James A. Crowder · Shelli Friess

Agile Project Management: Managing for Success

Springer

Agile Project Management: Managing for Success

James A. Crowder • Shelli Friess

Agile Project Management: Managing for Success

Springer

James A. Crowder
Raytheon
Denver, CO, USA

Shelli Friess
Englewood, CO, USA

ISBN 978-3-319-09017-7 ISBN 978-3-319-09018-4 (eBook)
DOI 10.1007/978-3-319-09018-4
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014945570

© Springer International Publishing Switzerland 2015
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Dr. Crowder has been involved in the research, design, development, implementation, and installation of engineering systems from several thousand dollars up to a few billion dollars. Both Dr. Crowder and Ms. Friess have been involved in raving successes and dismal failures (ok, let's call them learning opportunities) not only in development efforts but in team building and team dynamics as well. Having been involved in agile development projects and team building exercises, both have seen the major pitfalls associated with trying to build teams and, in particular, create successful agile development teams. A general lack of management commitment to the agile development process and a lack of training provided for people working in development teams are two of the major reasons agile teams so often falter or fail. Here we endeavor to discuss some of the major topics associated with team dynamics, individual empowerment, and helping management get comfortable with a new paradigm that is not going away.

Having taught both classical program management methods and agile development and management methods for many years, there are always arguments as to whether the proper term is program management or project management. To settle the matter and not create issues, in the course of this book, we will use the term program/project management. It may seem redundant, but it covers both bases.

There are several case studies throughout the book. These case studies came from a variety of government, aerospace, and commercial companies/groups, and no company should be inferred from a given case study, unless the company name is specifically mentioned. In some instances, the case study may represent a collection of very similar stories from several different companies.

Lastly, we want to emphasize that this is not a book on how to perform agile development, but how to manage the process of agile development and how managers can facilitate successful and efficient agile development programs/project. This book is written to give managers the tools required to be successful as an agile manager.

Denver, CO — James A. Crowder
Englewood, CO — Shelli Friess

Contents

Chapter 1
Introduction: The Agile Manager

Modern productivity teams demand modern leadership, one that understands modern development needs, stresses, teams, and other aspects of agile development team dynamics [22]. The purpose of the book is to introduce managers to the new productivity environments, including geographically, culturally, and generationally diverse teams. The Agile development paradigm embodies a set of principles which at first may seem contrary to classical business practices:

1. Satisfy the customer through early and continuous delivery of software capabilities and services through short software "sprints," lasting from 2 weeks to 2 months, with a preference toward shorter sprints.
2. Embracing the environment of change. The Agile development process harnesses change to gain a competitive advantage in the software development marketplace [28].
3. Business development, management, and developers must cooperate and collaborate throughout the development project.
4. Communication needs to be face-to-face, even if that means teleconferencing over diverse geographical locations. Face-to-face communication is essential for efficiently and effectively conveying necessary information across an agile development team.
5. The Agile development process is designed to allow a sustainable, constant pace development throughout the entire development project.
6. The primary measure of success (Earned Value) is working software and capabilities, *NOT* Equivalent Software Lines of Code (ESLOC).
7. Agile development does *NOT* mean a lack of design. A good design (architecture) enhances agility and allows continuous attention to technical excellence.
8. Simplicity is essential in agile development. The importance of agile software development demands the art of maximizing the work *NOT* done.
9. The best architectures, requirements, and software designs emerge from self-organizing teams, *NOT* from management mandated team structures.

© Springer International Publishing Switzerland 2015

J.A. Crowder, S. Friess, *Agile Project Management: Managing for Success*,
DOI 10.1007/978-3-319-09018-4_1

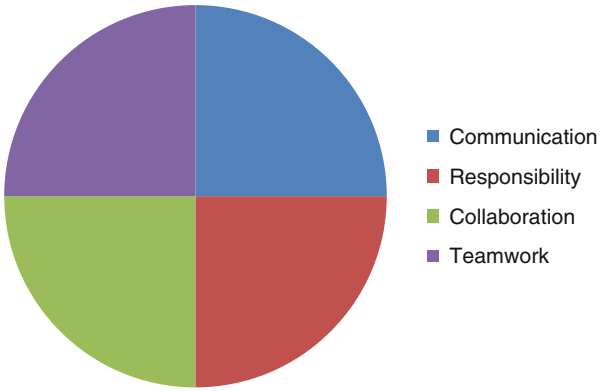

Fig. 1.1 What Agile development projects are supposed to teach you

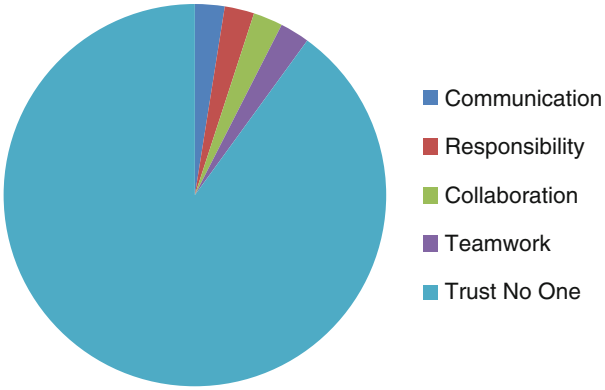

Fig. 1.2 What Agile projects that are managed badly actually teach you

The following illustrates this notion, as well as the results of not following these concepts in managing agile development teams (Figs. 1.1 and 1.2).

1.1 Agile Development Demands Agile Management

Agile software design methods are now commonplace; however, management skills, in general, have not kept pace with the advances in software development practices. There is a major push among companies, both government and commercial, to embrace the concepts of diversity and inclusiveness. Managers need to

be trained in how to manage teams of diverse personnel. Much has been made of managing different personalities, but managers need to be aware of soft people skills, how to manage them, use them effectively, and how they affect people of different backgrounds [29].

It is commonplace to have to work with teams across geographically, ethnically, generationally, and culturally diverse backgrounds within the same team, not to mention a range of skill levels. This book will be helpful in understanding how to manage an agile development team that includes such dynamics [30]. This book will take the project/program manager beyond the concepts of transformational leadership, which provides methodologies to connect to employees' sense of identity, to include human psychological concepts such as "Locus of Control," which will help the manager understand team members' view of how to manage their "world" contributes (enhances or detracts) from their ability to work within team dynamics.

Agile design methods have been utilized since the mid-1990s, and yet program/project managers have been slow to adapt to the changes required for effective agile development [31]. And while there are basic management techniques like Scrum available for the mechanics of managing agile teams, none of these address the dynamics agile development, which include:

- How to choose the right agile development team
- How to facilitate, not control, an agile team
- How to trust your team: trust is an important factor in change [18]

The agile development process demands trust, transparency, accountability, communication, and knowledge sharing [50]. Allowing agile teams to develop these qualities requires understanding and allowing people to evolve and exercise aspects of their internal development processes like Locus of Control. Locus of Control refers to the extent to which individuals believe that they can control events that affect them. Individuals with high Locus of Control believe that events result primarily from their own behavior and actions. Those with a high external Locus of Control believe that powerful others, or chance, primarily determine events that affect them [62].

The purpose of the book is not to emphasize any particular Agile Development Management style (e.g., Scrum), but instead to investigate and present methods for effective Agile team development and management philosophies. Just because you use Scrum does not mean you are Agile. Scrum is a one of many methods for managing Agile Software Development teams, assuming you have an agile development team. Many management organizations confuse this, with disastrous results. To overcome this and to provide the Agile Manager with the skills required to efficiently manage agile development teams, the book includes discussion agile management techniques [18], the psychology of agile management (Locus of Empowerment [53]), as well as new metrics and methodologies for measuring the efficacies of Agile Development teams (agile EVMS).

1.2 Software and Systems Engineering: Where Did They Come From?

Many think the discipline of Software Engineering is relatively new, and that before the invention of "modern" software techniques, the discipline was nothing more than structured coding. But actually, the first two conferences on Software Engineering were sponsored by the NATO Science Committee in 1968 and 1969 in Garmisch, Germany [58, 74].

1.2.1 Software Engineering History

In 1968 and 1969, the NATO Science Committee sponsored two conferences on software engineering, seen by many as the official start of formal discipline of Software Engineering. The discipline grew, based on what has been deemed the "Software Crisis" of the late 1960s, 1970s, and 1980s, in which very many major software projects ran over budget and over schedule; many even caused loss of life and property. Part of the issues involved in software engineering efforts throughout the 1970s and 1980s is that they emphasized productivity, often at the cost of quality [75]. Fred Brooks, in the Mythical Man Month [14], admits that he made a multi-million dollar mistake by not completing the architecture before beginning software development, a major problem that has been repeated over and over, even today. We will discuss the notion of the importance of having a complete architecture on agile development later in the book. This does not mean that the architecture can't change, as it often does throughout the project, but system's engineering must keep up with changes so that the development teams clearly understand the architecture they are developing to during every sprint [19]. We will discuss this at length in subsequent chapters, as the intent here is just to provide a brief history.

1.2.2 System Engineering History

Systems engineering began its development as a formal discipline much earlier than software engineering, during the 1940s and 1950s at Bell Laboratories. It was further refined and formalized during the 1960s during the NASA Apollo program. Given the aggressive schedule of the Apollo program, NASA recognized that a formal methodology of systems engineering was needed, allowing each subsystem across the Apollo project to be integrated into a whole system, even though it was composed of hundreds of diverse, specialized structures, sub-functions, and components. The discipline of system engineering allows designers to deal with a system that has multiple objectives, and that a balance must be struck between objectives that differ wildly from system to system. System engineering seeks to optimize the

overall system functionality, utilizing weighted objectives and trade-offs in order to achieve overall system compatibility and functionality [19]. During the 1970s and 1980s as engineering systems continued to increase in complexity, it became increasingly difficult to design each new system with a blank page. As system quality attributes like reliability, maintainability, re-usability, availability, etc. became more and more important, the concept of Object-Oriented design techniques was developed. The first Object-Oriented languages began to emerge during the 1970s and 1980s. By the 1990s, the first books on Object-Oriented Analysis and Design (OOAD) were published and available. Unfortunately there were many different OOAD methodologies. There was no consistency among methods. At one point in the 1990s, there were over 50 different OOAD methods. This became increasingly difficult for the Department of Defense (DoD), because contract proposals from competing contractors utilized entirely different OOAD methods to design their systems, making comparison between proposals nearly impossible. Finally, in 1993, the Rational Software Company began the development of a Unified Modeling Language (UML), based on methodologies by Grady Booch [13], James Rumbaugh [64], and Ivar Jacobsen [41], coupled with elements of other methods. Here the Rational Software Company simplified current methods from several authors into a set of OOAD methods that included Class Diagrams, Use Case Diagrams, State Diagrams, Activity Diagrams, Data Flow Diagrams, and many others.

1.3 The Need for New Leadership

While Software and Systems Engineering has matured and evolved over the decades to accommodate a faster-paced, ever-changing development environment, program/project management still tends to cling to rigid management techniques and principles that critically hamper the agile process [51]. A great example is described below:

Case Study #1: The Non-Agile Manager

Project Length	12 months
Number of Sprints	8 Sprints
Number of Teams	4 Teams
Average Sprint Duration	6 weeks

Description During the planning for Sprint #4, Team #3 discovered that team #4 had a capability scheduled for Sprint #5 that Team #3 needed for their development in Sprint #5 to accommodate their sprint #5 development. Team #3 negotiated with team #4 and they found a set of capabilities that team #4 had scheduled for sprint #4 that were not needed till sprint #6. The capabilities team #3 needed constituted the same number of story-points, and similar complexities, and therefore would not overly tax team #4's sprint work to swap the work between sprint #4 and #5 to accommodate team #3. Team #4 accomplished all of their development for sprint #4. During the progress evaluation with the program manager, the manager was very

upset that the planned work had not been accomplished, but it had been changed, with work moved from Sprint #4 to Sprint #5. Team #4 explained that the work represented the same number of story-points and similar complexity, therefore not perturbing the overall cost and schedule of the program. When the manager pushed back, still upset that the work scheduled had not been accomplished, both Team #3 and Team #4 explained that this is a classical part of the Agile Development Process, being flexible to move capability development around, based on changing needs and requirements. The manager's reply was (and hence the reason for the book), "Then I guess we need a more *rigid* agile process."

1.3.1 Agile Management: Leader, Manager, Facilitator

Many mangers shudder at the thought of agile development projects, feeling like their authority has been eroded. I have heard more than one project manager declare, when the agile methodology is explained, ask, "So what am I going to do?" Many managers are used to being intimately involved in the development process, even though they may, or may not, actually have been software developers [34]. It is true that project/program managers must learn to adapt and take on different roles in the world of agile development, becoming facilitators and leaders and not so much traditional managers. The notion of classical line management, or boss, which many managers still cling to, is no longer relevant in the paradigm of agile software development projects. Figure 1.3 illustrates this, albeit a bit dramatically.

For effective management of agile development projects, the manager (we'll refer to the manager as the Agile Manager throughout the book) needs to have many skills [33], including effective communication, a diplomat, and other skills shown in Fig. 1.4, and will be explained in detail in Chap. 2.

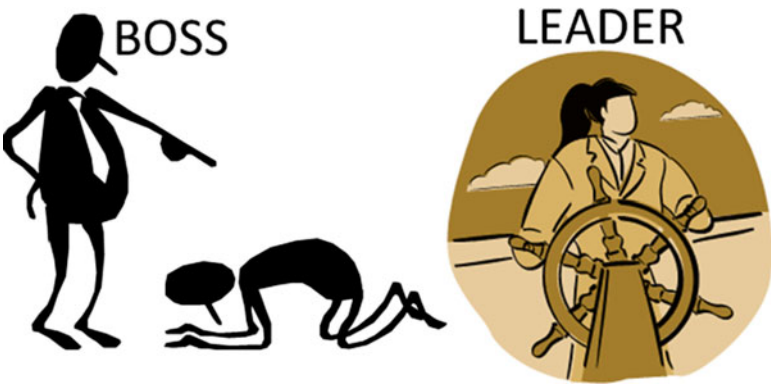

Fig. 1.3 Boss vs. Leader

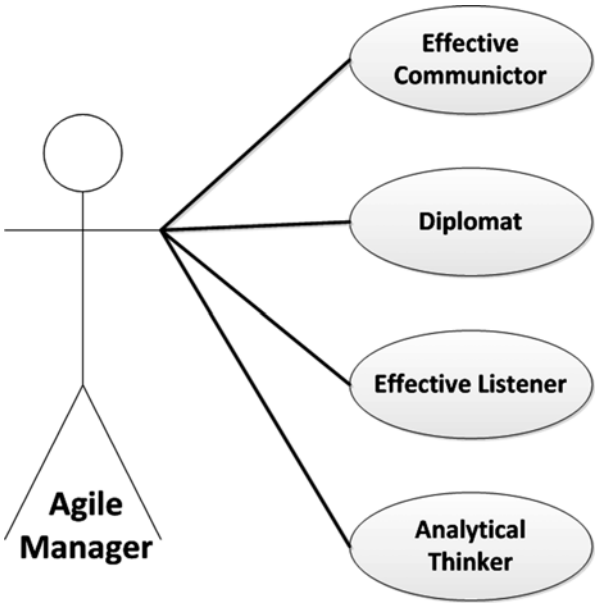

Fig. 1.4 Skills of the Agile Manager/Leader

The rest of the book is dedicated to an in-depth discussion of the new management paradigm required to enable, encourage, and fully embrace Agile Software Development and make them the successes they can be. To accomplish this, the structure of the book is outlined in the next section.

1.4 Layout of the Book

We have arranged the book to build up to new methods for Agile Software Program/Project management.

Chapter 2—The Psychology of Agile Team Leadership This describes the new "soft" people skills required for modern managers, and how they add/detract from modern agile development. How to recognize the skills, how to utilize the skills, and how to build teams with the right "mix" of personalities and soft people skills for effective and efficient development efforts.

Chapter 3—Understanding the Agile Team Success in the modern development era needs managers and leaders who truly understand what agile development means and how agile teams collaborate, cooperate, and function in various situations, particularly in geographically and culturally diverse environments. Understanding the variety of personalities and soft people skills, coupled with how these manifest themselves across gender, ethnic backgrounds, and cultural and generational diversities, and other considerations, will become essential for modern development leadership

and management. This includes discussion of overall inclusiveness and diversity within the agile development process. Diversity and Inclusiveness are important dynamics that companies are embracing. Building development teams that are not just effective but embrace the concepts of diversity and inclusiveness are important [35], but most leaders and managers have not been trained for the dynamics these bring (both good and bad) to teams.

Chapter 4—Productivity Tools for the Modern Team Providing an agile development team with tools to be productive goes beyond handing each one of them a laptop with compilers. Communication and collaboration tools, whether face-to-face or geographically diverse, are crucial in modern teams. Here we discuss collaboration tools and other tools that will be crucial today and in the future. The proper use of Information Systems can provide management and leadership with effective ways of monitoring and managing teams. Here we discuss the new management information systems environments and tools that are available, will be available in the future, and need to be utilized within the new agile development paradigm.

Chapter 5—Measuring Success in an Agile World: Agile EVMS The Earned Value Measurement System (EVMS) has become a mainstay in Commercial and Government groups to measure progress and success of a project. EVMS is effective (albeit subjective) measure, but does not play well with agile development efforts, due to its requirement of static schedules and work plans. Here we introduce a new paradigm for EVMS that will accommodate and be effective in measuring progress and problems within agile development efforts.

Chapter 6—Conclusion: Modern Design Methodologies One of the things that need to be understood by leadership and management in the future is that just because you deliver a product on time and on budget doesn't mean the project was an overall success. Delivering a product on budget and on schedule but decimating a development team is not, in the long run, a success for the company. Managers and Leaders must understand all aspects of development teams for long-term success.

Chapter 2
The Psychology of Agile Team Leadership

For modern managers, one has to adopt a new philosophy, or psychology, for dealing with agile development teams. While process is important to ensure the team delivers quality software that meets customer requirements, it is important to understand that the Agile Method is geared around more of an informal approach to management, while putting more time, effort, and emphasis on flexibility, communication, and transparency between team members and between the team and management. It promotes an environment of less control by managers and more facilitation by managers. The role of the manager takes on a new psychological role, one of removing roadblocks, encouraging openness and communication, and keeping track of the change-driven environment to ensure that the overall product meets in goals and requirements, while not putting too much control on the ebb and flow of the agile development process. Change is no longer wrong, the lack of ability to change is now wrong. Here we discuss the new "soft" people skills required for modern managers, and how they add/detract from modern agile development. How to recognize the skills, how to utilize the skills, and how to build teams with the right "mix" of personalities and soft people skills for effective and efficient development efforts [71].

