THE EXPERT'S VOICE® IN ORACLE

Agile Oracle Application Express

CREATE A POWERFUL AND SELF-ORGANIZING TEAM BY APPLYING AGILE PRACTICES TO APPLICATION EXPRESS DEVELOPMENT

Patrick Cimolini and Karen Cannell

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

CHAPTER 1

Agile Software Development

Before Oracle Application Express (APEX) can be discussed in the light of Agile software development, the stage must be set by defining, for the purposes of this book, what is meant by Agile software development.

This chapter introduces you to the core principles of Agile software development. The core principles were developed by the team of leading software developers who created the Agile Alliance in 2011. The core principles are expressed in the *Agile Manifesto,* which is further supported by *The Twelve Principles of Agile Software*; the up-to-date versions of these very short and concise principles can be found at the Agile Alliance web site (www.AgileAlliance.org). These core principles are the common ground that is shared by a number of lightweight software development methodologies. These methodologies grew up and evolved during the latter part of the twentieth century. Some of the common lightweight methodologies are summarized in this chapter because they are useful in the APEX context.

An in-depth discussion of Agile software development is beyond the scope of this book; however, you will leave this chapter with a solid overview of Agile software development. The rest of the book shows how APEX can be configured to directly support the core principles of Agile software development, turning groups of highly skilled and motivated individual developers into effective teams that lead their organizations to technical, strategic, and commercial success.

Agile History

In February 2001, 17 seasoned senior software developers met in Snowbird, Utah. These developers shared a passion for finding better ways of building software and a passion for sharing their thoughts and methodologies with the software development community. The meeting's goal was to find and articulate the common principles, if any, that underpin the various lightweight software development methodologies that the group had been using to manage their software development projects.

The lightweight software development methodologies were conceived, born, and nurtured because the group was frustrated with the classic engineering and project management approaches to software development. The classic approaches often failed, some with spectacularly embarrassing and very public negative results.

The Snowbird meeting produced four significant results:

- The word *Agile*
- The Agile Manifesto
- The Twelve Principles of Agile Software
- The Agile Alliance

The Word Agile

The word Agile was an important result from the Snowbird meeting. It effectively branded the software industry's intuitive movement toward lightweight project governance processes that embrace an iterative approach to discovering what a finished software product should do and look like.

The branding step was highly successful. One Snowbird participant, Alistair Cockburn, observed that an Agile conference was held within six months of the initial Snowbird meeting. Now the word *Agile*, in the context of software development, has become synonymous with the phrase *Agile software development*. Agile refers to the core concepts that are shared by all of the lightweight software development methodologies.

Agile Manifesto

The current version of the Agile Manifesto is (www.AgileAlliance.org) as follows:

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

The Snowbird group, despite the strong commitment of individual members to individual lightweight methodologies, quickly found this common ground.

The Agile Manifesto itself is a shining example of the Agile core principles. It is concise; it says much in a few words. It is lightweight, effective, and sufficient.

The first sentence celebrates the Snowbird group's altruistic commitment to the software industry. The four values distill software development into four core activities. The concluding statement is important because it explicitly states that while the group values a lightweight approach to software development, software development governance can never be zero-weight. The items on the right do have value and play an important part in building software. Without the items on the right, software development falls into the hellish abyss of endless, undisciplined hacking and cowboy coding.

The Twelve Principles of Agile Software

The current version of the Agile Manifesto's Twelve Principles of Agile Software is as follows (www.AgileAlliance.org):

- *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*
- *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*
- *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*
- *Working software is the primary measure of progress.*

- *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*
- *Close, daily co-operation between business people and developers*
- *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*
- *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*
- *Continuous attention to technical excellence and good design enhances agility.*
- *Simplicity—the art of maximizing the amount of work not done—is essential.*
- *The best architectures, requirements, and designs emerge from self-organizing teams.*
- *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*

The Snowbird group felt that the Agile Manifesto required some clarifying points. After a long day, the group agreed on the Twelve Principles of Agile Software. The principles start to give concrete direction to how the Agile Manifesto can be applied.

The Agile Alliance

The Agile Alliance describes itself as follows:

> *. . . a non profit organization with global membership, committed to advancing Agile development principles and practices. We believe that Agile approaches deliver higher value faster, and mak e the so ftware industry more productive, hum ane, and sustainable.*

You can delve into the Agile Alliance's offerings at www.agilealliance.org. The Agile Alliance is a hotbed of inspiration. The web site contains the most recent versions of the Agile Manifesto and its Twelve Principles of Agile Software. The web site acts as an international hub for the Agile community. Membership gives you access to a wide variety of resources that include an article library, videos, presentations, local user group listings, and links to additional Agile resources. The Agile Alliance organizes an annual conference, where you can attend presentations and network with the leading proponents of Agile. The Agile Alliance also provides financial and organizational support to scores of local, regional, and international special interest conferences and user groups. For anyone interested in Agile, it is an ideal starting point.

Agile Software Development Methodologies

While the individuals in the Snowbird group agreed on the Agile Manifesto and its Twelve Principles of Agile Software, they agreed to disagree on what methodologies are appropriate for applying the Manifesto and Principles to work on the shop floor. The disagreement recognizes that a single, one-size-fits-all methodology that is universally applicable in all situations will never be found. This is probably a wise conclusion.

The following sections outline the high-level features of some of the key lightweight methodologies. There are books devoted to each methodology that take great pains to describe their inner workings. This discussion limits itself to only the methodology features that are useful in the context of APEX and its relationship with Agile.

All of the methodologies support the Agile Manifesto and its Twelve Principles of Agile Software. Therefore, it is not surprising to find that there is a great deal of overlap between the methodologies. The methodologies differ in terminology and the emphasis that they put on the various Agile approaches, tools, and techniques.

