural modeling to identify a viable technical strategy for your solution.
- **Document continuously:** Write documentation for your deliverables throughout the life cycle in parallel to the creation of the rest of the solution. Some teams choose to write the documentation one iteration behind to focus on capturing stable information.
- **Document late:** Write deliverable documentation as late as possible to avoid speculative ideas likely to change in favor of stable information.
- **Executable specifications:** Specify detailed requirements in the form of executable customer tests and your detailed design as executable developer tests.
- **Iteration modeling:** Iteration modeling helps identify what needs to be built and how.
- **Just barely good enough artifacts:** A model needs to be sufficient for the situation at hand and no more.
- **Look-ahead modeling:** Invest time modeling requirements you intend to implement in upcoming iterations. Requirements near the top of your work item list are fairly complex so explore them before they're popped off the top to reduce overall project risk.
- **Model storming:** Do JIT modeling to explore the details behind a requirement or to think through a design issue.
- **Multiple models:** An effective developer has a range of models in his toolkit, enabling him to apply the right model for the situation at hand.

- **Prioritized requirements:** Implement requirements in priority order, as defined by your stakeholders.
- **Requirements envisioning:** Invest your time at the start of an agile project to identify the scope of the project and create the initial prioritized stack of requirements.
- **Single-source information:** Capture info in one place only.
- **TDD:** Quickly code a new test and update your functional code to make it pass the new test.

Unified Process (UP)

The *Unified Process* (UP) uses iterative and incremental approaches within a set life cycle. UP focuses on the collaborative nature of software development and works with tools in a low-ceremony way. It can be extended to address a broad variety of project types, including OpenUP, Agile Unified Process (AUP), and Rational Unified Process (RUP).

UP divides the project into iterations focused on delivering incremental value to stakeholders in a predictable manner. The iteration plan defines what should be delivered within the iteration, and the result is ready for iteration review or shipping. UP teams like to self-organize around how to accomplish iteration objectives and commit to delivering the results. They do that by defining and "pulling" fine-grained tasks from a work items list. UP applies an *iteration life cycle* that structures how micro-increments are applied to deliver stable, cohesive builds of the system that incrementally progress toward the iteration objectives.

UP structures the project life cycle into four phases: Inception, Elaboration, Construction, and Transition. The project life cycle provides stakeholders and team members with visibility and decision points throughout the project. This enables effective oversight and allows you to make "go or no-go" decisions at appropriate times. A project plan defines the life cycle, and the end result is a released application.

Chapter 5

Using Disciplined Agile Delivery (DAD)

In This Chapter

- Listing the characteristics of DAD
- Getting to know the DAD life cycle

Disciplined Agile Delivery (DAD) is a process framework that encompasses the entire solution life cycle, acting like an umbrella over best practices from many agile approaches (for the varied approaches, flip back to Chapter 4). DAD sees the solution through initiation of the project through construction to the point of releasing the solution into production.

TIP

When you adopt agile, you're likely to encounter a few speed bumps along the way, so it's important to collaborate with other agile practitioners to get ideas on what methods to start with, how to grow your practice, and what common pitfalls to avoid (see Chapter 9). To get started, join an online agile community, such as the Agile Transformation Zone. Visit `http://ibm.co/getagile`.

At this point, DAD works splendidly to help you avoid the pitfalls, all while providing a comprehensive life cycle management solution.

Understanding the Attributes of DAD

The DAD process framework has several important characteristics that are detailed in this section.

People first

With DAD, people, and the way they work together, are the first order of business. In an agile environment, the boundaries between disciplines are torn down and handoffs are minimized in the interest of working as a team instead of a group of specialized individuals. But in DAD, cross-functional teams are made up of cross-functional people. You don't have hierarchy within the team, and team members are encouraged to be cross-functional in their skill sets and perform work related to disciplines other than their specialty. Because team members gain understanding beyond their primary discipline, resources are used more effectively, and formal documentation and sign-offs, by and large, aren't necessary.

WARNING!

When you adopt an agile approach, people are pushed outside their comfort zone. Taking on work outside of their established skill sets may not be something they're accustomed to doing. They also may be reluctant to give up the work where they feel they have the most expertise, for fear of losing relevance. By eliminating hierarchy and providing roles that incorporate cross-functional skill sets, DAD paves the way for this transition to go a little more smoothly.

The five primary roles of DAD include the following:

- Stakeholder
- Product owner
- Team member
- Team lead
- Architecture owner

For details on each of these roles, check out Chapter 2.

Learning-oriented

Organizations working with traditional approaches to life cycle management often don't get the most out of every opportunity for their staff to learn about the way effective solutions are produced, yet the most effective organizations are the ones that promote a learning-oriented environment for their staff.

An important aspect of DAD is the iteration retrospective meetings (covered in Chapter 3). These meetings occur throughout the solution life cycle. In this way, team members can make corrections to their processes as they go, leading to a more efficient and effective outcome and learning from their own mistakes to make each sprint better.

Agile

The DAD process framework adheres to and enhances the values and principles of the Agile Manifesto (described in Chapter 1).Teams following either iterative or agile processes have been shown to

- **Produce higher quality:** High quality is achieved through techniques such as continuous integration (CI), developer regression testing, test-first development, and refactoring.
- **Provide greater return on investment (ROI):** Improved ROI comes from focusing more on high-value activities, working in priority order, automating as much of the IT drudgery as possible, self-organizing, close collaborating, and working smarter, not harder.
- **Provide greater stakeholder satisfaction:** Greater stakeholder satisfaction is achieved by enabling active stakeholder participation, incrementally delivering a potentially consumable solution with each iteration, and enabling stakeholders to evolve their requirements throughout the project.
- **Deliver quicker:** Quicker delivery as compared to either a traditional/waterfall approach or an ad-hoc (no defined process) approach.

Hybrid

The DAD process framework is a *hybrid:* It adopts and tailors strategies from a variety of sources. A common pattern within organizations is that they adopt the Scrum process framework and then do significant work to tailor ideas from other sources to flesh it out. This sounds like a great strategy, and it certainly is if you're a consultant specializing in agile adoption, until you notice that organizations tend to tailor Scrum in the same sort of way.

