

3

Preparing for “Production”

This chapter examines some of the key areas you should consider when setting up your software-development production line. With a well-oiled production line, you should be able to develop anything and get high quality products out the end. We’re not concentrating so much on the final solution, although its size and scale will affect some of the decisions you make. As with any kind of manufacturing, you need the right raw materials, the right processing, and the right quality controls.

There are many processes that you need to establish before starting full-on production (e.g., development) to ensure that you are in a position to produce high quality outputs and I tend to use the word “defining” for a lot of the activities described. In the majority of cases, I am actually referring to “defining and implementing.” As with most things in life, we’re very often constrained by certain factors, and software development is no different. Aside from timescales and budgets, we can also be constrained by a particular methodology, technology, or policy, for example. When setting up your factory, you must understand whether there are any specific constraints that the production line must follow. If not, then you’re free to implement your own. Think of these as the formal regulations for your production line. In the construction industry, buildings must be inspected at regular intervals. There may be regulations whereby your code will be inspected at regular intervals by an external party. The regulations can be set by your own company, the consumer, an official body or third-party, or, finally, by yourself. However, before you start making your own regulations, you need to determine if there are regulations that you *must* adhere to.

You’re going to look at some of the types of applications, environments, processes, and tools that should be assessed and put in place (where appropriate). Everything you’ve seen so far needs to be cost effective and implemented in a controlled and orderly fashion. This chapter focuses primarily on getting your software production line established.

Part I: Production-Ready Software

This chapter is organized into the following sections:

- ❑ **Preparation Is Paramount** — Examines the high-level activities that you perform during the project and shows how they can be broken down to ensure they are mobilized effectively. It also discusses the key constraints that should be identified when mobilizing any activities.
- ❑ **Development Mobilization** — Discusses the specifics regarding the activities, choices, and decisions that should be taken into account when mobilizing the development activities. This includes ensuring that you have the right inputs and that they are well-defined, ensuring that you have all the right processes and practices in place, and ensuring the outputs from the development activities are fit for purpose.
- ❑ **Build, Release, and Regular Integration Mobilization** — The build and regular integration activities are vital to ensuring that you can build and release your software. This section discusses this process along with some of the different releases and their typical contents.
- ❑ **Test Mobilization** — Testing is where you're going to prove your application works the way it should. Having solid foundations in place for testing will help your project. This section looks at some of the specifics in mobilizing your testing activities.
- ❑ **Defect Management (and Fix) Mobilization** — Testing, as well as other activities, is going to raise defects. This section looks at defect management, including the workflow and the contents of defects, and discusses some ways of how defects can be categorized for management (and reporting) purposes.
- ❑ **Configuration Management Mobilization** — You need to store all your artifacts somewhere, and this section discusses the configuration-management system and processes. It looks at some of the items that you can choose to place under configuration management and how they can be organized to meet the needs of the project. It also discusses branching and merging, which may be required.
- ❑ **Change Control Mobilization** — Things are going to change, so you need to be ready to assess changes appropriately. You also need to ensure that the change process is followed.

I am not going to cover the analysis and design mobilization activities, although you should also consider them accordingly. The activities outlined in this chapter will give you some ideas for identifying and defining analysis and design mobilization. Mobilizing your project effectively is a key factor for success and the activities discussed in this chapter should be included within the overall scope of the project to ensure that the costs and timescales are accounted for.

Preparation Is Paramount

Chapter 1 discussed the high-level activities involved in software development and implementation. You also looked at the production-readiness process and saw that your applications, environments, processes, and tools must be fit for purpose and your users must be trained and ready. All of these are brought together and shown in Figure 3-1.

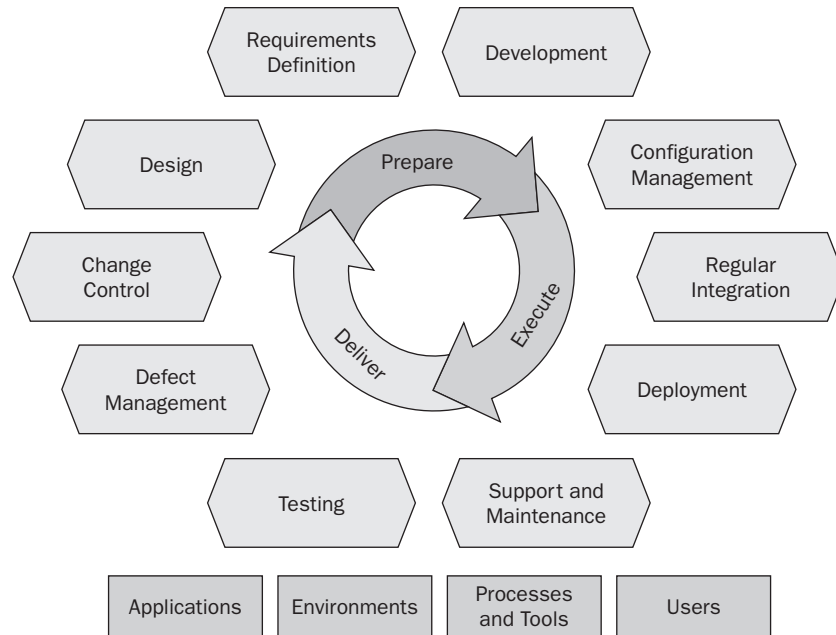


Figure 3-1

Hindsight is a wonderful thing. So is learning from our triumphs and mistakes. When planning your project, you need to remember your past experiences and look at what was good and what wasn't so good. You need to improve everything each time until you have the ideal process. Although the process will differ from project to project and organization to organization, the best time to reflect on past experience is early on. Once your project is off and running, the time to change and the impact of change generally increases, although you should reflect at very regular intervals and improve where necessary.

