CHAPTER

1

Introduction

"We understand that the only competitive advantage the company of the future will have is its managers' ability to learn faster than then their competitors."

Arie de Geus (1988)

COPYRIGHTED MATERIAL

Software development, in all its forms, is an exercise in learning. Learning occurs within the teams that develop the software – not just the amongst the managers. Then learning occurs with the people who use the software. If we exploit this learning, we can enhance the competitive advantage for our companies.

In order to recognize the value of learning, it's necessary to change things: to change what we do and the way we do it. Without change we can't truly learn, and we certainly don't exploit our learning. The process of learning and changing is an exercise in knowledge creation. Knowledge itself is learning with action: this action often manifests itself as change. This idea, summarized in Figure 1.1, runs through this book.

Knowledge is the underpinning of our modern economy – hence the 'knowledge economy' – and IT is a key part of this economy. Modern IT wouldn't be what it is without software, and that software needs to be written. Yet the people who develop software, and those who manage them, seldom talk about knowledge and the role that IT can play in enhancing learning. All too often, we prefer to view software development as some sort of factory production line process.

This view runs far beyond the development process. Organizations buy software and other IT products in order to create change. Introducing the

Changing Software Development: Learning to Become Agile Allan Kelly
© 2008 John Wiley & Sons, Ltd.

Knowledge = Learning + Action

Figure 1.1 Knowledge is.

software creates change for the users, and subsequent changes to the software create more change.

Today's software developers and their managers face three major forms of change. Firstly, there's the need to adopt Agile methods. Agile and Lean development techniques are now established and are moving into the mainstream arena. In order to adopt these techniques, development teams must change the way in which they work.

Secondly, having adopted Agile development, these best-performing teams need to move far beyond the current methods and best practices. Before long, simply adopting the prescribed practices of a methodology such as eXtreme Programming won't be enough. Each team must learn for itself what works best.

Finally, IT is all about creating change in others. IT deployments that inflict change on helpless users don't recognize the true benefits of IT; indeed, many such projects are outright failures. Those developing and deploying software must appreciate the need for change and learning by the end users.

Learning and change are complicated fields. All too often, people see change as the simple application of raw authority: *tell someone to do it differently, tell someone to use a new system, punish them if they get it wrong*. Unfortunately, this technique doesn't work very well. In particular, it isn't effective with knowledge workers who may actually know more about the problem than anyone in a position of authority. So we need a new view of change to help us with these problems.

Fortunately, the best people in IT – and, in particular, the actual software development side – like learning. Much, if not most, IT work is problem solving, which is itself a form of learning. Therefore, we need to help people learn, help them learn the right things and ensure that this learning is maximized through meaningful change.

1.1 Why Read this Book?

Even if you don't wish to embrace Agile software development, there are good reasons to embrace learning and create a learning culture. Building a learning environment and culture can help improve the way in which you create software and benefit your company in many ways.

This book is primarily written for software developers and managers who want to improve the way in which they, and their teams, develop software. Software developers who are making, or have recently made, the transition to team leadership and development management should find the ideas particularly interesting.

There's an additional group of people who I hope will find this book interesting: those dependent on the work of a software development team.

Such people often view IT, and specifically development activities, as a foreign land. Viewing IT as a learning activity, rather than an engineering or scientific activity, can help explain much of what goes on in that land.

1.1.1 Learning for Agility

The book aims to help in three ways:

- *For teams that want to be Agile:* Increasingly, we know what Agile software development is. The problem facing those who aren't Agile is not 'What is Agile?' or 'What do Agile teams do differently?' The problem is rather 'How do we change so that we're Agile?' This book presents learning as a mechanism for creating change.
- *For teams that are Agile and want to improve further:* For teams that have achieved Agility, the challenge is slightly different. Such teams will already be seeing the benefits of the Agile approach. However, there's still a need to improve and become even better. The learning-based approach can help here too.
- *By explaining the role of learning in software development:* During the past 40 years, there have been many attempts to make software development fit within the engineering and process metaphors. Despite this, software projects have continued to fail. In this book, I suggest that software development is an exercise in learning and knowledge management. Changing our perspective offers new insights and approaches. In particular, this perspective allows us to harness the tools and experience of the organizational learning movement instead of the tools of engineering.