DAD is a more robust process framework that has already done this common work. The DAD process framework adopts strategies from the following methods:

- **Scrum:** DAD adopts and tailors many ideas from Scrum, such as completing a stack of work items in priority order, having a product owner responsible for representing stakeholders, and producing a potentially consumable solution from each iteration. See Chapter 4 for more background information on Scrum.
- **Extreme programming (XP):** XP is an important source of development practices for DAD, including but not limited to continuous integration (CI), refactoring, test-driven development (TDD), collective ownership, and many more. See Chapter 4 for more info on XP.
- **Agile Modeling (AM):** DAD models its documentation practices after requirements envisioning, architecture envisioning, iteration modeling, continuous documentation, and just-in-time (JIT) model storming. See Chapter 4 for more info about AM.
- **Unified Process (UP):** DAD adopts several governance strategies from UP. In particular, this includes strategies such as having lightweight milestones and explicit phases and focusing on the importance of proving out the architecture in the early iterations and reducing all types of risk early in the life cycle. Read more about UP in Chapter 4.
- **Agile Data (AD):** DAD adopts several agile database practices from AD such as database refactoring, database test-in, and agile data modeling. It is also an important source of agile enterprise strategies, such as how agile teams can work effectively with enterprise architects and enterprise data administrators.
- **Kanban:** DAD adopts two critical concepts — limiting work in progress and visualizing work — from Kanban. Read more about Kanban in Chapter 4.

IT solution focused

Much of the focus within the agile community is on software development. DAD teams realize that in addition to software they must also address hardware, documentation, business process, and even organizational structure issues pertaining to their overall solution.

The DAD process frameworks promotes activities that explicitly address user experience (UX), database, business process, and documentation issues (to name a few) to help project teams think beyond software development alone.

Delivery focused

DAD addresses the entire life cycle from the point of initiating the project through construction to the point of releasing the solution into production. The project is carved into phases with lightweight milestones to ensure that the project is focused on the right things at the right time, such as initial visioning, architectural modeling, risk management, and deployment planning.

This differs from methods such as Scrum and XP, which focus on the construction aspects of the life cycle. Details about how to perform initiation and release activities, or even how they fit into the overall life cycle, are typically vague and left up to you.

The life cycle of a DAD project is shown in Figure 5-1.This life cycle has three critical features:

- **A delivery life cycle:** The DAD life cycle extends the Scrum construction life cycle to explicitly show the full delivery life cycle from the beginning of a project to the release of the solution into production (or the marketplace).
- **Explicit phases:** The DAD life cycle is organized into three distinct phases. See the section “Understanding the DAD Life Cycle” for more information on these phases.
- **Context:** The DAD life cycle indicates that pre-project activities as well as post-project activities occur.

See Figure 5-2 for the advanced DAD life cycle based on Kanban.

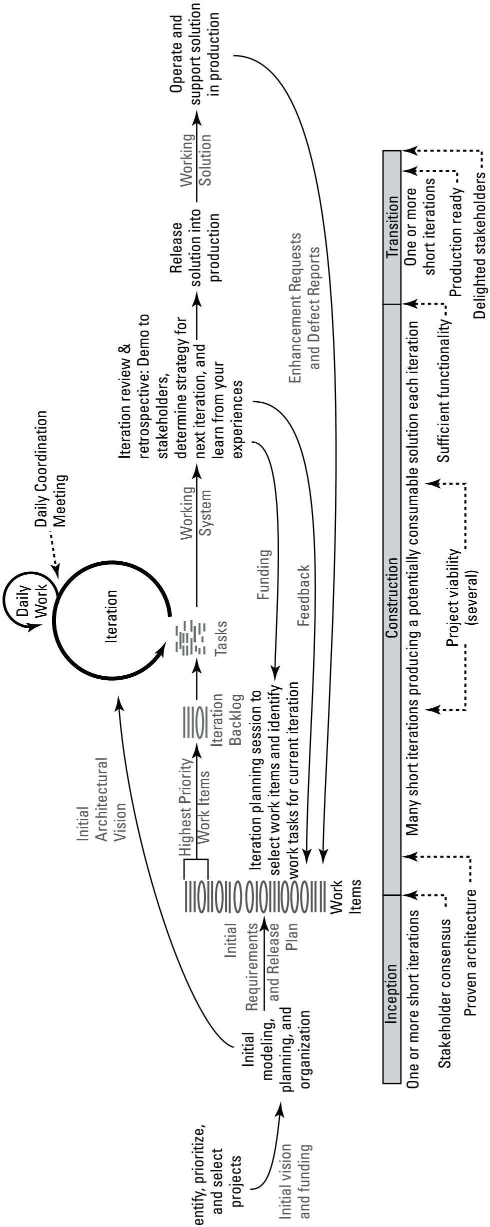

Identify, prioritize, and select projects
Initial vision and funding
Initial modeling, planning, and organization
Initial Requirements and Release Plan
Initial Architectural Vision
Work Items
Highest Priority Work Items
Iteration Backlog
Iteration planning session to select work items and identify work tasks for current iteration
Tasks
Iteration
Daily Work
Daily Coordination Meeting
Working System
Iteration review & retrospective: Demo to stakeholders, determine strategy for next iteration, and learn from your experiences
Funding
Feedback
Release solution into production
Working Solution
Operate and support solution in production
Enhancement Requests and Defect Reports
Inception
One or more short iterations
Stakeholder consensus
Proven architecture
Construction
Many short iterations producing a potentially consumable solution each iteration
Project viability (several)
Sufficient functionality
Transition
One or more short iterations
Production ready
Delighted stakeholders

Figure 5-1: The basic DAD life cycle.

