Part I

Production-Ready Software

1

"Production" Readiness

Developing and implementing a software system is a complicated and tricky business. In fact, "developing" and "implementing" are really two different but very interrelated disciplines. For the purposes of this book, "implementing a software system" refers to the activities and processes required to get a software system from an initial concept into live service or production, whereas "developing a software system" refers to the activities and processes of actual software construction and proving. Although the two disciplines are interrelated, they can also be very far apart. Just because a piece of software has been developed doesn't necessarily mean it will be implemented. Many software projects don't even get off the ground or are shelved part way through. This is especially true when the overall project isn't or hasn't been planned, executed, and delivered well. While the development is essential, the implementation is paramount. That said, the project needs to have a sound business case, and the project needs to be firmly planned, executed, and delivered.

This chapter looks at the high-level criteria for production readiness as it relates to both software development and its ultimate implementation. This chapter is organized into the following sections:

- **What Is Production Readiness?** On the one hand, production readiness assesses whether your system meets all the necessary criteria for live service. On the other hand, production readiness also refers to your readiness to produce or manufacture software. This section reviews the high-level activities involved in the system development lifecycle and how they map to some mainstream software development methodologies. Software systems, whether large or small, include a variety of *applications*, *environments*, *processes*, and *tools*, as well as a number of different *users*. The foundation criteria for software *development* and *implementation* are:
  - Applications must be fit for purpose.
  - Environments must be fit for purpose.
  - Processes and tools must be fit for purpose.
  - Users must be trained.

- **Why Is Production Readiness Important?** Some software projects fail or are seen to be a failure. You need to do everything that you can to ensure that your software development and implementation projects are successful. This section discusses some of the most important and common contributors to project failure, including poor scope, poor planning and execution, and poor quality.
- **The Production-Readiness "Process"** — This section builds on the previous ones to provide some foundation principles for software development and implementation, which provide the basis of the production-readiness "process." I've put "process" in quotes because it is not really a formal process; rather, it is a mindset. You need to really think about what you're going to do, how you're going to it, and who you're doing it for.

In this chapter, I don't look at specific technologies or vendors, nor do I go into a huge amount of detail. This chapter provides a high-level overview of the production-readiness landscape and some of the high-level actions to achieve it. At the end of this chapter, I'll recap on what's been covered and as the book progresses, I will cover some of these items in more detail to show how everything fits together to achieve the primary goal — successful software *development* and *implementation*.

What Is Production Readiness?

The term *production readiness* will mean different things to different people. In the world of software systems implementation, the term refers to whether a software system is ready for live service. In its simplest form, this means "Is the system ready for implementation?" It doesn't matter whether you're developing software for external clients, for internal purposes, for general sale, or even for yourself — the question remains the same.

The word *production* also means to produce or manufacture. In achieving production readiness for your system, you need to ensure that you're not only ready for its final implementation, but that you're also ready for everything that leads up to it.

Modern software systems encompass many different applications, environments, processes, and tools, as well a wide variety of users and uses. To fully assess whether a system is ready for go-live or production, you must truly understand the production-readiness criteria. You should ask yourself the following questions:

- Are your applications fit for purpose?
- Are your environments fit for purpose?
- Are your processes and tools fit for purpose?
- Are your users trained and ready?

Production readiness is underpinned by the criteria associated with these questions. However, it is very difficult to answer yes or no to any of these questions without fully understanding their true and entire meaning. For instance, what is meant by "Are your environments fit for purpose?" or "Are your users trained and ready?" Successful software development and implementation depend entirely on how well you *define*, *agree*, and *understand* all the necessary criteria.

When you start development and unit testing, you need a fit-for-purpose development environment. You need a development machine that has all the right applications and tools installed on it. You need to know how to use these properly, and, finally, you need to know what you're doing and how you're going to do it. In short, you've just taken a little step in defining some criteria to assess your own readiness to start producing software.

I've called this chapter "'Production' Readiness" to capture this dual meaning of the word *production* and to highlight the very essence of this book: You must be ready for production and you must assess and ensure your production readiness every step of the way, whether it is *development* or *implementation*. Producing software is your job, irrespective of who you do it for. The activities involved in software development are shown in Figure 1-1.

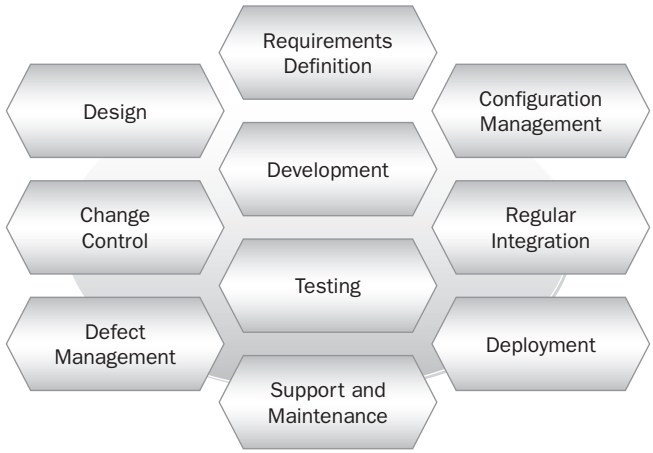

Figure 1-1

How well you *plan*, *execute*, and *deliver* these activities will determine the outcome of your project. The activities will be explained in more detail throughout this book. However, having a set of production-readiness criteria that relates to software development is just as important as a set of criteria for assessing the end result for implementation. The following provides a brief overview of each of the activities shown in the diagram:

- **Requirements definition** — This activity relates to analyzing the problem and defining and solidifying the requirements for the solution. Requirements are often categorized into functional requirements and non-functional requirements. Functional requirements document the solution's features and functions from a usage perspective. They can include business rules, calculations, and other functional and transaction-processing rules. Non-functional requirements document the solution's technical characteristics rather than functional processing.

Non-functional requirements are often referred to as "technical requirements" or the quality characteristics that the system must incorporate.

- **Design** — The design of a software system is typically performed in multiple stages. The design starts out at a high-level and subsequent phases and activities break the information down and provide additional low-level information. Traditionally, software design involves functional specification, technical specification, architecture specification, and so forth.
- **Development** — This is the actual business of "coding" and "building" the solution. In a custom-built application there will be a large proportion of coding, whereas a solution that involves third-party products generally has a mix of coding, configuration, or customization. Third-party products can often have their own proprietary programming language, although some do support extensions that can be developed in mainstream languages and tools, such as C# and Java.
- **Regular integration** — *Continuous integration* is a process whereby developers check-in their code and the very act of checking in triggers the solution to be built and regression tested. This ensures that all the code and artifacts are continually integrated and work together. Best practices have for a long time stated that build and regression testing should be performed at least once a day. The business of integration is about bringing all the components and artifacts together, compiling them, and running the regression tests. This is typically referred to as *software builds*. Although *continuous integration* is a good thing in certain situations, I'm more inclined to use the term "regular integration" to cover all situations. Bringing the artifacts together at regular intervals reduces the number of integration issues and the effort required to fix them.
- **Testing** — The testing that's performed on most software development projects includes unit testing, integration testing, and system testing. Unit testing validates that a single unit of software works correctly. A unit is typically a single class, module or method/function. Integration testing validates that a set of units work together in an integrated fashion. Integration testing can often be referred to as "assembly testing" because the set of units is generally referred to as an "assemblage" or "assembly." This is not to be confused with a .NET Assembly or Dynamic Link Library (DLL), which is also a collection of classes, interfaces, and so forth. System testing validates that the entire solution works correctly. System testing is often subdivided into multiple activities, including functional testing, technical testing, and acceptance testing, where each focuses on particular aspects of the system. Regression testing, mentioned earlier, executes as many of the test scenarios as possible from unit, integration, and system testing. "Smoke" testing is also often used to test the system end-to-end prior to passing the software on for further testing, and, as such, the test scenarios are more aligned to, and a subset of, the formal functional (and possibly technical) test scenarios.
- **Deployment** — This covers a few discrete activities. The first activity, release management, deals with the business of determining what constitutes a release, e.g., the actual contents of the release package. The second activity is packaging the release so that it can be deployed to a particular environment. And the third activity is the actual deployment, installation, configuration, and verification of the software (and content) of a release in a particular environment.

- **Configuration management** — In terms of software development, configuration management refers to the repository that is used to store documentation, source code, and other artifacts relating to the system or solution being implemented.
- **Change control** — As projects progress, changes to functionality and requirements are often identified. Including these changes in the overall scope of the project or release is referred to as *change control*. Incorporating change is typically based on an impact analysis that determines the costs and timescales associated with incorporating the change. These costs and timescales can be reviewed to determine whether there is a sufficiently valid business case for the change. It depends on the chosen development methodology, but traditionally a "change" refers to "a change to the signed-off requirements." Changes that are identified late in the cycle can have a dramatic affect on costs and timescales. Agile development methodologies embrace change, even late in the development cycle. However, changes still need to be validated for their business relevance and their impact in terms of costs and timescales.
- **Defect management** — Defects are identified during testing and review activities and can be raised at any point in the project lifecycle. A defect is not a change because the system (or solution) doesn't do what it is supposed to do according to the requirements and specifications. Defect management is the business of managing defects and their implementation within the system. It is not uncommon for some defects to be debated by the project team because they're actually a change to the requirements, and not really a defect at all.
- **Support and maintenance** — Once a system is in live service, it will need to be supported and maintained effectively. Back-end support concentrates on ensuring that the system is up and running and performing how it should. The system is also backed up and kept in good working order by the support team. Support also involves incident management and investigation when the system fails for some reason or another. Where defects are identified, or additional functionality is required, these will typically be implemented by the application maintenance team, which can sometimes be the support team too. Support, and more importantly maintenance activities, can be quite far-reaching and involve all the activities previously mentioned. Front-end support typically deals with user queries. Part of production readiness is ensuring that the system can be supported and maintained properly.