2.1 Individuals over Process and Tools

Companies have spent decades designing, creating, implementing, and executing tools required to bid and manage development projects. One major category of tools is prediction tools like *CiteSeer*© and COCOMO© (Constructive Cost Model) that have been used since the late 1900s to provide "objective" cost bids for software development. A later version of COCOMO, COSYSMO© (Constructive Systems Engineering Model), attempts to provide objective systems engineering bids also. All of them are based on the antiquated notion of Software Lines of Code (SLOC). Productivity metrics are all based on the lines of code written/unit time. They try to estimate the life-cycle cost of software, including designing, coding, testing,

© Springer International Publishing Switzerland 2015

J.A. Crowder, S. Friess, *Agile Project Management: Managing for Success*,
DOI 10.1007/978-3-319-09018-4_2

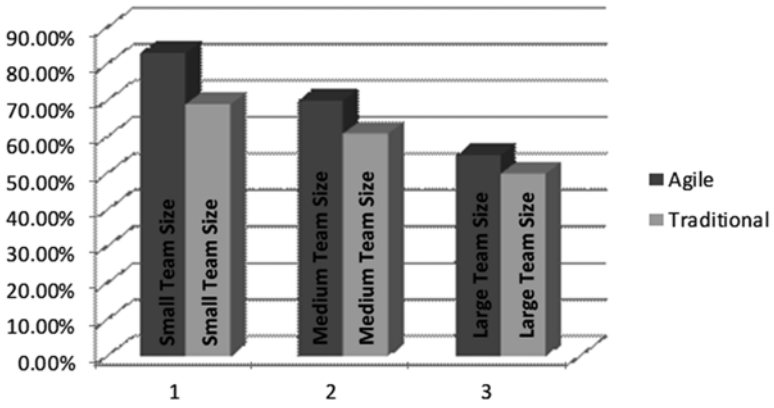

Fig. 2.1 Efficiencies between traditional and agile development

bug-fixes, and maintenance of the software. But ultimately it comes down to Software Lines of Code/Month (SLOC/Month). While many will claim these are objective tools for helping to determine the staff loading necessary for a software/systems development project. In each tool there are dozens of parameters which are input by the operator, each of which has an effect on the outcome of the cost model. Parameters like efficiency (average SLOC/Month), familiarity with the software language used, average experience level, etc., can be manipulated, and usually are, to arrive at the answer that was determined before the prediction tool was used [43].

Many other tools are utilized to measure the performance (cost and schedule) of projects once they are in execution. These measurement tools measure how the project is progressing against its preestablished cost and schedule profile, determined in the planning phase of the program/project. What none of these tools, cost estimation, performance metrics tools, etc., take into account is the actual agile team and their dynamics. The makeup of the each agile team and the facilitation of each team is as important, if not more important, than the initial planning of the project. If the Agile Manager/Leader is not cognizant of the skills necessary not to just write code, but to work cohesively as an agile team, then success is as random as how the teams were chosen (usually by who is available at the time). Grabbing the available software engineers, throwing them randomly into teams, and sending them off to do good agile things will usually result in abject failure of the project, or at least seriously reduced efficiency. This may sound like an extreme example, but you would be surprised how many agile development projects are staffed in just this fashion. Many managers point to the following graph (Fig. 2.1) as the reasons not to go to the expense of changing all their processes to accommodate agile development.

While in each category agile development produces a higher efficiency than traditional software development methods, the increase is not as dramatic as the promises made by agile advocates and zealots. Classical managers find this graph disturbing and feel smugly justified in their classical software development/execution/control methods. This is especially true for large teams. The data for this graph was taken from

Fig. 2.2 Four main components of the agile development process

50 of each size project, both agile and traditional. What are not taken into illustrated by this graph are the management methods utilized across the traditional vs. agile programs/projects: the team makeup, how the teams were chosen, or any discussion of the types of issues that were encountered during the development process. And while it's clear that under any team size agile development has increased efficiency over traditional methods, and, as expected, smaller team sizes produce better results with agile methods, understanding the true nature of the agile team process and applying the psychology of agile management can achieve even greater efficiencies.

Placing the emphasis on the individuals in the agile development teams rather than on process or tools means understanding people, recognizing their strengths (not only in terms of programming skills, but also in terms of soft people skills), and understanding the differences between people of different backgrounds and how the differences affect team dynamics. This is the first generation where it is possible to have 60-year-old software engineers in the same agile development teams with software engineers in their early 20s. The generational differences in perspectives can severely hamper team dynamics, and therefore team efficiencies will suffer greatly if they are not dealt with appropriately and the team members are not trained in how to function in an agile development team. All members of the teams need to be able to understand and come to grips with four main components of agile development, illustrated below in Fig. 2.2. While there are other components that are important,

without a good handle and agreement on these, agile development teams are in trouble from the start. These and other issues relating to team dynamics will be explored in Chap. 3.

As explained, Fig. 2.2 represents four of the major components of the Agile Development Process that must be embraced by the agile development team in order to have a successful and efficient development process. As important are the skills, or philosophies, that the manager of the program/project must embrace and practice in order for the teams to be able to function in an agile environment and have the best chance for success. Figure 1.4 provided a high-level look at the skills of the effective agile manager/leader. The descriptions of these skills are:

1. **Effective Communicator**: The effective communicator fosters and increases trust, is transparent, considers cultural differences, is able to be flexible in delivery of communications, encourages autonomy and role models, exudes confidence to solve problems and handle whatever comes up, and has the courage to admit when they are not sure and willingness to find out. They are willing to work side by side versus competitive with followers. They have the ability to communicate clear professional identity and integrity, their values are clear, and so are their expectations. The effective communicator communicates congruence with values and goals, as well as being a role model of ethical and culturally sensitive behavior and values.
2. **Diplomat**: The diplomat considers the impacts on all stakeholders and how to follow up with all those affected, even if it is delegated. There is willingness to consult cultural experts.
3. **Effective Listener**: The effective listener checks that they understand the meaning being portrayed, and goes with an idea even if they disagree until the whole idea is expressed and the originator can think through the complete thoughts with the leader.
4. **Analytical Thinker**: The analytical thinker must be able to see the forest and the trees. The analytical thinking manager/leader must be able to anticipate outcomes and problems, and explore how they might anticipate handling them, walking through possible solutions. They must initiate Professional Development of team members. They think about the how, not just the what-ifs.

2.2 The Agile Manager: Establishing Agile Goals

For the effective agile project/program manager, it is crucial early on to establish goals and objectives that establish the atmosphere for each sprint development team. Understanding how much independence each developer is allowed, how much interdependence each team member and each team should expect, and creating an environment that supports the agile development style will provide your teams with the best chance for success. Below is a list of agile team characteristics and constraints that must be defined in order for the teams to establish a business or development

"rhythm" throughout the agile development cycle for the program/project. Each will be explained in detail in its own section, but general definitions are given below:

1. **Define and Create Independence**: Independence is something many developers crave. In order for agile development to be successful, there must be a large degree of independence and need to feel an atmosphere of empowerment, where the developers are free to create and code the capabilities laid out during the planning phase of each sprint. This requires a level of trust. Trust that the developers and the leader all have stakeholders in mind. Trust that the developer is working toward the end product [23]. Empowerment at the organizational level provides structure and clear expectations [17]. At the individual level it allows for creativity. Independence means having a voice and yet operating under company structure of policies and procedures. Independence is also a sense of knowing that the developer is good and what they do. There is no need to check in too frequently with the leader, but enough to keep the teamwork cohesive.
2. **Define and Create Interdependence**: While independence is a desired and necessary atmosphere for agile teams, the agile manager must also establish the boundaries where individual developers, and development teams, must be interdependent on each other, given that the goal is to create an integrated, whole system, not just independent parts. Interdependence is being able to rely on team members [16]. The end goal will require a level of commitment from each person with a common mission in mind. The trust that all individuals on the team have all stakeholders in mind. This gets the whole team to the common goal and reduces each member motivated solely for their own end goal.
3. **Establish Overall, Individual, and Team Goals and Objectives**: setting the project/program overall goals, team goals, and individual goals and objectives up front and at the beginning of each sprint helps each team and individual team member to work success at all levels of the program/project. This can help to identify strengths of individuals so that the team can use its assets to their highest production. This also allows room for individual development and growth along with a place for passions. This also sets up clear expectations, say, of the overall and team goals. There may be some individual development that is between the leader and the developer that stays between them. This would also build individual trust between members of the team and between the leader and the developers.
4. **Establish Self-Organization Concepts**: self-organizing teams is one of the holy grails of agile development teams. However, self-organization is sometimes a myth, mostly because teams are not trained into how to self-organize. People do not just inherently self-organize well. If not trained, the stronger personalities will always run the teams, whether they are the best candidates or not [24]. Self-organization can be nearly impossible when there are very structured people coupled with not-so-structured people. There may be some work that the leader can do to promote self-organization. Part of that is opening communication, building dyads, calling behavior what it is, and being transparent so that others will follow. It may be helpful for team members to get to know strengths of other members and how each member can be helpful to each individual.

5. **Establish Feedback and Collaboration Timelines and Objectives**: given the loose structure and nature of agile development, feedback early and often is crucial to allowing the teams to adapt to changing requirements or development environments. Also, customer collaboration and feedback at each level in the development allows the teams to adjust and vector their development efforts, requirements, etc., to match customer expectations at all points in the development cycle. Feedback timelines can increase trust and clarify all expectations. It is nice to know when you need to change a direction, when you need to change it, instead of later when you had already put so much work into the project. The more feedback is modeled and practiced, the more natural it becomes and becomes more automatic. This builds on the independence and interdependence of the team and individual stakeholders.
6. **Establish Stable Sprint Team Membership**: choosing the right teams is important for success in an agile development program/project. Creating teams that are not volatile (changing members often) is essential to continued success across multiple sprints. If the teams constantly have to integrate new members, efficiency will suffer greatly. New expectations and explanations will take up much time that could be used for developing. A trusting team can be an efficient team. The more often it changes the more work needs to be done to build the trust. There may be increased commitment from those that work on a cohesive team with high trust levels and knowledge of one another [18].
7. **Establish Team's Ability to Challenge and Question Sprints**: if the teams are going to be allowed individual and team empowerment, then they must be allowed to challenge and question sprint capabilities and content across the development cycle. Forcing solutions on the teams fosters resentment and a lack of commitment to the program/project. If you've built the right team, you should listen to them. It seems more productive to work on something that makes sense to you, instead of handed down by others. The ability to challenge and question will lead to better understanding and more commitment to the end goal.
8. **Establish an Environment of Mentoring, Learning, and Creativity**: invariably, teams are composed of a combination of experience levels. This provides an excellent atmosphere of mentoring and learning, if the agile manager allows this. This must be built into the sprint schedules, understanding that an atmosphere of mentoring, learning, and creativity will increase efficiencies as the team progresses, not just on this project, but on future projects as well, as the team members learn from each other. Keep in mind that experienced developers can learn from junior developer too, as the more junior developer may have learned techniques and skills that were not previously available to more senior developers. The learning environment promotes growth. An environment that fosters learning decreases negative feelings of one's self, and thus other people. An environment that fosters learning isn't run by guilt, or feelings of not being good enough, or doing something wrong. A learning environment allows people to grow and the mentor helps the individuals self-determine the direction they want to develop. The learning environment will foster older members learning from younger members as well. People will want to learn more and more and

reduce competitiveness that can destroy a team. The competitiveness can come out as a good product not team dynamics. Transparency can help individuals feel more comfortable with learning. This can show that is ok to have areas of development and that everyone has room to grow.

9. **Keep Mission Vision always out in Front of Teams**: many believe that an established architecture is not required for agile development. This is absolutely wrong; a solid architecture is even more important during agile development, so each team and team member understands the end goals for the system. However, in order for the architecture and software to stay in sync, the systems engineering must also be agile enough to change as the system is redesigned (or adapted) over time [19]. Agility is not free from structure but the ability to move about within the structure.

2.3 Independence and Interdependence: Locus of Empowerment

Locus of Empowerment has been conceptualized as a function of informed choice and self-determination and has been linked to the concepts of self-efficacy and locus of control as it applies to agile team membership [6]. Self-understanding and empowerment, in relation to development opportunities and factual strength/weakness assessment, represents an important underlying component of feelings of self-empowerment within an agile development team [74]. Locus of empowerment and its counterpart, Locus of Control, help to establish both independence and interdependence for agile team members. Determining those things each team member is "empowered" to make decision on and work independently provides each person with a sense of autonomy, allowing them to work at their peak efficiency without interference or too much oversight control over their work. Establishing the Interdependence, or those things which are outside of the control of the team member, defines communication lines and those things which are necessary to collaborate on, or get inputs from other team members to facilitate integration and validation of "system-wide" capabilities [7]. What follows is a discussion of Locus of Empowerment. Locus of Control will be discussed in Sect. 2.4.

2.3.1 Locus of Empowerment

The notion of Locus of Empowerment is an interactive process that involves an individual team member's interaction with the team and the manager [70], allowing each team member to develop a sense of acceptance into the team, develop a sense of where they belong in the team, self-assessment of skills, and determination of their self-efficacy—their ability to function and participate both on an individual level and as part of an agile development team [49]. These allow each

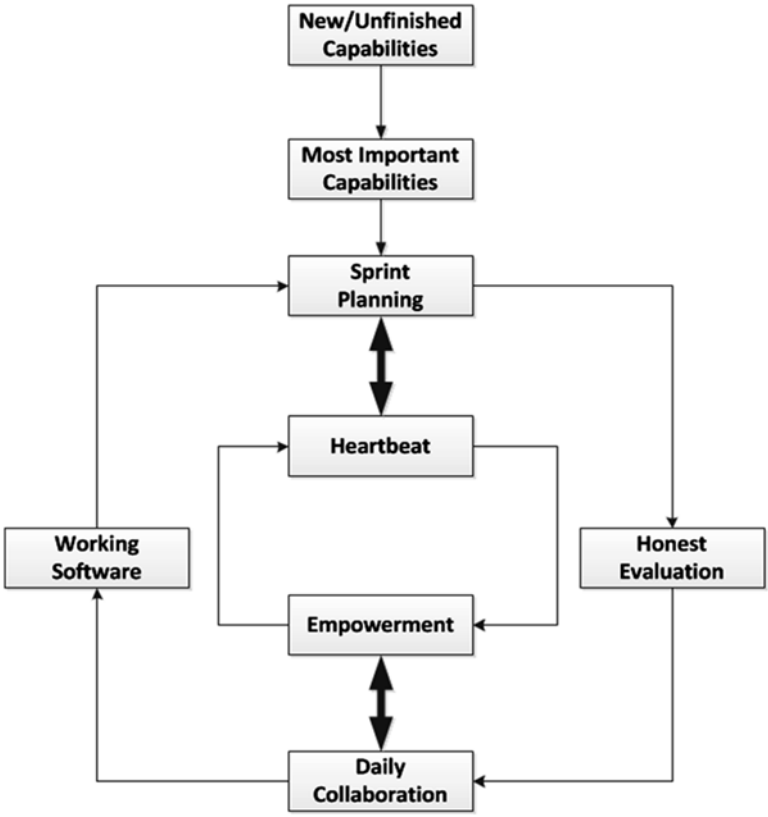

Fig. 2.3 The agile development process with empowerment

individual team member to participate with others, based on their understanding of their independence and interdependence from and to the team, allowing them to deal with the daily, weekly, monthly, etc., rhythms of the agile development cycles throughout the program/project [53].

The process of team and team member empowerment is a continual and active process; the form and efficacy of the empowerment process is determined by past, current, and ongoing circumstances and events [69]. In essence, the empowerment process is an ebb and flow of independence and interdependence relationships that change throughout the agile development process, including each daily Scrum, each Sprint planning session, and each Lessons Learned session, throughout the entire agile development cycle of the program/project. Figure 2.3 illustrates this process.

In Fig. 2.3, empowerment becomes an integral part of the overall agile development process, with evaluation of the team members' abilities, roles, independence, and interdependence, based on the capabilities needed to be developed within a given Sprint, the honest evaluation of skills and abilities; i.e., how to develop the heartbeat, or development rhythm required for each development Sprint. Without an environment of Empowerment, the team has no real focus, since each team member

does not have a sense of what they are individually responsible for, what the other team members are individually responsible for, and what communication is required throughout the Sprint development [68]. There will eventually be a breakdown of the team, a loss of efficiency, and the team will not be successful in their development efforts within cost and schedule constraints. Next we discuss the concepts of goal setting for an agile development project: project/program, team, and individual goals within the context of Locus of Control.

2.4 Overall, Individual, and Team Goals: Locus of Control

As explained above, the very nature of agile software development is to create a loose structure both within each Sprint team and across the Sprint team structure. The purpose of agile development is to allow developers, and subsequently the system being developed to adapt and change as requirements, features, capabilities, and/or development environment change over time (and they will change). However, this does not mean that there are not system-level, team-level, and individual-level goals at each point in time. In fact, it is more important in agile development to have well-defined goals as teams and individual developers write and test software, to ensure the software integrates and, more importantly, creates a set of capabilities and a system the customer wanted and is paying for. Customer and cross-team collaboration and feedback at each level is crucial to allow the teams to adjust, either from customer needs or inter-team needs across the agile Sprint developments. Again, independence and interdependence is essential for overall successful development. Further refinement of the Empowerment concept is to define, for each individual developer, what things are within their own control, and those things are outside of their control, even if they affect the individual.

This notion of internal vs. external control is called "Locus of Control." Locus of control refers to the extent to which individuals believe that they can control events that affect them [63]. Individuals with a high internal locus of control believe that events result primarily from their own behavior and actions. Those with a high external locus of control believe that powerful others, fate, or chance primarily determine events (in this case other team members, other teams, the program/project manager, and/or the customer). Those with a high internal locus of control have better control of their behavior, tend to exhibit better interactive behaviors, and are more likely to attempt to influence other people than those with a high external locus of control; they are more likely to assume that their efforts will be successful [12]. They are more active in seeking information and knowledge concerning their situation.

Locus of control is an individual's belief system regarding the causes of his or her experiences and the factors to which that person attributes success of failure. It can be assessed with the Rotter Internal–External Locus of Control Scale (see Fig. 2.4) [63]. Think about humans, and how each person experiences an event. Each person will see reality differently and uniquely. There is also the notion of how one interprets not just their local reality, but also the world reality [79]. This world reality may be based on fact or impression.