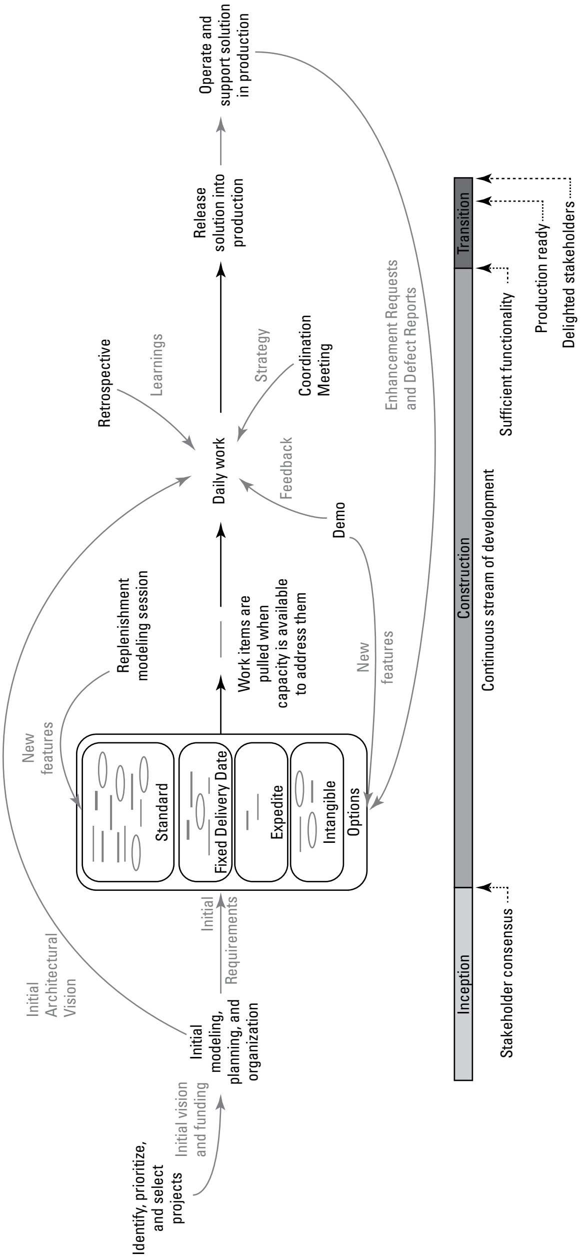

Identify, prioritize, and select projects
Initial vision and funding
Initial modeling, planning, and organization
Initial Architectural Vision
Initial Requirements
Standard
Fixed Delivery Date
Expedite
Intangible
Options
New features
Replenishment modeling session
Work items are pulled when capacity is available to address them
New features
Demo
Feedback
Daily work
Retrospective
Learnings
Strategy
Coordination Meeting
Release solution into production
Operate and support solution in production
Enhancement Requests and Defect Reports
Inception
Construction
Transition
Stakeholder consensus
Continuous stream of development
Sufficient functionality
Production ready
Delighted stakeholders

Figure 5-2: The advanced DAD life cycle.

Goal driven

The DAD process framework strives to meet goals in each phase (Inception, Construction, Transition). For example, goals during the inception phase include understanding the initial scope, identifying a technical strategy, performing initial release planning, and initiating the team. Each goal then has different issues to be addressed.

Instead of prescribing a single approach, DAD describes the goals you need to address, potential strategies for doing so, and the trade-offs that you face. It also suggests some good default options to help get you started.

Risk and value driven

The DAD process framework adopts what is called a *risk-value life cycle;* effectively, this is a lightweight version of the strategy promoted by the UP. DAD teams strive to address common project risks, such as coming to stakeholder consensus around the vision and proving the architecture, early in the life cycle. DAD also includes explicit checks for continued project viability, whether sufficient functionality has been produced, and whether the solution is production ready. It is also value-driven, in that DAD teams produce potentially consumable solutions regularly.

Enterprise aware

DAD teams work within your organization's enterprise ecosystem, as do other teams, and explicitly try to take advantage of the opportunities presented to them. Disciplined agilists act locally and think globally.

These teams work closely with the following teams and individuals:

- Enterprise technical architects and reuse engineers to leverage and enhance the existing and "to be" technical infrastructure
- Enterprise business architects and portfolio managers to fit into the overall business ecosystem
- Senior managers who govern the various teams appropriately
- Data administrators to access and improve existing data sources
- IT development support people to understand and follow enterprise IT guidance (such as coding, user interface, security, and data conventions)
- Operations and support staff to understand their needs to ensure a successful deployment (one part of DAD's overall support for Development and Operations)

REMEMBER

With the exception of start-up companies, agile delivery teams don't work in a vacuum. There are often existing systems currently in production, and minimally your solution shouldn't impact them although your solution should leverage existing functionality and data available in production.

Understanding the DAD Life Cycle

The ongoing goals of DAD include the following:

- Fulfill the project mission
- Grow team members' skills
- Enhance existing infrastructure
- Improve team process and environment
- Leverage existing infrastructure

The DAD life cycle has three phases.

Inception

The first phase of DAD is inception. Before going full-speed ahead, this phase takes typically between a few days and a few weeks to initiate the project. The inception phase ends when the team has developed a vision for the release that the stakeholders agree to and has obtained support for the rest of the project (or at least the next stage of it).

Construction

The construction phase in DAD is the period of time when the required functionality is built. The timeline is split up into a number of time-boxed iterations. The time-boxed iterations should be the same duration for a given project — typically one to four weeks, with two and four weeks being the most common — and typically don't overlap. At the end of each iteration, demonstrable increments of a potentially consumable solution have been produced and regression tested.

The construction phase ends where there's sufficient functionality to justify the cost of transition — sometimes referred to as minimally marketable release (MMR) — and which the stakeholders believe is acceptable to them.

Transition

The *transition phase* focuses on delivering the solution into production (or into the marketplace in the case of a consumer product). The time and effort spent transitioning varies from project to project. This phase ends when the solution is released into production.

For more information on this topic, see *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise,* by Scott W. Ambler and Mark Lines (IBM Press, 2012).

Chapter 6

Scaling Agile Practices

In This Chapter

- Seeing how agile can scale
- Incorporating additional roles on large teams

Agile approaches have been adapted to work in a wide range of situations, not just the small, collocated team environments that dominate the early agile literature. Agile strategies are being applied throughout the entire software delivery life cycle, not just construction, and very often in very complex environments that require far more than a small, collocated team.

Every project team finds itself in a unique situation, with its own goals, its abilities, and challenges to overcome. What they have in common is the need to adopt and then tailor agile methods, practices, and tools to address those unique situations.

Understanding What It Means to Scale

In the early days of agile, projects managed via agile development techniques were small in scope and relatively straightforward. The small, collocated team strategies of mainstream agile processes still get the job done in these situations. Today, the picture has changed significantly and organizations want to apply agile development to a broader set of projects.

Organizations often deal with problems that require large teams; they want to leverage a distributed workforce; they want to partner with other organizations; they need to comply with regulations and industry standards; they have significant technical or cultural environmental issues to overcome; and they want to go beyond the single-system mindset and truly consider cross-system enterprise issues effectively.

REMEMBER

Not every project team faces all these scaling factors or each scaling factor to the same extent, but all these issues add complexity to your situation, and you must find strategies to overcome these challenges.

To deal with the many business, organization, and technical complexities your development organization faces, your disciplined agile delivery process needs to adapt, or *scale.* The following sections describe the factors most organizations encounter when scaling their agile projects.

Large teams

Mainstream agile processes work very well for smaller teams of 10 to 15 people, but what if the team is much larger? What if it's 50 people? 100 people? 1,000 people? As your team-size grows, the communication risks increase and coordination becomes more difficult. As a result paper-based, face-to-face strategies start to fall apart.

Distributed teams

What happens when the team is distributed — perhaps on floors within the same building, different locations within the same city, or even in different countries? What happens if you allow some of your engineers to work from home? What happens when you have team members in different time zones? Suddenly, effective collaboration becomes more challenging and disconnects are more likely to occur.

Compliance

What if regulatory issues — such as Sarbanes Oxley, ISO 9000, or FDA CFR 21 — are applicable? This may mean that the team has to increase the formality of the work that it does and the artifacts that it creates.