Adaptive Software Development (ASD)

Adaptive software development (ASD) grew out of rapid application development (RAD) work by Jim Highsmith, one of the Snowbird group, and Sam Bayer.

This methodology gives us a key insight; an explicit recognition that stakeholders do not know everything about a problem at hand and have probably made some false assumptions about the problem. The lack of knowledge and erroneous assumptions are corrected by using an iterative series of speculate, collaborate, and learn cycles. The word *speculate* is used in place of planning; this recognizes the uncertainty involved building an accurate model of a business problem. The word *collaboration* pays homage to the open and frank communication between all of the stakeholders. The word *learn* describes short, time-boxed development iterations that include design, build, and testing. The learn cycle allows the extended team to learn and quickly adapt the solution based on real working software. Thus, the team acquires a deeper understanding of the problem and is able to correct false assumptions and to fix outright mistakes in the original specification.

ASD uses the following terminology to describe its life cycle: mission-focused, feature-based, iterative, time-boxed, risk-driven, and change-tolerant.

Extreme Programming (XP)

Extreme programming (XP) is a software development methodology that takes some software development best practices to extreme levels. Its key features are time-boxing, paired programming, test-driven development, coding features only when they are required, flat management structure, simplicity and clarity in the code, expecting requirement changes, and frequent communication with customers and fellow programmers.

The term extreme is used because the methodology is intolerant of activities that do not produce useful results immediately. For example, adding a column to a table because it will be useful in the future is not done. The future column is not added because it adds cost and complication today with no off-setting benefit. There is also a significant risk that the future column might never be required. The extreme strategy is to add the column only when it is, in fact, required.

Scrum

Scrum is an iterative, incremental framework for project management. It was originally conceived for managing product development and was adopted by the software development industry.

The word scrum, in the context of rugby, describes a play where the entire team interlocks their arms and pushes as a single unit with the aim of getting control of the ball. This analogy of aggressive and cooperative teamwork has attraction for software developers.

The word scrum refers to the daily stand-up meeting where the day's immediate work is planned. The daily scrum meeting is held at the same time and in the same place. It starts on time and ends on time after 15 minutes. Anyone can sit in on the meeting, but only the direct team members speak. The team members briefly summarize what was accomplished on the previous day and what will be

accomplished on the coming day, and raise a flag if they are experiencing a problem. Problem resolution is done after the meeting. This meeting style keeps all members accountable for their work and gets individual problems resolved quickly.

Work is done in time-boxed sprints that last approximately two to four weeks. In this context, the word time-box implies a hard deadline. Working software is always released to the test or acceptance teams exactly on schedule. If some software features run behind schedule, they are removed from the sprint and put back into the backlog list, where they can be included in a later sprint.

Sprints are controlled through three meetings:

- *Planning*: The sprint planning meeting brings the team together with the product owner. The meeting's output is a list of features that will be included in the sprint.
- *Technical review*: A sprint review meeting is held at the end of a sprint. The meeting looks at the work that was completed and the work that was not completed. The completed work is demonstrated to the stakeholders.
- *Process review*: A sprint retrospective meeting is also held at the end of a sprint. This meeting is the team's "lessons learned" exercise. In this meeting, the team reviews the sprint from a process perspective: what worked, what failed, how can the next sprint be improved.

All scrum meetings require strict discipline; they are time-boxed and must stick to the agenda. The central management artifacts that organize scrum work are the product backlog and the sprint backlog.

- *Product backlog*: The product backlog is a list of high-level features that describe the product that is under construction.
- *Sprint backlog*: The sprint backlog is a list of tasks that are required to build the items in the product backlog. During a sprint, individual programmers or programmer pairs check out the sprint backlog items, code them, and then mark them as completed. The check-out process occurs in the context of the daily scrum, where the team members check out sprint backlog items in an open and transparent manner, which fosters a spirit of teamwork and camaraderie.

The scrum's management structure is flat and has a limited number of roles. The scrum master is the team's project manager or team lead. The team consists of up to seven or eight cross-functional team members. The product owner represents the customer stakeholders.

Crystal

Crystal is a family of related methodologies. The families are color-coded as clear, yellow, orange, and red. The colors darken as projects grow in size and more programmers are added to the team; thus more governance artifacts are added as the project size increases. A second dimension deals with the cost of failure. If failure costs are high, then more quality-assurance rigor and formality are added to the development process.

Crystal values people and community over processes and tools. While the processes and tools are important, they exist only to support the people who are the critical success factor in software development. Crystal is also tolerant of different software development cultures. It encourages teams to pick and choose between the various Agile tools and processes in order to select the right mix for the project at hand.

Within the Crystal culture of tolerance, there are, however, two rules that are common to all of the colors: projects must use incremental development with increments of four months or less, and the team must hold reflection workshops before and after each increment.

The author of Crystal, Alistair Cockburn, notes that successful teams are concerned with the properties of a successful project and choose processes and tools that will ensure that their projects acquire these properties. The properties are as follows:

- Frequent delivery
- Reflective improvement
- Close/osmotic communication
- Personal safety
- Focus
- Easy access to expert users
- Technical environment with automated tests, configuration management, and frequent integration
- Collaboration across organizational boundaries

Feature-Driven Development (FDD)

Feature-driven development (FDD) is model-centric. FDD focuses on building a high-level model of a problem solution before coding begins. The model consists of a list of features that are visible and important to the client. The features are broken down into pieces that can be coded and delivered within short development cycles, typically two weeks.

Five core activities make up FDD. They are as follows:

- Develop overall model
- Build feature list
- Plan by feature
- Design by feature
- Build by feature

Progress is tracked by milestones and an easy-to-implement method of tracking percentage complete for both individual features and the overall project.

Dynamic System Development Method (DSDM)

The Dynamic System Development Method (DSDM) was born in the United Kingdom in the early 1990s. DSDM is a more mature and disciplined version of rapid application development (RAD). Over the last two decades, DSDM has been guided by the DSDM Consortium. The Consortium has published a number of DSDM revisions that have evolved in parallel with Agile, and DSDM continues to evolve.