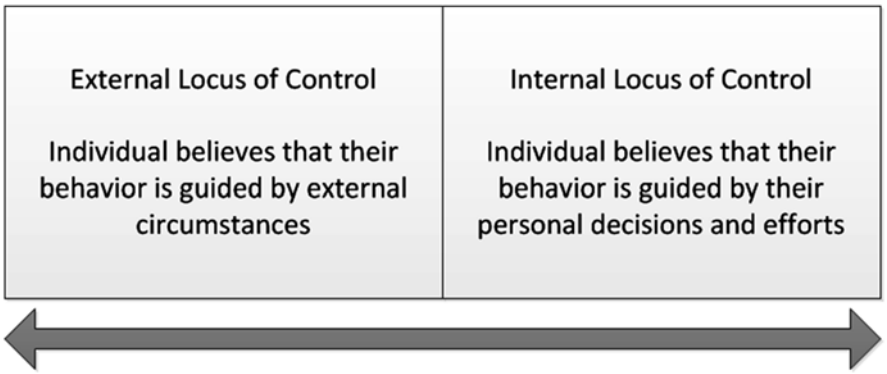

Fig. 2.4 The locus of control scale

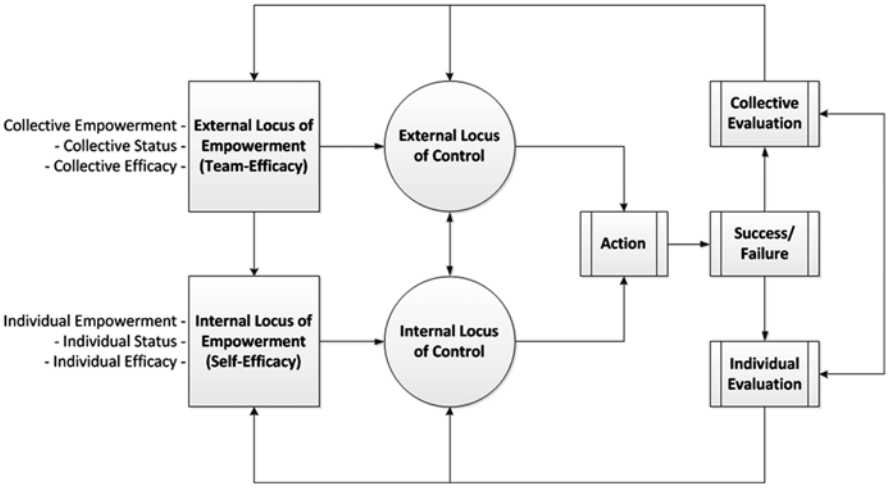

Fig. 2.5 Locus of control within an empowerment cycle

For further thought let's then consider Constructivist Psychology. According to "The internet Encyclopedia of Personal Construct Psychology" the Constructivist philosophy is interested more in the people's construction of the world than they are in evaluating the extent to which such constructions are "true" in representing a presumable external reality. It makes sense to look at this in the form of legitimacies. What is true is factually legitimate and what is people's construction of the external reality is another form of legitimacy. In order to have an efficient, successful agile development team [62], each member must understand and accept their internal and external level of Locus of Control, as well as their Locus of Empowerment level. Figure 2.5 illustrates how this flows throughout the Sprint development cycles.

How an individual sees the external vs. internal empowerment drives their view of internal vs. external Locus of Control. During each development cycle, evaluations are made (whether the individual is aware of it or not) as to their internal and external Empowerment, and subsequent Locus of Control. Actions are determined, based on this self-assessment, and self-efficacy determination. Based on the results of their efforts, individuals, as well as the team, and the entire program/project reevaluate the efficacy of the levels of internal vs. external Empowerment that are allowed, and adjustments are made. These adjustments to Empowerment levels drive changes in Locus of Control perception, which drives further actions. This process is repeated throughout the project/program. The manager must understand this process and make the necessary adjustment so that each individual can operate at their peak self-efficacy, as well as support team efficacy, providing the best atmosphere for successful development.

2.5 Self-Organization: The Myths and the Realities

One of the holy grails of agile development is self-organizing teams. Many software developers dream of having a team with complete autonomy, able to organize however works for them, completely without management involvement or interference. However, what most developers fail to realize is that given to their own devices, without training as to how to organize and what "organizing" actually means, most would fail miserably. Often, agile development efforts fail, even with efforts to educate the team about agile principles [53]. That is because the team doesn't fail because they don't understand agile software development. It's because they don't understand human nature and the difficulties in taking a team of highly motivated, strong personalities, and get them to automatically give up their egos, preconceived notions, and past experiences, and embrace the agile team dynamics required to put together a highly successful agile development effort. We call this "Agile Team Dysfunctionality," and there are many common dysfunctions that plague improperly trained teams and team members. Figure 2.6 illustrates several of the most serious dysfunctions, most of which come both from basic human nature and from people's experience with work on programs/projects in the past. Nothing drives failure of agile development like past failures. Remember Figs. 1.1 and 1.2. Teams that have experienced Fig. 1.2 are hard pressed to throw off their suspicions and embrace agile development processes, team dynamics, and the entire agile agenda fresh. Management must be cognizant of these dysfunctions and work within the teams to dispel them.

Inability to recognize or deal with agile team dysfunctions can destabilize the team(s) and derail the agile development process faster than anything else. Keeping a stable set of Sprints teams is important, as constantly changing out team members radically changes team dynamics, and affects both personal and team Empowerment and Locus of Control [58]. Section 2.6 discusses the concept of stable team membership.

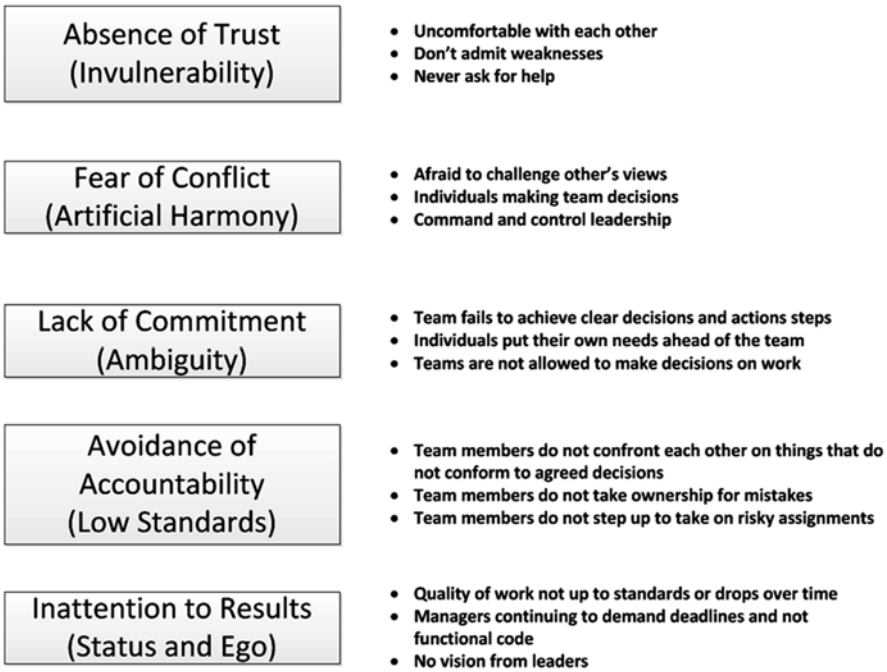

Fig. 2.6 Common agile team dysfunctions

2.6 Creating a Stable Team Membership: Containing Entropy

As previously discussed, it is vital to choose the right teams for any program/project, but it is even more important for agile development. Teams with stable memberships across Sprints are vital, as team members develop trust over time, gain an understanding of each member's strengths and idiosyncrasies, and, with proper training, mentorship, and facilitation by the manager, settle into an agile development "rhythm" throughout the program/project. If the team has to integrate new members, efficiency will always suffer until the new team member is properly integrated into the rhythm. New expectations are created; the new person will most likely have an entirely different notion of Empowerment and Locus of Control than the previous team member, throwing the overall team out of balance. A stable team can be a trusting and efficient team [56]. There is generally an increase in commitment over time with a stable team [52]. In order to facilitate creation of stable agile sprint teams, the Agile Manager must recognize, understand, and know how to deal with the dysfunctionalities discussed in Sect. 2.2. For each dysfunction, the Agile Manager must take on a role, or provide guidance that dispels the dysfunction and allows the team to move toward and independent cohesiveness between the team members [60]. Figure 2.7 illustrates the Agile Manger's role in dealing with classical agile team

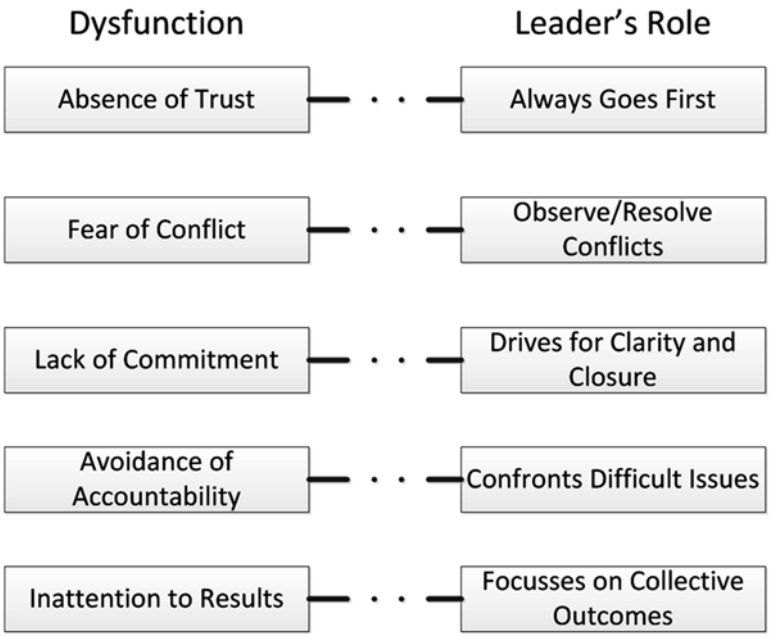

Fig. 2.7 The Agile Manager's response to team dysfunctions

dysfunctions, creating a team that works together, in Empowered independence and dependence, to develop software in an efficient agile environment.

As depicted in Fig. 2.7, for each of the agile team dysfunctions described in Fig. 2.6, Fig. 2.7 illustrates the Agile Manager's response required to eliminate the dysfunction and allow the agile development teams to function effectively and efficiently:

1. **Absence of Trust**: In order to build trust within the teams, the Agile Manager must always be willing to take the lead and prove to the team members that they will "roll up their sleeves" and do whatever is necessary to either get the program/project moving or to keep it moving along.
2. **Fear of Conflict**: Many developers are fearful of bringing up issues, not wanting to start controversy within the team. Many people, particularly strong introverts, may internalize the conflict, never bringing it up, but eventually the conflict will drive controversy between the developers, create a lack of trust, and may drive the team to withdraw from each other, destroying the collaborative nature of agile development teams. In order to diffuse these situations before they begin, the Agile Manager must be observant and cue in on body language and utilize the soft people skills like paying attention to changes in personal habits, language, friendliness, and other clues apparent between team members, and facial expressions to understand when such nonverbal controversies exist and work to resolve the conflict before they begin to negatively impact the development efforts.

3. **Lack of Commitment**: A lack of commitment to either the agile development team or the agile process in general can destroy an agile program/project before it gets started. Observing a low quality of work, absenteeism, lack of willingness to communicate, or constantly seeming to be overwhelmed by the volume of work may be indications of a lack of commitment. The Agile Manager needs to understand the developer's reasons for the lack of commitment, clarifying for the developer what is expected, clearing up any misconceptions the developer may have. In the end, if the Agile Manager does not feel they have dispelled the lack of commitment, the developer must be removed from the team or there is little hope for successful agile development. I know this sounds harsh, but agile only works if all parties have a buy-in to the agile development process.
4. **Avoidance of Accountability**: There may be issues getting developers to step up and take on rolls of responsibility within the agile teams because they are afraid that if they take responsibility for the team's activities during a given Sprint and there are problems, they will be punished. This lack of accountability needs to be dealt with in order for the Sprint development teams to develop a good business rhythm and operate effectively. It is up to the Agile Manager to confront issues, while not assigning blame or punishment, but working through difficult issues, helping each developer learn from the issues in order to solidify the teams and allow the developers to grow and mature as members of an agile development team. This will pay off in the future as each developer becomes more embedded in the agile process and learns to be effective in and excited about agile programs/projects.
5. **Inattention to Results**: Some developers like the agile team process because they feel they can just write code and let other people worry about the details, results, testing, etc. But, it is vitally important that the entire team focus on the results: working, error-free code with capabilities required for each Sprint that can be demonstrated. If any of the developers/team members are not focused on the results, the team will never develop a good agile development rhythm. Also, one member being inattentive to details and results will breed mistrust between the members, reducing the effectiveness of the team(s). Therefore, the Agile Manger must keep the program/project vision in front of all developers and teams, making sure everyone is marching down the same path, ensuring that the collective outcomes of all the Sprint teams, across all of the Sprints, integrate together and are heading toward a common, customer-focused goal.

2.7 Challenging and Questioning Sprints: Individual Responsibility

Creating a team of highly motivated, capable, and experienced developers that are expected to work in an agile development environment and not allowing them the freedom, or Empowerment, to question and/or challenge Sprint capabilities, planning, sequencing, etc., will destabilize the team quickly. The Agile Manager should

not force solutions on the team, for this fosters resentment and breeds an attitude of lack of commitment to the program/project. If you build and train the teams correctly, they should be able to discuss and come to agreement on how capabilities are spread across Sprints, who is the best choice for what role across each Sprint, and to work together when collaboration is needed. The ability to challenge and question leads team members to a better understanding and more commitment to the end goal, not only for each Sprint, but to the entire program/project as well [54].

2.8 Mentoring, Learning, and Creativity: Creating an Environment of Growth

Agile development teams, at least the majority of teams, will be composed of developers at a variety of experience levels. Each member comes with their own strengths and weaknesses and should be provided an atmosphere that not only allows them to succeed, but to grow and learn, both from the experience of developing code for the program/project across the Sprints, but from each other as well. If facilitated correctly by the Agile Manager, the agile development program/project will allow opportunities for mentoring and learning. However, this must be designed into the Sprints, both in schedule and in capability distribution across the team members. Creating an atmosphere of mentoring, learning, and creativity increases efficiencies, as the team progresses through the Sprints, and helps future programs/projects as well. Given the probable diversity of team members, the Agile Manager should make sure everyone has the opportunity and personal attitude of both mentoring and learning from each other. New software techniques brought by junior developers may be necessary for certain capabilities that older more experienced software developers may not be aware of. At the same time, junior developers should also bring an attitude of mentoring and learning, as the experienced developers can aid junior developers from going down disastrous roads already traveled by senior developers. In short, the atmosphere the Agile Manager must *NOT* bring to the agile development teams is illustrated in Fig. 2.8.

2.9 Keeping the Vision in Front of the Team: Ensuring System Integration

I have heard many developers tell me that the one advantage with agile development is that they are free to do what they want, because it isn't necessary to establish a systems and software architecture for agile programs/projects. Such notions lead to serious problems later in the development cycle. Without a systems and software architecture, integration and final testing of the system is problematic at best and normally results in much rework and recoding to create a complete system [26].

Fig. 2.8 The "Rigid" Agile Manager

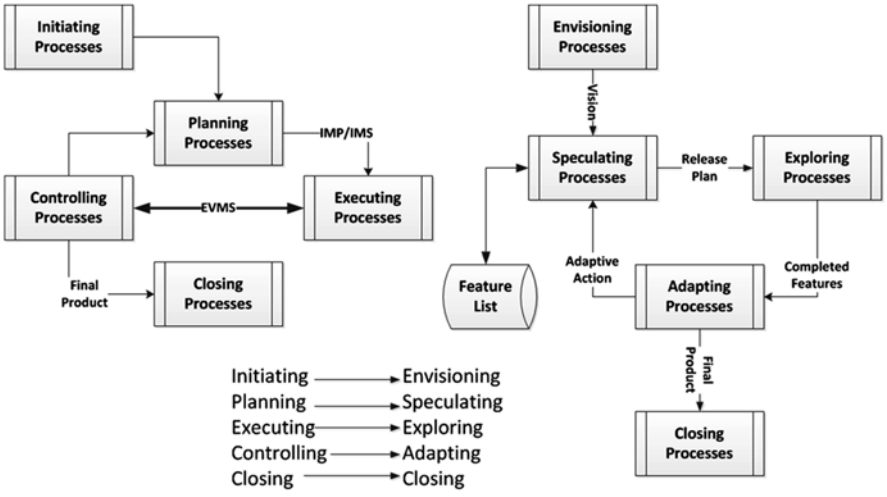

Fig. 2.9 The traditional vs. agile development process

Each team member and each team must understand the end goals for the system. You must remember that agility does not mean there is no structure, but the agile development methodology provides the abilities to move about within a given architecture or structure. Figure 2.9 illustrates the differences between the traditional development process and the agile development process [57].

Every aspect of the classical development cycle has a counterpart for the agile design process, but is designed, or intended, to promote the agile mind-set: one of adaptability to changing requirements or environments. Many are uncomfortable in the agile software development paradigm. Many like the structure of classical software development. So how does one understand who is and is not comfortable with agile development. Can any developer be made to function within the "freedom" of agile? In Chap. 3 we will explore all of the dynamics of agile development teams, how to create, manage, facilitate, and empower agile development teams.

Where the traditional development process involves and is focused on detailed planning, budgeting, controlling, and program/project execution, the agile development process must be adaptive and innovative, deriving solutions to a changing requirements/capabilities baseline, the focus being on working software, not cost and schedule. This is not to say that cost and schedule are not important in agile development, because cost and schedule are always important in any program/project execution. However, the agile development process has much more flexibility to deal with risks or issues that arise than the classical development process, giving the Agile Manager more tools and more opportunities to adjust without major rework in extensive schedules and budgets.

Chapter 3
Understanding the Agile Team

Success in the modern, agile development era required Agile Managers and Leaders who truly understand what agile development means, how to create agile teams, and how agile teams collaborate, cooperate, and function in various situations, particularly in geographically and culturally diverse environments. In addition, they must understand the agile development cycle vs. the traditional development cycle [15]. Figure 3.1 illustrates the classical program/project development cycle.

This program/project development cycle has been used for many decades. Requirements are allocated and/or derived, program plans are made, budgets and schedules established, and you march forward. For this development cycle, change is bad, for it causes rework of the entire development cycle, driving up cost and schedule. And, the farther into the development cycle you are when you discover problems, or changes happen (new or changing requirements), the costlier they are to "fix" both from a cost and schedule perspective. Figure 3.2 illustrates the increasing costs associated with changes, depending on when in the development cycle the changes or problems are encountered.

As mentioned, when problems arise, or requirement changes happen, it drives a replan of the program schedule and a re-estimation of costs to complete the program/project, either an Estimate at Completion (EAC) or an Estimate to Complete (ETC) or both. A new program plan must be put in place, new budgets and schedule, and the development cycle restarts, as illustrated in Fig. 3.3.