Domain complexity

Some project teams find themselves addressing a very straightforward problem, such as developing a data entry application or an informational website. Sometimes the problem domain is more intricate, such as the need to monitor a bio-chemical process or air traffic control. Or perhaps the situation is changing quickly, such as financial derivatives trading or electronic security assurance. More complex domains require greater emphasis on exploration and experimentation, including — but not limited to — prototyping, modeling, and simulation.

Organization distribution

Sometimes a project team includes members from different divisions, different partner companies, or from external services firms. The more organizationally distributed teams are, the more likely the relationship will be contractual in nature instead of collaborative.

WARNING! A lack of organizational cohesion can greatly increase risk to your project due to lack of trust, which may reduce willingness to collaborate and may even increase the risks associated with ownership of intellectual property (IP).

Technical complexity

Some applications are more complex than others. Sometimes you're working with existing legacy systems and legacy data sources that are less than perfect. Other times, you're building a system running on several platforms or by using several different technologies. And at different times the nature of the problem your team is trying to solve is very complex in its own right, requiring a complex solution.

Organizational complexity

Your existing organization structure and culture may reflect waterfall values, increasing the complexity of adopting and scaling agile strategies within your organization. Or some groups within your organization may wish to follow strategies that aren't perfectly compatible with the way yours wants to work.

Enterprise discipline

Most organizations want to leverage common infrastructure platforms to lower cost, reduce time to market, improve consistency, and promote a sustainable pace. This can be very difficult if your project teams focus only on their immediate needs.

To leverage common infrastructure, project teams need to take advantage of effective enterprise architecture, enterprise business modeling, strategic reuse, and portfolio management disciplines. These disciplines must work in concert with, and better yet enhance, your disciplined agile delivery processes. But this doesn't come free.

Your agile development teams need to include as stakeholders Enterprise Architecture professionals — such as enterprise and solution architects and reuse engineers — if not development team members in their own right. The enterprise professionals also need to learn to work in an agile manner, a manner that may be very different compared to the way that they work with more traditional teams. For more information on this topic, visit `http://bit.ly/79mLoJ`.

Organizing Large Teams

When a disciplined agile team consists of 30 or more people, it's considered large. A large team is divided into subteams (see Figure 6-1). Large teams add explicit roles required for coordination, particularly those within the leadership team. These roles for coordination are sometimes referred to as the *coordination team*. Large teams sometimes incorporate an *integrator,* but this is an optional role and not always used.

The leadership team is typically headed up by someone in the role of program manager, sometimes referred to as a *project manager,* who's responsible for overall team coordination. As Figure 6-1 indicates, the leadership team consists of the people in senior roles on the individual subteams. Together, these people address the following aspects of team collaboration:

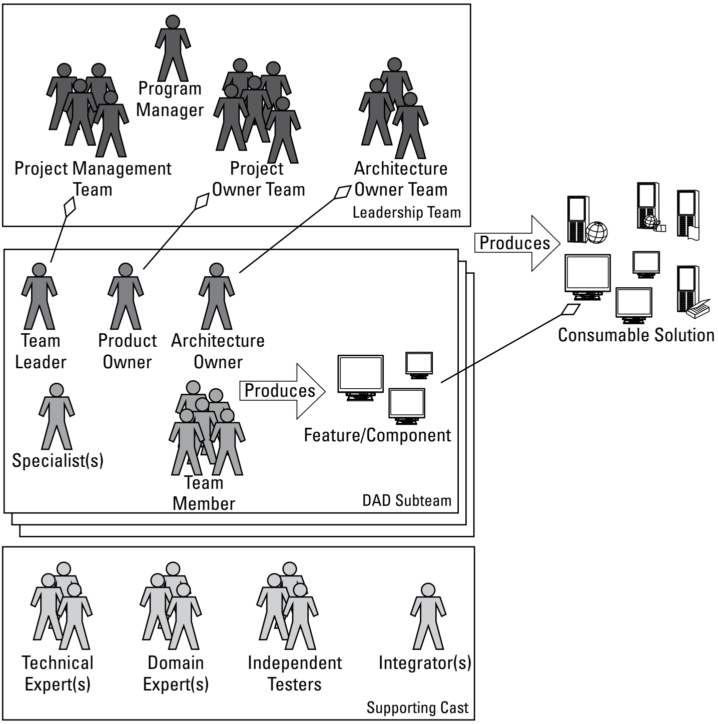


Figure 6-1: The structure of a large DAD team.

- **Project management coordination:** The individual team leads are each part of the project management team. They're responsible for coordinating fundamental management issues, such as schedule dependencies, staffing, conflicts between subteams, and overall cost and schedule tracking. The program manager typically heads up the project management team.
- **Requirements coordination:** Because there are dependencies between requirements and between work items in general, product owners must coordinate requirements and work items across subteams, including ensuring that the same requirement isn't being worked on unknowingly by two or more subteams and that the priorities of each subteam are consistent.

If requirement dependencies aren't coordinated across subteams, producing a consumable solution for each iteration becomes nearly impossible, and the development process can fall apart. The Chief Product Owner leads the team of product owners in this effort.

- **Technical coordination:** Technical dependencies, such as the need to invoke services or functions provided by another part of the solution, exist between each subsystem/component or feature. This requires the appropriate subteams to coordinate with one another — work that's typically initiated by the architecture owners on each subteam but often involves other team members as needed.

  Another aspect of technical coordination is regular integration of the overall system. Very large or complex teams have one or more people in the role of integrator — while the architecture owner is responsible for integration within their subteam, the integrator(s) is responsible for solution integration and coordinates accordingly with the architecture owners and other team members of the subteams.

The leadership subteams (project management, product owners, architecture owners) coordinate any issues via team coordination meetings and electronic means as needed. Many teams discover that these coordination issues have different cadences. For example, requirements and technical coordination occur daily at the beginning of an iteration but diminish later in the iteration, but project management coordination is needed daily throughout the iteration.

A greater need exists for shared models, documentation, and plans, particularly if the team is geographically dispersed. Use of integrated tooling that's instrumented to provide key measures, which in turn are displayed on project dashboards, can provide greater visibility of what is happening within the team and thereby aid coordination.

For more information on this topic, see *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise,* by Scott W. Ambler and Mark Lines (IBM Press, 2012).

Chapter 7

Evaluating Agile Tools

In This Chapter

- Knowing the key criteria for selecting agile tools
- Recognizing the value of an open platform solution
- Managing your agile process with Rational Team Concert

To support and automate the agile process into your organization, you can consider incorporating tools that facilitate a streamlined process for your agile teams. This chapter helps you make smart decisions when considering software purchases that serve your agile development needs.

Considering Key Criteria for Selecting Agile Tools

Each team approaches agile in a different way. Some teams start small, adopting Scrum and simple approaches using open source tools, while others require more extensive agile life cycle management solutions. Regardless of where they attain them, most agile teams make use of specific agile tools that help them get the most from their agile process.