You want to give your projects the best chance of success and a good way of achieving this is to ensure that you are prepared for each of the activities in Figure 3-1. You need to think about what you're going to do and how you're going to do it. This can be thought of as *mobilization*. Before starting actual development, you should mobilize the development activities — for example, ensuring that the best practice is in place, ensuring the developers have been on the right training or fit the right profile, and so on.

The ultimate goal of setting up your production line is to get as much of it as possible defined and working up front, before full-scale production starts. The purpose of the mobilization activity is really to iron out the processes, practices, and procedures, to plug as many gaps as possible, and to provide a solid foundation before launching into it full-scale. Every activity in the development lifecycle has its own prerequisites and ideally needs to be mobilized, stabilized, and documented. On a normal production line, you wouldn't want to start production of millions of units, only to find half way through there was a flaw in the process.

Remember that even with these mobilization activities in place, you probably won't get it right the first time. Tweaks and adjustments will need to be made throughout the project in order to fully refine the process. Depending on the number of mobilization activities and the time required, you could perform them simultaneously. For instance, if you want to use an agile approach to construction, you'd need to mobilize quite a few activities beforehand.

Part I: Production-Ready Software

As a developer, you are used to breaking down problems into their component parts, and preparation is no different. You need to look at what you will be doing during these activities and break it down into discrete tasks. Your success ultimately is based on how well you *prepare*, *execute*, and *deliver* the activities throughout the project lifecycle. This section (and chapter) looks at some of these high-level preparation activities.

I recently took my car to the garage after I, unfortunately, scraped and dented a panel. I was told that because there was nowhere to blend the paint, it required a complete re-spray. Although the design of the car is beautiful, it's a shock to the system when you're told that what appears to be a minor repair is actually a major one because of the car's design, not to mention the expense of it. I use this story for two important reasons. First, a total re-spray is all about the preparation. Actually spraying the car takes little time in comparison to the sanding down, pulling out all the little dinks and chinks, and making good anything else. Second, it's a costly business when it seems like a simple matter to be addressed. It's just a minor dent, right? These two principles apply to software development. What may seem like a simple change can have huge cost implications if not considered early enough in the process. Considering the solution and the way it will be implemented, tested, deployed, and operated is nothing more than preparation. The actual development can be very little in comparison if the foundations are set correctly.

Breaking Down the Activities

To define a really low-level set of requirements (functional or technical), some form of design activity needs to be conducted. Similarly, to flesh out the design in more detail, some form of development activity will need to be conducted. This usually involves building a prototype or a reference application that models the basic functionality. Let's not forget that the testing activities often raise a series of changes based on real usage of the system. This is the nature of software development. It's very hard to get it right on paper, and it's very hard to get it right the first time.

Figure 3-2 shows each of the key activities split in two parts. Each part has a high-level and a low-level or detailed counterpart that provides more detailed information.

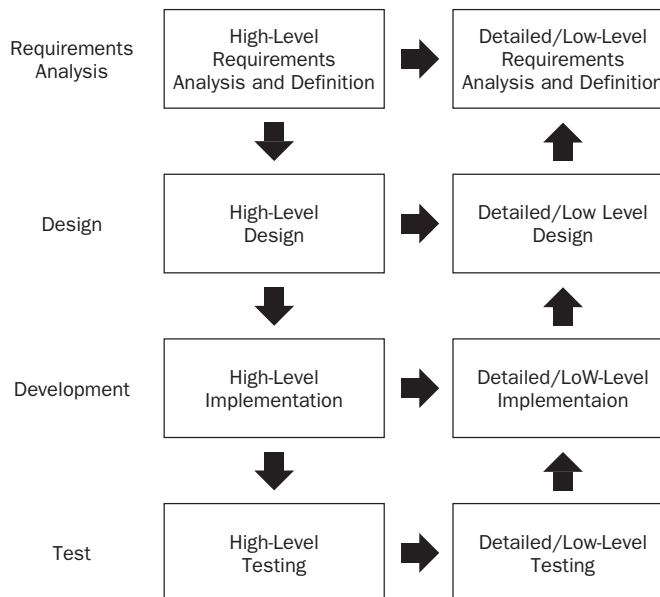


Figure 3-2

These activities essentially apply to everything you do and use. For example, you need to record your system’s functional and technical requirements somewhere. This task itself actually starts off with requirements analysis. You need to understand the requirements for your Requirements Management Solution (RMS). You then need to design a suitable solution to your problem, develop it, and test it. Each of these activities is a process of refinement. You start off with a high-level concept and refine it as you progress. Similarly, you need to store your system’s source code and configuration. This again starts with analyzing the requirements for the Configuration Management Solution (CMS). Again, you’ll produce a high-level design, develop the solution, test it, and so on.