This book doesn't attempt to be the last word on any of these subjects. I've tried to point you to many sources for further investigation. Instead, this book aims, firstly, to introduce each of these topics and, secondly, to weave them into a coherent narrative to explain software development as a learning activity.

1.1.2 Learning Creates Competitive Advantage

Modern business is constantly searching for competitive advantage: the ability to out-compete rivals, to sell more products, to sell more expensive products and to increase production. Once upon a time, competitive advantage could be gained by having better physical access than your competitors to some resource, land, labour or capital.

Today, firms seek competitive advantage through better access to knowledge and by their ability to act on this knowledge. Knowledge is the result of learning; therefore, as suggested in the opening quote, a firm's ability to learn may be its only competitive advantage.

By learning, we're able to create better products: we learn more about our customers, we learn more about the technology our products are built from and we learn how to produce the products more efficiently. Using this learning, we're able to improve our products. Learning about our customers, products and manufacturing process may allow us to create better products.

Learning also allows us to increase our productivity. Through learning, we're able to build products faster, more efficiently and with less waste. This allows us to maximize the returns from our investment – whether capital or workers' time – and generate more profit. In these cases, the firm's ability to learn is key to helping the firm improve and succeed. The firm that learns fastest wins.

But learning isn't just essential in order to win: it's also essential in order to survive. Modern businesses exist in a changing environment, new competitors enter markets, customer expectations change, and technologies and regulations change. Firms that don't learn and adapt to a changing environment may not survive.

So, learning isn't an optional extra. Firms and individuals must learn if they are to survive. For those that master learning and can learn faster than others, there are rewards.

1.1.3 Good People Like Learning

Humans are natural learners. Our ability to learn faster than many other animals is one of the reasons why we humans have advanced as far as we have. Within software development, those who enjoy and excel at learning tend to perform better than those who dislike learning new things. There are always new technologies and application domains to learn. Anyone who dislikes learning would be well advised to avoid a career in software development.

The search for competitive advantage outlined above isn't the only reason to embrace learning. People who enjoy learning are more motivated when given an environment in which they can learn more. Motivated people get more job satisfaction and are more productive.

Naturally, when people are motivated and happy with their work they are more likely to remain with the same employer. Therefore, creating a learning environment should help improve staff retention. Recruitment may also become easier, as word spreads of a positive work environment, filled with motivated people who are learning new things.

1.2 Who are Software Developers?

The term *software developer* is most often used to describe the engineers who write program code. In truth, there are many more roles necessary to develop software: testers, business analysts, designers, product managers, architects,

deployment specialists, project managers, development managers and others all have a hand in developing the software.

The IT community doesn't have a standard set of job titles and pre-defined roles; what one company calls a 'product manager' is an 'architect' elsewhere, one company's 'project manager' is another's 'development manager', a 'team leader' in one is a 'manager' in another, and so on. All these people are in some way contributing to the development of a software system.

The level of knowledge and experience required to develop a successful system causes the old 'blue-collar'/'white-collar' division to fade. Someone who thinks of a programmer as analogous to a factory worker is making a mistake: the level of knowledge required by a programmer is several orders of magnitude greater than that required by an assembly line worker.

The profile of a modern development team looks more like a group of white-collar managers than a set of blue-collar workers: highly skilled people with specific knowledge who spend their days making informed decisions – not to mention working in air-conditioned offices. Consequently, when looking outside the IT arena, research, advice and inspiration are often to be found in texts that discuss management challenges.

Thinking Point: Why Do You Want To Change?

This book is going to discuss changing the way in which you create software. Specifically, I'm going to describe how you can help your team adopt Agile software practices. Before getting stuck into the task in hand, it is worth taking a step back and asking: *Why? – Why do we want to change the way in which we do things?*

Before you read any further, put this book down and make a list of five reasons why you'd like to change the way in which your organization develops software:

- Try to think beyond immediate reasons such as a recently failed project.
- Try to think about *why*, not *what*.
- Try to think about big reasons rather than small ones.
- Try to think about your company as a whole rather than just your team: *What benefit will this bring?*
- Be honest: if you want to change the team to further your own career, recognize it – you don't have to tell anyone else.

You might also want to think about the opportunities that you can see if you can change.

Now that you've made the list, put it to one side. (If you want to hide it, do so!)