Before you decide to use a new tool, determine whether it's the best tool for your needs. The easiest way to do this is to keep these seven key criteria in mind as you make your selection:

- **Core agile capabilities:** First and foremost, the agile tool must support people over process by facilitating collaboration, planning, and productivity among the agile team, product owners, and other stakeholders.

- **Integrated and open agile delivery:** The agile tool must allow you to protect investments in existing tooling and minimize tool maintenance with simplified integrations.
- **Team collaboration in context:** The ability for team members, customers, managers and stakeholders to work together in context of the task at hand allows the team to focus on delivering working software.
- **Life cycle traceability:** The ability to understand how your actions affect others, to find gaps in test coverage, and be aware of defects that are blocking progress help your team identify and mitigate potential risks and reduce friction across the agile delivery life cycle.
- **Agile development analytics:** Arming managers and agile team members with real-time metrics to make informed decisions helps reduce friction and accelerate the velocity of agile teams.
- **Adaptability and flexibility:** Your tool should support your team regardless of its process or size. Look for adaptable process support that evolves as your needs change.
- **Agility at scale:** As your organization grows, it most likely needs a well-defined agile scaling process, team structure, and tooling to address real-world complexities, such as those faced by distributed teams or teams addressing compliance requirements.

The right tools can help you succeed, regardless of your entry point. Your challenge is to identify your greatest need today, the improvements you need to make to current practices, and whether new tools are really the solution.

For more information on evaluating agile tools, visit `www.ibm.com/rational/agile`.

Exploring the Jazz Initiative

Inspired by the artists who transformed musical expression, Jazz is an initiative to transform software and systems delivery by making it more collaborative, productive, and transparent through integration of information and tasks throughout the delivery life cycle. The Jazz initiative consists of three elements:

- **Platform:** The Jazz platform is an open and flexible platform, which allows for greater integration capability.
- **Products:** In keeping with agile principles, Jazz products are designed to put the team first.
- **Community:** The Jazz community site is the live development infrastructure for the Jazz technology and products.

Using the Best Tool for the Job

The ideal agile tool addresses project planning, work item tracking, source code management, continuous builds, and adaptive process support, enabling agile project teams and stakeholders to work together effectively.

IBM Rational Team Concert (RTC) is built on the Jazz platform and provides all these features in one integrated tool to help the project team collaboratively plan, execute, and deliver working applications. The following sections describe the key capabilities RTC offers to support agile teams. For more information on RTC features, visit `https://jazz.net/projects/rational-team-concert/features`.

RTC provides core capabilities, including source code management optimized for distributed teams; continuous integration supporting personal, team, and integration builds; and customizable work item tracking to support agile and formal teams.

Process awareness and customizability

Regardless of the size or maturity of your agile team, your agile tooling should give you the ability to deploy, customize, enact, and improve agile processes. RTC can improve the productivity of your teams and the quality of the work they produce by allowing each team to teach the tool its best practices. RTC uses this knowledge to automate team processes, allowing team members to focus on building great software.

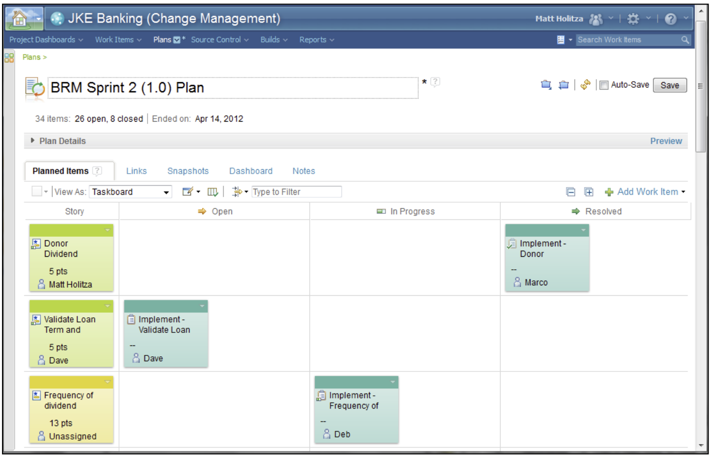
Team awareness

Your agile tooling should make collaboration easier across all stages of the life cycle. All team members, including stakeholders, should have access to real-time, role-relevant information with full traceability to related tasks, as well as the ability to easily make contributions to projects.

RTC knows your project teams, their internal organization, and the artifacts they are working on. It greatly simplifies the access to team-related information or performing team-related operations. In addition, RTC integrates with Lotus Sametime, GoogleTalk, and Skype, which help enhance collaboration when teams are geographically distributed.

Planning

Agile planning is all about keeping everyone on the same page and marching to the same beat. RTC provides capabilities to create product, release, and iteration plans for teams; to create individual plans for developers; to track the progress during an iteration; and to balance the workload of developers. In addition, RTC provides planning views to support different agile approaches including Taskboards for daily standups and Kanban to manage flow (see Figure 7-1). RTC also provides cross-project plans that allow tracking of work items that have dependencies on other work items in other projects.



BRM Sprint 2 (1.0) Plan

34 items: 26 open, 8 closed | Ended on: Apr 14, 2012

View As: Taskboard

Story	Open	In Progress	Resolved
Donor Dividend 5 pts Matt Holitza			Implement - Donor -- Marco
Validate Loan Term and 5 pts Dave	Implement - Validate Loan -- Dave		
Frequency of dividend 13 pts Unassigned		Implement - Frequency of -- Deb	

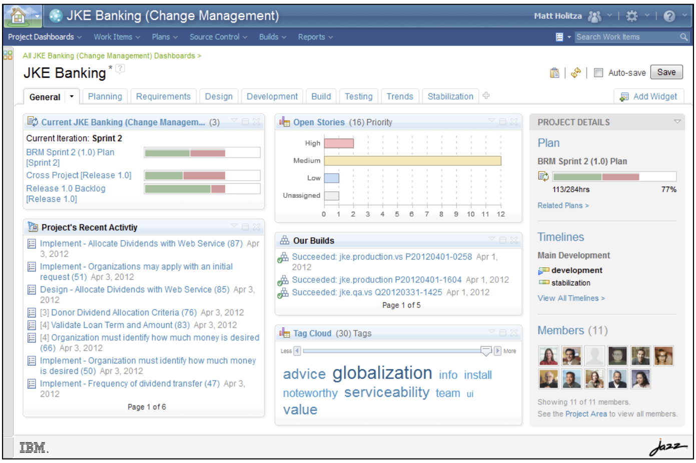
Figure 7-1: The Taskboard planning view.

Regardless of the process used, plans are accessible to everyone on the team via the web, Eclipse, or Visual Studio. All plans change dynamically over the course of the project to reflect the team's position and direction.