DSDM combines formal project management structures with iterative development. Projects are divided into three high-level phases. The second phase is further divided into five stages.

- Phase 1: The pre-project
- Phase 2: The project life cycle

    - Feasibility study
    - Business study
    - Functional model iteration
    - Design and build iteration
    - Implementation
- Phase 3: The post-project

The three phases and the two studies fit nicely into a classic project management view of software development. The modeling, design and build, and implementation stages are divided into iterative efforts that deliver working software on a scheduled basis. The iterative stages are the Agile part of the methodology.

PRAGMATIC PROGRAMMING

You'll sometimes hear the term *pragmatic programming* bandied about as if it referred to a methodology in the same sense as, say, extreme programming. Pragmatic programming is not a formal methodology. Rather, it is a mindset that is used by experienced Agile programmers who assemble individual Agile techniques to build a unique Agile framework for each project.

Unified Processes (UP)

Several lightweight versions of IBM Rational Unified Process (RUP) have achieved traction in the Agile world. They are as follows:

- Agile Unified Process (AUP)
- Essential Unified Process (EssUP)
- Open Unified Process (OpenUP)

Agile Unified Process (AUP) has been described as a methodology that lies somewhere between the formal RUP methodology and the informal extreme programming (XP) methodology.

The Agile UP is based on the following philosophies:

- *Your staff knows what they're doing*: People are not going to read detailed process documentation, but they will want some high-level guidance and/or training from time to time.
- *Simplicity*: Everything is described concisely using a handful of pages, not thousands of them.
- *Agility*: The Agile UP conforms to the values and principles of Agile software development.
- *Focus on high-value activities*: The focus is on the activities that actually count, not every possible thing that could happen to you on a project.

- *Tool independence*: You can use any toolset that you want with the Agile UP. The recommendation is that you use the tools that are best suited for the job, which are often simple tools.
- *Tailoring*: You'll want to tailor the AUP to meet your own needs.

The Agile Unified Process distinguishes between two types of iterations. A development release iteration results in a deployment to the quality assurance and/or demo area. A production release iteration results in a deployment to the production area.

Essential Unified Process (EssUP) is similar to the Agile Unified Process in that it evolved from the IBM Rational Unified Process. EssUP encourages teams to customize their project governance by picking the practices that suit their environment and tailoring them to their culture.

Open Unified Process (OpenUP) takes advantage of the overall structural aspects of RUP. It discards some of the heavier processes and artifacts and streamlines others. The result is a lightweight, iterative methodology that works within a high-level project plan that lays out the complete project life cycle. The actual work is subdivided into time-boxed iterations that span several weeks. The iterations are further broken down into micro-increments that span hours or days.

Agile and Your Team

The Snowbird group agreed that there is no magic methodology that suits all situations at all times. So how does a team pick the right methodology? The answer can be found by asking a few probing questions:

- How much will failure cost?
- How large is your project?
- What culture exists in your shop?

Cost of Failure

What follows is a true story. In a fluid dynamics engineering class, a student asked the professor how many terms of the Bernoulli equation are typically used in the field. The professor turned from the whiteboard and said, "It depends on how much the screw-up is going to cost you."

The Bernoulli equation can be used to calculate the amount of fluid that flows through a pipe. The equation contains many terms and variations that account for the size of the pipe, the roughness of the pipe's walls, the viscosity of the fluid, etc. The professor's point was that when you are on a construction site and a hole starts filling with water, you can just eyeball the situation and then run down to the nearest equipment rental company and get a submersible pump. Your pump size choices are limited to small, medium, large, and extra large. You make a guess, rent one, throw it in the hole, and see what happens. If the water starts going down, you are finished. If not, go back to the rental company and get another pump. In this scenario, you do not even use the Bernoulli equation; there is no formal analysis or planning as the cost of not getting the right size of pump is minimal. In short, this situation can be managed successfully by an Agile methodology that gets a working solution in place quickly and can be quickly changed based on immediate feedback.

On the other hand, if you are asked to deliver a critical amount of fluid to a process on a space shuttle, the situation changes dramatically. Getting the design wrong is simply not acceptable because the cost is horrific in financial terms and, more importantly, loss of life terms. You cannot correct a problem in the space shuttle by fixing it on the fly. An Agile approach in this case is clearly not an option.

Project governance weight and formality must increase in step with the cost of failure.

Project Size

One criticism of Agile is that it does not scale. This might be a fair observation. A daily 15-minute stand-up meeting does not work for a team of several hundred persons. On the other hand, one anecdote claims that a 10-programmer Agile team succeeded in a short time frame where a 26-programmer team that used a heavy project governance failed over a period of several years. Team efficiency must be factored into the conversation when talking about scalability.

Big projects can be subdivided so that smaller teams can apply Agile to their work. This is also a reasonable strategy with the caveat that inter-team communication must be addressed.

Figure 1-1 is a simplistic illustration that shows the relationship between cost of failure and size against the lightness or heaviness of project governance. As the cost of failure increases, the risk is addressed by putting heavier, more formal project management processes in place.

As project size increases, communication becomes more complicated. Again, the problem is addressed by putting heavier, more formal communication processes in place.

The important observation here, from an Agile perspective, is that the quadrant boundaries are not hard. There are gentle gradations between the highs and lows. Most Agile methodologies recommend an incremental approach to increasing the heaviness of project governance.

Agile's approach strives to follow the Goldilocks principle: do not add too much governance, do not add too little governance, but add just the right amount of governance.