There are many formal and informal methodologies that can be adopted for software development, ranging from the traditional waterfall-style methods through to agile and spiral development methods. In almost all cases, the methodology will encompass the primary activities shown in Figure 1-1. However, the extent to which the activities are performed is entirely dependent on the methodology.

Figure 1-2 shows a very simple and high-level block plan based on the waterfall development approach. The plan is not complete and the phases do not represent a true scale. The diagram is used simply to highlight the key activities in the project lifecycle and when they are required and/or performed.

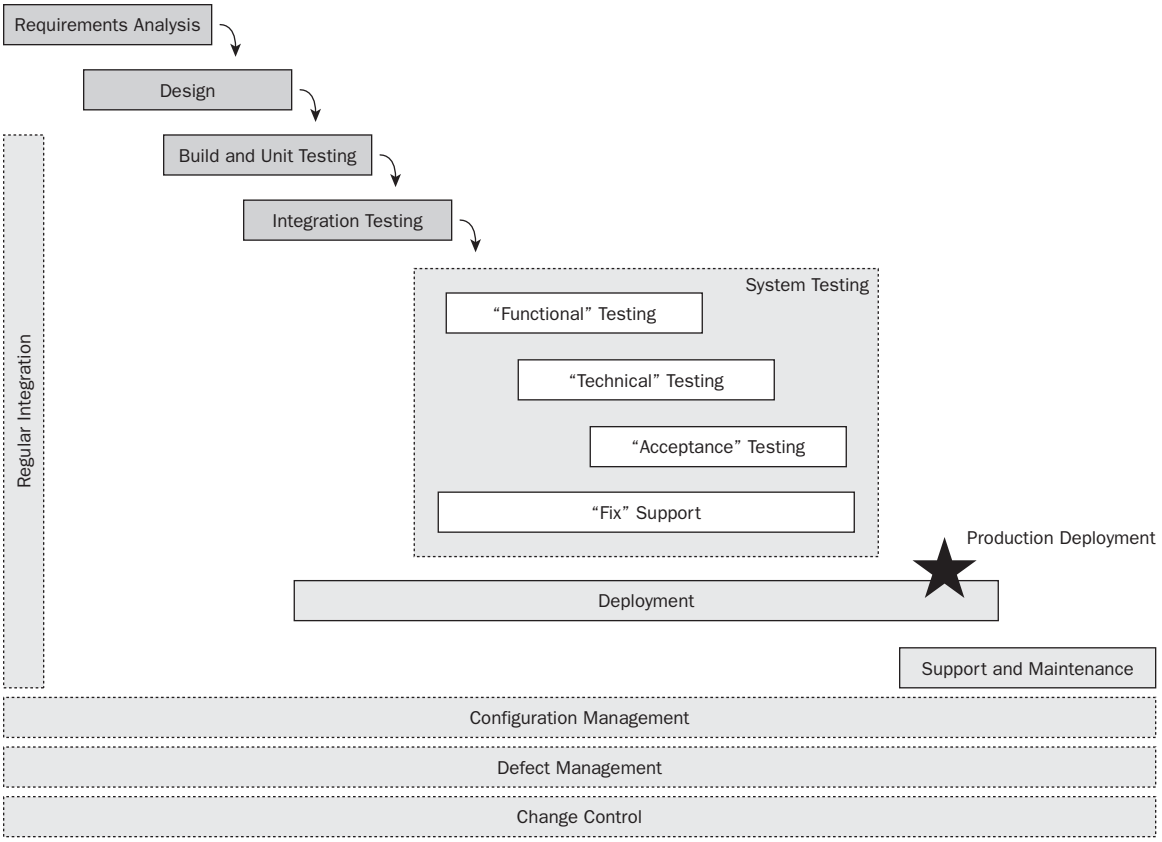

Figure 1-2

The waterfall approach focuses on phase containment. That is, one phase should not begin until the previous one is complete. For example, the development or build phases should not begin until the design phase is complete and signed off. Design should not start until the requirements are agreed and signed off. However, the sample plan shows the activities overlapping — for example, design overlapping with the build and unit test phases. It is very often the case that the development team will start building certain elements of the solution prior to the entire design being complete. For example, framework and utility components can often be started very early in the lifecycle. Functional testing can start when enough of the application is in place. Integration testing can continue while this is ongoing to finalize the design or development of other components. The plan shows functional testing, technical testing, and acceptance testing overlapping with one another, while the development team provides fix support — a very common scenario in the waterfall approach. In the early days of the project lifecycle, one phase drops into the other nicely, or at least it is meant to. It is important that each phase be ready for production (e.g. manufacture), and the waterfall approach provides some time to plan and mobilize for the follow-up phases. For example, during requirements analysis, you can prepare for design; in design, you can prepare for development; and so on.

The waterfall approach can also be used with iterative development, whereby the system is developed over a number of "iterations" or "releases." Each release delivers a certain amount of functionality that realizes business benefits and can be implemented in live service while future releases are under construction. Each release would typically encompass all the activities previously highlighted. There

could be many releases being developed in parallel (as shown in Figure 1-3), which can be very difficult to manage.

Figure 1-3

The release strategy is not associated with the methodology; it is associated with the project and the business priorities. Multiple releases all running in parallel have an impact on how you mobilize your projects and should be considered early. You'll see throughout this book how multiple releases affect some of the activities that you perform.

Although the high-level plan shown in Figure 1-2 would look somewhat different for an "agile" approach, the activities performed are again similar to those shown in Figure 1-1. A sample construction iteration outline is shown in Figure 1-4. The outline does not follow any specific agile development methodology. It is simply used to demonstrate the similarity in the activities and tasks that are performed during the project lifecycle.

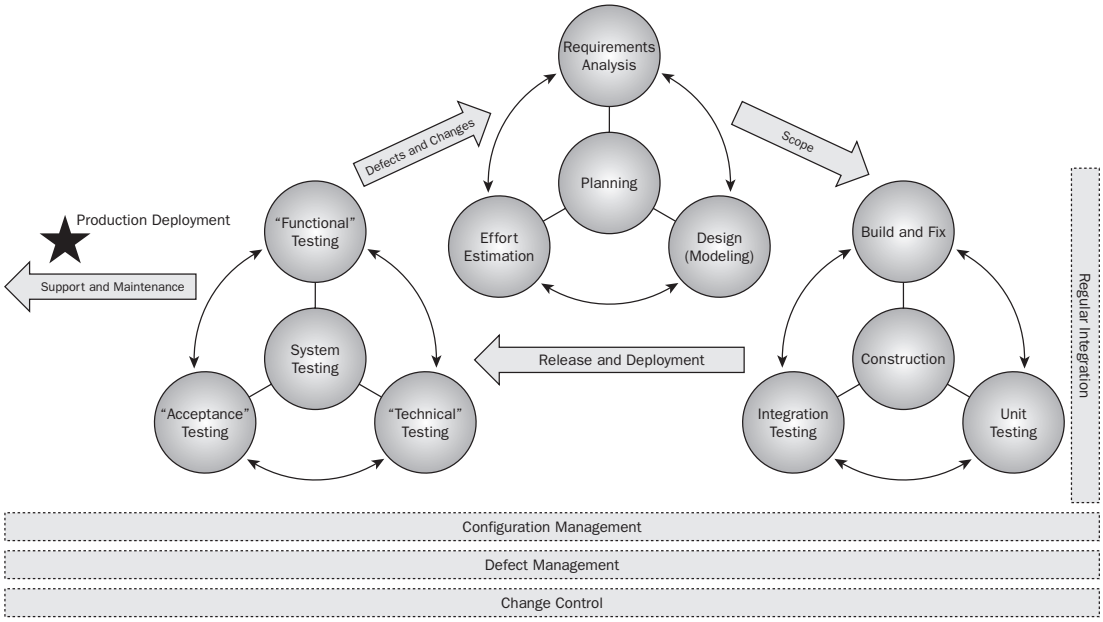

Figure 1-4

Agile software development methodologies focus primarily on developing working software rather than writing documentation (specifications and so forth). The software is developed over multiple iterations. Each of the iterations lasts for a fairly short period of time, usually somewhere between 4 to 8 weeks, and produces a working version of the system. An agile iteration would normally include requirements gathering and prioritization, as well as estimating and planning. A certain amount of design and modeling is also required. The code needs to be developed, tested, and fixed where defects are identified. The output at the end of the iteration doesn't necessarily contain all the features and functions required for live service. However, there's usually a discussion by the stakeholders, as to whether the functionality would realize true business benefits. If so, the project team has further discussions and plans meetings around what needs to be implemented to ensure the software is production-ready and can be deployed into live service. The necessary production readiness activities and requirements are then prioritized for the next iteration. Given that agile development methods focus on developing working software at (almost) every stage, it is even more important that all the applications, environments, processes, tools, and users be ready.

The plans have shown that in both methodologies the activities pretty much remain the same. The degree and quality to which each of the activities is performed ultimately determine the quality and readiness of the outputs. Deciding where to set the quality bar depends entirely on the budget and time constraints of the project, as discussed later. If the right processes and practices are in place, tuned, and understood, then all development can follow the same practice and provide the same level of quality even during fix and later application maintenance.

The applications, environments, processes, and tools that underpin all these activities must be fit for purpose. The development methodology, along with the activities it encompasses, is simply a component of the overall scope of the project. As you can see from the figures, the development methodology ultimately plays its part in the overall *preparation*, *execution*, and *delivery* of the project.

This chapter, and this book, promote and examine production readiness as it relates to both the actual *development* of a software system and its ultimate *implementation*. It focuses primarily on the activities involved in the project and what developers, architects, and team leaders can do to help ensure a successful outcome. To achieve your goals for production readiness, you need to ensure that:

- ❑ Your applications are fit for purpose.
- ❑ Your environments are fit for purpose.
- ❑ Your processes and tools are fit for purpose.
- ❑ Your users are trained.