Transparency/project health

Transparency and the ability to monitor status in real-time is vital to a successful agile project. Team members should have visibility to potential issues that could impede their progress, and managers need to have real-time information to proactively manage risks.

The RTC dashboards and reports help all team members keep tabs on the health of their projects. The Web Dashboard (Figure 7-2) provides a personalized view of plan status, work items, event feeds, reports, and other items that are critical to understanding your progress. Reports provide both real-time views and historical trends of velocity, builds, streams, work items, and other artifacts that your team works with. In addition, RTC supports the use of OpenSocial Gadgets and IBM iWidgets to extend visibility to other commercial and open source life cycle tools.



Current JKE Banking (Change Managem... (3)

Current Iteration: **Sprint 2**

BRM Sprint 2 (1.0) Plan [Sprint 2]

Cross Project [Release 1.0]

Release 1.0 Backlog [Release 1.0]

Project's Recent Activtiy

- Implement - Allocate Dividends with Web Service (87) Apr 3, 2012
- Implement - Organizations may apply with an initial request (51) Apr 3, 2012
- Design - Allocate Dividends with Web Service (85) Apr 3, 2012
- [3] Donor Dividend Allocation Criteria (76) Apr 3, 2012
- [4] Validate Loan Term and Amount (83) Apr 3, 2012
- [4] Organization must identify how much money is desired (66) Apr 3, 2012
- Implement - Organization must identify how much money is desired (50) Apr 3, 2012
- Implement - Frequency of dividend transfer (47) Apr 3, 2012

Page 1 of 6

Open Stories (16) Priority

Our Builds

- Succeeded: jke.production.vs P20120401-0258 Apr 1, 2012
- Succeeded: jke.production P20120401-1604 Apr 1, 2012
- Succeeded: jke.qa.vs Q20120331-1425 Apr 1, 2012

Page 1 of 5

Tag Cloud (30) Tags

PROJECT DETAILS

Plan

BRM Sprint 2 (1.0) Plan

113/284hrs 77%

Related Plans >

Timelines

Main Development

development

stabilization

View All Timelines >

Members (11)

Showing 11 of 11 members.
See the Project Area to view all members.

Figure 7-2: Web Dashboard.

Broad Platform Support

The trend toward more interconnected mobile and cloud systems is leading to an increasing number of cross technology development projects that are deployed on multiple target platforms. RTC provides the ability to centrally manage, develop, and track multi-platform projects, providing visibility to all team members and stakeholders. RTC supports popular development platforms, including Windows, Linux, IBM System z, IBM Power, and Solaris.

RTC not only provides cross platform support but also provides integrated development environments (IDE) for Eclipse and Microsoft Visual Studio that allow developers to collaborate across teams, track projects, manage source code, resolve defects, and execute builds.

Extending tooling beyond core agile development

While RTC serves as the core agile development tool, IBM Rational also provides depth and breadth of tools that allow organizations to adapt and automate beyond development across the entire life cycle. Depending on which scaling factors apply (see Chapter 6), organizations can extend RTC with a vast portfolio of capabilities including: more vigorous requirements envisioning and management, test management, deployment automation, architecture and asset management, and business collaboration.

In addition, leveraging the Jazz platform RTC can be integrated with popular open source tools, including Subversion, Git, CVS, Maven, Hudson, and Jenkins. For a complete list of integrations see `www.jazz.net/projects/rational-team-concert/integrations`.

Chapter 8

Making the Move to Agile: IBM's Story

In This Chapter

- ▶ Empowering development teams
- ▶ Creating processes to work with distributed teams
- ▶ Tooling up for success
- ▶ Seeing the benefits of using agile

In 2006, IBM faced a unique challenge. In a period of about five years, the company made 70 software acquisitions with 90 major labs and employed a variety of people who worked remotely. This feat translated to approximately 27,000 people working over five continents. And, yet, IBM struggled in the software development area.

In the IBM Rational organization, for example, IBM only delivered software on time 47 percent of the time. The cost of poor quality was high. The company released products with defects or delayed releases until defects were fixed. As a result, products that were supposed to be high-flyers actually crash-landed because the cost of fixing defects was so high. IBM also used tools that didn't integrate with each other.

IBM needed to be more agile. The success of IBM's agile adoption depended on significant changes in three key areas: the people, the adopted processes, and the implemented tools for success.

Setting Teams Up for Success

Agile requires empowering people to make the significant change in how software is developed. IBM made adjustments with the areas covered in this section.

Training

IBM helped the software teams change by setting up a center of excellence, holding a two-day disciplined agile workshop that trained approximately 9,000 people over a period of two years, and running additional focused workshops for in-depth training on practices that included a collaborative leadership workshop. IBM identified key resource dependencies (either people or technology) and made them available to the teams when they needed help adopting agile.

Collaboration capabilities

Although studies show that face-to-face collaboration is best for agile development, in IBM's case, it wasn't always possible. So the company deployed tools, such as IBM Rational Team Concert (see Chapter 7 for more info), that could bring large teams together. This included support for team members sharing immediate office space or sitting on the other side of the globe. These tools informed individual users, teams, and teams of teams about project changes as they happened.

IBM also enabled continuous planning and change management. So, if a project lead and management had a conversation about a change, everyone on the team was notified about it, the reason for it, who's affected, and how they're affected. These tools also have repositories so plan and work items can be stored and used in the future by others.

Changing culture

A change in culture can be a challenge for some people. In a number of cases, particularly at the beginning, some teams struggled to accept or adopt agile because they were so dispersed, and their components were done in different geographic locations. So, IBM sent a coach who understood agile to each location for three to six months. Having an agile practitioner in these locations was worth the cost, because the rate of failure dropped.

IBM also had to change the views of development at the top. The executives were used to having an initial state and plan and relating them to waterfall measurements. They worked to accept that scope changes over time and that this was a good thing — as long as IBM kept release rates and quality levels the same (or better).

Changing roles

To support evolving culture to a more agile strategy, IBM used automation to support new habits and get rid of old ones. For example, traditional project managers used to canvas individuals about progress, but by the time they could enter the notes into a spreadsheet, the team had moved to the next activity. Now, tools track progress and agile team leads focus on how people work with each other. At the same time, the team leads refocused on leadership activities over technical management activities, thereby adding greater value to their teams and to IBM as a whole.

Team structure

Because IBM's complex products vary from testing tools to compilers to database management systems to application servers and more, it needed to have more than one type of team. So, IBM created teams, depending on the types of capabilities it was looking for, via the following strategies:

- **Feature teams:** Responsible for a complete customer feature (products and components). The goal was to optimize customer value. Feature teams minimize dependencies, use iterative development, and track dependencies between adoptions with an adoption tool.
- **Component teams:** Responsible for only part of a customer feature. The development is more sequential and dependencies between teams lead to additional planning. They track execution in plan items — either in work areas or work items.