In this instance, your consumers are the folks who are going to use all these items. In the case of the RMS, this will probably be everyone involved in the project. You need to assess and define each of the quality characteristics and ensure that the solution meets them. You also need to ensure that all the consumers (users) can access the solution and understand how to use it.

Identifying the Constraints

I find the best place to start when mobilizing any activities is to identify the constraints. Constraints are always a very good place to start because they don’t normally have any bearing on the functionality or technical aspects of the system or sub-systems — they simply shape the way the final state solution is implemented. Gathering, understanding, and defining the constraints serve as the starting point for building your software-development production line.

Figure 3-3 shows (and the following sections discuss) the high-level categories of the constraints you need to understand, plan for, or put in place yourself.

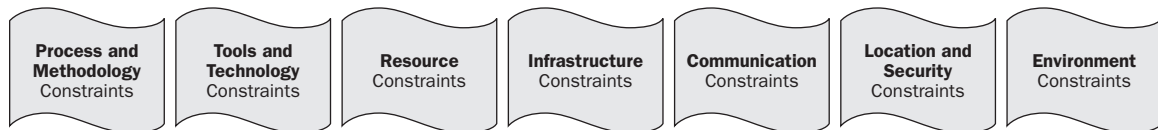


Figure 3-3

Constraints can be specified in both high-level and low-level form. For example, a high-level constraint could be “Code coverage must be checked and reported,” whereas a low-level constraint would be “Code coverage must be 100 percent during unit testing and checked using code coverage version 8.7.” The more detailed the constraints, the better.

Process and Methodology Constraints

You must first determine (and understand) the methodology that the overall project will follow. The methodology generally stipulates what the phases and/or activities are within the project, what each phase encompasses (including its inputs and outputs), and how the phases interact (and possibly overlap) with each other.

There are many formal methodologies (and you can define your own custom approaches), some of which were highlighted in the previous chapters. The size, scale, and complexity of the project (or activity) typically dictate the approach that should be taken. You’re in the very early stages of setting up your development factory, and you have a number of “activities” to initiate.

Part I: Production-Ready Software

It's possible that your "implementing the Requirements Management Solution" project could follow a very strict waterfall model, especially if the solution is going to be used for many other projects. Each phase would follow the previous phase, such as analysis, requirements definition, design, build, test, deployment, and so on. For example, analysis would involve talking to stakeholders and gathering their high-level requirements. It would also involve gathering the user's high-level requirements. The high-level requirements would be defined, documented, and reviewed as part of the requirements definition activity. The requirements would be fed into the design activities, and so on, and so forth.

Alternatively, "implementing the Requirements Management Solution" could follow a far more rapid development approach, especially if it's only for a single project. For example, it could be as simple as performing a cursory search on the Internet for some "best practice requirements gathering" tips, pulling together a template spreadsheet, storing it on a shared drive, and telling everyone where it is.

The approach could also be anywhere in between the two extremes. Every project is different, so the reality is that most projects use a mix of methodologies without necessarily knowing it or putting it in place formally. Large, mission-critical projects that have multiple sub-projects may follow a structured waterfall approach overall. However, some of the sub-projects and activities may follow more agile and rapid development approaches. For instance, tools development will more often than not follow a more agile approach. It's quite likely that scripts will be developed around the Configuration Management Solution to extract files and feed them into the Regular Integration Solution. These development activities could essentially follow a strict waterfall approach, although more than likely they won't.

The term *mission-critical* refers to items and software that are crucial to the successful completion of a project. It also means the items and software are vital to the organization's mission. It is, therefore, possible (although perhaps somewhat tenuous) to classify your mobilization activities as "mission-critical." Without a development environment and development tools, you can't develop any software at all. While every project is different, there are some high-level categories listed below:

- Non-critical systems** — Systems that are not critical to the success of the project or organization
- Mission-critical systems** — System that are critical to the success of the project or organization
- Life-critical systems** — Systems that are critical to safety and lives

We rely on software and systems so much today that it's often hard to categorize under the "non-critical systems" heading. You need to determine whether the project and/or organization can succeed without it. Identifying what is mission-critical is the first step.

There are no hard-and-fast rules as to the development approach that should be taken for each system type. The system and the approach need to be assessed for their suitability. Each methodology will have its pros and cons. Although I could make some general recommendations for each, it's really a process of discussion and agreement. The project might have the opportunity to define its own development approach; however, organizations often have their own formal approaches to software development that must be followed. It is important to determine and understand whether there are any rigid constraints that *must* be adhered to regarding the software development approach. If the organization doesn't stipulate a formal approach, the project needs to define its own approach.

In addition to software development approaches, organizations often have their own internal processes that must be followed. This is another critical factor in your scope and preparation activities. It's not uncommon for organizations to stipulate external or third-party reviews and penetration tests, in which case you need to understand this process and ensure that it's planned for.