"Fit for purpose" doesn't necessarily mean best of breed. It simply means that everything must be fit for the purpose its intended for. If you were working on next-generation software, it is quite likely that you'd require the best of breed as well.

Applications Must Be Fit for Purpose

A software system doesn't normally just involve a single application — for example, the one you're producing. Many other applications are usually involved. Some will be custom built and others will be off-the-shelf and configured or customized. It is important that all of these be fit for purpose. The system is really only as strong as its weakest links. It doesn't help to say "It's the database. It just keeps falling over." Or "The logs don't actually tell me anything . . . . " All of your applications need to be production-ready and

meet all the necessary quality characteristics (which are discussed in the next chapter). Your system is usually built from many custom and third-party applications. This is referred to as the "solution stack." Which applications are included in the final state solution stack will depend entirely on the size and scale of the system being implemented. Figure 1-5 shows a sample final state solution stack for the purposes of this book.

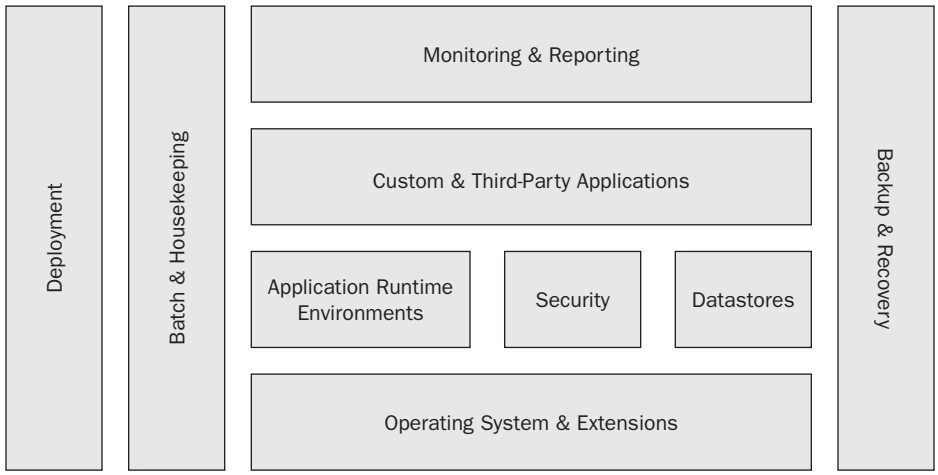

Figure 1-5

You need to ensure that all the chosen applications, as well as the ones you develop, display the necessary quality characteristics. The systems that you implement today will undoubtedly contain the ones you develop yourself. However, they also include many other applications, as described here:

- **The operating system and extensions** — This is pretty obvious but it is worth mentioning as everything else sits on top of the operating system and, if it is inherently flawed in some way, this could impact your system. You must determine and understand which aspects and features of the operating system you are using so that you can ensure they are properly tested and evaluated for production readiness. The requirements of the system and the applications need to be mapped to the features of the operating system and extensions. For example, a system may provide failover capabilities and in a Windows environment, some of these could make use of Windows Clustering Services. In this case, you need to ensure they meet the needs of the system. For instance, you might have a requirement whereby the system needs to failover in 20 seconds. If the underlying services do not support this requirement, you're going to have a problem. Understanding all the applications you have in the stack will enable you to determine which ones can be clustered properly. Some third-party applications might not support clustering at all and, as such, alternative failure and recovery solutions need to be put in place to ensure that they are fit for purpose. Furthermore, knowing which specific features of the operating system your application is going to use enables you to better develop your application. For example, knowing which application components will run as services, clustered applications, or a combination of both will enable you to determine what you can and should incorporate during development and testing.
- **The Application Runtime Environment and extensions** — This is again pretty obvious, but whichever runtime environments your applications are going to run within must also be fit for

purpose. You need to understand the requirements of your applications to determine that the appropriate runtime environment is in place and is tested appropriately. Third-party applications often have specific system requirements that stipulate specific runtime versions to be used. For example, a third-party Customer Relationship Management (CRM) system might stipulate that the .NET Framework version 2.0 should be used. Understanding these requirements will ensure that you have compatible runtimes in place. If your custom applications are built to run on the .NET Framework version 3.5, you might have a problem running both applications on the same machine. Trying to run applications on an inappropriate runtime can cause issues and delays. Choosing applications that can all run on the same version of a runtime is often the best way forward. Understanding the runtime environment enables you to determine what you can incorporate during development. For instance, knowing which application components will run within Internet Information Services (IIS) will help you to determine what you can and should incorporate during development and testing. Furthermore, knowing the exact version of the runtime will enable you to avoid using incompatible, deprecated, or even unsupported features during development, which can also cause issues further down the line.

- **The data stores —** Again, this sounds pretty obvious, but the data stores (or databases) will be used by many of the applications and tools you put in place. The database needs to support all the requirements and needs of the applications and tools that use it. You need to map the requirements of the applications to the features of the database. Applications that all use the same database engine can help to reduce costs because you do not have to deal with multiple database technologies, although this is sometimes not possible. Knowing the features of the database that you are going to use will help you to determine what you can and should do during development and testing. For instance, the system might utilize multiple databases to improve performance and separate functionality. All these databases will require some form of housekeeping, purging of old data, re-indexing, and perhaps even recompiling stored procedures for optimal performance. Understanding the types of databases and engines that you are going to use will help you to better design, build, and run your system.

- **The security and encryption solutions** — Some systems are often required to use an external security or authentication system — for example, using Microsoft's Active Directory to store user- and role-based information. These applications will again have their own requirements and considerations for design, development, and implementation. It is important that all of these applications be production-ready and scaled appropriately for the entire system. The project may also need to use an isolated encryption server or service. Hardware encryption/decryption is quite common in large-scale secure systems, and it will also have its own considerations and usage scenarios. It is important to understand these external systems because, again, they need to be scaled and used appropriately. A single instance could cause a bottleneck in the final solution, and finding this out toward the go-live date could be a very costly business to rectify.

- **The batch solution** — Even in these days of 24/7 availability and service-orientated architectures (SOA), batch can still form a large part of an overall solution. Batch is traditionally thought of as an overnight process, with a lot of number crunching and data processing — and this is still the case in a lot of systems. However, batch covers many different types of jobs. Batch jobs and processes can be as fundamental as clearing out old data or taking the system down in

readiness for routine housekeeping. It is important to ensure that the batch solution is fit for purpose and meets all the necessary quality characteristics. The "batch window" (the time in which all batch jobs must be completed) is ever decreasing due to the high availability demands of today's systems. In some situations, the batch window is almost non-existent and the majority of the system must remain up and running while batch or routine maintenance is performed. You need to ensure that your batch solution takes the batch window into account. Understanding the batch solution and its features will help you to develop better batch jobs and frameworks within your applications. Knowing which application components will run as batch or background processes will also determine what you can and should do during design, development, and testing.

- **The reporting and analytics solutions** — I've often said, "It's not what you put into a system, it's what you get out of it." The truth is that the two are related. "Rubbish in, rubbish out" is an old saying. The reporting solution and the resultant reports also need to be fit for purpose. The information in the reports is used for many different purposes by many different people. Some reports can be used to make financial or investment decisions; others might be used to assess performance; and some can be used during incident investigation. The reporting solution needs to be understood so that you can determine what you can and should do during design, development, and testing. For instance, reports can often be executed against a replicated database and a report can be as simple as running a SQL query. There are also third-party reporting applications, such as SQL Server Reporting Services which allow you to design, build and execute custom reports. These reporting applications require specific knowledge, not only on how to develop reports but also how to implement the overall reporting solution. Knowing which reports you need to produce and how and where they will run will help you to better design, build, and test your application. There are also many applications that provide web analytics. These applications very often analyze web logs and provide usage and trend analysis reports. The use of these applications also introduces the need for additional development and training. The applications will have their own hardware and software requirements, as well as various deployment considerations. Additional batch jobs or scripts might be required to copy the log files from various servers to a central location for processing, analyzing, and reporting.

- **The monitoring solution** — All your applications and infrastructure will require monitoring. You need to know when something goes wrong in order to investigate and correct it. You may also need to know when something good happens in the system. The monitoring solution needs to be fit for purpose. If the monitoring solution is unstable or ineffective, you risk entering a black hole. You don't know what the application is doing because you don't know what the monitoring solution is doing. Monitoring is often divided into different levels and can involve different monitoring applications. Understanding the various applications involved will help you to determine what you can and should do during development and testing. For instance, most monitoring solutions are capable of extracting information from the Windows Event Log, WMI (Windows Management Instrumentation) events, and performance counters, which you can use to provide better monitoring and instrumentation capabilities in your applications.

- **The backup (and recovery) solution** — A number of artifacts go along with the applications you put in place, each artifact having its own backup requirements. Understanding what needs to be backed up and when is important. Your backup strategy needs to be fit for purpose. You saw earlier that the batch window is ever decreasing, and backups are generally included in this window. You need to ensure that you back up only what you have to. For instance, if you lose a server, you can generally re-image it or a new one (apply the base operating system, applications

and configuration) much faster than restoring from a backup. Furthermore, most companies send backup tapes off-site at regular intervals. You may need to take this into account, as the time it takes to get a backup restored could also cover the time it takes to get the tape back. Understanding the recovery strategy also helps to better design the application for recoverability.

- **The deployment solution** — You have many applications that need to be deployed to your different environments. Your deployment solution needs to support all these requirements and needs. If you can't get the software out in the environment, you're not going to be able to run it. Understanding the deployment requirements and applications will help you to better design, build, and test your applications. Automated deployment tools still require configuration and customization, and knowing what needs to be installed and where will help you to better design, build, and test the solution.

There are usually many other applications apart from those listed, and these include applications such as anti-virus software that will also need to be production-ready. However, I am not going to cover all of these in this book. The key point is that all the applications and solutions (including your own) need to be production-ready and display all the necessary quality characteristics in every environment that they are going to be used in.