Problems occurring (particularly requirements changes) 1/3 or more through the development cycle create major rework for the entire program, as the requirement changes must be flowed throughout all parts of the system and those affected systems, subsystems, configuration items, components, etc., that are affected must be reworked, replanned, rescheduled, and a new baseline budget and schedule created for execution. This can extend by 50 % or more the overall timeline and budget for the program/project. In contrast, the agile development cycle is broken up into small Sprints that are typically 2–6 weeks in length, although there is much debate over the

© Springer International Publishing Switzerland 2015

J.A. Crowder, S. Friess, *Agile Project Management: Managing for Success*,
DOI 10.1007/978-3-319-09018-4_3

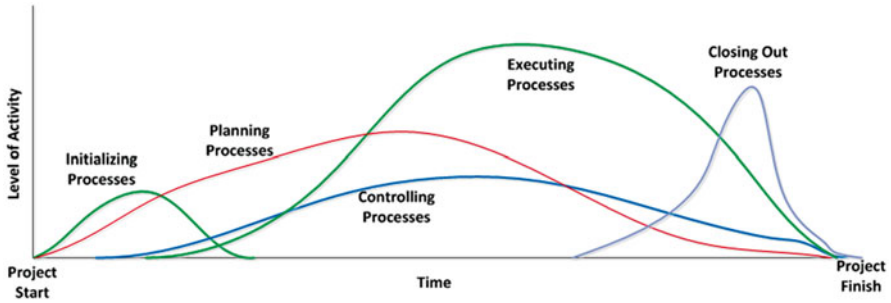

Fig. 3.1 Traditional program development cycle

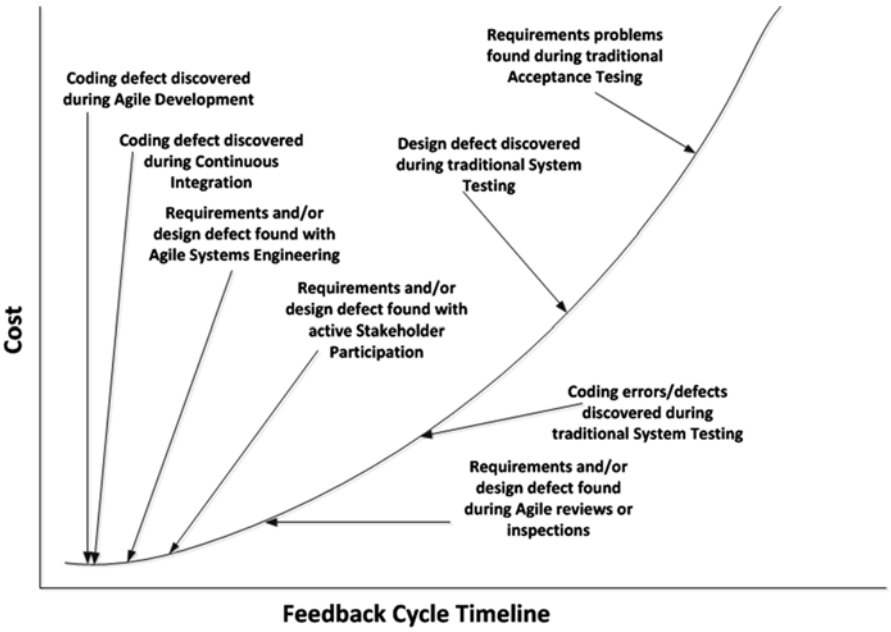

Fig. 3.2 Costs associated with changes along the development timeline

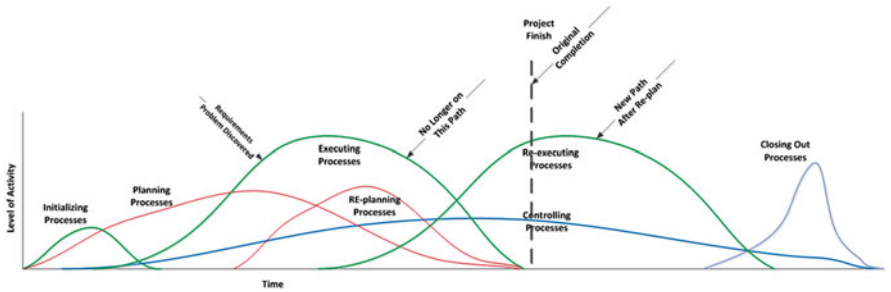

Fig. 3.3 Replanning of the classical development cycle

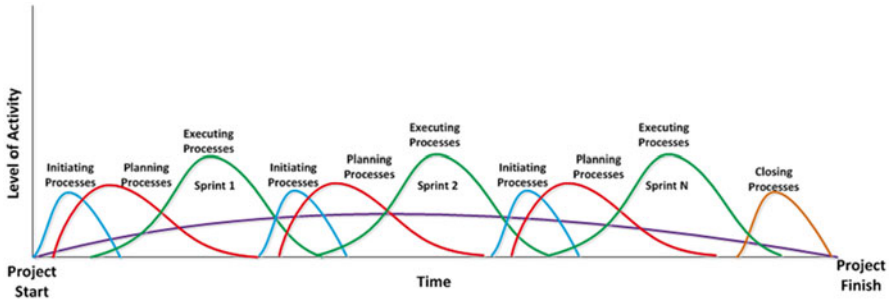

Fig. 3.4 Typical agile program development cycle

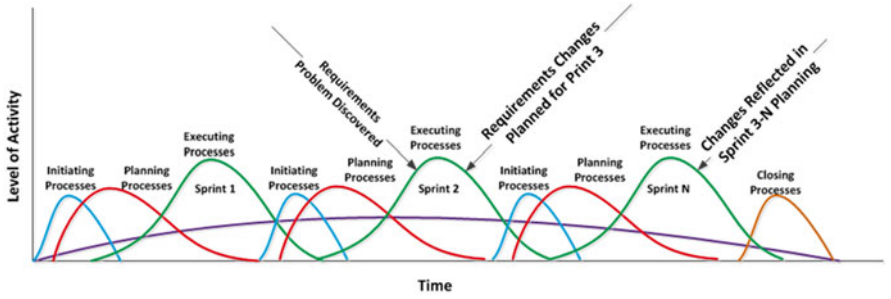

Fig. 3.5 How requirements changes are handled in the agile process

proper length of a Sprint. Figure 3.4 illustrates a typical agile program development cycle. In Fig. 3.4 we see the typical Sprint rhythm of initiating, planning, and execution of agile Sprints during the agile development process.

Again, the length of the Sprints is dependent on the program/project, the complexity of the software, total duration, and team membership. The duration of the Sprints is not relevant to this discussion. One of the big advantages to agile development is due to the adaptive nature of the Sprints. If a problem is discovered it can be rolled into the next and subsequent Sprint planning sessions and rolled into the schedules and does not require a major replan of the entire project, the way it does in classical development. Figure 3.5 demonstrates this process. Normally, because there is working software after each Sprint, problems are discovered earlier in the development process than in conventional or classical development methodologies. Customer and management feedback after each Sprint provide the opportunities to re-vector the development efforts before major cost and schedule have been expended.

We will not try to argue here over the length of Sprints, except to mention that the length of Sprints is tied to the overall length and complexity of the program/project development effort. Typically Sprints build in some sort of demo or evaluation to demonstrate the goals of the Sprint have been achieved. If the Sprint is too short, the work/evaluation ratios will be small; therefore the accomplishment/cost will be low. The other problem with short Sprints is that it will be difficult to achieve goals for complicated capabilities. Typically 30–45 days is a reasonable length for

a Sprint, again, depending on the complexities and overall goals for the software. The goal is to find the right rhythm for the project. Sprints should be long enough to produce working software that can be properly evaluated to determine the program/project is progressing along the correct paths [78]. But if the Sprints are too long, too many capabilities will be included in the Sprint, making end-of-Sprint demos and evaluations lengthy, and any issues discovered during the Sprint make it more difficult to recover. Also, if Sprints are too long, the retrospectives evaluation at the end of the Sprint is more difficult as it is more difficult to remember what happened at the beginning of the Sprint. Also, the clients/customers will be flooded with features to evaluate and provide feedback; in other words, if the Sprints are too long, they are not "agile" enough. Also, there should not be time between Sprints. The demonstration/evaluation, retrospective, backlog evaluations, etc., should be done and then the planning for the next Sprint should commence immediately. Again, Sprint length is tied into the quality and abilities of the teams, the technology and architecture of the systems, and the complexity of the software to be developed. Sprint length should be kept as constant as possible, since changing the Sprint length disrupts the rhythm of the project and the teams constantly have to reevaluate what can get accomplished during any given Sprint.

3.1 Agile Team Dynamics

Understanding the mix of personalities and soft people skills, coupled with how these manifest themselves across gender, ethnic backgrounds, cultural and generational diversities, and other considerations, will become essential for modern development leadership and management [5]. This new Agile Team paradigm is uncomfortable to many managers, who may feel like they now have nothing to do, given that central control and authority is given to the team, not to the Agile Manager. However, the Agile Manager is as busy as or busier in this environment than in a classical development effort [77]. The role of the facilitator across the agile teams is a critical role to the overall success of the development program/project. Many of the responsibilities of the Agile Manager are not all that different from those in classical management, except that in the Agile Development world, the Agile Manager must focus on the team's ability to deliver working software, rather than focus on traditional program/project measurement metrics [36]. The Agile Manager must lead, inspire, and provide empowerment (coaching) to the agile development team. The Agile Manager must be involved and facilitate, from a "hands-off" perspective, those supporting functions that allow the agile development team to be effective:

1. IT support and governance policies
2. Human Resource issues
3. Finance responsibilities
4. Support Teams—administrative, security, etc.

For the Agile Manager, this often involves adjusting one's person management/leadership style, and may require the Agile Manager to even work to affect and change the company culture to allow the agile teams to be effective and successful. It means working to acquire and utilize the abilities to understand and choose individuals who possess competencies that are agile-value based [37]. Understanding and being able to assess developers' skills, not just in designing, coding, and testing, but in those skills that can make or break agile teams, are equally important. The Agile Manager must ensure that they, as well as the development team, understand the agile style, cultural dynamics, and team dynamics [38]. It is essential that the team understand and be comfortable with the rhythm of the agile development style, Sprint after Sprint, each with its own cycle of planning, coding, testing, lessons learned, and then immediately moving on to the next Sprint. Figure 3.6 illustrates this recurring agile Sprint development rhythm.

Figure 3.6 illustrates the recursive nature of the agile development process. Once the initial project vision and initial scope has been established (with stakeholder buy-in), the teams are selected, the initial number and length of Sprints are determined, IT environments established, and the initial Sprint is initiated. From Sprint 0, the agile development program/project continues through the center two blocks shown in Fig. 3.6 until the program/project is completed and it is time for final product release. Blocks 2 and 3 shown in Fig. 3.6 are the heart and soul of agile development. Block 2 represents the Sprint software development, which includes working with the stakeholders to establish the priorities and capabilities needed for the current Sprint. Brain/Model storming is used to determine the test-driven software designs required for the current Sprint capabilities, keeping in mind that the current Sprint software must integrate with the previous volume of software [48]. This is accomplished with a continuous integration process that ensures working software at the end of the Sprint. Block 3 includes the final Sprint system/acceptance testing, releasing software through Sprint N into production and demonstrating the software to ensure functionalities are acceptable. Stakeholder involvement at each step ensures the final software will be accepted at the end of the project. After the final Sprint, the final software is released and final documentation is delivered, along with any training that is required.

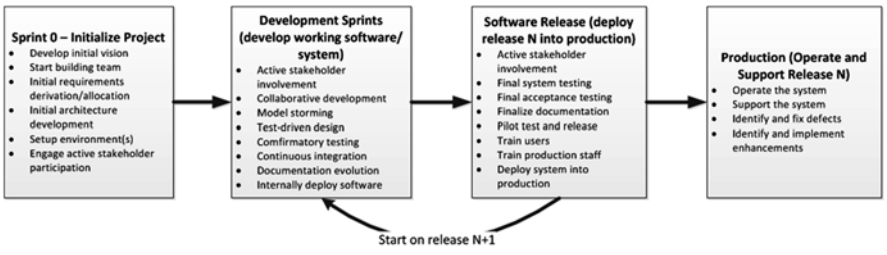

Fig. 3.6 The agile development process rhythm

This agile development process rhythm is important to the overall success of the program/project. Without understanding, facilitating, and protecting this rhythm, it will be difficult to achieve overall success from an agile development program/project. Keeping the teams functioning smoothly is the best thing the Agile Manager can do to promote a successful project. Case Study #2 illustrates this.

Case Study #2: The Disruptive Scrum Master

Project Length	24 months
Number of Sprints	9 Sprints
Number of Teams	5 Teams
Average Sprint Duration	8 weeks

Description This project was struggling early on as personnel from each of the agile development teams were hesitant to take on the role of Scrum Master for the Sprints, not wanting to be tagged as responsible if the team did not meet their capability development goals for any given Sprint. To solve this, the Agile Manager brought in a Senior Scrum Master from one of the company's "showcase" programs/projects. The teams assumed they would be coming in to train them on how to be effective as a Scrum Master, and to teach management how to allow the Scrum Masters to handle each development team across the Sprints, and to facilitate the teams choosing the right Scrum Master for each Sprint. Instead, the Senior Scrum Master took on the role of scrum master for all five teams. This was inherently a bad idea, as having one person act as Scrum Master for five agile development teams violates many of the roles that the Scrum Master needs to take on during each of the development Sprints. The first problem with one person acting as Scrum Master for all the teams across all of the Sprints is that the person taking on the role of Scrum Master should change at each Sprint, depending on the complexity points required for that Sprint and the expertise of the team members, based on the complexity points required for that Sprint. The role of the Scrum Master includes, but is not restricted to:

1. Assists the product owner with the backlog, both in its creation and with its maintenance.
2. Works with the agile development team to define completion of the Sprint. What are the criteria for "done?"
3. Removes stumbling blocks or impediments to progress, whether they are internal to the agile development team, or outside the team (e.g., support needed from another agile development team, or the IT team, etc.).
4. Facilitates the team's self-organization; helps the team remove roadblocks as part of the Sprint development team.
5. Helps the agile development team follow the Scrum process. As such, the Scrum Master should have an excellent understanding of the Scrum agile development management structures and must be willing to teach or mentor junior developers in the process.
6. In the end, the Scrum Master should spend the Sprint making the rest of the development team as productive as possible.

Having described the role(s) of the Scrum Master, having one person act as the Scrum Master across multiple agile development teams and across all Sprints within the development process is inherently inefficient, violates the precepts of a Scrum Master, and produces major problems that will ultimately hurt efficiencies. The issues that developed with the overall project are:

1. Each individual agile development team did not have access to the Scrum Master throughout the development cycle, since the Scrum Master's time and attention were spread over five development teams. This delayed problem resolution, road block removal, etc.
2. The Scrum Master was not a domain expert on any part of the system being developed, and therefore was unavailable as part of the development team throughout all of the Sprint development cycles.
3. Much time was wasted each morning as a single Scrum Master had to cycle through five Scrum meetings, trying to keep track of which team expressed which issues, and then having to prioritize whose issues got worked first. This resulted in much wasted time across the Sprint development teams and fostered a sense of resentment for the Scrum Master.
4. Since the Scrum Master was not a domain expert in any part of the system being developed, he could not act as mentor for junior engineers.
5. Having the same Scrum Master across five agile development teams kills the notion of self-organization and moves the teams Locus of Control outside to the outside Scrum Master and not within the team. This causes a general lack of ownership by the development teams [45].
6. Since there was only one Scrum Master, they tended to fix problems by moving people from one development team to another, destroying the cohesiveness of the teams. The teams never knew who was going to be on the team, and frequently the team members changed out in the middle of the Sprint.

The result was that by the third Sprint, efficiency had dropped to below 70 % and the entire program was behind at least one full Sprint. Attempts to make the developers work 15–20 hours of overtime was unsuccessful, since the environment was not conducive to self-organization [46]. Even those that agreed to work the overtime were not as productive as they should have been, since it took too long to overcome any obstacles the teams encountered, since the Scrum Master is beholding to all five agile development teams.

In the end, program management had to step in, remove the Scrum Master, and regroup all the development teams, putting the project behind three full Sprints. The program/project recovered, but not before increasing cost and schedule by over 40 %. The result was that management declared Agile Development "no better that when we did it the old way. Might as well go back to a development method we all understand." Agile development was abandoned and many people left the organization.

Lessons Learned: "If you go out of your way to violate the precepts of agile development, don't expect great results."

3.2 Team Member Dynamics

While there are many issues the Agile Manager must learn to deal with as we move into the future of software development projects, some of the major ones deal with interpersonal issues between developers that are a result of culture, learned behavior, or other factors beyond the control of the Agile Manager, but, nonetheless, must be dealt with in order to have an efficient, successful program/project. Some of the major issues are:

1. Differences between Agile teams and Classical development team structure and dynamics
2. Generational Differences—"Old Software Engineers" vs. "Young Software Engineers"
3. Cultural and Diversity Differences
4. Virtual Team Dynamics

Any one of these issues, if not dealt with carefully and skillfully, can derail a development project. Often, more than one of these issues is prevalent in any given team, adding to the complexity of creating a cohesive, efficient agile software development team. We will deal with each of these separately.

3.2.1 Differences Between Classical and Agile Team Dynamics

One of the first things Agile Managers must understand and come to grips with is the inherent differences in team dynamics, based on the nature of Agile vs. Classical development teams. Table 3.1 illustrates these differences.

As Table 3.1 indicates, the major differences between classical development team dynamics and the dynamics of an agile development team have to do with where the empowerment lies. For the classical development program/project, empowerment lies with the manager. Then in decreasing level of authority, you have a senior software developer (programmer), a lead system architect, and a lead test engineer. At the bottom of the authority chain are the software developers, systems engineers, testers, etc. Planning is done by the manager, design is done by specific people (or a specific team), and the requirements and design artifacts (Use Cases, Activity Diagrams, etc.) are flowed down to the software developers to code. It is often difficult for software developers at the lower levels to feel "ownership" of the final product, since they had no say in its design.