Furthermore, many organizations incorporate the following processes:

- ❑ **Procurement** — Anything that you want to buy may have to go through the organization’s procurement process. This might mean using the organization’s preferred suppliers and vendors instead of your own. This could result in not being able to purchase the items you need and force you to seek alternatives. If this is the case, it’s very important to engage the procurement team early and understand the process, timescales, and constraints.
- ❑ **Recruitment** — I’ll discuss resource constraints shortly, but a number of organizations have their own recruitment processes. Mobilizing your team might require new resources to be hired. If this is the case, the recruitment team needs to be engaged early to understand the process. In addition, you might want to engage outside specialists or contractors, a process that could come under the heading of recruitment or procurement, so it’s well worth understanding.
- ❑ **Policy compliance** — Organizations usually have a number of internal policies that need to be adhered to. For example, an organization may have strict guidelines regarding the use of freeware, shareware, open source software, or even the use of trial licenses. The policy may have to be adhered to even if the software isn’t a part of the final solution. Information exchange is another area that could require strict policy compliance. It’s very important that you understand any policies or regulations that you *must* adhere to during the project.

There may be other categories of process and methodology constraints, but in summary, you need to ensure that you fully understand any and all of them. If none are in place, you are free to implement your own approach, assuming it is agreed on by the organization.

Tools and Technology Constraints

Most organizations have their own preferred tools and technologies. For example, there may be a constraint to use a particular configuration-management tool, load-testing tool, or profiling tool. It is quite rare that you get to choose everything from scratch, although it does sometimes happen. You generally get the opportunity to fill in the gaps with a combination of custom tools and scripts along with some other purchased tools. If there are no formal constraints, you’re free to implement your own tools and technologies, again, assuming they are agreed on by the organization.

One particular constraint to watch out for is the use of “internal” components. For example, you may be integrating with the Order Management System (OMS), and the team responsible for the OMS has developed a specific integration component. If you’re required to use any of these, you must ensure that they are fit for purpose and that you understand exactly how they are used, what they are used for, and, most importantly, whether they actually work. Specifications are no substitute for the real thing. What is the real thing? You need to get hold of it, install it, and make sure it works the way *you* need it to work. You may also find that you need to stub it or even simulate it (if suitable stubs and simulators aren’t already available). When “internal” components *must* be used, it’s worth engaging the teams responsible for them and understanding as much as possible regarding their installation, configuration, usage (in all situations), and, most importantly, support and maintenance. You may need to raise defects or changes against the component, so you should fully understand these processes, too. It’s exactly the same for third-party products, as well.

The tools and technology constraints need to be listed along with:

- ❑ The exact version that must be used.
- ❑ What they are used for.

Part I: Production-Ready Software

- Whom they are used by (designer, developer, tester, and so on).
- Instructions for obtaining them, installing them, and accessing them. It's equally important to determine whether they are hosted centrally or locally on the user's machine.
- Specific usage guidelines.
- Any licensing model and associated costs.
- The number of licenses required and available, and how to extend or obtain new ones and how long this takes.
- The current license keys and their expiry dates.
- Whether they require configuration, customization, or enhancement.
- The system requirements, deployment and backup requirements, and best practices.
- The support arrangements (office hours, out-of-hours, and so on).

All this information will help when mobilizing the activities and defining the additional tasks that may be required.

Resource Constraints

You need to understand if there are any key resources that must be utilized. In this case, you're mainly looking at human resources that need to be engaged. For example, a lot of companies have a central technical architecture or DBA function. It's often a constraint that all designs and outputs are reviewed and signed off by them. In some cases, this can sometimes create a bottleneck for projects. If you're producing large numbers of items, the individual or team could be overloaded. The resource constraints should be identified and documented. When attempting to identify resource constraints, you should ask yourself (and the organization) the following questions:

- Whom do I need to engage?
- Why am I engaging them?
- How do I engage them?
- What lead time do I need to give them?
- What do I need to provide them with?
- What are they going to provide back to me?
- How long can I expect them to take to complete their activities?
- What do I do if they are late or delayed?

It is imperative that you mitigate these risks and ensure that the project can continue even if it is waiting on sign-offs. If the team is going to continue with their core activities, this is referred to as *proceeding at risk*. Development that starts before the designs have been signed off is a very clear risk and needs to be carefully balanced. Developers can begin a number of activities while waiting for formally signed-off designs and other documentation; however, the delay could have a more far-reaching impact on the timescales.

If you’re doing the project on behalf of another organization, it might be a constraint that all development staff pass their in-house test or have to be certified to a certain level before being able to work on the project. It may also be a constraint that all people working on the project must be cleared by security. This could apply equally to contractors and other temporary staff within your own organization.

The following questions help to quantify these constraints:

- What skills do the development staff, contractors, and/or temporary staff need?
- Are any specific qualifications/certifications required?
- Are there any specific processes to obtain them?

The final resource constraint is working hours and availability. A lot of companies (and regulations) stipulate maximum working hours. You need to understand this because not only will it affect how you manage your own people, it will also affect how you engage others. For instance, if the support organization works a set of core working hours, you may have to coordinate well in advance any out-of-hours arrangements.

All the resource constraints need to be listed and captured appropriately so that they can be carried forward throughout the mobilization activities (and, of course, the remainder of the project).