I've chosen to highlight the preceding areas because it is in these areas that we as software designers and developers can have a major impact, both positively and negatively, during the project lifecycle. Understanding the purpose of these applications, what they are used for, and how they integrate with the system is a key factor which will improve the quality and success of your software. For example, it is very important to understand all the features that are being used in the applications and that they are fully documented (and supported) by the appropriate vendor. It is not uncommon to use *undocumented features* directly or indirectly, which can cause issues for the entire project further down the line. It can lead to applications not being supported. It can lead to upgrade and maintenance issues, as well as different behaviors in testing and production environments.

Environments Must Be Fit for Purpose

Applications don't usually jump straight from the development environment into the production environment without going through other environments and additional testing. Throughout the project lifecycle there are a number of different environments used along the way. Production or live service environments are generally not a single environment either; they can often involve a disaster recovery counterpart and a pre-production counterpart. Every environment you design and use must be fit for purpose and meet the necessary quality characteristics. It is important that each environment is scaled and configured appropriately for its use. In some cases, it is not possible to test certain features of the system without having access to an environment that is sized and scaled appropriately and contains all the required features and functionality. For instance, testing the clustering failure and recovery scenarios requires an environment with multiple clustered servers.

Your environments all have their own requirements and specific uses. Depending on the size and scale of the project, the number of environments will vary and some environments may be shared and used for multiple purposes, which also introduces its own set of considerations. Figure 1-6 shows just a few very basic and common environment configurations.

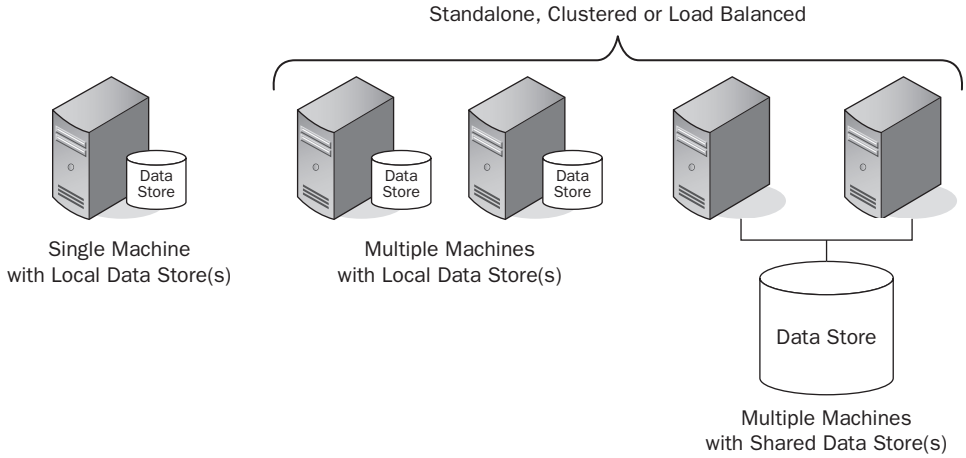

Figure 1-6

Whether an environment is a single machine or multiple machines; whether the servers are standalone, clustered, or load balanced; and whether they have local or shared data stores is entirely dependent on what the environment is being used for. However, the environment configuration will typically bring its own considerations for management, usage, and maintenance. The development methodology will generally dictate the various activities that are performed during the project lifecycle, and these activities will require an environment in which they can be performed (including all the required access, applications, and other required resources). It is possible that some of these activities will be performed in the same environment, which may also need to be considered. The following lists some of the environments required to perform the preceding activities:

- **Design environment** — The design environment is used to produce the system and solution designs. It is an environment that is often taken for granted; however, the design environment needs to be fit for purpose and sized and scaled appropriately. Modern software design tools are becoming more and more prevalent and require increasing amounts of memory. The size and scale of the design environments will depend entirely on the design tools and technologies being used. In the race for better quality and reduced timescales, software design tools often produce a lot of code and templates, which can help to reduce the development timescales. However, this would also need to be considered and assessed. It is possible that the code generated by the tool doesn't meet the coding standards of the project. It could be configurable, but then again, it could also require a large amount of rework following its generation. In addition, when a machine doesn't have enough memory or when it is not fast enough to support the design activities, the effects can ripple through the project, affecting development as well as hindering the design activities. There may be multiple design applications, each of which has its own requirements and constraints that need to be considered when defining the design environment.
- **Development environment** — The development environment is primarily used to write and fix your code, write reports and batch jobs, configure applications, and perform unit testing and integration testing. It is the environment that you use on a day-to-day basis to get your job done. If this environment is not fit for purpose, you really don't have a chance of developing anything. That is not to say that it has to be the most powerful environment available. It needs to be sized and scaled appropriately. The tools and technologies that are being used determine the

development environment requirements. Development machines are often multi-purpose and this needs to be taken into account. For instance, some development machines may be used to develop architecture and application components; some machines may be used to develop batch components or reports; and still other machines may be used to develop packaging and deployment solutions. The development environment needs to support the requirements of its usage and the tools that will be used in it.

- **Regular integration environment** — The regular integration and build environment needs to be fit for purpose; otherwise, you're not going to be able to build a complete software release. The build process usually involves compiling everything from the bottom up and includes compiling the tools and tests you've developed as well. Understanding the build process often helps during design and development to ensure that components are placed in the correct libraries or assemblies, as this can affect the way the solution is built, packaged, and deployed. For example, you could have a code generator that generates data access objects from the database. If this is built correctly, it should really use common low-level framework components for logging, tracing, and instrumentation. This really means that the common low-level framework components need to be built before the code generator. But, if these common low-level components rely on some generated database access objects, a cyclic dependency between the components results, which can lead to trouble. The number of applications that need to be built and the dependencies can sometimes require more than one machine to be used to build the various components of the solution. This can often reduce the time it takes to build a complete release. Where this book refers to the *build environment*, it is also essentially referring to the regular integration environment. Builds and releases can be executed on a daily or weekly basis, depending on the project's needs. They can also be taken more frequently, if required. Committing artifacts, building, and regression testing regularly reduce the number of failed or broken builds and issues. There are many practices for the regular integration "process," but they typically stipulate the following three principles:
  - **Maintaining a source repository and checking in code and artifacts at regular intervals** — Typically, the latest code is taken into the build, although other techniques and labeling can also be employed to mark files as "ready to build." The source code repository stores all the source code and related artifacts.
  - **Automating the build** — The application build shouldn't really require manual intervention or processes, unless, of course, it fails for some reason. There are a lot of tools to help with automating builds and releases. However, these tools often need configuration and often require custom scripting and development, which needs to be considered.
  - **Automatically testing the release** — The built software should be tested, and this testing should be automated to reduce manual effort. This is referred to as *regression testing*.
- **Configuration management environment** — Although this environment is typically accessed from other environments, I've listed it separately because it is a very important environment and one that does need to be fit for purpose and meet all the necessary quality characteristics. If this environment isn't backed up, there's a possibility of losing all the source code and other project artifacts. If it is unstable or connectivity to it is slow, it could cause delays to the project by increasing down time.
- **Regression test environment** — The regression test environment is used to test the application to a *suitable* degree. However, it could be the same environment as the regular integration environment. I much prefer to use a "clean machine" because it ensures that nothing is lingering on the build machine that's not included in the release. It is not always possible to test

everything in the regression test environment, so it is good to have an understanding of what the limitations are. Ideally, as much as possible will be tested in this environment, so it needs to be sized and scaled accordingly. The number of tests that need to be run and the time it takes end-to-end will determine whether more than one machine needs to be used to execute different tests in parallel. Wherever possible, all regression testing should be automated to reduce manual effort, and the regression test data should be aligned to that of a functional test and/or technical test to avoid unnecessary issues in other environments or later testing.

- **Test environments** — Test environments are where the application will be put through its paces, functionally and technically. You need to ensure that these are fit for purpose and scaled appropriately. Understanding the different test environments will help you to understand your deployment requirements and your test tool requirements. Each test has its own purpose, and as such the applications and environment may be configured in different ways to support these requirements. The size and scale of the project, as well as the methodology, will typically dictate the number of test activities or phases and the environments required to perform them in. Some test environments need to compress multiple days of testing into one day to save time and effort. For instance, lifecycle testing usually involves testing Day 0 (the day prior to go-live), Day 1 (the day of go-live), Day 2 (the day after), and so on. For example, a transaction entered on Day 1 that is not processed (for any reason) may need to be picked up and processed on Day 2. Another example is data expiration whereby a record has a specific expiry date or elapsed time. The application needs to be able to support this level of testing and it can sometimes be as simple as allowing the application to have a specific date/time configuration rather than relying on the system clock or, even worse, having to test in "real time."

- **Training environments** — The training environments are often forgotten but provide an invaluable service. Training environments can sometimes be referred to as *sandboxes* — environments where users can experiment using the applications and try out features and functions. The training environments can also help to pull together the training documentation and other associated artifacts. It is important to understand the training requirements and the potential training environment requirements. A training environment can sometimes be as simple as a single desktop machine, although it could also be a full-scale, production-like environment.

- **Production environments** — The production environments are ultimately where the system belongs. There may be disaster recovery environments and pre-production environments that can and often are used in the event of failures in the live environment. The pre-production environment can be thought of as a separate test environment because of its usage, although I prefer to include it as a production environment because it is typically more controlled and can be used as a production fall-back in certain situations. You need to ensure that the production environments are fit for purpose. The production environments will generally have their own configuration settings, and sometimes the first time you get to test something is in a production environment. For instance, usernames and credentials will often be different across the production environments. The access rights of these users and credentials need to be clearly stated to avoid unnecessary issues after deployment.

- **Operations and support environments** — Although these can be considered part of the production environments, I've listed them separately because they, too, must be fit for purpose. A system that is monitoring another system needs to be fit for purpose. It is no good if the operations and support environments don't meet all the necessary quality criteria. If the monitoring environment isn't robust, you won't be able to tell whether your own system is up and running. If the batch environment isn't robust, you won't be able to perform routine maintenance of your applications and environments.