In contrast, for the agile development process, the dynamics are quite different. Empowerment happens at the Sprint team level. The development teams are cross-functional, containing team members from software, systems, test, etc., so that the team can plan and execute each Sprint successfully, with all disciplines having buy-in on what is being developed across each Sprint, and why. The teams should be as self-organizing as possible. We will speak more about this later. The ability of a team to self-organize assumes the team understands how to self-organize, and all the

Table 3.1 Agile vs. non-agile team dynamics

Agile teams	Non-agile teams
Teams are self-organizing	Teams are directed and managed by manager
Teams are cross-functional, with all skills necessary for deliving a product sprint (e.g., systems, software, test, etc.)	Teams contain sub-teams that have specific skill sets. Each person specializes in an particular skill area, such as designing, programming, testing, etc.
All team members are called developers, regardless of the work done	There are specific titles, such as programmer, senior programmer, project manager, tester, systems architect, etc.
Recommended size is 5–12 nominally	There is no recommended size for the team
There is no longer a manager that leads the team from the front. There are management roles to help the team run smoothly. Manager becomes more of a facilitator for the team	There is a manager, who directs and leads the team from the front
All functions for sprint development (Planning, estimating, design, coding, testing, release, and customer collaboration) are done by the team	Planning functions are performed by manager. Design, coding, and testing, are done by specific team members with specific skill sets for each part. Releases are done by a separate team
Knowledge and power are colocated throughout the team, creating their own center of authority	Knowledge and power are located within management
Responsibility and attachment are shared as a whole within the team	Responsibility and commitment are attached only to a single job for project/sub-project

team members are on board with taking on different roles as the Sprints progress through the development cycles [8]. One of the important precepts of successful agile development teams is that each team member leave their egos at the door. The purpose is to produce working software that supports an operationally viable system, not to promote any one person. The Agile Manager must understand this as well. The role of the facilitator is vitally important to the overall success of the project. And, if the teams are successful, the Agile Manager looks good and will be considered a success. The whole team owns the successes, and the whole team owns the failures. Again, sometimes this is a difficult concept for people to understand and accept as they enter into an agile development program/project.

3.2.2 Generational Differences in Team Members

In Sect. 1.2.1, we discussed the history of software engineering. While there was computer programming before the late 1960s, it was really in the late 1960s and early 1970s that software engineering as a discipline really took off. What that means is here, in 2014, those earliest software engineers would be in there early 60s. If you know many software developers, you understand that the classical view of programmers is a geek programmer, jacked in with headphones on, hunched over

his computer programming for hours and hours on end. And, by the way, that's not an unreasonable view. What one finds is the 60+ year old software engineers are the same in that respect as young software engineers. One software engineer, 63 years old and still coding, more now than ever, was heard to say, "Just put me at my computer, throw a few hunks of meat to me now and then and leave me alone and let me code." In this respect, young and old developers are alike. However, inevitably generational biases come into play as you form agile development teams. While the diversity of age brings the possibility of new and exciting solutions to the complex software problems of our ever-continuing technologies, getting all parties to play well together can be a daunting task for the Agile Manager. The older, more experienced software engineers may feel like there is nothing they can learn from these young upstarts, while the young software engineers assume the older software engineers couldn't possibly understand modern programming techniques and methodologies. We have found that those software developers that have stayed in development and have not gone the management route have, in general, kept up with modern methods and languages. Figure 3.7 illustrates, in an over top sense, how each generation of software developer sees the other.

One thing is sure, if given the chance, each can bring their own expertise to the table and each can learn from the other one. It is up to the Agile Manager to facilitate this mutual respect for the other's skills. The way to *NOT* accomplish this is to throw them in a room, to tell them they are a team, and to "do good agile things."

How 25-30 year old developers view 60+ year old developers

How 60+ year old developers view 25-30 year old developers

Fig. 3.7 How each generation of software developers sees the other

The Agile Manager needs to foster respect between each of their developers. One excellent way is to facilitate brainstorming sessions during the beginning stages of the project, allowing very free-form thinking between the team to come up with innovative ways of solving the program/project before them. This allows each team member to express their views, based on their own experiences and expertise, illustrating to the other team members what they bring to the table. This also helps the Agile Manager understand each team member better, by seeing them in this environment. The Agile Manager needs to understand their team, and not just programming skills, but personalities, idiosyncrasies, biases, their sense of Locus of Control and Empowerment, as well as understanding cultural differences that may exist [10].

Tom Perry[1] has compiled a number of learning games for agile development teams that the Agile Manager might find useful. Another important aspect of agile development team dynamics is the retrospective after each Sprint. It is important that the Agile Manager allow the team to express their blunt and honest views as to what went right and what went wrong so the team and the Agile Manager can learn how the team dynamics are working. It may be necessary for the Agile Manager to draw out certain team members, making sure they get the chance to express their views. It is important for the agile development team members feel they are self-organizing, rather than having structure imposed upon them. It is also important for the team members to feel like they are allowed to "play" with different ideas, techniques, and software structures in forming solutions for each Sprint.

Case Study #3: Scaling Agile Development Methodologies Many believe that the agile development philosophy works for small to medium programs/projects, but will ultimately fail for large development efforts. However, if agile methodologies are applied correctly, and if there is a commitment throughout the project to agile, even very large programs/projects can benefit from the agile development paradigm. While many people envision agile development as a small group of develops huddled together in the same room, handling everything within their own team, not required to communicate or collaborate with other teams, and while many may think this is nirvana for agile development, it is not reality, even for small programs/projects. But is it possible to realize the benefits of agile development for a large-scale development program (e.g., satellite system, commercial space flight software, etc.)?

The real challenge for large-scale agile development programs/projects is keeping the big picture out in front of all of the teams. Since requirements, architecture, Use Cases, database designs, and dependencies, by definition, evolve in an agile development effort, it is incumbent upon the Agile Manager to ensure that the architects and systems engineers within each team communicate and collaborate often to ensure that all teams are on the same page. Admittedly this resembles herding cats, but the goal is to make sure the teams aren't too far afield of each other, and are generally headed in the same direction, as specified by the overall systems

[1] http://agiletools.wordpress.com/2009/06/22/learning-games/

and software architecture. According to Joe McKendrick, "…agile has a fractal, scalable nature that allows for growth."[2]

In one large-scale agile development program/project (launch vehicle), most of the teams were housed in an aircraft hangar, with a very large white board at one end of the hangar that had the entire system and software architecture drawn on the white board. The program/project consisted of 10 on-site and 2 off-site Scrum teams, each with 20 team members, consisting of a system and software architect, systems engineers (2), test engineers (2), and 14 developers. Every morning the system and software architects held a 15–20 minute Scrum to review the overall architecture to ensure that everyone was on the same page. Then each Scrum team held their own 15-minute Scrum to plan the day, find out what roadblocks existed, and get ready to roll for the day. The reason the team was all housed in a hangar is that the actual launch vehicle was being built at the other end of the hangar. Not only was the system and software architecture kept in front of the teams at all times, the actual vehicle was at the end of the hangar and could be viewed at any time, even for the teams that were off-site (as described there were two of the teams that were off-site) but continually connected by several closed circuit cameras and VTCs so they could be involved all throughout every day and throughout the entire development process.

This seems like an extreme example of scaled agile development, but the result was startling, radically reducing the time to delivery, and vastly reducing the amount of rework that is typical with large-scale space vehicle projects.

3.2.3 Cultural and Diversity Differences

As we have discussed at length throughout the book, the Agile Manager must incorporate new management concepts that include the notion of becoming a facilitator for a group of self-organizing software development teams [4]. The agile development methodologies grew out of an understanding that the classical "factory" model of management and development hindered, rather than enhanced, software development. Slowly, over time, the efficiencies gained by collaboration between developers, and between the developers and their customers, allowed software to be delivered that was more useful, or "operationally viable," than when software was developed in a vacuum [76]. The crucial part of agile development, and the attitude the Agile Manager must continually foster and facilitate, is constant, open collaboration.

But how does the Agile Manager foster collaboration. As we have discussed, agile development teams are typically 8–12 developers. Collaboration is not facilitated by having them each have their own office, working independently. However, having them all crammed into a small office with no chance to "think" is not the answer either. If you want agile software development teams that collaborate and work as a team, you need to provide a team atmosphere; a "team room" for each agile development team with tables that allow several people to sit and work.

[2] http://www.zdnet.com/yes-agile-works-in-larger-enterprise-projects-too-7000020875/

You can almost foster collaboration "by accident" as team members overhear conversations. However, there must also be space where team members can go to have privacy to think through issues. The driving point is that the norm for the team should be working in a collaborative work environment, unencumbered by too much management oversight.

One aspect of working in a "team environment" that must be taken into account by the Agile Manager is cultural differences between team members. In our culturally diverse world, even though software engineers may have similar personalities, which may have driven them into software development, their cultural differences will undoubtedly come out during the hours and hours of collaboration across the Sprints. Many managers disappointingly try to assemble "homogeneous" teams of similar developers. The experienced Agile Manager understands that while the homogeneous team may experience increased efficiencies in the beginning. The diverse teams pass them as the diversity of ideas and experiences drive the development of innovative solutions to difficult software problems and complexities. In an atmosphere of continuous collaboration, you want people who can generalize and people who are serious detail people. Self-organization demands an emergent and collaborative workplace. In the diverse agile team development environment, new and different ideas, varying skill sets, and experiences drive innovation solutions and efficient development [47]. The better the collaborative environment, the greater the chance for "AHA" moments among the team, where the team realizes that together they have derived a solution that is better than any of them had thought of individually [47].

Having said that diverse teams can improve innovation and efficiencies, there are challenges for the Agile Manager in dealing with diversity and cultural differences. Geert Hofstede, a Dutch social psychologist, defined culture in terms of "software of the mind" [38], describing how cultural differences carry "cultural programming" that drives them as to how they understand what they experience, which influences their point of view on almost everything. For the Agile Manager, understanding where developers from different cultures are coming from helps them understand how to help facilitate the team, how to remove roadblocks, and how to bring about an atmosphere of collaboration and cooperation. One of the big challenges for teams with diversity and cultural differences is to understand and realize that many software development processes, including many aspects of agile development, were fostered, matured, and "grew up" in the United States and therefore tend to be described from an American individualist point of view [37]. Hofstede explains that many cultural differences arise from viewpoints of individualist vs. collectivist viewpoints. If the team is allowed to downplay collaboration, and act more as individuals in their software development, it is easy for misunderstandings to develop between developers as to how and why certain things were done by each individual developer. Misunderstandings foster a lack of trust between developers. Individualism in agile development teams fosters attitudes of ego, mistrust, and "not invented here," when presented with differing viewpoints. If the Agile Manager creates an environment of openness and trust, it is very possible to work together in

teams where you have different viewpoints: using those viewpoints to arrive at solutions none of the team could have created on their own.

3.2.4 Virtual Team Dynamics

Given the discussion above, the Agile Manager might ask, "Does this mean virtual teams won't work?" This is a fair question, as many experts in the field of agile development espouse that true collaboration can only happen if everyone is in the same place. And while this may be preferable, it is not always possible. However, there are ways the Agile Manager can facilitate geographically diverse "virtual teams" and still create, to a great extent, an atmosphere of collaboration and cooperation. One methodology that has been used by some organizations is to initially bring the team together for the start of the program/project to facilitate the initial collaboration between the team members. This could be for 2–6 weeks at the beginning, depending on the length of the development program/project. Then the team members return home, but keep in daily contact, either by Video Teleconference (VTC), by Skype, or even by phone. After a given length of time, the team is then reassembled, possibly at one of the other locations, and the collaboration rekindled between the teams. This allows them to foster trust and cooperation, to create an atmosphere and feeling of collaboration, even though they are geographically separated. Obviously there are challenges to be overcome with virtual teams, but it is possible, and it's up to the Agile Manager to keep track of what is going on (not interfere, but observe) and remove any roadblocks encountered so the teams can work efficiently.

The main point of all of this discussion is that the agile development project succeeds or fails, based on the vigilance, understanding, and commitment of the Agile Manager, first to the agile process in general, but also to creating the right team dynamics throughout the agile development life cycle. While the Agile Manager can develop the skills to form and facilitate agile development teams, there are tools the manager will need to provide to the teams so that the collaborations are captured, along with retrospectives, to allow each team member to grow, and so the Agile Manager can review what worked and didn't work for each program/project. Also, the Agile Manager will need their own productivity and metrics tools to measure progress, efficiency, and quality of the software development across the Sprints.

3.2.5 Diversity and Inclusiveness

Diversity and Inclusiveness are important dynamics that companies are embracing. Building development teams that are not just effective but embrace the concepts of diversity and inclusiveness is important, but most leaders and managers have not been trained for the dynamics these bring (both good and bad) to teams [20]. The concept of inclusiveness and diversity has grown over the last two decades and is

now an integral part of modern organizations. Most large companies have corporate Diversity and Inclusiveness Officers that ensure the companies policies and practices are in-line with the diversity and inclusiveness protocols [27]. The basic precepts of diversity and inclusiveness came out of the progressive models developed in the 1960s that originally drove affirmative action and the need to comply with equal opportunity laws. In today's workforce, diversity and inclusiveness defines [11]:

> The similarities and differences among coworkers in terms of their ages, cultural backgrounds, physical abilities and disabilities, race, religion, gender, and gender orientation.

Where it is understood that such differences have no bearing on abilities within the workplace and allows teams to be built from the best personnel, in terms of technical skills as well as soft people skills discussed earlier in the book. The most efficient and effective agile development teams have very little to do with the differences between diverse groups, and everything to do with aligning skills, personalities, and team dynamics. Diversity and inclusiveness models today reflect the understanding by companies that a multicultural, multi-diverse atmosphere promotes diversity of thought and perspectives, as well as better problem-solving capabilities—traits crucial to successful agile development, where innovative solutions may be required at each Sprint.

The heart of a company's ability to capture and retain a diverse workforce is reflected in their leadership. The transformational Agile Manager focuses on competencies that allow cohesive team dynamics among the agile development teams, in terms of both technical and interpersonal (hard and soft) skills. The inclusive and diverse Agile Manager seeks to create a team culture where they see diversity and inclusiveness as an influential resource to enhance the overall Sprint development team effectiveness [59]. And while recognizing and embracing diversity and inclusiveness helps the Agile Manager to link a variety of talents together to form strong agile development teams, the act of embracing and recognizing the need for diversity and inclusiveness provides personnel an atmosphere that fosters a sense of belonging, to feel empowered, and to internalize their Locus of Control, allowing their talents, thoughts, and innovations to come out and be utilized throughout the agile development program/project, allowing each developer to contribute in a unique way that reflects the diversity of the group. The next chapter describes some of these productivity tools available to developers, the team as a whole, and the Agile Manager [44].

Chapter 4
Productivity Tools for the Modern Team

Providing an agile development team with tools to be productive goes beyond handing each one of them a laptop with compilers. Communication and collaboration tools, whether face-to-face or geographically diverse, are crucial in modern teams. Here we discuss collaboration tools and other tools that will be crucial today and in the future. We will discuss three major types of productivity tools for agile development programs/projects:

1. Productivity tools for the Agile Manager
2. Productivity tools for the agile software developer
3. Collaboration tools for agile development teams

4.1 Productivity Tools for the Agile Manager

For the Agile Manager, planning, tracking, and managing a program/project is very different from managing a classical development project. For the Agile Manager, a coordinated and integrated set of productivity tools should allow the Agile Manager to manage overall and individual Sprint team's backlogs, overall and individual burn-down and burn-up charts, capture Sprint reviews and retrospectives, and plan Sprints, including changes associated with the entire agile development process. Integration of requirements, UML artifacts, and designs is more important in an agile development effort, given the dynamic nature of agile development. Without productivity tools to easily track all of the moving parts within an agile development program/project, it is easy for the Agile Manager to lose control of progress and issues. One important piece that is often missed is the issue of release control. High-level release planning and coordination across the Sprint teams is important. Given the fluidity of moving capabilities between Sprints, based on needs and coordination between Sprint teams, keeping track of releases and what is in and out of a given software release is essential and helps manage customer expectations as

© Springer International Publishing Switzerland 2015

J.A. Crowder, S. Friess, *Agile Project Management: Managing for Success*,
DOI 10.1007/978-3-319-09018-4_4

demonstrations and review meetings occur. This should all be integrated together into comprehensive agile management software tools. Below is a review of the features of most of the current agile management commercial software packages that are available. Some are free, open source software; some are Commercial Off-the-Shelf (COTS) software, each with its own advantages and disadvantages. Each Agile Manager must assess the tools they feel are necessary, given the size and complexity of their agile development program/project. The following discussion describes the general features COTS software packages designed to manage agile development programs/projects include, obviously with various levels of success and features unique to each COTS software package.

4.1.1 Agile Management Software

COTS agile program/project management software packages are specifically designed to be productivity tools for agile program/project management. Most are designed to work well with Scrum, ScrumBan, Kanban, and many include provisions for custom agile management methods [66]. Most include integrated tools for:

1. Product and Sprint backlog management
2. High-level and Sprint planning
3. Capture of daily Scrum meetings
4. Capture of Sprint retrospectives
5. High-level and Sprint release management
6. Variety of graphics, including burn-down charts, including Sprint, overall program/project, and release metrics

4.2 Productivity Tools for the Agile Developer

In his work on Global Software Development, Iyengar [40] explains that the rapid increase in agile software development has led to an explosion of productivity tools. Most are geared at the Agile Manager, but tools are needed to enhance collaboration and cooperation among agile team members, whether they are collocated or in different continents. If the Agile Manager has not fully adopted the philosophy of the agile process, productivity tools provided to agile team members may be limited to software tools like Microsoft Project©, spreadsheets like Microsoft Excel©, or tools like OneNote©. While these are fine tools for the purposes they were intended, they fall very short of providing collaboration, idea generation and capture, retrospective analysis, or capturing software and its context for future use. Ron Jeffries [42], the father of Extreme Programming, says it this way:

> I think that people and how they interact on a project are the most important thing, and I think that they need to create a way of working – a process – that works best for them. Because their interactions are critical to project success, I suggest that teams begin the work with an approach that will bring them together as people, not one that will let them remain apart, communicating electronically.—Ron Jeffries

Now there are obvious simple collaboration/brainstorming tools like white boards. Of particular use are electronic white boards that electronically capture what is written on the white board and stores it for transfer and review later. There are many COTS software products that promise to be the end-all for agile development, but the majority are focused on the management of agile teams and not on actual team productivity, collaboration, and cooperative development.

While it is important to have individual workstations that enable stand-alone work environments for software developers, you still have engineering artifacts and data stored separately. These personal environments must be turned into interpersonal collaborative enterprise environments [19]. As the pace of design and implementation increases, there is a need for increased collaboration and continuous capture of brainstorming, design, coding, and test artifacts. Realization of this environment is crucial to overall efficiency and success.