Infrastructure Constraints

Production hardware and related infrastructure is typically installed in a data center or server room. In addition, there may be constraints for specific hardware to be purchased and used. It is well worth understanding the number of servers required and the infrastructure constraints. Space is often at a premium, and sometimes to reduce the number of servers there’s a constraint to consolidate and host multiple applications on the same server. This can have an effect on your solution design. In addition, you may need to buy additional infrastructure components and this can have a dramatic effect on costs. You will also need storage space for all your applications and this will need to be backed up too.

If all servers need to be housed in a central data center, you need to ensure that your team can access them. Your team could be spread across multiple locations, and their inability to access the servers via a remote access solution could be a problem. Furthermore, if work is going to be carried out across multiple locations, you need to know where the servers are located and define the appropriate connectivity. If all development machines are located in the U.K. and developers are located around the globe, it’s critical that high-speed communications and tools are in place to avoid inefficient teams.

Where multi-site development takes place, the processes and tools also need to support this. When different teams are using different processes it causes major headaches for the project. You also need to consider security because data transmitted out of the country could also breach regulations or the company’s internal policies. Even transferring data between networks could be a breach.

Time-zone differences can also cause problems for distributed projects. For instance, when should the backups kick off? It’s not that uncommon when you’re testing the system and see that it’s running slowly, only to find that a backup is running at the same time. It’s usually only after many hours of investigation that you find this out. Backups should be performed when no one is working on the system to ensure consistency and to not impact the project. It’s not just backups that can impact testing and distributed teams like this; anti-virus software and other background tasks can affect the team in

Part I: Production-Ready Software

similar ways. It's important to understand when background tasks will be executed and what the task's potential impact will be.

The following questions will help you to identify or define infrastructure constraints:

- How much hardware do we need?
- Do we need to buy specific hardware?
- Where does it need to be located?
- Is there enough room (and resources) at this location?
- When is it required?
- Who will be using it?
- What is it used for?
- How should it be installed?
- How should it be accessed and from where?
- How should it be maintained and supported?

There are a lot more questions that you could ask to determine whether there were any other specific constraints regarding infrastructure. However, the preceding list provides an overview of what you're trying to define and ensure.

Communication Constraints

We're all used to communicating by telephone, live messaging software, and e-mail in our daily lives. We're also used to surfing the Net to find out information and browse articles. A lot of companies block certain sites, as well as disallow live messaging software. Your organization may have its own messaging software (instant, e-mail, or both). Although e-mail is commonplace, a lot of organizations don't allow instant messaging software and web-cams for communication. Some companies don't even allow pen drives in the building. So, transferring and sharing large artifacts has to be done on the network, thus preventing them and any classified material from being shipped outside the organization.

Although these items might seem rudimentary, they can sometimes affect how quickly you can mobilize and get up-and-running. Effective communications need to be in place right from the start. This is especially true if the team is distributed across multiple locations and/or time-zones.

It's important to understand how the team and its members need to communicate with each other and to ensure that the processes and tools are available or alternative methods are defined and in place.

Environment Constraints

You need to understand whether any shared environment constraints need to be considered. It is often necessary to install the software in an environment that is shared with other projects. For example, the pre-production environment might be shared across a number of different projects and initiatives. It is important to understand and document the process for using the environment as well as the person or persons you need to contact to reserve timeslots within which you can perform your activities. The following questions help to gather and document environment constraints:

Chapter 3: Preparing for “Production”

- Do we need to share any environments with anyone else?
- Who is the main contact for the environment?
- How do we go about booking timeslots?
- How do we escalate issues with this environment?

Environment constraints can also include office and desk space. The number of “bums on seats” that you need may differ from the space available. There may also be a policy that prohibits desk sharing. You need to ensure that the project resources can be located effectively and that there’s room for expansion (where necessary). Location and security constraints are discussed shortly.

Technical environments can also be constrained because of infrastructure constraints. You need to ensure that you capture any and all environment constraints so that you can effectively plan and mobilize your activities. To do so, answer the following questions:

- Which environments do we need to use?
- How many of them do we need?
- What should they be used for?
- How should they be sized and scaled?
- How should they be used?
- How should they be maintained and supported?

Location (and Security) Constraints

Location constraints include any specific locations where activities *must* be performed. Location opening and access times can cause problems — for example, if access to the building is only available during typical business hours or access to the data center is only allowed outside of hours. These constraints could impact when certain tasks are performed.

It is important to answer the following questions to understand the locations of resources, the activities that will be carried out, and the tools that will be required and used:

- Who needs to be where?
- What are they going to be doing there?
- What hours can they be there?
- What do they need to do their job?
- How long do they need to be there?
- Are there any security constraints at this location?

The final location constraints are travel and visa requirements. In a globally distributed team, resources are often required to travel to different locations to perform various tasks. Traveling to certain countries

Part I: Production-Ready Software

requires a number of injections that often take time to organize and are spread out over time. In addition, visa applications can often take a long time to obtain. It's worth asking the following questions:

- Will this person need to travel? If so, to where?
- Will this require any injections?
- Will this person need a visa?
- Are there any specific visa requirements that could be a problem?
- How do we go about getting a visa?
- How long does it take?