There are various reasons why you might want to change your development practices. Here are a few reasons, all of them legitimate:

- To improve the competitiveness of your team or company.
- To improve the quality of your software.
- To increase the productivity of your team.
- To create new business opportunities, products and/or services.
- To address a problem that you're having today.
- To save your own job, perhaps by preventing your work being outsourced and/or sent off shore.
- To better serve the business.
- To enjoy your job more.

This isn't an exhaustive list; nor are the items in the list distinct – they all overlap. Depending on your situation, some will be cause and others effect: improving the quality may allow you to support your business better and prevent your department being outsourced, thereby saving your job.

In fact, everything could be reduced to the first item: *improve company competitiveness*. However, this is so general as to be of little use. Most of the other reasons can be reduced to either quality or productivity, but to do so means losing useful information about motivation.

1.3 Software Developers are Knowledge Workers

If we look at the definition of knowledge workers, it is clear that it includes developers:

> ''Knowledge workers have high degrees of expertise, education, or experience, and the primary purpose of their jobs involves the creation, distribution, or application of knowledge.''
>
> Thomas Davenport (2005)

Indeed, writers and experts on the knowledge economy and knowledge workers frequently cite software developers, and IT people in general, as prime examples of knowledge workers. These are individuals who work primarily with their knowledge. Yet it is rare for those in IT, or writers about IT, to discuss software developers as knowledge workers. But then: Why would they? What difference does it make?

This book will argue that by viewing software developers as knowledge workers, and considering development activities as knowledge creation with

active learning processes, we gain many useful insights into the process by which software is developed and deployed. By recognizing IT staff as knowledge workers, a rich field of literature and experience opens up from which we may learn from to help improve our own practice.

From the same book quoted above, we can distil a list of knowledge work characteristics:

- Knowledge workers like autonomy: they don't like being told what to do.
- Specifying detailed steps to follow is less valuable than in other types of work.
- Knowledge workers find it difficult to describe what they do in detail: if you want to know, you're better off watching.
- Not only do knowledge workers find it difficult to describe what they do, but they're aware of the value of knowledge and don't share it without a motivation.
- Even though they may not be able to describe what they do, these workers often have good reason for doing what they do and have often thought in advance about the way in which they work.
- Commitment matters and makes a huge difference in productivity.

Looking at this list, two things stand out: firstly, this is a list of developer characteristics too, so any doubt that developers are knowledge workers should be dispelled. Secondly, an individual with these characteristics is unlikely to relish routine, factory-like, work. The traditional view of management isn't applicable to these workers.

Recognizing that IT workers are knowledge workers also recognizes that they're not unique. They share the same characteristics as other knowledge workers. Nor are the problems that they encounter unique. The opportunities and problems faced by IT staff and their managers are quite legitimate, and are shared by other modern knowledge workers. Consequently, it is wrong to think of the 'IT geek' as a class apart.

Once we recognize software developers as knowledge workers, it becomes clear that development activities – specifying, designing, coding and testing new software – are themselves knowledge activities. Such activities are completely different from traditional factory production line processes, where a worker's individual knowledge makes little immediate difference to the end product. Having recognized this critical difference, it becomes meaningless to characterize software development as a factory process.

Many previous attempts to change the way in which IT staff work were misplaced because they failed to recognize the roles of knowledge and the characteristics of knowledge workers. Naive attempts at quality improvement, productivity enhancement and cost cutting that draw on manufacturing experience are simply wrong.

1.4 Drucker's Challenge

Defining software development as knowledge work doesn't allow us to ignore the issue of productivity. Productivity and quality are still very important to the success of a business venture. The management guru Peter Drucker forecast the emergence of this issue as long ago as 1969:

> ''Knowledge work is not easily defined in quantitative terms, . . . To make knowledge work productive will be the great management task of this century, just as to make manual work productive was the great management task of the last century.''
>
> Peter Drucker (1969)

How you measure productivity in software development is a good question. It is most certainly not lines of code, function points or hours worked. Still, no matter how difficult it is to measure, we are producing something and we can always improve productivity and quality. Perhaps we just have to live with this ambiguity.

Any attempts to quantify software development productivity must make allowance for the multiple results of such work. In developing a piece of software we create a deliverable executable, but there are by-products. The developers themselves increase their stock of knowledge – about their tools, about the subject of the software and about the creation process. Similarly, managers, users and others involved with the specification, implementation and delivery of the software will learn as a by-product.