Increased collaboration and increased capture of artifacts help increase quality, since all development teams and individuals understand the end goal of the system, increasing the efficiency of software integration after each Sprint. Many managers believe the push to increase quality increases cost of development, and it should be left to the built-in time after each Sprint to handle software deficiencies (bugs). However, this is a fallacy, as increasing quality without increasing cost is possible. What are required are overall analysis, design, development, and testing environments where all aspects of development efforts across developers are captured and shared easily throughout the program/project. What is required is a holistic approach to software analysis, design, implementation, and test that forms a fluid, adaptive, and quality/test-driven development philosophy that can take software engineering into the future. In short, the productivity tools the Agile Manager needs to make available to the agile development teams, and to themselves, need to account for the three main domains of agile development (see Fig. 4.1). Management must first embrace new organizational structures required for agile development. Second, they must provide real productivity and collaboration tools specifically designed to

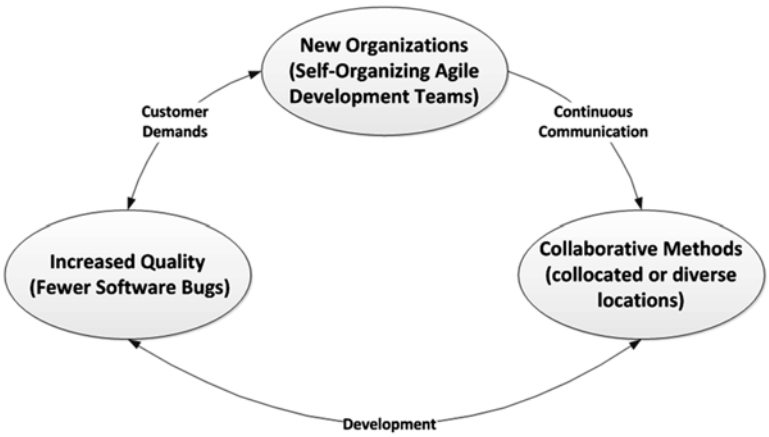

Fig. 4.1 The three domains to master for real agile development

facilitate agile development (collaboration and complete capture of analysis, design, implementation, and test artifacts). Third, the atmosphere and methodologies should provide increased quality up front; quality should be built into the test-driven design and implementation, not forced in by fixing the code after the Sprints.

4.3 The Future of Agile Development Productivity Tools

So what does the future hold for agile development teams that will enhance their productivity? Productivity tools that are geared to software engineers engaged in agile development must take a paradigm shift in foundation. Instead of process-central tools, developers need people-centric productivity tools (see Fig. 4.2) that foster an environment of innovation, collaboration, efficiency, and as much automation as is reasonable for the developer. Not everything can be automated, but mundane tasks like documentation, capture of ideas, requirements, and designs, resource management, etc. Many classical mangers may view automation as threatening. However, automation has always resulted in allowing engineers, managers, and others to move on to more important tasks, allowing more innovation, mentoring, and continued education.

Organizations must strive to provide an environment where budget is not centered on time and motion (Earned Value only based on Cost and Schedule), but must be centered on the perceived values of the objectives of the project [25]. Working software should be front and center in determination of "Sprint Value." The agile development teams must be free to self-organize, generate work plans, resource management plans, and pick the productivity tools appropriate for the project and teams [73]. However, this freedom must include the Agile Manager, whose role of

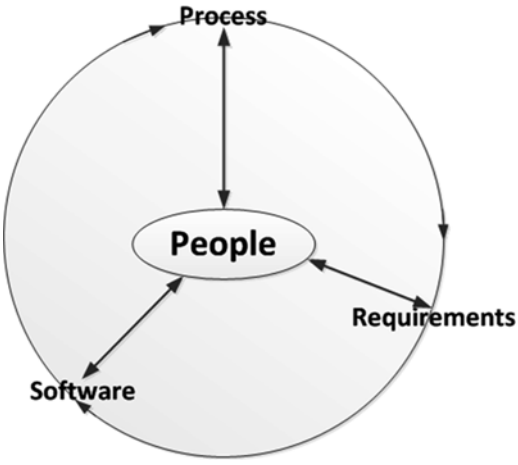

Fig. 4.2 People-centric paradigm for agile development

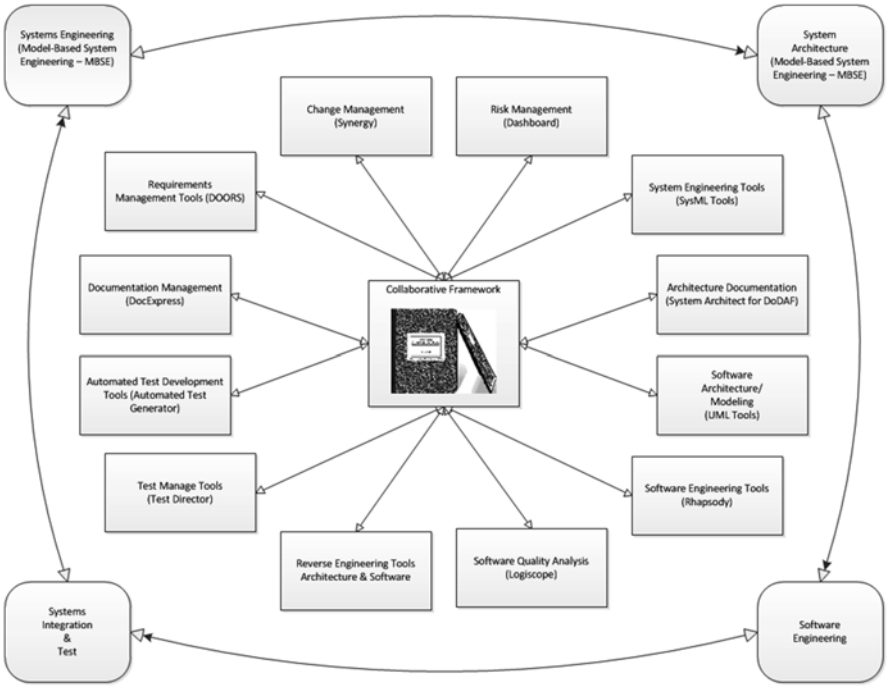

Fig. 4.3 The modern agile development productivity environment

facilitator is essential, providing the tools and environment that allows the Sprint teams to be successful. The Agile Manager must provide an atmosphere of collaboration and freedom, while at the same time demanding discipline across the individuals and their teams. Agile development does not mean free-for-all, but allows the developers to be as productive as possible.

Lastly, the organizational, physical, and IT environments must promote continual process and product improvement over the life cycle of the program/project. Quality characteristics and working software must be the Sprint teams' central objectives. This provides the teams focus throughout the development process. Future productivity environments/systems that allow automation, collaboration, and inclusion of all aspects of product development may look similar to Fig. 4.3 [19].

If one studies Fig. 4.3 you will notice many of the products that are used today. You will find Unified Modeling Language (UML) tools, requirements tools (DOORS©), change management tools, etc. The difference is the central automated complete system design tool, the Collaboration Framework, the heart of which is the Electronic, Engineering Notebook. The Electronic, Engineering Notebook is an electronic version of the old "Engineering Notebook" that started in the 1940s and 1950s. The difference is it automatically captures, logs, correlates, and publishes all aspects of the program/project. Figure 4.4 illustrates this [19].

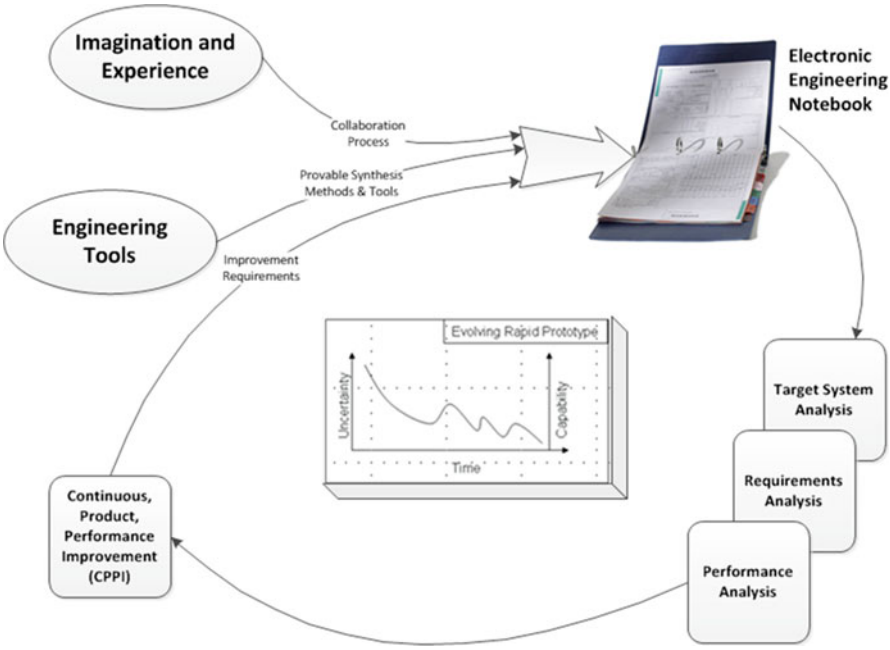

Fig. 4.4 The electronic, engineering notebook

One very important aspect of the agile development process is how to accurately and appropriately measure the productivity and efficiency of the development throughout the agile life cycle. Chapter 5 deals with how the Earned Value process must be completely revamped, and a new Earned Value Management System must be created to adequately handle agile software development.

Chapter 5
Measuring Success in an Agile World: Agile EVMS

The Earned Value Measurement System (EVMS) has become a mainstay in Commercial and Government groups to measure progress and success of a project. EVMS is espoused to be an effective (albeit subjective) measure, but it does not play well with agile development efforts, due to its requirement of static schedules and work plans [55]. Here we introduce a new paradigm for EVMS that will accommodate and be effective in measuring progress and problems within agile development efforts.

5.1 Brief History of the Earned Value Management System

The government instituted the formal practice of Earned Value in the 1960s as a methodology for program/project management in terms of scope, cost, and schedule. Earned Value promises to provide accurate measurements of program/project performance as well as identifying problems, a crucial component of program/project management [73]. The basic 11 precepts or elements of the Earned Value Management system are:

1. Define Authorized Work Elements
2. Identify Program Organizational Structure
3. Integrate the Work Breakdown Structure (WBS) and Organizational Breakdown Structure (OBS)
4. Schedule the Work
5. Identify Products and Milestones
6. Set Time-Phased Budget
7. Record Direct Costs
8. Determine Variances
9. Sum Data and Variances
10. Manage Action Plans
11. Incorporate Changes

© Springer International Publishing Switzerland 2015

J.A. Crowder, S. Friess, *Agile Project Management: Managing for Success*,
DOI 10.1007/978-3-319-09018-4_5

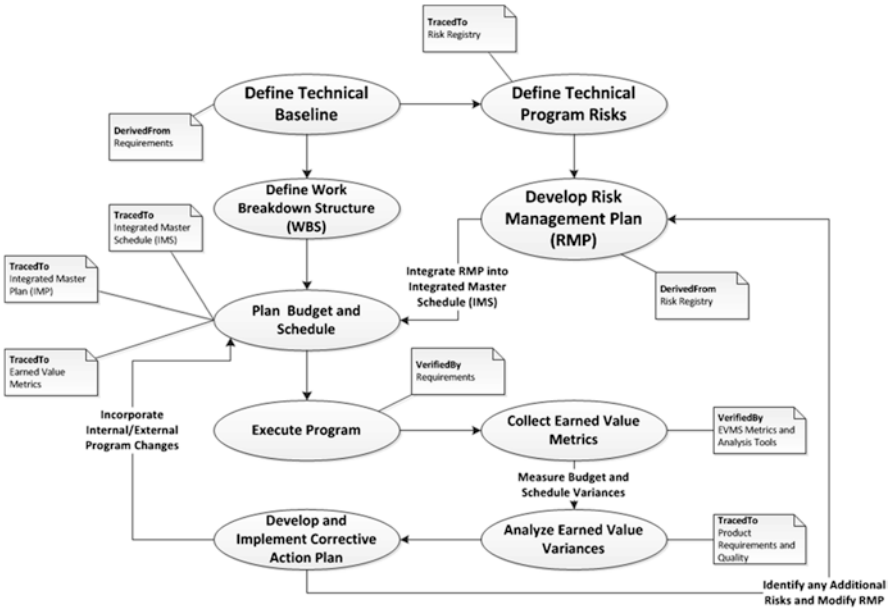

Fig. 5.1 Classical waterfall program/project high-level execution

By the late 1980s and early 1990s, the EVMS became a mainstay tool for managing, measuring, and executing programs/projects among the Department of Defense (DoD) and their respective Defense Contractors, Department of Energy (DoE), and NASA [21]. Since then many large commercial companies have adopted EVMS as well, like Boeing Commercial Airplane Division [65]. There are many Earned Value COTS software packages available for classical Earned Value. Some of the most popular ones are Microsoft Project©, Open Plan©, and Deltek wInsight©. These products are geared toward helping to plan, measure, analyze, and execute the classical waterfall development methodology [1]. Figure 5.1 illustrates this process, which includes measurement and analysis of earned value metrics.

Contrasting Fig. 5.1, Fig. 5.2 illustrates the changes associated with creating a similar execution plan for agile development programs/projects. As you can see from Fig. 5.2, the flow is quite different, and includes the recursive Sprint development process that continually refines the requirements as the Sprints progress [72]. The emphasis at the end of each Sprint is on working software that integrates together at the end of each Sprint. Customer input is sought and the Sprint plans and capability plans adjusted, based on the Sprint perspectives, integration and test, along with customer input. Included in the agile program/project execution process is the ability for the Sprint teams to self-organize for each Sprint: team members taking on different roles across the Sprints based on their capabilities and expertise [50].

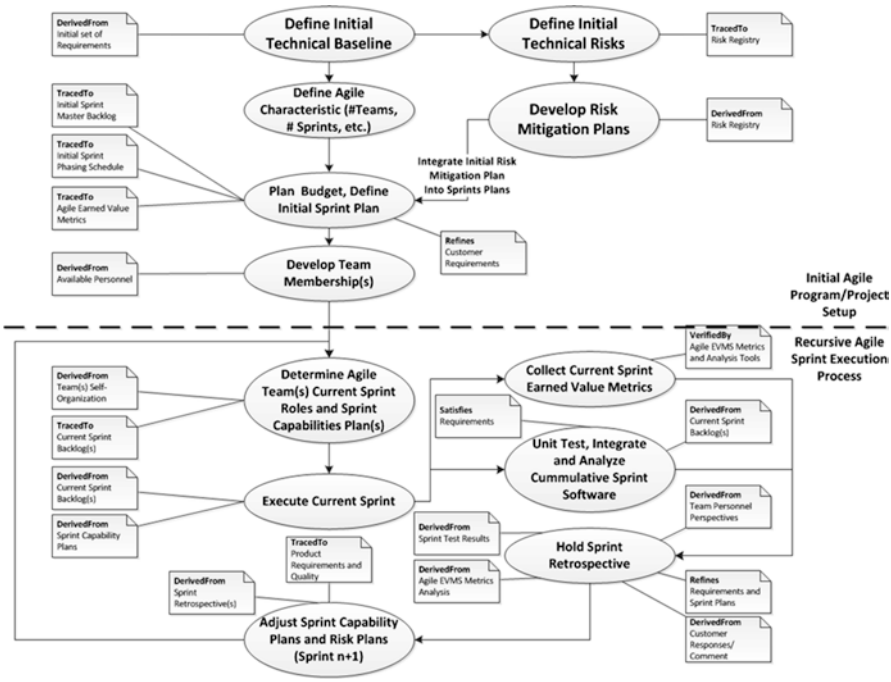

Fig. 5.2 The agile program/project high-level execution

Case Study #4: Classical EVMS for Agile Development (FAILURE)

Project Length	9 Months
Number of Sprints	5 Sprints
Number of Teams	3 Teams
Average Sprint Duration	5 weeks

Case Study 4 illustrates that just because you use a Scrum Management style, agile software development will never just show up as an emergent property of the program/project by accident. Many a program manager has exclaimed, "We have a backlog, we have Scrum meetings; we are most certainly agile." While both of these are properties that may be utilized for agile programs/projects, their presence alone does not define agile. They are simply tools to help you, assuming you are committed to the agile development methodology, have established an actual agile development program/project, and manage it like an agile manager, and not a "rigid agile manager." Likewise, if the manager attempts to measure the progress of an agile development program/project using the same methods, and performs the same analysis as waterfall development, not only will the manager be constantly frustrated, they will never understand the real progress, or success/failure of their program/project. What follows is a classic example of classical EVMS used to measure an agile development program/project, albeit pseudo-agile.

The manager began the effort espousing the precepts of agile; however the entire program/project was laid out in detail in the Integrate Master Schedule across the entire development cycle, with each task being 4–5 weeks in length, tied together with predecessors and successors. The program was not laid out by Sprints, but laid out as a classical waterfall development project, and collections of tasks that were coincident (or done in parallel) and about the same length were identified as "Sprints." There were overarching systems and software leads, and development teams were software engineers only, not systems, software, test, etc., or cross-functional, as you would expect for a true agile development effort.

Trying to keep up the appearance of agile, the developers were allowed to function autonomously and write software based on what capabilities they deemed were required at each "Sprint." Meanwhile, the manager made sure the Earned Value Cost Performance Index (CPI) and Schedule Performance Index (SPI) at the end of each reporting period were always 1, assuming the teams would end up finishing the software on time, would meet the customer requirements, and in the end, no one would ask any questions. Unfortunately, midstream, right before one of the End-of-Sprint demonstrations, the customer informed the manager that is was necessary to change the requirements, as expectations had changed; congressional budgets had changed, and it was necessary to scale back on the capabilities to be delivered. Unfortunately, some of those capabilities were already done and folded into the demonstration that followed; capabilities that the waterfall, iterative schedule showed weren't scheduled for months. Unfortunately for the manager, the agile development teams had decided that they were easier incorporated into the original software framework for the program/project and decided to add them in earlier. Had the manager acted like an agile manager and paid attention to what the agile development teams were actually producing, they would have known what was happening. The emphasis was on showing a stable Earned Value Metrics, software productivity numbers were on track (not taking into account what capabilities were being coded), and things looked great to upper management. What was discovered was that the capabilities the customer had to have, had been placed on the backlog for later Sprints, meaning it wasn't possible to scale back on the requirements, or cost estimate, or schedule estimate. The customer ordered an immediate Estimate-to-Complete, and discovered the inconsistencies in Earned Value reporting Metrics. The project was cancelled, the manager was fired, and it was a long time before the company successfully procured another contract.

Committing to agile up front is crucial to its success. Having truly agile earned value metrics that reflect the rhythms and processes of agile software development is essential as agile development becomes more prevalent in the industry. The next section defines a new Earned Value system, one with agile EVMS metrics that reflect the agile process, while being just as useful for the manager to assess the cost, schedule, productivity, and quality of their program/project. I know managers love their tried and true EVMS metrics, and change is hard and often frightening. Please refrain from the pitchforks and torches till you get through the entire discussion (see Fig. 5.3).