You should plan for this accordingly, and it's also worth discussing with the individual, just in case he or she has any prior commitments or thinks it could be a problem.

Although there may be other constraints, the important thing is that they are all identified, documented, and well understood. Equally important, as you will see later, is that you do not need to know the answers to all the preceding questions to be able to proceed with a particular activity. Once you have a core set of constraints defined, you can start to outline the low-level processes and practices you need, and fine-tune them accordingly.

Development Mobilization

Development mobilization activities ensure that the actual development activities are completely defined and ready to use. The output of development mobilization feeds into a number of other mobilization activities. For example, once you've defined the inputs that you need for development, these will be fed into the design mobilization activities to ensure that they're accounted for. There can be a number of tasks carried out in this mobilization activity that could be split into parallel streams, where necessary.

The previous chapter looked at a high-level development plan and the activities involved. I also discussed scope in the previous chapters, so I'm not going to dwell on what has already been covered. Ultimately, the scope defines the inputs, approach, and outputs of construction. The next chapter looks at some of the inputs and outputs of construction. In this section, I'm simply going to provide some clarity for the scope and approach that will be taken during the formal development activities.

Defining the Development Scope

Initially, it is important to clarify the scope and approach of development. You need to ensure that the development team understands:

- What it needs to be doing
- How it's supposed to be doing it
- What it's using to do it

- ❑ When it needs to complete it
- ❑ What it needs to deliver at the end

It is important that all the artifacts that the development team is responsible for are clearly stated and understood. It really depends on the types of systems being developed, but the following provides a list of some high-level items that could be within the development scope for a medium-sized solution:

- ❑ **Technical designs** — It is important to understand whether the technical designs need to be signed off before development begins. If so, you might need to mitigate for delays. You could proceed at risk, which would need to be mitigated because the technical design could be reviewed and come back with an abundance of comments, which could have a dramatic impact on the overall development timescales. Having an agreed template in place for each of the component types in the solution will help. The template should contain all the relevant sections that need to be completed and the guidelines for completing them. You should consider developing a single technical design for each component type and ensure that it captures all the necessary information required by the development team as well as by the stakeholders. The initial technical designs will provide a basis for the remaining component designs and can be used as examples.
- ❑ **Database components** — Although most developers are somewhat familiar with the basics of database development, not everyone is a full-fledged DBA. There’s a lot of tuning, indexing, security, and other technical tricks that can’t always be performed within the development environment or by the development team. For example, identifying the index that should be clustered is generally easy, but determining how the tables and indexes should be split across the file system is something else. In these situations, it is important to understand whether the development team is responsible for making recommendations, implementing the actual solution, or simply providing support and input.
- ❑ **Framework and common service components** — The development teams could be split so that one team is working on framework components and the other is working on application components. It is important to have a clear line of division, to document who’s responsible for what, and to ensure that everyone understands their respective responsibilities. If the development team is responsible for developing framework components, this needs to be factored into the scope. It will also need to be factored into the solution design. If the development team is not responsible for framework components, which are probably inputs to the process, they will need to be very clearly tested, documented, and understood.
- ❑ **Integration with third-party components** — It is important that any integration with components developed outside the core development team be clearly understood. This even goes for framework components developed by another team. Major problems and delays can arise when it is taken for granted that something works or is supposed to work. More often than not, there’s a requirement to use a third-party product or a custom solution. If this is the case, then it needs to be thoroughly tested and prototyped beforehand to ensure that it can be used by the development team. If not, it may need to be simulated, which could take time to develop, especially if the product is complex.
- ❑ **Test stubs and simulators** — It is important to understand whether the development team is going to need to develop stubs and simulators. As previously mentioned, these can take a while to develop, depending on their complexity. Where possible, the stubs and simulators should be developed as early as possible and tested so that they can be used by the development team without undue impact. The development team will undoubtedly need to develop stubs for unit testing purposes.