Despite the problems of measuring productivity, we can still discuss the issues, and we can still ask how we can address Peter Drucker's challenge. Much of this book is directed at addressing this challenge: *How can we make software developers more productive?*

The Agile and Lean schools give us the methods to increase developer productivity, but we still need to apply them. The challenge we face is less 'What can we do to be more productive?' and more 'How can we move from here to there – from where we are today to more productive practices?' and 'How can we continue to improve our productivity?'

In other words: *How do we change? How do we continue to change? How do we go beyond our current stock of knowledge?*

1.5 The Prototype of Future Knowledge Workers

Highlighting IT workers as knowledge workers allows us to learn from the existing body of knowledge on the subject. IT workers are not alone; they are knowledge workers and there's much to learn from other knowledge workers, and from research and literature about knowledge work in

general. There's no need for IT managers (and writers) to re-invent the wheel.

Yet, in another way, the existing literature, research and experience can't help IT workers and their managers. This is because IT workers, and software developers in particular, are at the cutting edge of knowledge work. In many ways, they're the prototype of the future knowledge worker; they're pushing the boundaries of twenty-first century knowledge work.

This occurs because, to paraphrase Karl Marx, software developers control the means of production. Modern knowledge work is enabled by and dependent on information technology: e-mail for communication, web sites for distribution, databases for storage, word processors for writing reports, spreadsheets for analysis – the list is endless! These technologies are created by software developers and used by legions of knowledge workers worldwide. The key difference between software knowledge workers and the others is that other knowledge workers can only use the tools that exist. If a tool doesn't exist, they can't use it. Conversely, software developers have the means to create any tool they can imagine.

Consequently, it was a programmer, Ward Cunningham, who invented the Wiki. Programmers Dan Bricklin and Bob Frankston invented the electronic spreadsheet. Even earlier, it was another programmer, Ray Tomlinson, who invented inter-machine e-mail. This doesn't mean that non-programmers can't invent electronic tools. Others can invent tools, but for programmers the barriers between imagining a tool and creating the tool are far lower.

Lower barriers mean that programmers create many more tools than other types of worker. Some tools fail, while others are very specific to a specific problem, organization or task in hand, but when tools do work it is programmers who get to use them first. In addition, because IT people have had Internet access for far longer than any other group, the propensity to use it to find tools and share new tools is far greater. So tools such as Cunningham's Wiki were in common use by software developers years before they were used by other knowledge workers.

Early Internet access has had other effects too: IT workers were early adopters of remote working, either as individual home workers or as members of remote development teams; IT people are far more likely to turn to the Web for assistance with problems and more likely to find it, because IT information has been stored on the Web since the very beginning.

The net effect of these factors and others means that software developers are often the first to adopt new tools and techniques in their knowledge work. They're also the first to find problems with such tools and techniques. Consequently, these workers are at the cutting edge of twenty-first century knowledge work; they are the prototype for other knowledge workers. Other knowledge workers, and their managers, can learn from the way in which IT people work today, provided that we recognize these workers as knowledge workers.

1.6 Software: Embedded Knowledge

When we program, we teach a computer to do something. We use our knowledge of computers and programming to create an automated system that embodies knowledge. For example, accounts software contains knowledge of accounting principles and practices, the software in a telephone exchange contains knowledge of call handling and routing, and so on.

As we shall see later (Section 4.1), software brings together three knowledge domains: knowledge of the technical tools to create the software, knowledge of software creation process and knowledge of the problem that we're trying to solve. Sometimes one person will be accomplished in all three domains – say, an experienced compiler writer. On other occasions, different individuals will embody different knowledge: a programmer knows the tools, a manager knows the process and a product expert knows the problem that we're trying to solve.

At the end of the process we have a piece of software that we expect to function without the presence of any of these individuals. The software itself doesn't know anything; even when running on a computer, it has no self-awareness. However, the software does, to a greater or lesser degree, embody knowledge from all those who were part of its creation.

1.7 Authority and Leadership

One question that inevitably pops up when discussing change is: *Do I have the authority to introduce change?*

This book will argue that change and learning are merely different sides of the same coin, in which case we could rephrase the original question as follows: *Do I have the authority to enhance learning?* This is a much less confrontational question and one that it is perhaps easier to answer *Yes*.

A much more difficult question to answer is: *Does having authority make it easier to introduce change and enhance learning?* Before you rush to answer, consider two facts. Firstly, as already noted, knowledge workers don't like being told what to do. So even if you can order someone to do something, you might not get the results that you wanted.