Fig. 5.3 How managers may view a change to the EVMS

5.2 Assessing Agile Development: Agile EVMS

Earned Value and the Earned Value Management System (EVMS) provides cost and schedule performance metrics that, if handled carefully and honestly, can be useful in helping the program/project manager track the progress and get early indications of problems. Basically, Earned Value measures whether you have earned the right to spend that much money, and whether you have earned the right to spend that much schedule [61]. Some of the metrics that Earned Value uses the measure program/project progress are:

1. BCWS: Budgeted Cost of Work Scheduled. This represents the "planned" value of the work scheduled for a given time period.
2. BCWP: Budgeted Cost of Work Performed. This represents the cost (from the original budget) of the work that was actually performed during a given time period. This metric is used in both of the Earned Value performance calculations.
3. ACWP: Actual Cost of Work Performed. This represents the actual collected costs over a given time period for the work that was actually performed. This may or may not represent the amount of work that was supposed to be accomplished during a given time period.
4. BAC: Budget at Completion. This is the total cost of the program/project at completion, or the BCWS at the end of the program/project.

5. EAC: Estimate at Completion. This is the ACWP to date, plus the estimate to complete the remaining work.
6. CV: Cost Variance. CV = BCWP – ACWP, or the Budget cost of the work actually performed during a given time period (what they work should have cost), minus what the actual costs were for the work performed during the same time period. If CV is negative, then the cost of the work performed was underestimated. If CV is positive, then the actual cost of the work performed was less than was budgeted (there's an urban legend that this has actually happened once, but we don't put much credence in it). Typically, managers are looking for a CV as close to zero as possible. For even if the actual cost is less than the budgeted cost, it means the estimates were wrong, making the rest of the budgeted work suspect.
7. SV: Schedule Variance. SV = BCWP – BCWS. Since both BCWP and BCWS represent the same time period, a negative SV means there is still work left to do that was not accomplished during the time period, which will take more time (i.e., schedule) to work off the remaining tasks. Again, the goal is for SV to be as close to zero as possible [2]. And while it is possible for SV to be positive, which means there was more progress during the time period than was scheduled, this too is the stuff of Earned Value folklore.
8. VAC: Variance at Completion. VAC = BAC – EAC. This represents the complete variance for the program/project at the conclusion. Again, the goal is to have VAC as close to zero as possible, for if VAC is large positive, then the program/project was grossly over-budgeted, while a VAC that is large negative indicates a grossly under-budgeted program/project. Both are hazardous because it calls into question the company's budgeting practices.

If you read through metrics 1–8, they seem like reasonable measures of a program/project. However, the issues with classical Earned Value are that the entire program/project must be planned out in detail, often down to 2–4 week tasks, and detailed budgets put in place for the program/project schedule. Any variances from this budget or schedule are considered problems (variances) and variance reports must be written and explained; in short, in classical Earned Value change is *bad* and uncertainty is *worse* [3]. In fact, the concepts of classical Earned Value can be broken down into seven major precepts:

1. Plan all work to completion.
2. Break down the work scope into finite pieces assigned to responsible persons for control of technical, cost, and schedule objectives.
3. Integrate work scope, cost, and schedule objectives into a performance baseline to measure progress against. Control changes to the baseline.
4. Use actual costs incurred and recorded in accomplishing the work performed.
5. Objectively assess accomplishments at the work performance level.
6. Analyze variances from the plan, forecast impacts, and prepare an EAC based on current performance.
7. Use Earned Value metrics to assess management processes.

In short, plan every detail of the project, including the work to be performed at every small increment, and create a detailed and complete schedule and budget across the entire project. Manage change carefully, for change is the *hobgoblin* of Earned Value. In classical waterfall development, working software is delivered at major milestones.

Now let us bounce the precepts of classical Earned Value against the precepts of agile software development to see if there may be some issues.

1. The emphasis is on early and continuously working software deliveries at the end of each of the Sprints.
2. Constant customer interaction and collaboration that includes welcoming changes to requirements. This allows the customer to adapt to changing environment and user needs to create products and services that are considered viable by the end users.
3. Business development, management, customers, and developers MUST work together throughout the project.
4. Sprints teams should be staffed with motivated individuals who are trained in both agile development and agile team dynamics.
5. Management needs to create an effective team environment and support the teams by being a facilitator and trusting the teams to develop the required software.
6. The most efficient and effective method of cooperation and collaboration within an agile development team is face-to-face conversation—even if it is over a Video Teleconference (VTC).
7. Working software and team/software entropy are the primary metrics.
8. Agile development processes promote sustainable development.
9. Continuous attention to technical excellence and good design enhances agility and promotes healthy cost and schedule metrics.
10. Simplicity is essential in agile development—work for work sake has no place in agile programs/projects.
11. The best architectures, requirements, and designs emerge from well-trained, self-organizing teams.
12. Teams must reflect at regular intervals (Sprint introspectives) on how to become more effective. The team must then tune and adjust its behavior accordingly.

5.2.1 Disconnects Between Classical EVMS and Agile Development

The notion of detailed planning of every task in the program/project across the entire schedule and striving to control and drive down changes is completely antithetic to the precepts of agile software development. In agile development, change is welcomed throughout the project. The entire reason agile development was

Table 5.1 Classical vs. agile EVMS concepts

Classical EVMS	Concepts for Agile EVMS
Plan all work to completion of the program/project.	Create capabilities backlog and loosely plan across Sprints.
Break down work scope into finite pieces assigned to responsible person(s) who control technical, cost, and schedule objectives.	Work is assigned to Sprint teams who control technical content of the Sprint backlogs.
Integrate work scope, cost, and schedule into a performance/program baseline. Control changes to baseline.	Create program/project backlog burndown plan, assigning capabilities across Sprints and teams. Results of demonstrable software at the end of each Sprint define how requirements/capabilities change across the program/project.
Use actual costs incurred and recorded in accomplishing the work performed.	Agile teams assess performance characteristics of the teams and tune the teams' needs and performance to improve over time, throughout the entire agile development cycle.
Objectively assess accomplishments at the work performance level.	Assess accomplishment by continuously integrated working software.
Analyze variances from the plan, forecast impacts, and prepare EAC, based on current performance.	Analyze, based on working software, new or changes in requirements for future Sprints. Assess efficiency and effectiveness, which includes volatility of teams and software, based on entropy measures.
Use Earned Value metrics to assess management process.	Use Agile Earned Value metrics to assess effectiveness of both management and Sprint teams.

created was to deal with the reality that requirements and necessary capabilities change over time, especially for a project that spans years. In today's environment where technology and customer, geopolitical, and cultural needs change rapidly, the need for embracing agile will only increase over time. The successful companies are those that not only embrace the mechanics of agile development, but are those that understand the need for management and developers with the nontechnical skills (soft people skills) necessary to empower and facilitate efficient and motivated agile development Sprint teams. One of the most important things to understand in today's environments is that it is possible to come in completely on budget and on schedule and yet the project fail because the program/project development did not adapt to changing customer needs. If no one wants the product once it's completed, it was not a success. Likewise, if the program/project comes in on schedule and on budget and meets customer needs, but your developers never want to work on a program/project with that manager ever again, the program/project failed. Table 5.1 illustrates classical EVMS versus the concepts for Agile EVMS.

Next we will discuss factors that often derail the use of EVMS on Agile programs/projects, and then Sect. 5.3 will introduce new EVMS assessment concepts, ones that are more adaptable and useable to assess agile development: the Agile Earned Value Management System (AEVMS).

5.2.2 *Factors That Can Derail Agile EVMS*

There are very many factors that can derail agile development teams and lead them to failure; the most prevalent are those revolving around a lack of management commitment and training in how to manage an agile program/project (i.e., how to be an agile manager). Even more issues arrive when managers must embrace a new paradigm of how to measure agile programs/projects, or how to use Agile Earned Value. What follows is a discussion of the factors that can most easily derail an agile development project, from an Agile EVMS perspective:

1. Lack of accountability: This pertains both to the members of the agile development teams and the agile manager. Many managers may feel like they have nothing to do, given the autonomy and control that the agile team need to have over the development efforts. In this case, the manager may feel like they are no longer accountable for the project, and therefore will not facilitate the agile teams, becoming apathetic toward the entire process. In this case, the program/project has very little chance of being successful. If the teams are not chosen well, some team members may feel like they are not individually accountable and that it's the teams' responsibility, not theirs, to make sure things work well. Individual accountability to the teams is crucial to the overall success of agile development.
2. Lack of commitment to Agile (holding on to classical EVMS): The manager that insists on using classical waterfall development Earned Value and management techniques on an agile development effort will not only be unsuccessful, but the manager will be very frustrated throughout the entire effort. However, this requires commitment from upper management to provide the proper management training on agile projects.
3. Poorly trained teams: Just being efficient at writing software and being adaptable to changes doesn't mean agile teams are successful. Agile development Sprint teams need to be trained in how to collaborate effectively and how to deal with generational, cultural, and other differences that can cause mistrust among team members. Understanding the teams personalities can go a long way toward the teams self-organizing in a way that allows the team to be effective across multiple Sprints and multiple programs/projects.
4. Poor documentation: Many developers feel that agile gives them the freedom to not worry about documentation: that documentation gets in the way of their freedom to self-organize and adapt. However, the right amount of documentation is essential in order for the team members and teams to understand the end goals, and to understand what each other is currently developing, how it fits into the Sprint, and how the Sprints will integrate together to form working software at the end of each Sprint.
5. Using unproven collaboration/automation tools: As we have discussed, providing productivity tools to the teams is necessary to keep the individual developers and the Sprint teams running at peak efficiency and can promote collaboration. However, introducing new tools into the teams during a development effort may

completely disrupt the rhythm of the agile development process while each team member comes up to speed on the tools and how to use them effectively. In addition, if it turns out the tool is not appropriate for the teams, additional efficiencies will be lost when teams and individuals try to re-adopt previous tools.
6. Inaccurate data: It is vitally important that the Agile Manager gather accurate data concerning the productivity and effectivity of the teams across Sprints. Retrospectives are difficult if the teams are not provided accurate information.
7. Manager holding everything at their level (failure to communicate issues to the teams): While inaccurate data causes incorrect decisions to be made among the teams and between the teams, the lack of information is more devastating to effective agile development efforts. There must be complete transparency between the Agile Manager and the teams, the Agile Manager and individual developers, and between Sprint teams. The adaptivity the agile development process promises is only achievable if there is effective communications all throughout the program/project.

5.2.3 Agile EVMS Metrics

In his paper on Assessing Agility [39], Lappo asserts that classical metrics are not much use for assessing agility. He goes on to explain that the use of metrics like Software Lines of Code (SLOC), function points, or quality metrics is not an effective measure of agile development, and assessments made should be made in terms of how the software, as well as the software development process, is effective in meeting the needs of the program/project, the customer, the end users, and the overall companies' business goals and visions.

This does not mean that these metrics are not useful throughout the program/project to help with the overall agile development efforts. For instance, complexity measures are useful in determining which capabilities from the backlog are scheduled in a given Sprint, and are useful in determining how to "swap out" one set of capabilities for another when it is determined that a set of capabilities must be moved forward or moved out within the overall Sprint development schedule. However, complexity measures tell very little about the maintainability of the code: how easy is it to adapt the code for other purposes (i.e., how agile is the code). Some very complex code may be written in a way that is easy to understand and structured in such a way that it fits easily into the agile development style. At the same time some very simple code can be written in such a convoluted way that it is almost impossible to understand, modify, or maintain. Lappo's view [39] is that these low-level measures don't measure or provide insight into higher-level effectiveness and efficiency measures of the overall program/project's agile process.

Agile Earned Value metrics must take into account the entire agile development life cycle, which includes assessments of the software, the program/project agile process, the environment that has been created for the developers and development (Sprint) teams, as well as assessment of the tools utilized in the agile development process.

In order to effectively measure agile development in terms of Earned Value one must take into all of these factors, for each of them drives cost, schedule, and quality across the entire agile development program/project. Assessing software in terms of complexity may not provide a high-level view of overall program effectiveness, but according to Abran [2], it is an essential characteristic of the agile software process and product and should be measured. According to Carbone [15], capturing and utilizing context in such measurements is essential to capture the overall measure of complexity. Software complexity, combined with context, allows the Agile Manager to measure the computational, structural, functional, and representational complexity of the software throughout the agile development life cycle. Abran [2] explains that measuring computational complexity (CC) provides classical Earned Value measurements of CV and SV, as it quantifies the time and resources required to write and test the software. This may be measured in terms of algorithmic efficiency of the software, coupled with the efficiency measure of each Sprint. Looking at the integrated, working software at the end of each Sprint from high-level dynamic event traces that are required to achieve the functional requirements of the system allows measurement of the functional complexity (FC). Representational complexity (RC) is measured from systems architecture (DoDAF[1]) perspective, looking at the graphical and textual notations for representations of the System Model (SV-1), System Interactions (SV-3), and System Behaviors (SV-4). Based on a measure from zero to one, the overall Sprint Complexity Factor (SCF) for a given team for a given Sprint is:

$$\mathrm{SCF} = \mathrm{CC} \times \mathrm{FC} \times \mathrm{RC}$$

and the overall agile cost and schedule metrics, Agile Cost Variance (ACV), and Agile Schedule Variance (ASV) become:

$$\mathrm{ACV} = \mathrm{CV} \times \mathrm{SCF}$$
$$\mathrm{ASV} = \mathrm{SV} \times \mathrm{SCF}$$

The overall Agile Effectiveness (AE) of a given Sprint for n number of Sprint teams is:

$$\mathrm{AE} = \sum_{i=1}^{n} \mathrm{ACV}_i \times \mathrm{ASV}_i$$

The Cumulative Agile Earned Value (CAEV) effectiveness measure, across m number or Sprints, then becomes:

$$\mathrm{CAEV} - \sum_{j=1}^{m} \sum_{i=1}^{n} \mathrm{ACV}_{i,j} \times \mathrm{ASV}_{i,j}$$

The next section described another important measure of agile software that must be taken into account in measuring Earned Value for agile development programs/projects, and that is Entropy. While some changes are embraced by the

[1] Department of Defense Architecture Framework.

agile design methodology, it is important to measure those phenomena that drive uncertainty into agile development and are indicators of impending problems within the overall development rhythms of the agile program/project. We will discuss two of these in the next section:

1. Volatility in Sprint team membership: As was discussed earlier, it is important to keep the Sprint teams as stable as possible across the development program/project in order to keep a stable and sustainable development rhythm.
2. Volatility or velocity of increase/decrease of Sprint software defects. As the Sprint teams work together, get used to each other, understand each other's strengths and expertise, and as they gain experience writing software for this project, one would expect the number of defects across each Sprint to decrease. One way to measure this is with Entropy or the measure of change across Sprints.

5.3 Entropy as an Earned Value Metric for Agile Development

Entropy is a concept in information theory proposed by Shannon [67], and generalized by Rényi [9]. Entropy is used in information theory to provide a quantitative measure of uncertainty in systems random variables. A simple way to describe the use of Entropy is to say that the more uncertainty there is in a given system, the more potential there is for volatility within the system. This is exactly the case we have with agile development programs/projects. However, the uncertainty here is not the uncertainty of requirements change, but the uncertainty of increase in Entropy of certain factors that drive the efficiency of agile development teams [32]. In particular we are talking about the uncertainty of teams (moving people between teams or brining new people into teams) and the uncertainty that can be measured in the software defect volatility.

5.3.1 Entropy Measures

The measurement of Entropy is based on the uncertainty associated with random variables. And while you might think that this does not apply to agile development teams, the following discussion may change your mind.

At the beginning of an agile development project, it is not possible to determine how many times an Agile Manager may change out team members, either because they choose to or because a team member leaves the company or the program/project. Therefore, we can think of the volatility of team membership as a random variable, X. Also, it is not possible to determine the number of software defects that will be created through software development across the teams and Sprints. Attempts are made to make predictions, but these also are treated as random variables and attempts are made to predict the average number of defects and the "bounds" on defects.

Again, we will call the number of defects random variable, ***Y***. Given a random variable ***X***, with probability distribution ***P***, Shannon's Entropy calculation is computed from the following:

$$H(X) = \sum_{i=1}^{n} p(x_i) I(x_i) = \sum_{i=1}^{n} p(x_i) \log_2 \frac{1}{p(x_i)} = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$

where $p(x_i)$ represents the point probability at time i. The next two sections will explore the use of Entropy to measure uncertainty within an agile development project and how to utilize this in the overall Agile Earned Value metrics.

5.3.2 Volatility of Teams

Earlier, we discussed the problems associated with changing out Sprint team members during the agile development program/project. This volatility of team members disrupts the agile development process and introduces uncertainty (or entropy) into the development efforts. In order to adequately measure the effectiveness and productivity of agile development, the Entropy of Team Volatility (ETV) must be a factor in the Agile Earned Value metrics. We will let X be a random variable that describes the probability of a change in team members across one or more agile teams, where the probability of team member volatility increases with the number of people in each team and increases with the number of teams. As the team size and the number of teams increase the probability of a change of one or more personnel increases also. We will assign an exponential random variable to the probability that there will be personnel changes, given a number of teams and number of personnel/team. Also, since removing a team member means adding a new team member, and changing out personnel from teams means moving at least two people (or an even number of personnel), for n number of changes there are $2n$ people changed. Therefore the uncertainty (or entropy) equation becomes:

$$p(X) = \lambda e^{-\lambda X},$$

where $X = \ln\left(\sum_{i=1}^{n} \text{size}(\text{team}_i)\right)$, $n = \#$ of teams.

For example we will let $\lambda = 0.5$. Cumulative Density Function for X is $1 - \lambda e^{-\lambda X}$ and looks like in Fig. 5.4.

For instance, if there are six Sprint teams, each with eight people, then the probability of a change occurring within the team structure is:

$$p(\text{change}) = 1 - 0.5e^{-0.5\ln(48)} = 0.97$$

This says it is 97 % likely that there will need to be a change across the teams at least once in the agile development cycle for the program/project.