Part I: Production-Ready Software

- ❑ **Application components** — The application components are typically the “bread and butter” of the application development team.
- ❑ **Batch components and scripts** — I discuss batch in more detail later, but it is important to understand whether the development team is responsible for developing batch processes and batch jobs. If the application requires batch jobs or background processes, it’s important to clearly define which jobs the development team is responsible for, how they need to be developed, and how they need to be tested. For example, it may be a requirement that all logs are shipped to a central system for auditing. First, the format of the logs would definitely need to be agreed to. Second, you’d need to understand whether it was a “push” or “pull” implementation, both of which have their implications on the development team.
- ❑ **Reports** — I discuss reports in more detail later, but, again, it is important to understand and document the reports that need to be produced by the development team and how they need to be implemented and tested.
- ❑ **Automated unit and integration tests** — It is very important to determine up front whether automated tests need to be developed. Furthermore, it needs to be noted where automated tests can’t be developed — for example, where they’re not supported by the applications being used. The development team should be developing automated tests and the level of code coverage should also be established early.
- ❑ **Test data and test scenarios** — Test data and test scenarios are often tricky. I mentioned in the previous chapter that common test data and scenarios should be used where possible. It’s important to ensure that all the necessary applications, tools, and environments are available to do this. It’s also important to ensure that changes to common test data don’t create a bottleneck in the process.
- ❑ **Configuration values** — You’re probably used to including configuration values in your software instead of hard-coded values. However, when it comes to deploying the software in other environments, the configuration settings need to be fully understood and calibrated for their optimum values. Consider the `MaximumQueueDepth` configuration value, for example. What is it? What should it be set to in the production environment? What happens when it is reached? Changing the value could have a dramatic effect on the solution and its performance characteristics. During technical testing, these will typically be calibrated for production. You need to understand whether the development team is responsible for making recommendations or simply documenting the settings and their ranges. In most cases, it’s the latter. However, you don’t want to get into tricky situations if someone were to ask “What should this value be set to in production?” It’s important to ensure that you know what you’re responsible for and when the values will be assessed and calibrated. The component may also have very specific configuration requirements which have been specified by the design team. For instance, the design team could specify that the low-level SQL components implement wait-retry logic and that the wait time and retry count are configurable. These values could be very different across the various environments. The wait-retry pattern is discussed in Chapter 19, “Designing for Resilience.”
- ❑ **Release, packaging, and deployment** — It is important to understand whether the development team is responsible for developing or even managing the packing and deployment solution. With distributed teams and differing time zones, it may be necessary for each development team to understand the solution, and thus manage releases and builds, rather than just submitting artifacts into a central process.

- ❑ **Release notes (including installation procedures)** — If the team is responsible for releasing, packaging, and deploying the solution, they’ll almost certainly be required to produce release notes. It’s important to establish the content of the release notes early to ensure that they contain all the relevant information.
- ❑ **Operational procedures** — It is vitally important to understand which operational procedures the development team is responsible for (if any) — for example, how and when the application should be backed up, which procedures need to be followed during a backup and restore, and what to do in failure scenarios. Writing operational documentation is a lengthy process and should be factored into the scope, if required.

Many more items could be included within the scope. The preceding list is only representative and should be used as a guideline. The primary objective is to define and agree on as much of the development scope, as early as possible, to avoid surprises and delays.

Clearly defining the requirements of each of these items will help to ensure that everyone is on the right track. You’re likely not going to get everything right the first time, but it’s important to at least understand what the team is responsible for and try to exercise the process early.

Defining the Inputs

I am going to discuss the inputs to construction in the next chapter (and throughout this book), so I won’t spend much time on them here. The following list represents just a handful of the possible inputs into the construction process:

- ❑ **Functional and non-functional requirements** — It is important that the development team understand the functional and the non-functional requirements, including the quality characteristics and constraints. Functional specifications might include (or incorporate) the functional requirements, and technical specifications might include (or incorporate) the technical requirements. For instance, a functional requirement such as “the system must support user security” can be incorporated in the functional specifications by including user administration pages and functions. However, it is sometimes harder to capture all the non-functional requirements, quality characteristics, and constraints in a single technical or detailed specification. For instance, where are you going to capture (and document) the constraint “the unit tests must achieve 100 percent code coverage”? Technical requirements and quality characteristics often affect a number of components in the solution. For instance, the requirement “the system must be capable of processing 1000 records per second” would need to be considered across a variety of components. All requirements and constraints need to be fully documented and understood by the development team.
- ❑ **Use-case diagrams** — A use-case diagram shows the primary actor (user) and the functions he or she can perform when using the system.
- ❑ **Use cases** — The use cases are typically a textual representation of the step-by-step flow and interactions between an actor (the user) and the system. The use cases also document the major alternative flows in the application’s functionality. It is very important in use cases to capture (and highlight) any manual steps that are involved in the overall process. These can often become the subject of debate later in the lifecycle.

Part I: Production-Ready Software

- ❑ **Navigation map / site map** — The site map is another visual representation of the application or website. The site map highlights the individual pages and paths to those pages. It can be used to highlight areas of inconsistency, as well as the journeys through the system.
- ❑ **Storyboards** — Storyboards are a sequence of illustrations often used in media, filmmaking, and other “visual” industries. Producing storyboards can be a tricky business when it comes to business applications, but I find them very useful because they complement the use cases. The storyboards walk through the end-to-end flow of the system and can be used to agree the functionality with the stakeholders, as well as highlight areas for further consideration, such as page transitions, caching, and additional application functionality.
- ❑ **Page layouts and specifications** — The page specifications show the individual pages (or forms) in either high- or low-resolution. High-resolution page specifications can be produced using development and modeling tools, whereas low-resolution specifications can be produced in any package that supports basic drawing functionality. The page specifications not only show a layout of the page, but they also document the controls on the page, such as how the controls interact with each other, and how the user interacts with the controls and what the controls do.
- ❑ **Style and usability guidelines** — The style guidelines complement the page layouts and, typically, include information about:
 - ❑ Standard fonts, styles, and colors
 - ❑ Standard graphics and positioning requirements
 - ❑ Standard usability criteria (ensuring tab order is correct, and so on)
- ❑ **Component models and specifications** — The component models and specifications are similar to the page specifications, although they document the components. The high-level component specifications describe the purpose of the component, whereas the low-level component specifications go into much greater detail regarding the component, its members (properties and methods), and usage. Component specifications can include as much or as little detail as necessary, including:
 - ❑ Component diagrams and hierarchy
 - ❑ Simple textual descriptions
 - ❑ Member outlines (property and method descriptions)
 - ❑ Class diagrams
 - ❑ Activity, sequence, or interaction diagrams
 - ❑ Configuration requirements
 - ❑ Logging and diagnostic requirements
 - ❑ Specific quality characteristics, such as automatic recovery in the event of a failure, wait-retry logic, or other specific patterns and practices