Secondly, people tend to work better when they're doing something that they want to do. Individuals who choose to do something voluntarily are more enthusiastic, and consequently more productive, more likely to do it well and happier overall.

Consequently, even if you do have a position in the organizational hierarchy that allows you to tell others to do something, you might be better off finding an alternative. Rather than exercising authority, it is better to exercise leadership and to work with people's own motivations. The subject of leadership is itself vast and isn't one that I intend to deal with in depth here. Suffice to say, a position of authority doesn't make you a leader: it does, however, confer on you legitimacy.

Legitimacy is important because it allows you to step forward as a leader; it allows you to create the right environment and remove blockages to learning and change. Legitimacy may also allow you to reward those who follow your leadership. We will return to leadership later.

Authority, leadership and legitimacy manifest themselves differently in different environments. This varies from country to country, from company to company and within companies. There's no guarantee that what works on a German factory production line will work in an American office.

Even in environments in which someone does exercise authority and people do what they're told, there's no monopoly on good ideas. Ideas on how to improve the product, the technology or the process can come from anywhere. Managers who rely on authority to get things done risk missing these ideas because individuals won't speak up and put their ideas forward – and even if they do speak up, the manager may not have time to listen.

This is part of the thinking behind the 'flat hierarchy' (something of a contradiction in terms) and 'empowerment' in the workforce. However sceptical we may be about management commitment and motivation for advocating empowerment, it is of itself a valid idea.

In trying to lead learning and change, we need to consider ourselves empowered – an individual who doesn't will find it hard to lead anything. We need to create change not through our own authority or through borrowing someone else's but, rather, through working with those around us who are receptive to new ideas. Not everyone will be receptive to our ideas, but some will. Sometimes it may seem like throwing mud at a wall: some will stick, some will fall off – you can't tell in advance what will stick and what won't.

On occasions, authority can be useful: sometimes it can be useful to stop people doing something, to ensure that someone takes a specific action or to do something quickly in a crisis. Authority isn't a cure, though, and in many cases you'll find that you don't have the authority to take your desired action. The tools of leadership and legitimacy are more useful and can be acquired and exercised wherever you are in the company hierarchy. If you're in a position to exercise authority, use it judiciously. You can order someone to change, but you can't guarantee that they will, and you certainly can't order anyone to learn.

1.8 Practical Theory

''There is nothing so practical as a good theory''

Kurt Lewin (1890–1947), psychologist, inventor of *action research* and change theorist

During the course of this book, we will look at a variety of theories, mostly about learning and change. For a book that tries to have a practical bent, this

might seem unusual. In fact, there are two good reasons to look at theories even when we're trying to be practical.

Firstly, theories allow us to consider and examine the world in ways that are otherwise very difficult. By abstracting away much detail and considering a few key factors, they allow us to look at the issue in hand in a new and potentially revealing way. This provides a grounding for conducting learning and change in practice.

Secondly, we all struggle to understand people and events around us. This understanding then informs our own actions. In order to make sense of the world, we all use our own set of theories. Some of these will be explicit and we will know that we're using a theory; other theories will be implicit and unspoken. By looking at different theories we open our minds to different models of the world: if these models make sense to us, they will inform our actions in the future and change the way in which we act.

Studying theories of learning and change should better prepare us for practising learning and change. Hopefully some of the theories given here will change the way you see the world and might prompt you to discard some of the theories that you're already using. This is the start of the change process.

Terminology

This book draws on a large variety of sources from software development, computing and information technology in general, and from the business world. These sources use different terms for what are essentially the same things. Although sometimes these terms refer to different things, the underlying concept is, from our point of view, the same.

For simplicity, I'm going to consider the terms *Information Technology* (IT), *Information Systems* (IS) and *Information Technology and Communications* (ITC) as synonymous. Some of the authors quoted discuss *Management Systems* (MS) and *Management Information Systems* (IMS). Strictly speaking, the terms refer to subsets of information systems, but the difference isn't important for our purposes.

This book is primarily concerned with the development of software; that is, software development. This is a discipline necessary to all kinds of IT(C) and it is a subset of IT. On the whole, I will use the term *software development* when I am specifically discussing some aspect of the development process and *IT* when I am discussing the wider dimensions.