- **Application maintenance environments** — The application maintenance environments are actually the same as the development environments. I've again listed these separately because they must be fit for purpose. Once a system has been handed over to live support and application maintenance, changes will probably need to be made and the environment must be fit for purpose and support all the necessary activities for development, testing, and deployment.

There may be other environments that have specific requirements and uses; however, most environments fit into one of the preceding categories. The key message is that all of your environments need to be understood, sized, and scaled appropriately, and display the required quality characteristics to ensure that you can smoothly progress through the project lifecycle. In addition, not all of them need to be on separate hardware. There are many situations where the same physical hardware is used for multiple activities, which also has its own considerations.

Processes and Tools Must Be Fit for Purpose

Throughout the project lifecycle there are a number of processes that need to be followed and a number of tools that are associated with them. Each process has its own purpose and often there are tools associated with them to enhance productivity. If the tools don't work or do not display the same quality characteristics of a production-ready solution, they can negatively impact the ability to perform a given task. In the same way, the processes need to be robust and streamlined to avoid unnecessary work or re-work. Figure 1-1 showed a number of different activities and processes that will be performed on a typical software development project. Each process will typically involve the use of a number of different tools and technologies. The following processes and tools must be fit for purpose:

- **Requirements analysis and tracking processes and tools** — You need to capture your requirements and constraints, and they need to be documented, agreed on, and accessible. You need to ensure that the requirements also capture the relevant quality characteristics to avoid issues later on. The appropriate processes and tools need to be in place to ensure that the solution requirements are tracked and implemented accordingly. It should be possible to trace your requirements through all the project artifacts to ensure that all have been captured and implemented accordingly.
- **Design processes and tools** — The design process involves ensuring that your requirements are validated and that your designs meet those requirements. The tools that you use can be as simple as diagrams and documents. Alternatively, you may be using some advanced Model Driven Engineering (MDE) or CASE (Computer-Aided Software Engineering) tools and techniques. MDE and CASE tools are used to model the software design. In most cases, these tools can be used to generate code and other artifacts, such as database scripts and even documentation. The tools you use must be fit for purpose and not impact your ability to design software. The process you are going to use for design needs to be in place, along with the appropriate review checkpoints. Your estimates need to take your process into account to avoid potential overruns or missed milestones. Design covers the application functionality, the hardware architecture, and the software architecture. Understanding the requirements and the design is crucial during the development process. That does not mean things are not going to change as you move on, but it ensures that everyone is working from the same page. If things change, you will need to update your designs, so the tools and processes need to be fit for this purpose as well.
- **Development processes and tools** — The development process is where you are going to do your coding. You've seen that development involves architecture, application, batch, reporting,

deployment, and tools development. These areas may involve a variety of different tools and it is important that all the development tools and processes be fit for purpose. Timescales can be increased dramatically by not having the right tools and processes in place. Your tools need to display the required quality characteristics of a production-ready application. To avoid slippages, your estimates need to take into account your development tools and processes. Understanding the development requirements is paramount to supporting the development activities. You also need to ensure that the components you develop during construction are fit for purpose and ready to build; otherwise, you could experience issues when you submit your development work into the build process.

- **Configuration management processes and tools** — The tools and processes that you use to manage your source code and other development and test artifacts need to be in place and fit for purpose. Not being able to check in or check out can have an impact on development and test timescales. The version/source control system may also have its own database and housekeeping recommendations or requirements. It is important that these are also understood to ensure that the source control system is kept in good shape and doesn't adversely affect your ability to develop and test. Releases are often developed in parallel, involving branching and merging activities. It is important that all these factors are understood to ensure that the source control tools and processes support these requirements as well as all the other required quality characteristics. If implemented incorrectly, branching and merging can be a painful business, so the process needs to be defined, understood, and executed well to avoid failures and delays.

- **Regular integration processes and tools** — It is important to define and understand this process so that you avoid failed builds and don't have to go back around the loop. The tools you use to build a release must be fit for purpose and have the required quality characteristics. If you can't build a release, you can't get it into the deployment process. Ensuring that you provide ready-to-build components and artifacts is vital to the successful building of a release. You need to ensure that your components work with the latest versions, and that your components are tested and that the tests work with the latest versions. Submitting incorrect or non-ready-to-build components has a negative impact on the build process and your ability to get the appropriate components into the deployment process.

- **Deployment processes and tools** — The deployment process covers everything from including all the final artifacts in a deployment package to getting releases into the environments with all the correct set-ups and configurations ready for use. It is important that the tools you use are fit for purpose and that you get your software out where it is required. The requirements of the environments need to be understood to ensure that the relevant artifacts are deployed, including binaries, tests, documentation, and data (including base data and test data). A lot of time can be wasted installing and re-installing releases because of incorrect options or settings being selected.

- **Testing processes and tools** — The testing process is where the software will be tested. It is important that the tools and processes used to test the software be fit for purpose. Simple things such as trying to bring the environment up can impact testing negatively. You need to ensure that you have all the productivity tools and processes in place so that testing can continue without environment or process issues. That's not to say that there will not be defects in your code, but it will ensure smoother testing of your software without unnecessary issues, and help you to find the real issues to concentrate on and fix.

- **Defect tracking processes and tools** — Software is very rarely defect-free, although we strive to achieve it. Having a good defect tracking system in place helps you understand the defects you need to take into account and plan for. It is important that this system is also fit for purpose and

displays the required quality characteristics. The defect tracking system is often customized to the project's needs, so understanding these needs helps greatly. These needs will also change as the project progresses, so you need to be able to react to this and change accordingly. Understanding the defect tracking system and what information needs to be included in a defect will help to improve your ability to react to them. There's nothing worse than trying to work on fixing a defect and finding that there isn't enough information to go on.

- **Change control processes and tools** — Things change as you move through the development lifecycle, and having a good change control process and the tools to support it is a key factor for success. Understanding and implementing a formal change control process and adhering to it will enable you to avoid unnecessary delays and improve the transparency of changes. Delays are often suffered during development by working on out-of-date information. Sometimes it is better to complete a development piece prior to taking updates into account, but knowing that it is going to change assists in planning and estimating and improves the entire process.
- **Application support and maintenance processes and tools** — Your system needs to be supported in the event that any issues arise that need to be fixed. This is not just for your own applications; it includes third-party applications, as well. You need to have the right processes and tools in place to assist with support queries. You can greatly improve the supportability of the applications you develop; however, third-party applications are not always the same. Understanding the support requirements of these will help you to define your support processes and the productivity tools that you should put in place. I've never had a support call in which the support person didn't ask for the version numbers of the operating system, patches, the hardware it is running on, the version of the application, the logs files, and so on. These are basic support requirements, and having robust and streamlined processes to obtain this information will help you better react to support calls and incidents.

There are potentially many other processes and tools involved during software development and implementation. However, the preceding list captures all the processes discussed in this book.

Users Must Be Trained

It is important that all the users of your system be educated correctly and sufficiently. There are a number of users of the system and each has his or her own agenda and work to do. Understanding the requirements of your users is important to the success of your project.

Education can be as simple as providing productivity guides and documentation, or it can be instructor-led training courses on the environments, applications, languages, and/or technologies. The processes that you put in place need to be well-documented so that everyone understands them and, more importantly, follows them.

A lot of different people and roles are involved in a software implementation project. Figure 1-7 shows just some of the roles and responsibilities that can be involved in an end-to-end software implementation project.

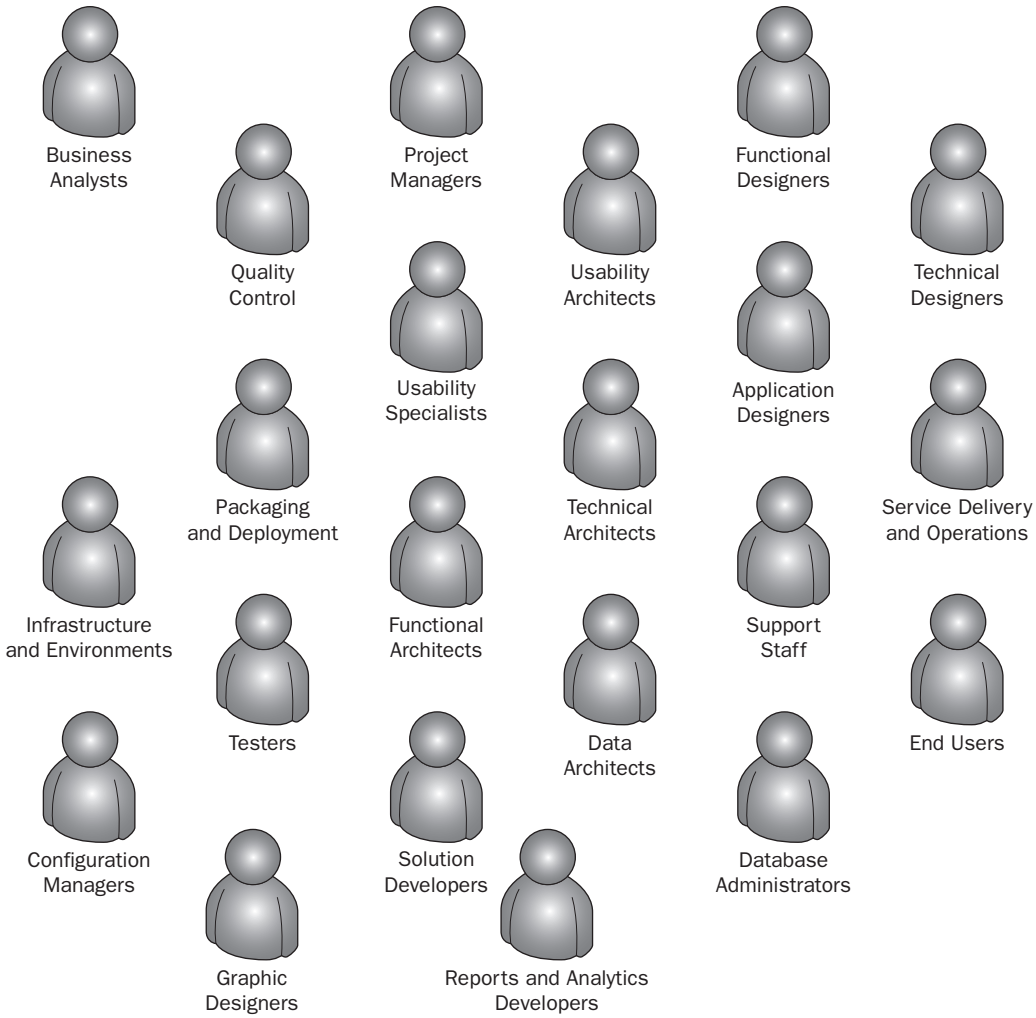

Figure 1-7

Although Figure 1-7 doesn't depict every person and role, as you can see, it is possible for a number of different people to be involved in the overall project, all of which need to understand the applications, tools, and processes that they use. Furthermore, in many cases a single individual can perform many roles, which could require knowledge of many different applications, tools, and processes.