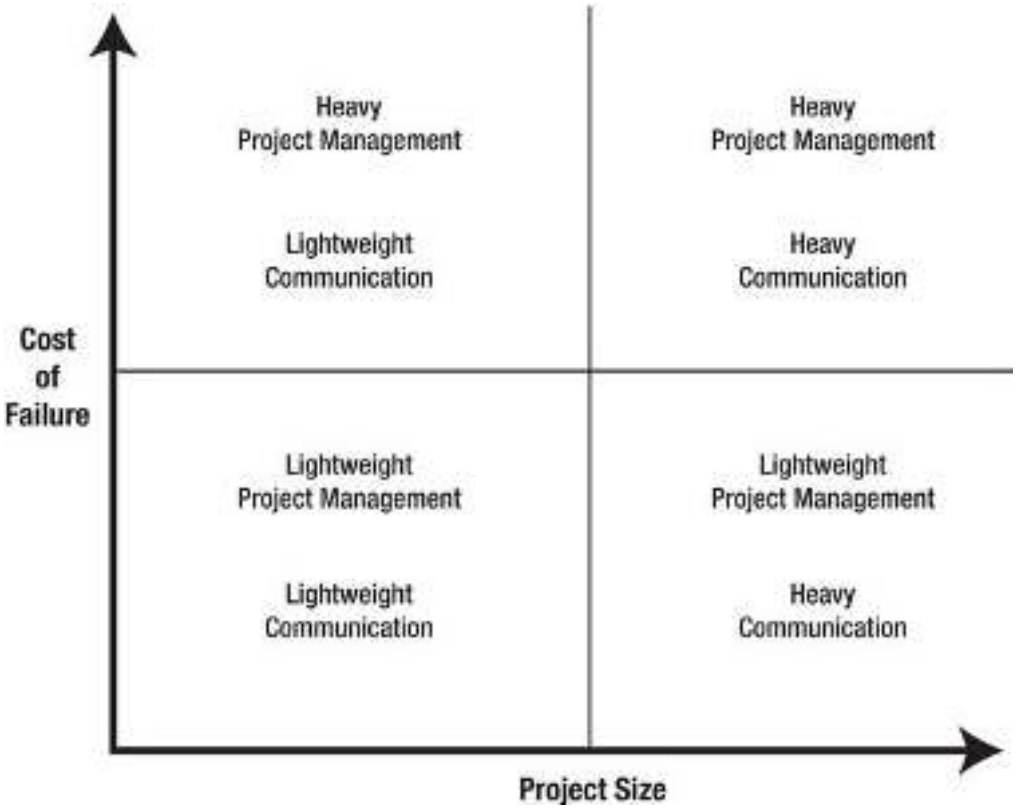

Figure 1-1. *Project governance trade-offs*

Fitting Agile into Your Culture

Is Agile right for your team? How much governance do you apply?

The answer to the first question is usually "yes." Many teams find that an Agile approach to software development is a natural way to work, even within and despite a formal governance setting. Years ago, I worked on a mid-sized project that started out with a good product description. A formal project plan was formulated, and a big Gantt chart was posted on a wall and regularly updated. The project started and working software was delivered to the client on schedule. The client's vision of the product quickly evolved as a result of seeing and working with the software. Together, the client and development teams found better ways to solve some of the business problems. As a result, the project drifted from following the plan and produced a product that was superior to the planned product. However, the formal governance processes did not adapt. Team members booked hours for actual tasks to planned tasks that were never started. The project management reports became quite useless. In the end, the team paid lip service to the formal project management processes while getting the real work done on the shop floor, using our own home-grown Agile methodology that was iterative and time-boxed. Of course, this was long before the word Agile was introduced to our industry. The lesson learned here is that Agile is a natural way to work and feels "right."

If you decide to adopt Agile, how do you go about getting it up and running in your shop? There are two broad options:

- *Adopt a methodology*: A team can adopt an existing Agile methodology that fits their culture. This approach takes advantage of existing documentation, templates, and possibly an active Agile community for support.
- *Build your own methodology*: Study the Agile Manifesto, and then pick Agile processes and techniques from several of the methodologies to produce a custom Agile solution for your situation.

Some of the individual Agile methodologies have been summarized earlier. The following is a sampling of the Agile processes and techniques that can be assembled into an Agile environment that is unique to a team or project. Most of these processes and techniques are shared among the lightweight methodologies; only the terminology and emphasis change.

- Time boxing
- Expecting requirement changes
- Frequent communication with customers and fellow programmers
- Daily stand-up meetings
- Paired programming
- Side-by-side programming
- Flat management structure
- Easy access to expert personnel
- Focus time
- Reflective learning
- Test-driven development

- Code features only when they are required
- Simplicity and clarity in the code

When you adopt Agile, how much governance do you apply to your projects? The answer is never crystal clear. Figure 1-2 helps visualize the approaches.

Zero weight is never an option. All projects, no matter how small, need some governance or they quickly degrade into costly chaos. In my own work on small projects where I am the only programmer, I always work from a written task list that is very terse. It is often written in pencil on a scrap of paper. I do this because of the following:

- Writing the list often uncovers issues that I did not think of.
- The list keeps me on track when I am interrupted in the middle of the work. I check off each task as I complete it so that when I return after an interruption, I pick up exactly where I left off. This keeps me efficient and prevents me from forgetting a step, which causes unnecessary debugging later on.

The main point here is that zero-weight governance has the attraction of zero capital cost; however, its operating cost can be huge.

Lightweight project governance is Agile's sweet spot. As a rule of thumb, your governance should be sufficient and effective. Anything more is a needless cost.

However, there are always valid reasons to adopt a heavyweight governance strategy. Some reasons are as follows:

- High cost of failure
- Large project
- Risk-averse environment
- Existing command and control environment that is entrenched

Where do you draw the line between lightweight and heavyweight project governance? There is no easy answer. The answer will come from open, frank, and sometimes tense communication between the stakeholders.