In addition, I will use the terms *firm*, *company* and *corporation* as synonyms. While these terms usually refer to profit-making entities, for our purposes I include not-for-profit organizations within them.

The word *organization* is a more flexible term that may refer to a large multinational corporation, a division of a large company, a branch office or a single team, depending on the context or your own terms of reference.

Finally, despite my personal dislike for the term *user* – which has too many negative overtones – there's no more suitable term in widespread use to describe the people who make use of our software. The term *customer* can sometimes substitute, but customers and users aren't always the same.

1.9 Begin with Yourself

The primary objective of this book is to give you, the reader, an understanding of how you can help software development teams improve their ability to learn, allow them to change the way in which they work and adopt a more Agile approach to development. During the course of the book, we will look at various theories of learning and change, we will discuss examples of learning and change and we will suggest some actions that you can take to help teams learn and change.

Naturally, this leads to the following questions: *Where do I begin? What do I do first?*

The answer is: *Begin with yourself*. First seek to improve your own learning and understanding of the situation in which you find yourself.

We will return to this theme again and again, because if you can't improve yourself, then you can't improve your team. Conversely, if you can improve yourself, then you're in a better position to help others and act as a role model and mentor.

Rather than wait until you've finished reading this book, I suggest that you start now. As you read the book, think about the ideas and suggestions presented and how they apply to you and your team.

In order to do this, you'll need to take some time to think about this book, your team, your organization and your current environment. You might like to schedule some time during the week when you can do this. You might also like to undertake your thinking with a partner – in which case the thinking becomes a discussion. It isn't essential that your partner also reads this book, but he or she should share your interest.

If you don't have a partner to work with, you can still do this by yourself. Keeping a personal journal, or diary, can be an effective mechanism for ordering and recording your thinking. You could use an online Blog for the same purposes, but if you do be aware that others – including your team-mates – might read your thoughts. You may not have anything to hide, but knowing that your thoughts are private allows you to express yourself in different ways and to speculate. Alternatively, you could try drawing mind-maps, talking to the dog or just taking long thoughtful baths. Whatever you do, try to think!

Try to think about your organization and environment. Do you understand what the organization is trying to achieve? Or what's happening around you? Or why recent decisions have been made?

Hopefully, such thinking will lead you to inquire more deeply into what's happening. To improve your understanding, ask questions of people around you. It might be that what you think is the case isn't, so it's best not to jump to assumptions. Recognize that different people see situations in different ways: there are multiple ways to see things, so there's no single right way to do things.

In trying to understand the world around you, it is probable that you'll find the need to give up some of your current beliefs and understanding of the way in which the organization operates. This is normal – the process of learning also entails the process of *unlearning*. If you aren't challenging what you think you know, then you aren't learning anything.

Taken together, the process of thinking, inquiring, learning, unlearning and understanding is called *reflection*. It simply means taking time out to think about what's happening.

At some points in the book, I will suggest questions that you might like to think about. These are intended to help you relate the material to your organization. Hopefully, this will help improve your understanding and reveal opportunities.

To help others learn and change, you have to begin with yourself. Nobody can tell you what to do; nobody can give you a recipe to improve your team – you have to decide what you want to do and you have to make it happen. This requires thought and understanding.

1.10 The Organization of the Book

By now, I hope you have a good idea of what this book is going to talk about. We will return to several key points:

- In the modern economy, knowledge is key to all business activities; knowledge can give your business competitive advantage and greater profits.
- Software development is a knowledge-based industry and the workers are knowledge workers.
- Knowledge results from learning and acting on that learning, which involves change.
- Without change we can't capitalize on what we learn, and without change we can't continue our learning.
- Agile methods are rooted in organizational learning; in order to become Agile, we must change the way in which we do things – in order to stay Agile and improve further, we must learn.