It is important that everyone involved in the project know how to use the *applications, environments, processes*, and *tools* that they are going to use throughout the project. The documentation and education that you put in place will generally be multi-purpose — that is, it will be used by a variety of different people. There are, of course, documents that are specific to a particular process or technology and, as such, target a particular group of users. Understanding your users and their needs is an important part

of production readiness, and having access to the relevant documentation is equally important. Your project will have a variety of documentation, including:

- **Project documentation and guides** — This set of documentation includes a variety of materials, including the project overview and business case, the requirements and specifications (both functional and technical), architecture diagrams and documentation, the project plans, and the organization structure. These documents are important for you to understand what you are building and why. It is important that they contain all the relevant information to allow you to do your job properly.
- **Application and technology guides** — This set of documentation covers all your applications and technologies. Third-party applications come with their own documentation that can be supplemented with your own additions and extensions. These may be as simple as installation and configuration guides. These documents need to contain the information necessary to ensure that whoever is using them can get up to speed quickly and efficiently. For example, one such guide would be about how to configure application-specific clustering, including the specific naming standards that must be adhered to and which prerequisites must be in place.
- **Environment guides** — The environment guides provide an overview of the environments you are using, how they are scaled, what they are used for and when. The information can come from many sources, such as architecture diagrams and other specifications. You need to understand the environments and how to use them, which may be as simple as knowing how to get access to the functional test environment. The important thing is that the information is available and includes everything required to use the environment effectively.
- **Process and productivity guides** — There are a number of processes that will be created and followed during the project. It is important that these processes be understood to ensure that everyone is working from the same page and understands what to do and when to do it. You will have a development process, a build process, a unit test process, an integration test process, as well as many more throughout the project. The process guides need to document the process and procedures in plain language to ensure that everyone can follow them.
- **Tool guides** — Understanding the tools, what they are used for, how to use them, and when to use them is important. This set of documentation covers all your tools and the specifics of how you are going to use them for your own purposes. These documents may cover how to use the source control system, how to use the defect tracking system, and how to use the unit test tools and integration test tools. The important thing is that all the tools are well-documented so that users of these tools can understand their purpose and use them effectively.
- **Completion reports and handover documents** — Part of the overall development cycle involves producing completion reports. Depending on the methodology chosen, each phase or activity may involve a hand-off between phases or teams. The completion criteria for the activity or phase will need to be met, and the completion report documents how this has been achieved — for example, documenting the test results, reviews and profiling results. The handover documentation will typically also include the number of defects that have been raised and addressed within the current activity. It may also include outstanding defects or other items that have not been addressed and need to be carried over to a further release.
- **Release notes** — Although release notes strictly belong within the handover documentation category, I've pulled them out separately as they are very important documents. In summary, a release note typically includes the current version, system requirements, installation instructions, and resolved issues as well as known issues. Whenever you produce a release, an

associated release note should be produced to go along with it. Although automated deployment tools might be available to help with the installation, the release note still needs to show the contents of the overall release.

There will probably be other useful documents that are produced throughout the project lifecycle. However, I've chosen the preceding because during design, development, and early testing, you can have a dramatic impact on their production and readiness. As this book progresses, you'll see how some of these documents are used and the people that will use them.

Having these documents and guides in place helps to reduce the time and effort you spend on education and one-on-one training. You might have many developers or testers starting on your project, and being in a position to provide them with an induction pack will greatly help to reduce your costs and improve their readiness to start developing or testing. All the documentation that you produce needs to be fit for purpose to ensure that everyone is ready to do their jobs. The documentation will be used throughout the project and will form a valuable part in the handover to live service.

It is also important to note that I'm not necessarily suggesting that all of this information be captured in formal "documents." Online help files, wikis, and other electronic-based methods can be employed to reduce overheads and improve the overall level of communication, collaboration, and education within the team. A *wiki* is really a set of web pages that can be read and/or updated by project resources. A wiki can often save a lot of documentation effort by focusing on the actual information that's required by the reader. I personally like guides and certain other documentation to be very article-based and quick to the point, and to provide all the necessary information and step-by-step instructions to perform a task.

I'm a firm believer in the "project portal" concept. It is great to have a single project website where people can go to find out information about the project, which can include status, plans, designs, specifications, process guides, training guides, induction guides, and so on. In my opinion, the project portal is essentially a one-stop shop for everyone and everything to do with the project. I try to think of the content as "day one developer," which basically just means that I imagine what a new resource would need to get up-to-speed when they first come to the project, aside from access and an environment to work in. It is not a full time-and-motion study, but it works very well to ensure that inducting a new resource is efficient and successful. If a new resource can get up-to-speed quickly, all resources should be up-to-speed. A well-laid-out portal can provide a rich user experience and ensure that everyone has access to the latest status, plans, information, templates, and project documentation. Setting up and maintaining a project portal, however, offer up their own challenges that need to be factored into the scope, budgets, and timescales. And, of course, it too, needs to be fit for purpose and production-ready.

Why Is Production Readiness Important?

Failure is *not* an option. I can only assume that almost all projects start off with this stance. Most of the projects that I've worked on certainly have. The trouble is that a number of projects do fail or are seen to be a failure in the eyes of customers or stakeholders. The question is not "Why do they fail?" The ultimate question is "How do we succeed?" A cursory search on the Internet would yield an abundance of results on both these subjects. However, I'm not going to go into the statistics and findings of these studies. I'm going to provide my personal point of view on what I think are the main causes of failure and what I think are the key factors for success. To answer the second question "How do we succeed?" it is often necessary to examine your past mistakes, understand what went wrong, and put measures in

place so that it doesn't happen again — which is just commonsense. In recent times, there have been some very newsworthy software outages, some of which have cost millions of dollars in lost revenues and crashing stock prices. Mission-critical systems can't sustain serious outages in live service. In some extreme circumstances, the outages have cost many high-ranking personnel their jobs and sometimes their careers. It is not always so bad, however. On the less extreme side, a project can be seen as a failure because of delays and budget overruns. A successful project is typically referred to as one that delivers all the required scope, on time and on budget. To achieve this goal requires you fully to understand the scope, timescales, and budgets for the project.

> *My first experience of "commercial" software development: I remember back to around the mid-80s when I wrote a program for a small business in my hometown. The program was written on a BBC Model B computer in BBC BASIC. The owner of the firm told me what he wanted the program to do, and I thought I could write it in a couple of days and said I'd do it for £50. For a couple of evenings I wrote the initial version and tested it (as best as I could) on the machine in his office. The following day he came in and "played around" with it. When I arrived that evening, he dictated his "improvements" and it took me literally weeks to include them. After which, I never heard from him again and I didn't get any more money for doing the additional work! I'm not trying to be funny, but the moral of the story is to get it right up front (or at least as best you can) because success is not solely based on customer satisfaction! It can cost an awful lot of time and effort to satisfy the customer. In a fix-priced agreement it is important to understand the implications of this very early on.*

It is a fact that some software implementation projects fail. There are many different reasons for why these projects fail. Sometimes a project is simply scrapped because it doesn't have a truly viable business case. Others are scrapped because of budget cuts, contractual disputes, and poor implementation. In my personal experience on the shop floor, projects fail or are seen to be a failure for these three reasons:

- **Poor scope** — Scope really refers to the requirements, constraints, and quality of the project, software, or even task. When the scope is poorly defined, the outcome can't be assessed effectively, the effects of which can have far-reaching consequences. Poorly defined or missing requirements as well as changes in scope lead to "scope creep." This can contribute to a project's failure by increasing costs and timescales and potentially delaying go-live. Changes or corrections to scope can also be caused by an incorrect or erroneous interpretation. For instance, consider what a response might be to the following extreme statement: "You told me you wanted an online store. You never actually said you wanted it to run on UNIX." Interpretation only works when the recipient of the final product actually agrees with what has been interpreted and delivered. If not, it has the potential to cause major disputes. More often than not, requirements and constraints are missing or misinterpreted and not sufficiently clarified, which results in an incorrect or incomplete solution. Furthermore, assumptions, like interpretation, can also lead to incorrect or incomplete deliveries, for example, "Oh, I thought that *you* were going to write that bit," or "I assumed it only needed to work with Internet Explorer."

Poor scope is not just about the functionality of the software, either. Poorly defined processes and tasks also affect the overall outcome of a project. For example, not stipulating the relevant quality criteria and reviews can also affect the final result. Having a well-defined and agreed scope will help to avoid these types of common problems. The quality criteria apply to all components of the delivery, including documentation, source code, and all other areas and artifacts. For instance, a technical specification can be seen as defective or sub-standard if it doesn't include certain elements, such as class diagrams, interaction diagrams, and component walkthroughs. The developer may not be in a position to develop the solution without such information. The impact and effort to include these items can then have a big

impact on the budget and timescales. In this example, the elements that need to be included in the document are simply part of the scope of work. Minor failings can build up over time and cause overruns and slippages, which can ultimately result in the project being seen as a failure. Rework is a costly business. For the purposes of this book, the term "scope" also includes all the applications, environments, processes, and tools. I'm also bundling customer and/or stakeholder involvement under this heading because without the appropriate level of involvement, it is nearly impossible to define (and agree on) the appropriate scope for a project, solution, or task. A very clear scope allows for much better preparation, execution, and delivery.