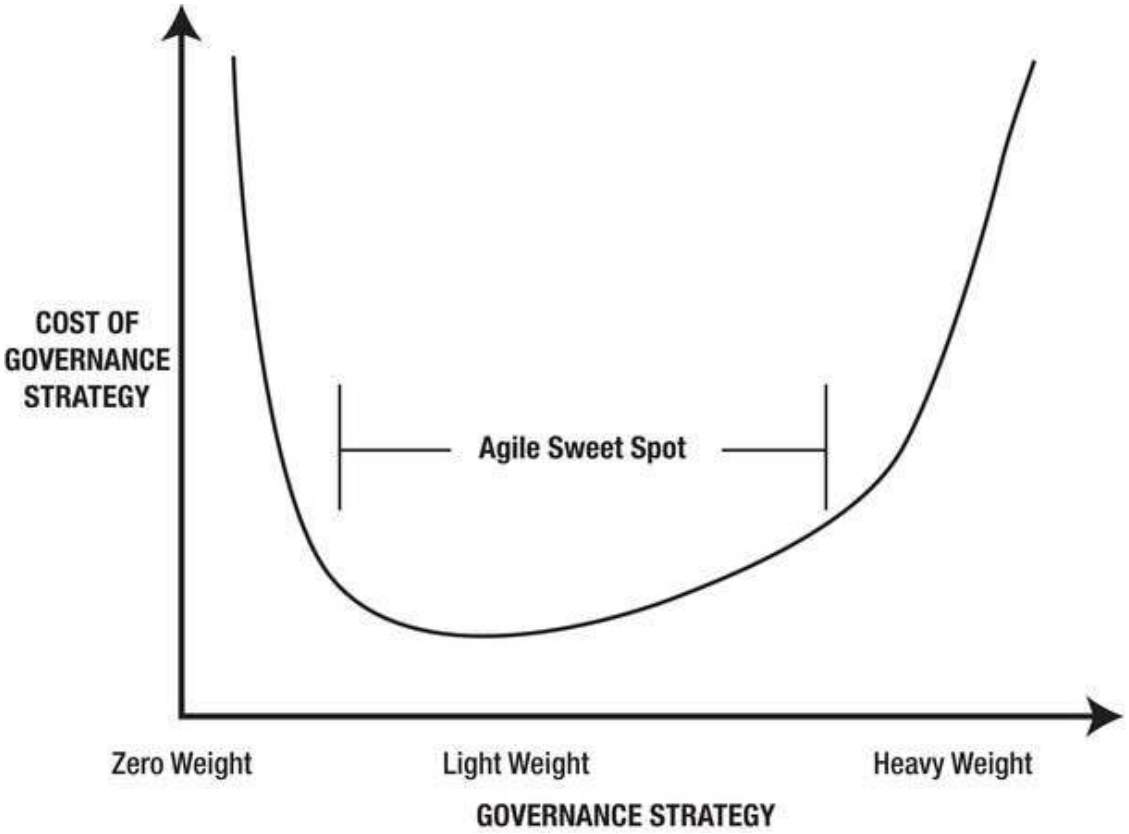

Figure 1-2. *The cost of governance strategies*

Tools, Process, People

Organizing your work environment involves three important areas: tools, process, and people. Figure 1-3 points out that the critical success factor is the people. It is easy to lose sight of this observation in the software development industry. We are constantly bombarded with aggressive advertising and sales calls from software and hardware vendors who claim that their tools will boost our productivity. There are many strident schools and institutes that promote their processes and methodologies as the key to hitting our cost, schedule, and quality targets. In order to work efficiently, people do need tools and process; however, tools and process are useless without good people. In Agile terms, this observation becomes as follows:

That is, while there is value in tools and process, we value people more.

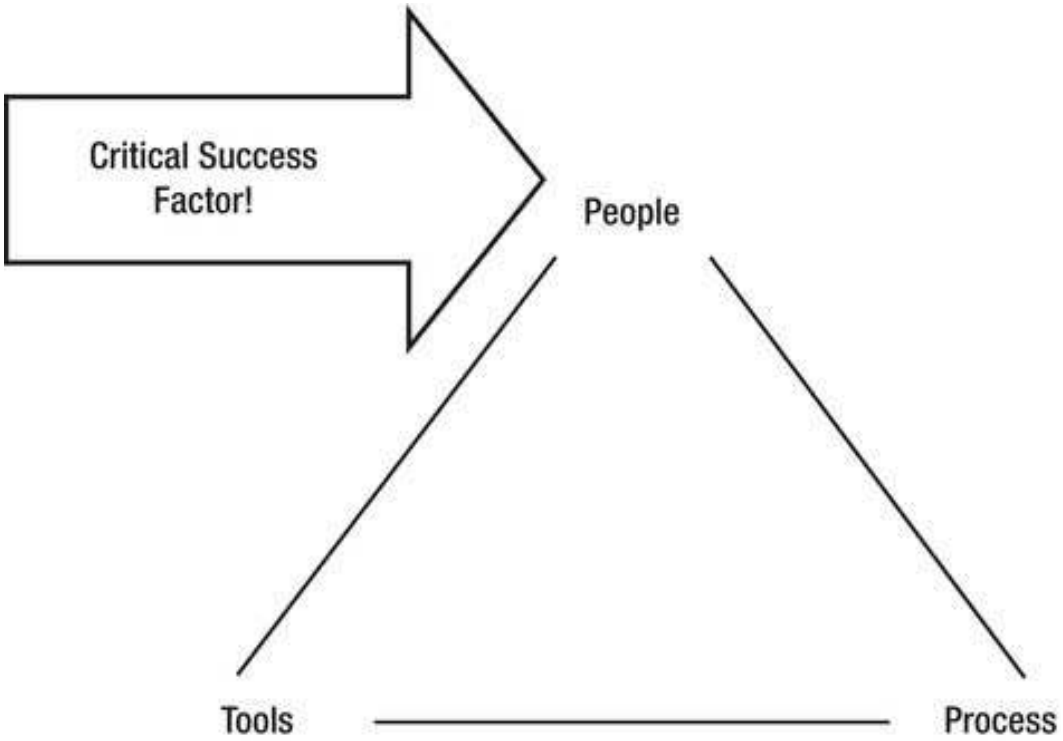

Figure 1-3. *The artifacts that keep your work environment organized*

Summary

This chapter is a lightweight introduction to Agile. The chapter has introduced the Agile Manifesto, its Twelve Principles of Agile Software, individual Agile methodologies, and some of the key Agile processes and techniques. For in-depth insights, go to the Agile Alliance web site at www.AgileAlliance.org or Google the keywords that were introduced earlier.