- **Internal open source:** When a component needs to evolve quickly yet is needed by multiple development teams, IBM uses open source strategies as an effective development method. IBM has deep experience in open source development, so this was an easy team structure to adopt.

Updating Processes for Distributed Teams

After agile was implemented in a number of development teams, some teams were successful, but many more teams had trouble. The teams with the most success were housed in a single location and collaborative. The teams that struggled were globally distributed with hundreds of people. IBM realized that it had to change some processes to make agile work better in these distributed teams.

A big change was in IBM's auditable process, which, in 2006, was to "ask for forgiveness" if quality, scope, resources, and time to market changed. This process took a great deal of time and effort, so IBM approached scope differently, while still locking in the other pieces of the project, by dividing it into two types:

- **Release-defining scope:** Usually consumes about 70 percent of the schedule, which is all the things that will cause the project to slip if problems arise.
- **Extended content:** Consists of things that won't cause problems if they aren't shipped.

IBM is now better at delivering what's release defining and, and in almost all cases, additional content that its customers and sales force don't know about in advance. This improvement reduced time and effort spent asking for forgiveness. Also, IBM teams have a clearer picture of what's important and what's optionally important.

Other changes IBM made to its processes are as follows:

- Work with stakeholders every iteration for timely and effective feedback
- Organize its agile methodology into smaller, consumable practices that can be adopted easily
- Make all parts of the production process agile, not just development

  The goal was to make it difficult to go back to old waterfall ways, and the process has been successful.
- Adopt a robust planning strategy

  Each activity, whether it was going on vacation or writing code or testing, is a work item that's estimated and prioritized.

Working with New Tools

Because IBM is a large organization, it has to be intentional about the way it communicates and collaborates. IBM determined that tooling was necessary to provide a way for teams to exchange information and stay on the same page — much like smaller teams do.

The key to collaboration is that teams want a way to collaborate and exchange information in a way that feels natural. Posting a bunch of information to a wiki won't get people to pay attention. Exchanging information has to feel natural to and fit into the context of the teams' work.

IBM instituted technology that unified people beyond just the development team, including documentation, training, translation, and release management. Everyone could see the status on complementary work activities that took place. This innovation created better visibility to overall team progress.

When you have a transparent view of the work items that are complete, the items that still need to be done, and team visibility into how certain tasks are taking more effort than originally scoped, they help the team get better at agile planning and drive conversations about whether the team needs to realign resources to stay on track.

The software reuse initiative

Component reuse is one the holy grails of software development. If you reuse code, you typically get better quality and productivity, reduced risk of development efforts, and more consistency for your user experience. You can try to *mandate* reuse, but that really doesn't work well. So instead, IBM created a site that emulated an open source environment and called it *community sourcing.* With this site, developers can create their own projects, share those projects with others, gain access to the code, and decide whether the projects meet their needs. By using a combination of Rational Team Concert, Rational Asset Manager, and Lotus Connections, IBM provided an environment conducive to community and sharing. IBM now has over 30,000 developers using the community source site and 4,800 uses of components in its products now. For example, a component that helps IBM's installation has been reused by 152 products, and one component for security has been used by 175 projects.

One of the nice benefits of this type of community sourcing initiative is that it connects people in new and exciting ways. IBM has seen people reaching across divisions and functional boundaries because they were using a common component. With the community source site, this type of collaboration happens organically.

Reaping the Benefits of Agile

IBM's agile transformation took some adjustments. The company made some mistakes along the way, sure, but it learned from them and has now experienced extensive improvements in quality, time-to-market, and customer satisfaction. IBM is proof that the rewards of agile adoption far outweigh the obstacles.

IBM's results from agile adoption include the following:

- A 31/69 percent ratio of maintenance to innovation (start was 42/58 and goal was 50/50)
- Customer touches increased from 10 to 400
- Customer calls decreased to 19 percent
- Customer defect arrivals decreased from 6,900 to 2,200
- Defect backlog reduced to 3 months from 9 months or more
- Reduced cost of poor quality (cut almost in half)

Chapter 9

Ten Common Agile Adoption Pitfalls

In This Chapter

- Knowing what mistakes to avoid
- Creating a better agile adoption experience

The ability to avoid common agile adoption pitfalls may seem daunting, but there's a light at the end of the tunnel. With over 10 years of experience helping customers manage and execute their agile transformations, we present you with this chapter to help briefly explore some common pitfalls organizations make when adopting agile strategies. And hopefully, with this advice, you can skip over making the same mistakes.

Focusing Only on Construction

You can realize the spirit of the Agile Manifesto through many approaches. Ironically, most of these approaches focus on one phase or discipline within the delivery life cycle — which goes against the spirit of lean, which advises to consider the whole. Most approaches focus on the construction phase.

Construction is typically a straightforward area to focus on when taking on an agile transformation, but if organizations only change the way they construct software, they can't necessarily call themselves agile. The development teams could be humming along, delivering new working software every two weeks, but if the processes in Operations only allow for deployment every six months or if the Help Desk is unable to handle the churn or if customer stakeholders aren't prepared to meet regularly, the organization isn't realizing all the benefits agile can provide.

Becoming Agile Zombies

Organizations fall into the trap that if they attend a class and mandate a certain out-of-the-box (OOTB) process, that they are now agile. They train their teams to blindly follow and enforce the anointed process not considering which practices may need to change to meet their organizations' unique needs.

REMEMBER

Agile isn't a prescribed process or set of practices; it's a philosophy that can be supported by a practice and no two agile approaches are the same. One OOTB methodology that fulfills all needs doesn't exist.

Improper Planning

That old adage, "If you fail to plan, plan to fail," is really true. Planning is core to the success of any agile adoption. Organizations should answer these questions:

- Why do we want to be agile, and what benefits will agile provide?
- How will we achieve and measure agility?
- What cultural, technological or governance barriers exist, and how do we overcome them?

  Without a plan that clearly shapes the initiative and includes addressing and resolving constraints to agility (for example, removing waterfall process checkpoints or getting support from other required entities), it is more difficult to shape the initiative, staff it, fund it, manage blockers and maintain continued executive sponsorship.

Excluding the Entire Organization

You can quickly short circuit an agile adoption by working in the vacuum of a single software or system delivery team. A single team can gain *some* benefit from agile, but to be truly successful, you need to look at the whole process around solution delivery. And many people are involved in that process.

Agile should be a change in culture for the entire organization. Find champions in Operations, lines of business, product management, Marketing, and other functional areas to increase your success.