- **Poor planning and execution** — This heading includes a variety of different tasks and the environments they need to be performed in. One of the most common planning errors is setting or agreeing to unrealistic timescales. Although aggressive timescales can sometimes positively and constructively stretch project resources, unrealistic timescales can often break them. Unrealistic timescales can totally compromise quality and functionality. For instance, the documentation may not be as complete as it could be, there may be a major rush to start coding, and the quality may be dropped over the race to deliver. In this scenario, the plans are based on when a task needs to be finished and not how long it will realistically take to perform (according to a reasonable set of estimates). Plans should be based on realistic estimates, and the estimates should be based on a very clear scope and understanding. Furthermore, just because one particular person can complete a given task to the specific scope in a given timeframe, doesn't necessarily mean that someone else can.

This leads to another common planning error: inappropriate resource planning and staffing — that is, insufficient numbers of resources, assigning the wrong resources to the tasks, or using insufficiently trained personnel. Other typical planning mistakes are — failing to include all the required tasks in the plan, overlapping tasks without fully understanding their dependencies, and not building in the appropriate review, hand-over, and potential corrective activities. When the plans are agreed and in place, they need to be executed. That doesn't mean that the plan won't change — you often need to make adjustments — it just means that there is a solid way forward and changes can be incorporated according to the agreed process. It is also not worth denying that poor workmanship can lead to poor execution. There are a lot of reasons why certain resources don't perform very well. It can certainly happen when the resources aren't sufficiently trained or qualified. Therefore, it is important to ensure that all resources be sufficiently trained and educated to perform the required tasks. Finally, unnecessary downtime can occur when there are conflicts and major differences of opinion. Disputes can often cause delays and overruns when rework is needed. The entire team needs to understand and subscribe to the scope and the plans.

- **Poor quality** — Poor quality can occur for many reasons, including insufficient quality controls and insufficient scope, insufficient planning and "squeezed" timescales (as previously mentioned). Inappropriate resource allocation can also lead to poor quality — for example, using inexperienced resources when experienced personnel are required. The term "quality" refers to more than just code quality; it encompasses all the components of the delivery. If the technical specification is poor, it typically results in delays and multiple iterations before it is acceptable. In the worst case, it results in a poor implementation, especially when the gaps or errors are not caught early enough.

Poor quality is generally uncovered during testing, reviews and/or checkpoints. When testing and reviews take place early and regularly, it is much easier to manage the outcome and corrective workload. There's a very good opportunity for planning the corrective actions that need to be taken — for instance, categorizing the defects into their appropriate severity, prioritizing in order of implementation, and/or justifying and discounting them. Re-work and updates can be a costly and time-consuming task,

especially when a lot of issues are discovered and even more so when they're discovered very late in the process. It can be even more costly when the issues are raised by an external body, which could be the customer or another organization and perhaps sometimes even both. There are costs and implications associated with the time and effort required to go through the issues, assess and agree on them (where necessary), and then plan and address them. It might not be possible to address all the issues within the timescales and/or budgets, which could incur severe commercial penalties and/or delay go-live dates. The implications can be very far-reaching and it is really not worth getting caught in these situations. Defining the scope early, and understanding and agreeing on the criteria for quality and regular check-pointing and correction will avoid unnecessary disputes. Quality is an extremely important part of the overall implementation, which is why I truly believe it is a component of scope. A poorly defined scope, that doesn't encompass quality, can lead to a perception of failure. Testing will uncover defects. If it didn't, you wouldn't need to test. Testing should really be done early and frequently to ensure that as many defects as possible are captured and addressed (where necessary) as early as possible.

The Production-Readiness "Process"

Production readiness is all-encompassing. It includes everything to do with your system and ensuring that you are ready for development as well as live service through all of your processes and practices. It is not actually a formal process. It is really just a collection of foundation principles for successful software development and implementation, including:

- ❑ Good preparation
- ❑ Good execution
- ❑ Good delivery

These are all just umbrella terms that encompass many discrete and varied tasks, although they all play a vital role in the production readiness process, which simply underpins these primary success factors by specifying the steps — *prepare, execute*, and *deliver* (see Figure 1-8).

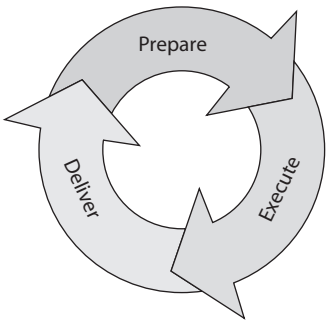

Figure 1-8

"Why do you need to prepare? Just do it!" Although this might seem like a good idea, racing ahead unprepared can cause major issues further down the line. There's no point in hiring a bunch of developers when there aren't enough development machines for them to work on. Similarly, it would be dangerous to start development on an application when the architecture and frameworks aren't sufficiently in place or the designs are inadequate. There's clearly a balance to the amount of upfront preparation that should be performed on a project, iteration, or task. However, the preparation really does need to consider what must *be in place and what the scope is to ensure effective execution and delivery. You look at this in more detail later.*

It is important to do the most critical upfront work as soon as possible to ensure that everything is fit for purpose and ready when it is required. This book places its focus on the elements of scope, execution, and delivery. As the book progresses, you'll see more on the tasks and activities that underpin these high-level, production-readiness steps. In the meantime, the following provides a brief overview of the steps in the process:

- **Prepare** — This step involves planning and defining the scope, including the applications, environments, processes, and tools, as well as defining the associated inputs, standards, checks, and outputs. It covers the definition of the acceptance criteria between one process and another. It also includes the definition of timescales and costs associated with each of these elements and the necessary checks and governance required for them.
- **Execute** — This step encompasses the actual execution of the plans and processes. It is the physical work being carried out on the production line (including any additional planning). For construction, this would cover the development team accepting inputs, producing designs, having them reviewed and agreed, coding, testing, packaging, releasing, and so forth. For testing this would include all the inputs required to produce test scripts, execute tests, and so on.
- **Deliver** — This step refers to the hand-off between processes and teams. In terms of software implementation, this would ultimately include handing over the solution to a support team or an application maintenance team once the system is in live service. Hand-offs are performed at many points in the development cycle, and one of the key areas is handing over a solution to a test team for testing or acceptance.

The activities should not be overly prescriptive, nor should they slow down the overall process. The processes and activities should always aim to reduce overheads and improve the performance of the overall project and the team. The processes that are implemented should be reviewed regularly and continually improved.

The Project Management Triangle

The commercial world of software implementation is typically all about squeezing as much as possible into given budgets and timescales. It is very rare that you're given a free hand and unlimited budgets to do what you want. Projects are typically constrained by a set of defined boundaries — *scope*, *budget*, and *timescales*, as shown in Figure 1-9. This is generally referred to as the *project management triangle*.

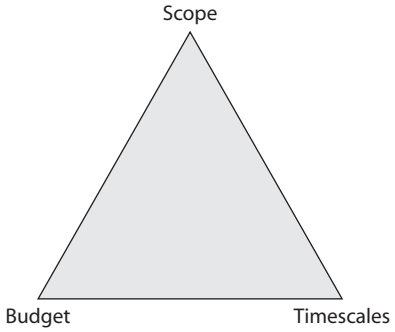

Figure 1-9

The figure shows the traditional project constraints and the axes are defined as follows:

- **Scope** — Refers to the requirements (functional and non-functional), constraints, and quality characteristics. In some cases, quality is treated as a separate dimension; however, as I've mentioned before, I treat quality as nothing more than a component of the overall scope.
- **Timescales** — Refers to the overall project timeline, from start to finish.
- **Budget (or costs)** — Refers to the amount of money available or required to complete the project.

The figure is drawn as an equilateral triangle, which simply reinforces the basic premise that changes to a single axis will affect the other two in a similar and consistent way. Although this point could very well be argued, an increase in the scope of a project will generally *increase* costs and timescales, whereas a reduction in scope will generally *decrease* costs and timescales (see Figure 1-10).

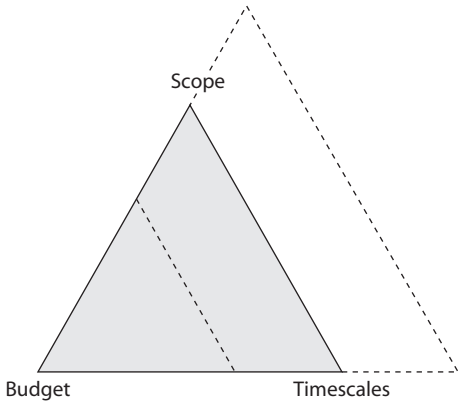

Figure 1-10

It is really just a matter of how much the changes affect the overall project. There's a theory that *two* of these boundaries can be constrained or fixed, but not all three (especially when initiating and planning a project). When everything is agreed and finalized, all three boundaries are typically set and the project is then underway. The following list details some of the various options when setting these boundaries:

- If the scope is fixed (that is, the solution must meet all the specified scope), the following options can be assessed:
  - Both the timescales and the budgets must be aligned according to the scope.
  - The timescale remains fixed and the budget is set to allow for more resources and parallel activities. This works on the principle that by using more people, the tasks can be done in a shorter timeframe. However, this in itself is sometimes challenging to achieve, given staffing and task dependencies. It is also possible that the budget or costs can be balanced elsewhere if performing the specific tasks in a shorter timeframe will reduce costs further down the line or in other areas.
  - The budget remains fixed and the timescales are set accordingly. This generally works on the assumption that by using fewer resources, the task (or tasks) will take longer to complete but the costs will remain the same. For example, if you assume each resource costs $100 per day, a task (or set of tasks) that takes five person days will cost $500. If the task can be completed with five people, it could theoretically be completed in a single day. Using only a single resource, it would take five days. However, in both cases the cost is the same.
- If the timescales are constrained (that is, the software must be delivered on a specified date), the following options can be assessed:
  - Both the scope and the budgets must be aligned according to the timescales.
  - The scope remains fixed and the budget is set accordingly — again, using more (or less) people to achieve a particular result.
  - The budget remains fixed and the scope is set accordingly. This works on the basis of delivering only what can be achieved within the timeframe and budget.
- If the budgets are fixed (that is, the software must be delivered within a specific amount of costs), the following options can be assessed:
  - Both the scope and timescales are aligned accordingly.
  - The scope remains fixed and the timescales are set accordingly. This works because you use far fewer resources over a longer period of time to deliver the appropriate scope.
  - The timescales are fixed and the scope is set accordingly. Again, this works because you set the scope to what can be delivered within the budget and timescales.