Is Agile a panacea? No! Software development is an inherently messy activity. No tool, process, or methodology will ever completely sanitize it. Developing software has a lot in common with playing a contact sport like football or rugby. The individual players work hard to develop their game skills. The team management buys the best equipment it can afford. The coaches devise plays and drill the players on play execution. Then, on game day, contact happens, it rains, it snows, the opposition is bigger and faster, plays go well, plays go badly, and the team scrambles to adjust to ever-changing conditions. At the end of the game, the team comes off the field, tired, limping, battered, bruised, bloodied, and laughing about all the fun they had.

CHAPTER 2

Agile and APEX

Delivery of high-quality working software to users on a fast and regular basis is a key goal of Agile software development. Oracle Application Express is a highly efficient rapid application development (RAD) environment. Therefore, it is not surprising that these two entities dovetail together extremely well.

This chapter steps through the Agile Manifesto with its Twelve Principles of Agile Software and highlights the areas where APEX supports Agile software development. The goal of this chapter is to give you insights into how you and your team (the people) can use Agile software development (the process) together with APEX (the tool) in ways that will enable you to reap the incredible commercial benefits that are inherent in these complementary environments.

The Agile Manifesto

The Agile Manifesto is short, simple, to the point, and incredibly wise. APEX can be viewed in a similar fashion. APEX's declarative environment and simple underlying architecture make many routine software development tasks short, simple, to the point, and robust.

Individuals and Interactions Over Processes and Tools

Agile software development is a process. It is ironic, but noteworthy too, that the authors of the Agile Manifesto value individuals and interactions more than their brainchild, the process of Agile software development (see Figure 2.1).

Agile software development supports individuals and interactions by promoting a strategic set of processes that fit well with the natural way in which people work. The Agile processes are lightweight and require a minimum of bookkeeping and bureaucracy. This strategy gives developers more time for producing working software.

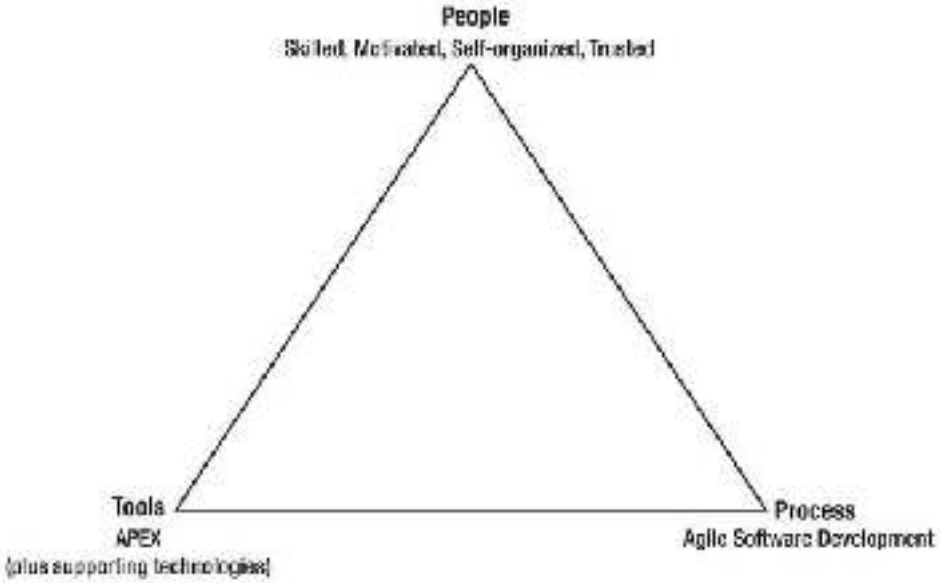

Figure 2-1. Although tools and process have value, we value people more.

APEX supports individuals and interactions by providing a set of tools that are lightweight and supportive of both traditional/virtual workcenter and team collaboration.

APEX wizards are one of the chief tools that support individual developers. These are wizards that help you create most of the artifacts in APEX. Each wizard guides you through a set of pages that contain related properties. In this way, you create an artifact by entering data for the 20% of the properties that do 80% of your work. This method is far more humane and efficient than going to a single page that has all the artifact's properties. The single page with all the properties can be confusing because the important 20% of the items are mixed in with the 80% that are rarely used. It is easy to miss an important property, which means you must return to the artifact later to debug what you missed.

APEX's Team Development module supports team collaboration. Team Development provides a light but rich framework that allows a skilled, motivated, and trusted team to self-organize. The feedback mechanism, features, to-dos, bugs, and milestones are used in concert by the team to more efficiently and effectively communicate among themselves and outside stakeholders. Team Development is designed so that it is relatively easy for developers to keep it up to date in near real-time. Chapter 6 discusses this important APEX module in detail.

Working Software Over Comprehensive Documentation

Software developers produce working software; that is our primary job. Everything else merely supports the primary purpose one must be looked at as overhead. The overhead is always necessary, but it must be ruthlessly minimized and must never, ever become an end in itself.

APEX's declarative environment is the tool's main mechanism for producing working software. Most of the underlying tough coding is taken care of by the APEX engine. Developers rarely, if ever, have to worry about routine things like record inserts, deletes, and updates. The APEX engine does an excellent job of making these database actions safe, quick, and reliable. The safety, quickness, and reliability stem from three sources: the APEX team, the APEX team's processes, and the tool used to create APEX.