Lack of Executive Support

An effective agile adoption requires executive sponsorship at the highest level. This involvement means more than showing up at a kickoff meeting to say a few words. Without executive sponsorship supporting the overall initiative, the agile adoption is often doomed because agile initiatives require an upfront investment of resources and funding — two areas that executives typically control.

Going Too Fast

Moving to agile is very exciting, and it can be tempting to jump right in, pick a process, get some tools, and hit the ground running. Unfortunately, if a proper roadmap for coaching, process, and tooling isn't outlined early in the adoption you can run into issues like the following:

- No defined processes for dealing with multiple dependent or distributed teams
- Scalability issues with the core agile tools
- Extending the tool to support deployment, testing, or business collaboration

Insufficient Coaching

Because an agile adoption isn't just a matter of a new delivery process, but is also major cultural shift, coaching is imperative. Developers don't like change and many people like working in their own world. As a result, the concept of not only changing the way they develop, but adding the concept that now they have to work closely with five, six, or ten other people all the time can be downright horrifying.

A coach can work with these team members and help them through the early phases of agile adoption. Have you known of a teacher or coach that possessed a unique ability to inspire students to stretch their skills and perform at higher levels? Good agile coaching can have the same affect and make the difference between the success and failure of an agile adoption.

Retaining Traditional Governance

When an organization plans its agile adoption, it needs to evaluate all current processes and procedures and whether they inhibit or enhance agility. Existing traditional governance processes can be very difficult to change due to internal politics, company history, or fear that compliance mandates may be negatively impacted. Some common governance areas that are overlooked but can have dramatic impacts to agility are project funding, change control, and phase gates.

Skimping on Training

Organizations often see agile practice training, like coaching, as an area where they can save money, sending only a few key leads to learn the new process in hopes that they can train the rest of the organization while trying to implement the new approach. Agile involves a change in behavior and process. It is critical to send all team members to the appropriate training and provide them with ongoing training to reinforce agile values and update team members on processes that may have changed.

Skimping on Tooling

Agile tooling should support and automate an organization's process. Ensuring all team members consistently use tools impacts the success of the project. If tools are used inconsistently, metrics may not correctly reflect the correct status, builds could be run incorrectly, and overall flow and quality issues result. See Chapter 7 for more information on agile tools.

Chapter 10

Ten Myths about Agile

In This Chapter

- Sorting agile facts from the fiction
- Understanding how agile can work for you

Although agile is an established development approach, it represents a new way of life for the organizations that adopt it. The prospect of working in iterations instead of with a linear approach is unsettling to managers and developers who are deciding whether to make the leap to agile. They fear that focused efforts will be compromised and that control over projects and development teams will be sacrificed. Nothing is further from the truth.

This chapter debunks these myths and others to show that agile is most organizations' best bet for success.

Agile Is a Fad

The agile approach to project management is far from a fad. Agile has been in use for many decades even though it was only recently formalized with the Agile Manifesto and its associated principles. Agile exists because it works. Compared with traditional project management approaches, agile is better at producing successful projects.

Agile Isn't Disciplined

Sometimes agile can seem chaotic because it's a very collaborative process. Agile is a departure from the rigid assembly-line process. The iterative approach requires rapid response times and flexibility from the team. In fact, agile demands greater discipline than what's typical of traditional teams. Agile requires teams to reduce the feedback cycle on many activities, incrementally deliver a consumable solution, work closely with stakeholders throughout the life cycle, and adopt individual practices which require discipline in their own right.

Agile Means "We Don't Plan"

With agile's reliance on collaboration instead of big documents, it may seem like no planning occurs. But in reality, the planning is incremental and evolutionary, which has been proven successful (instead of planning all at once early in a project).

Agile Means "No Documentation"

Agile teams keep documentation as lightweight as possible, but they do document their solutions as they go. They follow strategies, such as documenting continuously and writing executable specifications.

Agile Is Only Effective for Collocated Teams

Sure, *ideally* agile teams are located within proximity of one another, but in this day and age, most development teams are distributed. Just remember, to succeed, you need to adopt practices and tooling that build team cohesion. If you use the proper tools, your team doesn't have to be collocated to work effectively together.

Agile Doesn't Scale

Agile definitely scales. Large teams must be organized differently. They need more than index cards and whiteboard sketches. Large agile teams succeed by using products like the following:

- **IBM Rational Requirements Composer** for requirements modeling

- **IBM Rational Build Forge** for large-scale continuous integration
- **IBM Rational Quality Manager** to support parallel independent testing

REMEMBER IBM is one of the world's largest agile adoption projects, transforming teams ranging in size from 5 to 600 team members. For more information on agility at scale, visit `ibm.com/rational/talks`.

Agile Is Unsuitable for Regulated Environments

Regulated environments are those that are subject to some regulatory mandates, such as medical device companies, business in the finance area, governmental departments and offices, the healthcare field, and more. These organizations are audited from time to time for compliance with regulations. With agile, these organizations can feel confident when they endure these audits. They benefit from faster delivery of data and higher quality of their output.

Agile Means We Don't Know What Will Be Delivered

Because agile is an iterative process, it provides the opportunity not just for greater control but *better* control over building the right things in the life cycle than one would have with the more traditional Waterfall approaches. At the end of each iteration, the development team presents a completed product to the product owner for feedback. Furthermore, Disciplined Agile Delivery (DAD) teams explicitly explore high-level requirements at the beginning of the project and seek to gain stakeholder agreement around the requirements. See Chapter 5 for more details.

Agile Won't Work at My Company

For many companies, the biggest challenge they face when considering changing to agile is the cultural change making the change requires. Agile has explicit means of frequent feedback and loops, which means that developers and managers may feel more exposed to scrutiny. But that doesn't mean that agile won't work at your company.

Agile is a team approach. It's not like football where one player at a time moves the ball down the field to score a touchdown. Agile is more like rugby — the whole team moves the ball down the field together. Roles are cross-functional and shared. Developers become testers and more frequent delivery creates more exposure and personal accountability.

For an organization to successfully adopt agile, executive support is also critical. Agile *can* succeed without it, but if you hit a bump in the road, you'll want that vote of confidence to help you keep everything moving in the right direction.

It's Enough for My Development Team to Be Agile

For agile to work properly, *all* teams have to buy in. So if your development team is gung ho, but your testing team is blasé, you won't get your best results. Your agile delivery process is only going to be as effective as your slowest group. To make agile succeed at its greatest potential, make each piece of the chain as efficient as possible.

Agile Is a Silver Bullet

Agile isn't needed for every team in every situation. It isn't a cure-all. Agile is a superb solution for projects that are in development or undergoing radical changes. For other projects, such as those that are in maintenance mode, agile isn't as good a fit. If your project has a stable customer base and isn't undergoing a lot of change in the code, you may not need to use agile for that particular project. But for projects that are under new product or rapid development, agile really is the best way to go.