If a successful project is considered to be one that delivers *all the scope*, *on time* and *on budget*, to ensure this, the scope, budget and timescales need to be very clearly defined and understood. In my opinion, it really all starts with scope. Figure 1-11 shows the high-level components of scope related to software development and implementation.

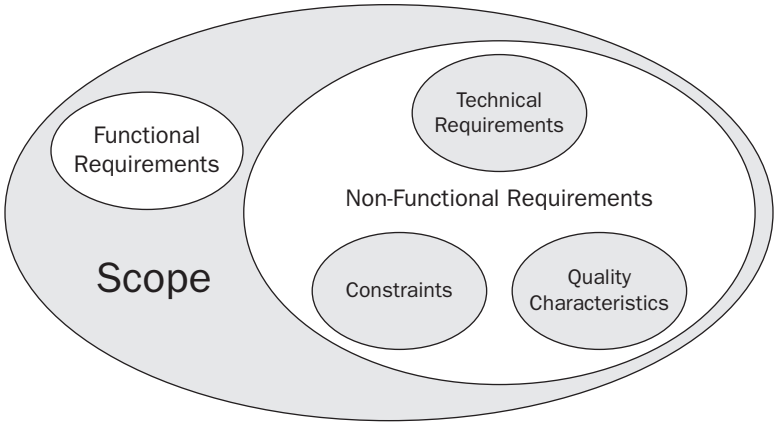

Figure 1-11

The scope will typically dictate the budgets and timescales. If the budgets and timescales are reduced, this could have an impact on the scope. I know I've labored on this point, but clearly defining the entire scope will ensure that everyone is on the same page and under no delusions or assumptions about what will be delivered and how it will be delivered, and its quality characteristics. The following provides a brief overview of the components of scope:

- **Functional requirements** — As the name implies, functional requirements define a set of functions for a software system, application, or its components. Functional requirements describe the functional behavior of the system. Functional requirements often include business rules, calculations, and processing requirements that describe how the software must work or behave from a functional perspective. Requirements are often born out of use-cases and high-level analysis and design activities.
- **Non-functional requirements** — Non-functional requirements, which can also be referred to as *technical requirements*, typically stipulate criteria for how the software should execute — for example, the system's expected performance characteristics, resource utilization, or security measures. Non-functional requirements are often referred to as the "ilities" primarily because they all end in "ility" — for example, reliability, scalability, availability, usability, maintainability, and so on. There's a fine line between functional requirements, technical requirements, and the quality characteristics of a software system. In fact, non-functional requirements are very often referred to as the *quality goals, quality attributes,* or *quality characteristics*. These types of overlapping terms and meanings are rife within the software industry, so I feel that it is really not worth getting hugely hung up on labeling and categorizing them and trying to define a truly perfect boundary. However, in terms of a software implementation project, the technical requirements could encompass execution qualities such as reliability and availability, whereas the quality characteristics could include the static and evolutionary aspects such as documentation, maintainability, and scalability. In the end, as far as I'm concerned, it is all just scope — functional, technical, or otherwise.
- **Quality characteristics** — The quality characteristics often encompass the non-functional requirements of a system. For instance, code quality can be seen as a technical or non-functional requirement and can be categorized under the readability or maintainability characteristics. But what really defines code quality? How is it measured, proven, and accepted? Chapter 2 examines some of the common quality characteristics and how they apply to not just the

software but the entire system (which, as you've already seen, encompasses *applications, environments* and *processes*, and *tools*). For example, a manual process that doesn't "scale" can easily impact a project. It is not just about the software, although the final application not being able to scale for future demand can also impact the project.

- **Constraints** — Requirements can also be referred to as *constraints*, so you again face overlapping terminology. However, I categorize requirements as the business rules, the functionality and features of the system, as well as the technical aspects of the system. Constraints, on the other hand, are restrictions on the degree of flexibility and freedom around implementing the system. For example, constraints could include, cost and budget, location, environment, tools and technologies, methodologies and process, resources, and timescales. A simple rule-of-thumb that I use when differentiating a requirement from a constraint is to determine whether it pertains to *how* and *when* the system will be implemented, as opposed to *what* is being implemented and *what* it needs to do.

While it is a very good idea to categorize all the various attributes of scope, it is actually far more important to ensure that they are captured, agreed, and fully understood. To help achieve success, your *preparation, execution,* and *delivery* need to capture all the attributes within your *scope, timescales*, and *budgets*.

Summary

In this first chapter you've seen that production readiness relates to both software development and implementation. It is about the quality and readiness of everything to do within the project and solution. If you're unfamiliar with some of the terms used in this chapter, don't worry — they'll be discussed further as the book progresses. Your application is just one part of an entire system, and you've seen what the individual parts can consist of, how they can be used, and who can use them. As you continue reading, you will start to see how everything fits together. However, keeping these factors for success in the back of your mind will help you to improve your own capabilities, help others to improve theirs, and should positively impact the overall outcome of the project.

The following are the key points to take away from this chapter:

- **There are a core set of "development" and "implementation" activities.** Irrespective of the chosen development approach, there are a core set of activities that are performed during the project lifecycle. The activities discussed in this chapter include:
  - Requirements definition
  - Design
  - Development
  - Regular integration
  - Deployment
  - Testing
  - Configuration management
  - Change control

- Defect management
- Support and maintenance

- **Failure is *not* an option.** Good planning, good execution, and good delivery will help to ensure successful development and implementation of software projects. As soon as the project starts, you're in a "production mode" of some sort, and any tools, technologies, environments, or processes that are being used need to be fit for purpose to avoid unnecessary delays moving forward. Considering all the production readiness and development readiness criteria will help you to *prepare* better, *execute* better, and *deliver* better, resulting in a truly fit-for-purpose system.
- **Clearly define the scope of the project, solution, or task.** Only when you know the true scope of a project, system, or task can you really determine what needs to be done, how it can be done, and the associated budgets and timescales. A successful project *delivers all the required scope on time and on budget.* Scope should include all the following for all your applications, environments, processes, and tools:
  - Functional requirements
  - Technical requirements
  - Quality characteristics
  - Constraints
- **Production readiness is all-encompassing.** It is is not just about producing production-ready code and testing it thoroughly. It encompasses everything to do with your system. You need to ensure that everything is fit for purpose and ready for production usage.
- **Applications must be fit for purpose.** Identify as many of the other applications early in the lifecycle, as this will help to ensure that the solution stack is well-defined and all the appropriate steps can be taken to assess, design, develop, document, and implement them. It is also important to understand all the features and functions of the applications being used, and that they are documented and supported to avoid potential issues later in the cycle. The applications highlighted and discussed in this chapter include:
  - Operating system and extensions
  - Application Runtime Environments and extensions
  - The data stores
  - The security and encryption solution
  - The batch solution
  - The reporting and analytics solutions
  - The monitoring solution
  - The backup and recovery solutions
  - The deployment solutions
  - Your own custom applications
- **Environments must be fit for purpose.** Key environments should be identified early. The environments should be sized and scaled appropriately for their purpose and ready to use when

they are required. The environment includes all the necessary hardware, software, access, and networking components. The environments highlighted and discussed in this chapter include:

- The design environments
- The development environments
- The regular integration and build environment
- The configuration management environment
- The regression test environments
- The test environments
- The training environments
- The production environments (including pre-production and disaster recovery)
- The support and operations environments
- The application maintenance environments

- **Processes and tools must be fit for purpose.** Ensure that all the processes that need to be followed are documented and fully understood. Where necessary, tools should be used to improve the overall performance and outcome of the process. The processes and tools discussed in this chapter support the "development" and "implementation" activities.

- **Users must be trained.** Your users include many different people, including designers, developers, infrastructure and release personnel, testers, support staff, as well as end users. Each of these user groups needs to be trained, and there's a variety of documentation that can be put in place to help with this. Your documentation needs to cover all your applications, environments, processes, and tools — not just the application you are developing. Documentation doesn't need to be exhaustive, but it does need to be fit for purpose, and a project portal, online help, and wikis can help to speed up its production and usage. The high-level user groups and roles discussed in this chapter include:
  - Business analysts
  - Designers
  - Developers
  - Testers
  - Configuration management
  - Release management
  - Service delivery and operations
  - Application maintenance
  - Customers and business users

The types of documentation and education material discussed in this chapter include:

- Project documentation and guides (including requirements, quality, and constraints)
- Application and technologies guides (covering all your applications)

- ❑ Environment guides (covering all your environments)
- ❑ Process and productivity guides (covering all your processes)
- ❑ Tools guides (covering all your tools)
- ❑ Completion reports
- ❑ Release notes

❑ **Your systems involve multiple applications, environments, processes and tools.** You need to ensure that all of them are industrial strength and meet all the required quality characteristics for production readiness.

The following chapter examines some of the quality characteristics that should be considered for all your environments, applications, processes, tools, and documentation.