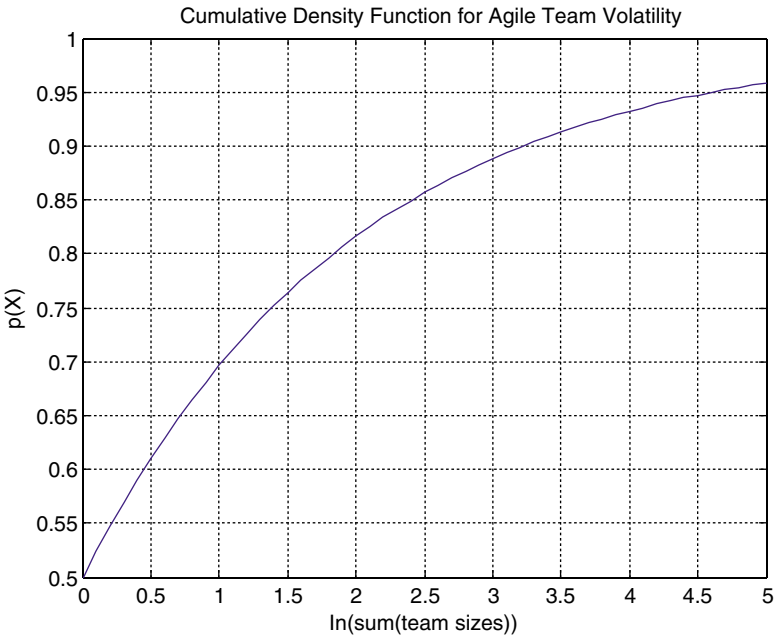

Fig. 5.4 Cumulative density function for agile team volatility

If we let the agile development project be comprised of five agile teams and eight developers/team, and if there are three personnel moves, which constitutes six changes, then the entropy factor caused by these changes is $p(X) = 0.079$, and the Entropy caused by this is 0.29, or there is an estimated 29 % degradation to future Agile Earned Value due to the Entropy introduced by three personnel moves. Therefore, the Agile Earned Value metrics must be reduced by 29 %: new Agile Earned Value = Measured Agile Earned Value × 0.71. This is a significant amount, but should be taken into account to have more accurate Agile Earned Value Metrics, and allows the Agile Manager to keep track of potential issues that they may introduce by changing out personnel throughout the program/project agile development cycle.

5.3.3 Volatility of Software Defects

Volatility of software defects is a measure of whether the software defects are decreasing over time, given that over time, the developers become more familiar with the overall system being developed and how all the services play together, and become more comfortable with the teams and team environment. For a given set of capabilities within each successive Sprint, if the complexity between the Sprints is normalized, one would expect the software defects to be decreasing. An increase in the normalized software defects over successive Sprints is increasing; this indicates

volatility or Entropy in the development process and must be measured and remedies determined and put into place across the Sprint teams.

The Software Defect Entropy is determined and measured, based on the first Sprint, setting the complexity factor for the first Sprint equal to 1. Then the complexity of each successive Sprint is measured against the first Sprint and a complexity factor determined. Based on the software defects Sprint 1, the Software Defect Factor for Sprint i is:

$$\mathrm{SDF}_1 = \#\,\mathrm{defects}_1$$
$$\mathrm{SDF}_i = \#\,\mathrm{defects}_i \times \text{normalized complexity factor}_i, i > 1$$

If $\mathrm{SDF}_{i+1} > \mathrm{SDF}_i$ it indicates Entropy has been introduced into the software development process and the causes must be determined and adjudicated in order to get the agile development effort back on track. The total Software Defect Entropy (SDE) across the agile development project is then measured, where we compute the change in normalized defects/Sprint team across each Sprint, or:

$$\mathrm{SDE} = \sum_{i=1}^{m}\sum_{j=1}^{n}\mathrm{SDF}_{i,j},$$

where m = #Sprints and n = #teams.

Adjusting Earned Value metrics for agile development will be a long paradigm shifting exercise for managers and may take time to get the Agile Manager to use different types of metrics and measures than they have been used to. The emphasis with agile development needs to be on measuring the agile process, and results (working code), not antiquated measures like SLOC. Only when we embrace measures that are effective for agile development will the Agile Manager be able to truly understand the dynamics and issues with their agile development programs/projects.

Having dealt with most of the issues surrounding the management of agile development, we move on to a subject that has been getting much more visibility in the last few years, and that is the subject of inclusiveness and diversity as part of the overall team dynamics for agile development.

Chapter 6
Conclusion: Modern Design Methodologies—Information and Knowledge Management

One of the things that must be understood by leadership and management in the future is that just because you deliver a product on time and on budget doesn't mean the project was an overall success. Delivering a product on budget and on schedule but decimating a development team is not, in the long run, a success for the company. Managers and Leaders must understand all aspects of development teams for long-term success. And while some managers may wonder if they have a role in the new agile development paradigm, the Agile Manager understands that their role is more important than ever to the overall success of the agile development program/project. The Agile Manager must split their time between facilitating each Sprint team to allow them to be as efficient and effective as possible, keeping track of agile productivity metrics, and keeping open and active collaboration and cooperation with the customer in terms of reviewing working software at the end of each Sprint and helping to shape the requirements, capabilities, and needs going forward. These are three paradigm domains that must be attended to by the Agile Manager in order to keep the agile development rhythms sustained throughout the agile development life cycle. A lack of commitment to any of these Agile Management domains or treating any one of them in isolation may result in ineffective Sprint development teams.

It is important for the Agile Manager to realize that quality software is not achieved through reacting to software failures during integration and test; instead, it is achieved through the time and desire on the part of the entire development team and the Agile Manager to produce a quality product. Both the developers and the Agile Manager must work together to create a cohesive program/project development team that works collaboratively within the Sprint teams, cooperatively across Sprint teams, and working in tandem with the Agile Manager, bringing issues up early to allow the Agile Manager to remove roadblocks from efficient development.

For the Agile Manager, in order to keep a working team it is important to utilize soft people skills as well as standard agile management skills. This includes understanding how to facilitate inclusion of cultural diversity, minimize judgment of people, and focus on expertise. The manager needs to have an understanding of

© Springer International Publishing Switzerland 2015

J.A. Crowder, S. Friess, *Agile Project Management: Managing for Success*,
DOI 10.1007/978-3-319-09018-4_6

Locus-of-Control so that the team feels accountable individually and to the team, such that each member feels a sense of agency. The manager works toward empowerment of all individuals on the team. Each member has a voice and is heard. Without the facilitator, team dynamics can easily be taken over by those who have a strong voice yet lack expertise of other members of the team. This is a side-by-side development style versus a completive individualistic style. Save the competition for the team as a whole and the end product.

In addition, embracing new productivity tools and new agile metrics (Agile Earned Value) will allow the Agile Manager to manage the diversity of information, lessons learned, etc. that results from the agile development process. The real question for the Agile Manager is: will they release control to the agile development teams? If the manager insists on holding on to old ways of managing, ineffective metrics that are antithetic to agile development, and will not allow empowerment within the teams, it is difficult to envision a truly successful agile development program/project. In the end, this paradigm shift will be borne out of economic necessity as companies lose their competitive edge to companies that will embrace the new paradigm of agile development, and Agile Managers embrace their new, important roles, facilitating efficient, effective, and robust agile development teams. In the end, the true Agile Manager will be those managers for whom these concepts become intuitive and not forced. We all understand that change is hard and many people don't deal well with change, but change will happen with or without the reluctant manager.

References

1. Abba W. How earned value got to prime time: a short look back and a glance ahead. PMI College of Performance Management; 2000 (www.pmi-cpm.org).
2. Abran A, Ormandjieva O, Abu Talib M. Information-theory-based functional complexity measures and function size with COSMIC-FFP. Montreal: Université du Québec à Montréal; 2001.
3. Alleman G. Herding Cats: issues with deploying earned value management; 2012. Available from http://zo-d.com/blog/archives/project-management-on-the-web/-pm-web-001-glen-b-allemans-herding-cats.html
4. Allen B. Diversity and organizational communication. J Appl Commun Res. 1995;23:143–55.
5. Astin H, Leland C. Women of influence, women of vision: a cross-generational study of leaders and social change. San Francisco, CA: Jossey-Bass; 1991.
6. Bandura A. Self-efficacy: the exercise of control. New York: W. H. Freeman; 1997.
7. Barnes K. Applying self-efficacy theory to counselor training and supervision: a comparison of two approaches. Counsel Educ Supervision. 2004;44(1):56–69.
8. Beck K. Extreme programming explained - embrace change. Boston, MA: Addison-Wesley; 1999.
9. Beck C, Friedrich A. Thermodynamics of chaotic systems: an introduction. Cambridge: Cambridge University Press; 1993. ISBN 0521433673.
10. Becker J, Kovach A, Gronseth D. Individual empowerment: how community health workers operationalize self-determination, self-sufficiency, and decision-making abilities of low-income mothers. J Commun Psychol. 2004;32(3):327–42.
11. Black L, Magnuson S. Women of spirit: leaders in the counseling profession. J Counsel Dev. 2005;83(3):337–42.
12. Bollen K. Indictor: Methodology. In: Smelser N, Baltes P, editors. International encyclopedia of the social and behavioral sciences. Oxford: Elsevier Science; 2001. p. 7282–87.
13. Booch G. Object solutions: managing the object-oriented project. Upper Saddle River, NJ: Pearson Education; 1995. ISBN 0-8053-0594-7.
14. Brooks F. The mythical man-month. Boston, MA: Addison-Wessley; 1975. ISBN 0-201-00650-2.
15. Campbell J, Trapnell P, Heine S, Katz E, Lavallee L, Lehman D. Self-concept clarity: measurement, personality correlates, and cultural boundaries. J Personal Soc Psychol. 1996;70:141–56.
16. Carless S. Does psychological empowerment mediate the relationship between psychological climate and job satisfaction? J Bus Psychol. 2004;18(4):405–25.
17. Crawford A. Empowerment and organizational climate: an investigation mediating effects on the core self-evaluation, job satisfaction, and organizational commitment relationship. ProQuest Dissertations and Theses; 2008. p. 147.

© Springer International Publishing Switzerland 2015

J.A. Crowder, S. Friess, *Agile Project Management: Managing for Success*,
DOI 10.1007/978-3-319-09018-4

18. Crowder JA, Carbone J. Recombinant knowledge relativity threads for contextual knowledge storage. Proceedings of the International Conference on Artificial Intelligence; 2011; Las Vegas.
19. Crowder J, Friess S. Systems engineering agile design methodologies. New York, NY: Springer; 2013. ISBN 1461466628.
20. Csikszentmihalyi M. Good business: leadership, flow, and the making of meaning. New York, NY: Penguin Group; 2003.
21. Defense Systems Management College. Earned value management textbook, Chapter 2. Fort Belvoir, VA: Defense Systems Management College; 1997.
22. Deming W. Out of the crisis. Cambridge, MA: Massachusetts Institute of Technology; 1986.
23. Dirks K, Ferrin D. Trust in leadership: meta-analytic findings and implications for research and practice. J Appl Psychol. 2002;87:611–28.
24. Eysenck H, Eysenck S. Personality structure and measurement. San Diego, CA: Robert R. Knapp; 1969.
25. Fleming Q, Koppelman J. Earned value project management. 3rd ed. Newton Square, PA: Project Management Institute; 2005. ISBN 1-930699-89-1.
26. Francisco M, Holcombe M, Gheorghe M. A formal experiment comparing extreme programming with traditional software construction. Proceedings of the Fourth Mexican International Conference on Computer Science; 2003.
27. Freeman S, Bourque C, Shelton C, editors. Women on power: leadership redefined. Boston, MA: Northeastern University Press; 2001. p. 3–24.
28. Frieze C, Blum L. Building an effective computer science student organization: the Carnegie Mellon Women@SCS Action Plan. Inroads SIGSCE Bull Women Comput. 2002;34(3):74–8.
29. Gardner W, Avolio B, Luthans F, May D, Walumbwa F. Can you see the real me? A self-based model of authentic leader and follower development. Leadersh Q. 2005;16:343–72.
30. Glib T. Evolutionary project management; 2013. Available from http://ebookbrowsee.net/document-evoprojectmanagement-pdf-d357611912
31. Goodpasture J. Quantitative methods in project management. Plantation, FL: J. Ross; 2004. p. 173–8. ISBN 1-932159-15-0.
32. Harrison W. An entropy-based measure of software complexity. IEEE Trans Software Eng. 2000;18(11):1025–9.
33. Harter S. Authenticity. In: Snyder CR, Lopez S, editors. Handbook of positive psychology. Oxford: Oxford University Press; 2002. p. 382–94.
34. Harter J, Schmidt F, Hayes T. Business-unit level relationship between employee satisfaction, employee engagement, and business outcomes: a meta-analysis. J Appl Psychol. 2002;87(2): 268–79.
35. Harvey C, Allard J. Understanding and managing diversity. 5th ed. Upper Saddle River, NJ: Pearson Education; 2012. ISBN 0-13-255311-2.
36. Hazzen O, Dubinsky Y. Empower gender diversity with agile software development. In: Trauth EM, editor. The encyclopedia of gender information and information technology. Hershey, PA: IGI Global; 2006. p. 249–56.
37. Highsmith J. Agile software developments ecosystems. Reading, MA: Addison-Wesley; 2002.
38. Hofstede G, Hofstede GJ, Minkov M. Cultures and organizations: software of the mind. New York, NY: McGraw-Hill; 2010.
39. ISO/IEC 19761. Software engineering – COSMIC-FFP-A functional size measurement method. International Organization of Standardization – ISO; 2003; Geneva, Switzerland.
40. Iyengar P. Application development is more global than ever, Publication G00124025. Stamford, CT: Gartner; 2004.
41. Jacobsen I. Object-oriented software engineering: a use case driven approach. Boston, MA: Addison-Wesley; 1992. ISBN 0-201-42289-1.
42. Jeffries R, Anderson A, Hendrickson C. Extreme programming installed. Boston, MA: Addison-Wesley; 2001.
43. Jewson N, Mason D. The theory of equal opportunity policies: liberal and radical approaches. Sociol Rev. 1986;34(2):307–34.

44. Jha SS, Nair SK. Influence of locus of control, job characteristics and superior-subordinate relationship on psychological empowerment: a study in five star hotels. J Manag Res. 2008;8(3):147–61.
45. Judge TA, Erez A, Bono JE, Thoresen CJ. Are measures of self-esteem, neuroticism, locus of control, and generalized self-efficacy indicators of a common core construct? J Personal Soc Psychol. 2002;83(3):693–710.
46. Kanter RM. The change masters: innovation for productivity in the American Corporation. New York, NY: Simon and Schuster; 1983.
47. Kanter RM. The challenges of organizational change. New York, NY: Simon and Schuster; 1992.
48. Karlesky M, Williams G, Bereze W, Fletcher M. Mocking the embedded world: test-driven development, continuous integration, and design patterns. Processings of the Embedded Systems Conference Silicon Value; 2007; San Jose, CA.
49. Kernis MH. Toward a conceptualization of optimal self-esteem. Psychol Inq. 2003;14:1–26.
50. Kurian T. Agility metrics: a quantitative, fuzzy-based approach for measuring agility of a software process. ISAM-Proceedings for the International Conference on Agile Manufacturing'06 (ICAM-2006); 2006; Norfolk, VA.
51. Lappo P, Andrew H. Assessing agility. In: Extreme programming and agile processes in software engineering, Lecture Notes in Computer Science, vol. 3092. Heidelberg: Springer; 2004. p. 331–8.
52. Larson R, Walker K, Pearce N. A comparison of youth-driven and adult-driven youth programs: balancing inputs from youth and adults. J Commun Psychol. 2005;33(1):57–74.
53. Lee L. The empowerment approach to social work practice. New York: Columbia University Press; 1994.
54. Mann HH. Empowerment in terms of theoretical perspectives: exploring a typology of the process and components across disciplines. J Commun Psychol. 2006;34(5):523.
55. Marshall R. The contribution of earned value management to project success of contracted efforts. J Contract Manag. 2007;Summer:21–33.
56. Nichols JD. Empowerment and relationships: a classroom model to enhance student motivation. Learn Environ Res. 2006;9(2):149–61.
57. Noll J, Atkinson D. Comparing extreme programming to traditional development for student projects: a case study. Proceedings of the 4th International Conference of Extreme Programming and Agile Processes in Software Engineering; 2003.
58. Page N, Czuba CE. Empowerment: what is it? J Extens. 1999;37(5).
59. Peters J, Pedrycx W. Software measures in software engineering: an engineering approach. New York, NY: Wiley; 2000.
60. Piotrowski CL. Quantum empowerment: a grounded theory for the realization of human potential (Order No. 3240834, Cardinal Stritch University). ProQuest Dissertations and Theses; 2006. p. 358.
61. Pisano N. Technical performance measurement, earned value, and risk management: an integrated diagnostic tool for program management. Defense Acquisition University Acquisition Research Symposium; 1999.
62. Roope K. Efficient authoring of software documentation using RaPiD7. ICSE, 25th International Conference on Software Engineering (ICSE'03); 2003, p. 255.
63. Rotter J. Generalized expectancies for internal versus external control of reinforcement. Psychol Monogr. 1966;80(1):1–28.
64. Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W. Object-oriented modeling and design. Upper Saddle River, NJ: Prentice Hall; 1990. ISBN 0-13-629842-9.
65. Schulze W, Gerking S, De Haan M. How earned value management is limited. J Risk Uncertainty. 2010;1(2):185–99.
66. Schwaber K. The scrum development process. Processings of the OOPSLA'95 Workshop on Business Object Design and Implementation; 1995.
67. Shannon C. The mathematical theory of communication. Urbana, IL: University of Illinois Press; 1969.

68. Skinner EA. A guide to constructs of control. J Personal Soc Psychol. 1996;71(3):549–70.
69. Speer PW. Intrapersonal and interactional empowerment: implication for theory. J Commun Psychol. 2000;20(1):51–61.
70. Spreitzer GM. Psychological empowerment in the workplace: dimensions, measurement and validation. Acad Manag J. 1995;38(5):1442–65.
71. Stanhope DS, Samuel B, Surface EA. Core self-evaluations and training effectiveness: prediction through motivational intervening mechanisms. J Appl Psychol. 2013;98(5):820–31.
72. Sulaiman T, Barton B, Blackburn T. Earned value management the agile way. Proceedings of the AGILE Conference; 2007. p. 10.
73. Sumara J, Goodpasture J. Everything you wanted to know about time-centric earned value. PM Network. 2000;14:51–4.
74. Thomas K, Velthouse B. Cognitive elements of empowerment. Acad Manag Rev. 1990;15:666–81.
75. Tom G. Principles of software engineering management. Boston, MA: Addison-Wesley; 1998. p. 133–58.
76. Walck C. Diverse approaches to managing diversity. J Appl Behav Sci. 1995;31:119–23.
77. Wienberg G. Iterative and incremental development: a brief history. Computer. 2003;36(6): 47–56.
78. Williams L, Succi G, Stefonovic J, Marchesi M. A metric suite for evaluating the effectiveness of an agile methodology. In: Marchesi M, Succi G, Wells D, Williams L, editors. Extreme programming perspectives. Boston, MA: Addison-Wesley; 2003.
79. Botella L. Personal construct theory, constructivism, and postmodern thought. In: Neimeyer RA, Neimeyer J, editors. Advances in personal construct psychology, vol. 3. Greenwich, CT: JAI Press; 1995. p. 3–35.

Index

© Springer International Publishing Switzerland 2015

J.A. Crowder, S. Friess, *Agile Project Management: Managing for Success*,
DOI 10.1007/978-3-319-09018-4