Over the years, I have attended numerous conferences where the APEX team has presented technical papers on APEX. The formal presentations together with informal networking have clearly demonstrated that the APEX team has all the right stuff; they are highly skilled and motivated. They are an embodiment of the Agile Manifesto's opening phrase, "We are uncovering better ways of developing software by doing it and helping others do it."

The APEX team used the Team Development module to manage the construction of APEX 4.0 and continues to use it for the following releases. This fact shows that the team has a good handle on Agile software development processes.

■ **Note** In my opinion, one of the main reasons that APEX produces working software safely and reliably is that it is written in PL/SQL, the Oracle database scripting language. PL/SQL is an old technology; it was added as a database feature in Oracle 6, which was released in 1991. It is now over 20 years old and has matured through many cycles of testing, debugging, and aggressive optimization. It is now extremely stable. Upon this rock, APEX was built.

Working software needs documentation. Documentation has value. This is true because no matter how hard software developers work to make the GUI easy to use and intuitive, there are always some parts that need explanation. This is true for both the external interface that is exposed to the end users and the internal code that the developers must understand in order to do maintenance work.

APEX provides hooks that enable developers to document all aspects of their work in the immediate context of the APEX Application Builder without resorting to weighty three-ring binders full of outdated paper. Websheets, team development, APEX utilities, and the help text features can be used in concert to efficiently produce practical and agile documentation within the APEX context. Chapter 8 covers Agile APEX documentation in detail.

Customer Collaboration Over Contract Negotiation

Customer collaboration that is ongoing throughout a product's development is imperative if the product is to be practical and useful. One of the critical success factors of a software project is customer buy-in. Buy-in is almost always the result of positive collaboration with the development team.

APEX fosters customer collaboration through

- Fast delivery of working software
- Feedback

APEX delivers working software to the customer quickly. This enables the customer to start working with the product to test, debug, and evaluate the requirements.

The feedback mechanism that is built into the APEX Team Development module is an ideal collaboration tool. It enables the customer to provide immediate and useful comments to the development team from within the context of the application. The feedback mechanism is embedded into every page in an APEX application. A customer who sees a bug or thinks of a design change can immediately capture the thought in the feedback page. The customer's comment is captured together with all the underlying details, such as the APEX session state, the customer's browser type and version, the customer's computer operating system, and other data that is often invaluable to the developers.

This cuts out the need for longwinded e-mails that require a sluggish back-and-forth set of questions and answers. Chapter 6 comments on the feedback mechanism in detail.

APEX is not a contract-management system; however, the Feature mechanism in APEX's Team Development module can be used to produce a high-level list of features that contains the start date, due date, and estimated effort in hours (see Figure 2-2). The features can be used as an input into the statement of work, which is a key component of the contract. Chapter 6 discusses the feature mechanism in detail.

Figure 2-2. *APEX features can be used in a contract's statement of work.*

Responding to Change Over Following a Plan

Change is a fact of life. We are always living in a state of flux, and software development is no exception. Agile software development deals with change by expecting it and planning for it. It does this by putting working software into the customer's hands as quickly as possible so that the team can iterate, multiple times if required, through these stages:

1. Design
2. Build
3. Evaluate

If you have a good understanding of the requirements and a good design, then why iterate through these steps? There are two primary reasons:

Bugs: Bugs are typically associated with the software; software bugs are fixed to make the software conform to the requirements and design. Bugs can also be associated with the requirements and the design; this type of bug can be scary because it can potentially involve a tremendous amount of rework. Delivering working software early is the only way to uncover requirement and design bugs that have not been identified during reviews of the requirements and design.

Knowledge transfer: During a project, the business users constantly learn more and more about the technology; the programmers constantly learn more and more about the business (see Figure 2-3). Knowledge transfer often causes "ah-hah" moments when someone on the team sees a much better solution to a business or technical problem. Planning for change allows the team to take advantage of these "ah-hah" moments to increase the quality of the end product. Sometimes the good ideas can come very close to the end of the project.

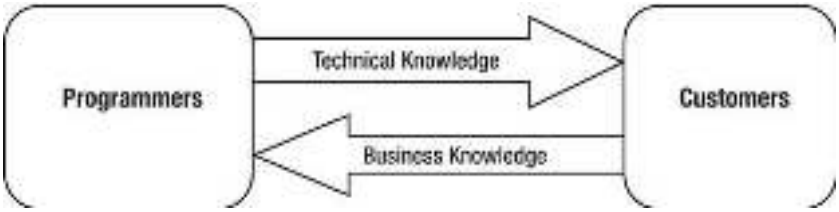

Figure 2-3. *Knowledge transfer continues throughout the entire project.*

How do you plan for change? You start with a high-level overview of the project. Senior resources look at the initial requirements and sketch in the major entities to form the initial design. Time estimates are the product of expert judgement and must account for the confidence that the team has in the requirements. The team estimates the time required to build the initial version of a module, and then a line item for change is added as a contingency. The team must be careful with this line item: programmers must not view it as a chunk of time to clean up sloppy code—adding error traps and global constants, for example. The initial production code must never be sloppy. The line item for change is reserved strictly for refactoring the product based on legitimate and knowledgeable feedback from the customer.

I recently attended a seminar given by a respected scrum master. The scrum master said that the correct answer from a programmer when asked how long an individual task will take is, "I don't know." This is scary when you are estimating the effort needed to complete a project, especially when a fixed-price contract will govern the work. Solving this dilemma requires expert judgment by senior personnel who can look at a high-level design and assign qualitative descriptions such as *easy*, *medium*, and *hard* to each entity. Quantitative units of measures, such as hours, are then assigned to the terms *easy*, *medium*, and *hard* to arrive at the final estimate. The quantitative units of measure are, in turn, affected by the computing environment and the skill levels of the available resources. As a project unfolds, you will come in under budget on some entities and over budget on others. The individual unders and overs usually average out if the initial quantitative estimates are reasonable. In other words, your overall estimate can be reasonably accurate even when estimates of individual line items are not precise.