Figure 1.2 shows graphically the philosophy behind this book, with learning at the heart. Initially, we start by seeding and motivating learning: most good software developers are eager learners. Frustration sets in when barriers are encountered. Many of these barriers come from implicit assumptions and the

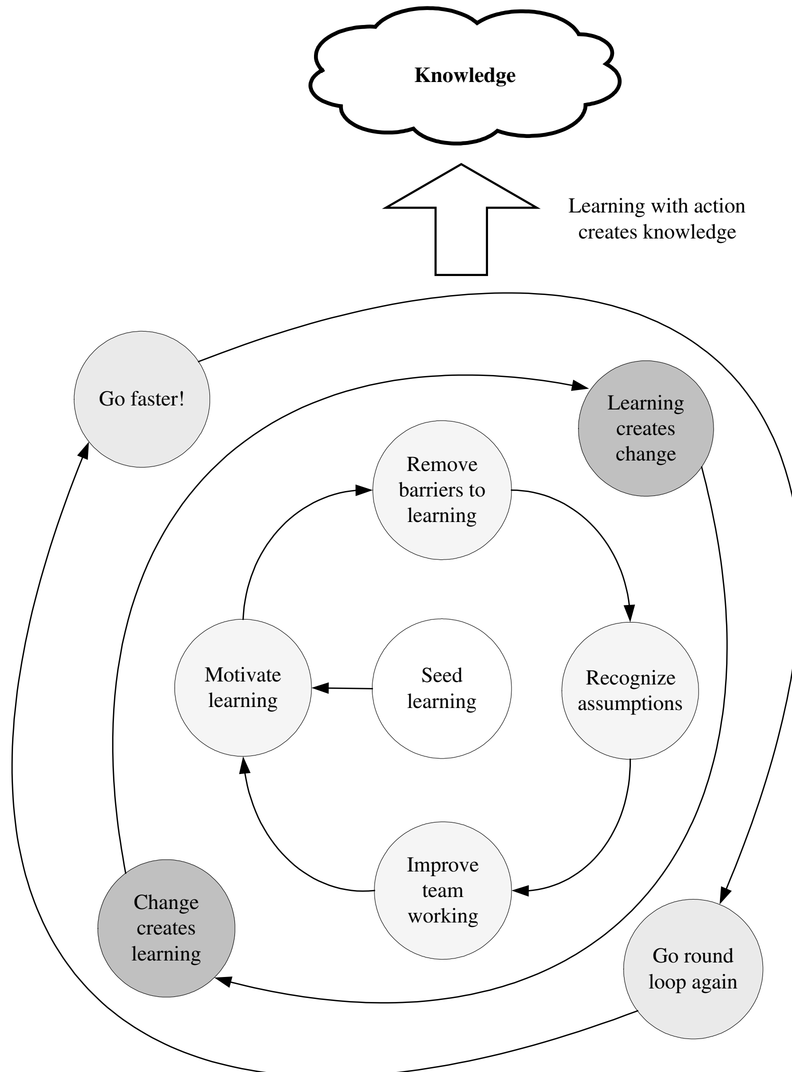

Figure 1.2 The philosophy of the book.

team environment: recognizing these assumptions speeds up and improves the learning process. Active learning leads to and requires change: this change then creates learning, so establishing a virtuous circle of improvement. The alternative is a vicious circle of decay, in which learning without change leads to frustration and delay.

Once we're learning and changing, we need to keep doing it. We can't simply declare our work done and stop. We need to do it again and again, each time getting better and faster. And out of this work knowledge is created.

In the chapters that follow, we will explore these points in more depth and consider what you can do to learn, how you can improve your organizational

learning, how learning can create change and how to manage change to create learning.

We start by looking at Agile software development in Chapter 2. Those of you who are already familiar with Agile may prefer to browse this chapter rather than read it in full. If you are new to the ideas of Agile, you should read the chapter more thoroughly.

The next three chapters look at knowledge and learning in more detail. Those anxious to start doing something soon might want to skip ahead and read Chapter 4, which discusses different types of learning and how we can enhance learning in our organizations. Chapter 5 expands on this to look at learning in organizations, and specifically the ideas of Peter Senge.

Having grounded ourselves in knowledge and learning, in the second half of the book we turn our attention to change specifically. Chapter 6 starts by looking beyond development at the wider picture of business change, and considers how software requirements change and how IT changes the people who use it.

Chapter 7 considers how we can classify change so we can recognize different types of change, and Chapter 8 follows this up with some theories of change. Taken together, these chapters help us to understand the nature of change and why it is difficult.

Chapters 9 and 10 try to pull learning and change together by discussing what action we can take to create learning and change in our organizations.

If you're happy in your understanding of learning and change, then you might want focus on Chapters 3, 8 and 9, where most of the hands-on advice for day-to-day action can be found. Chapters 10 and 11 contain some more involved techniques for promoting improvement.

Finally, Chapter 12 pulls everything together and considers where you can start turning ideas into action.