How does APEX support responding to change and building a plan? First, APEX is an efficient rapid application development (RAD) environment. A RAD environment enables you to quickly build the initial version of the application. This environment also enables you to quickly delete a page and then re-create it. Second, the Team Development module can be used to build your initial plan and then evolve with changes as they occur. Chapter 6 discusses Team Development in detail.

The Twelve Principles of Agile Software

Agile's Twelve Principles of Agile Software serve as stepping-stones between the somewhat abstract core values of the Agile Manifesto and the concrete world of software development methodologies. The Twelve Principles give developers a sense of how they can go about applying the Agile Manifesto's values to the real world without getting into the details of the individual Agile methodologies. This section lists the Twelve Principles and points out the features of APEX that support them.

Remember that APEX's primary function is to build a web-based graphical user interface (GUI) on top of an Oracle Relational Database Management System (RDBMS). The following discussion assumes that the database has been constructed and that it is relatively stable. In most cases, end users are content with a high-level overview of the database design; they generally need this information to help them understand the hows and whys of APEX GUI construction. End users are very interested in how their data is displayed so that they quickly get the information they need and clearly see what they must

do in order to get their work done. The desire to see information instead of data, and the desire to quickly and intuitively step through workflows, drive a GUI through many iterations of

1. Analyze
2. Design
3. Build
4. Evaluate

The iterative cycles, over time, tend to become shorter as the development team learns the business and the business team learns the technology. In baseball terms, the teams start hitting more and more home runs as the game unfolds.

Customer Satisfaction by Rapid Delivery of Useful Software

From a customer's point of view, software development looks somewhat like building a house. The customer first signs off on the architectural drawings. Then construction begins with lots of visible activity like digging the foundation, pouring the foundation, erecting the walls, and putting on the roof. These activities are highly visible and give the customer a concrete sense of progress. However, once the shell is built, it seems to take forever to reach the move-in date. The finishing work like plumbing, wiring, drywall finishing, and painting must all be completed before the customer can move in. The customer feels frustrated because no big and highly visible tasks are being completed. APEX software development is similar to building a house in that building the shell gives the customer a sense of rapid progress. The comparison breaks down during the finishing stage because in an APEX software development environment, the customer can start using parts of the application in production before the entire application is completed. In other words, they can move into the ground floor while the upper floor is still being finished; there is no paint smell or plaster dust to deal with.

APEX delivers useful software rapidly because it delivers a truly RAD environment. A typical delivery plan can include the following artifacts in quick succession:

- High-level navigation shell
- Administration pages
- Features/Modules

APEX's simple architecture is another major reason that working software is delivered quickly. The architecture enables an application's promotion from the development environment to the test and production environments to be automated. Typically, the promotion effort consumes much less than an hour. Application promotion is a low-risk activity that can be quickly done on a daily or weekly basis.

■ **Note** The technical details, tip, tricks, and insights for promoting APEX applications between environments are found in *Pro Oracle Application Express 4* by Tim Fox, John Scott, and Scott Spendolini (Apress, 2011) and *Expert Oracle Application Express* by Dietmar Aust et al. (Apress, 2011).

High-Level Navigation Shell

APEX uses lists and tabs to build the high-level navigation shell that knits the application together (see Figure 2-4). Buttons and report links are used for finer-grained navigation to the lower-level, more-detailed features.

Figure 2-4. *High-level navigation shell*

The high-level navigation shell takes only a few minutes to create, yet it has a great deal of value to the business users when it is delivered early. The navigation shell gives the business users their first hands-on experience with the application. It is the first concrete step in validating the requirements and design.

Business users and developers often have different visions as to how the requirements and design are translated into working software. The navigation shell is a great starting point to make sure the two visions come together in a spirit of collaboration. The shell is easy to change at this point, so the business users quickly get a sense of ownership when their suggestions appear in the application almost immediately.

The high-level navigation shell is the perfect place to refine terminology. Getting the terminology defined in the users' language at this point enables the developers to use the same terminology in the underlying code as the detailed design and construction unfolds. Agreement between the high-level and low-level terminology is one of the key factors that make an application easier to build, maintain, and document.

■ **Note** At this point, I want to give a word of caution to developers. Developers often trivialize features that are easy to code. Business users can easily be intimidated and feel marginalized when a developer responds to a request with comments and body language that indicates the request is trivial to code. Just because a request is easy to code in the programming domain does not mean that it is trivial in the business domain. Respectful collaboration is a major part of Agile.

Administration Pages

Building an application's administration pages is a great second step after the high-level navigation is in place. The functionality in this area includes application support objects. Lookup tables are a good example. APEX's declarative environment enables the developers to rapidly build the maintenance pages for the support objects. Typically, the maintenance pages are based on tabular forms and interactive reports.

Administrative tabular forms most often are built on top of a single table that has a small number of rows and columns (see Figure 2-5). Building a tabular form on a single table typically takes less than five minutes.

Figure 2-5. *Simple tabular form that maintains a lookup table*

Interactive reports can also be used to maintain support objects. Interactive reports are generally used for tables that have a large number of rows and require a search function that enables the user to quickly find an individual row or group of related rows (see Figure 2-6). The interactive report page contains links to a page where a new row is created or an existing row is edited (see Figure 2-7). The create and edit functions are both handled by a single APEX page that was created using the form on a table or view wizard. Building an interactive report together with its create and edit supporting page usually takes less than half an hour, assuming the support table has a small number of columns and no complicated dependencies.

sample content of Agile Oracle Application Express (Expert's Voice in Oracle)