- ❑ **Report specifications** — Report specifications are also very similar to page specifications. Reports can contain lists of records, grouped records, charts, and graphs. A report specification would typically document the following:
 - ❑ The report selection criteria (and controls)
 - ❑ The output (along with sample results)
- ❑ **Batch job specifications** — Batch job specifications document various background or offline tasks that need to be performed. A batch job specification can be as simple as stating the basic function of the job. For example, it can state “The Remove Old Records job is used to remove the old records from the database.” However, this doesn’t describe what constitutes “old records,” although this may be described in the requirements. Batch is also discussed in more detail later in this book. A reasonable batch job specification would include:
 - ❑ The name and purpose of the job
 - ❑ Its processing rules and criteria
 - ❑ Its dependencies on other jobs
- ❑ **Data model** — The data model can be specified as logical or physical. The logical data model typically outlines the tables and the relationships (constraints) between them. The logical data model is described using business terminology and is technology (and implementation) independent. The logical data model doesn’t generally include the specification of indexes, views, triggers, and stored procedures. The logical data model feeds in to the physical data model design, which is an important distinction because the physical data model would typically include the following:
 - ❑ Tables and relationships
 - ❑ Indexes (clustered and non-clustered)
 - ❑ Views
 - ❑ Stored procedures
 - ❑ Security and roles
 - ❑ File system usage and storage requirements

It is important to understand and agree on who will be responsible for the design, development, and implementation of all the database artifacts.

- ❑ **Reference data** — The reference data typically encompasses the base data for the solution — for example, the items in a “Security Question” drop-down list or the product categories for an online store. The reference data is based on the data model, which may contain as little or as much data as necessary. The more data specified, however, the better. This saves you from having to make up base data for testing purposes.
- ❑ **Test scenarios and test data** — The test scenarios and test data include specific areas of testing that need to be covered and their input data, conditions, and expected results. There may be specific processing and calculations that must be thoroughly tested for which a series of test conditions and scenarios must be executed. You may also be sharing common scripts and data, and as such, they are an input into the construction activities.

Part I: Production-Ready Software

- ❑ **Third-party components** — It is important that all the components that the development team needs to use are captured in the scope and inputs. This includes specific usage scenarios and guidelines as well as any stubs or simulators required for thorough testing. The following list represents a few items that can be considered inputs:
 - ❑ Architecture and framework components
 - ❑ Integration components
 - ❑ Third-party components
- ❑ **Architecture blueprints and deployment diagrams** — It is important that the development team understand the holistic architecture and the overall deployment concept. These inputs can describe the architecture at either a logical or physical level. The blueprint will highlight the following components and how they are interconnected:
 - ❑ Client computers
 - ❑ Network components
 - ❑ Servers
 - ❑ Storage devices

These items help to flesh out the “requirements and design validation” steps in the construction process. The tasks should be complemented with checklists for each of the items to ensure that all the information is specified, and the associated query log should be used in conjunction with the checks.

Depending on the project and the team, the inputs will vary, but the preceding lists provide a good starting point. The key is to define and document the specific inputs that are required for each type of component and task, along with any required supplementary information.

This activity focuses on answering the question “Is there enough information to build the component?” If there isn’t, the inputs need to be revised.

Defining the Process

The previous chapter showed a high-level construction plan, which would also be part of defining the construction process. It also includes all the constraint groups discussed earlier, as well as all the activities on the plan. The important thing is to ensure that these are documented appropriately and understood. This includes any mitigating actions that might need to be taken into account to reduce risk — for example, if there are specific skill-level requirements, listing specific training courses that can provide the relevant level of skills. If there are potential resource bottlenecks, clearly identify them and state what actions will be taken — for example, that development will go on at risk, that other development activities will be performed, or even that training will be scheduled during the period.

Based on the technology constraints and the component groups, the following activities define the hardware and software required to effectively develop, unit test, and integration test each of the components in the scope:

- ❑ **Gathering and compiling the system requirements for all the applications and tools being used, and defining the development architecture and infrastructure** — Each development machine needs to be fit for purpose and support the applications and tools that are going to be used.
- ❑ **Developing software installation and configuration guides** — The software needs to be installed and configured appropriately. As new team members come on board, they can use these instructions to set up their development environment appropriately. Alternatively, the information can be used for managing the roll-out of the applications from a central source.
- ❑ **Developing usage guides for common tasks** — In many cases, these documents are included with the software. However, they should be complemented with your own, specific usage guides. In addition, the tools developed internally need usage guides.
- ❑ **Defining the maintenance and governance structure** — This establishes who maintains each tool and document. As previously mentioned, you’re probably not going to get everything right the first time, so be prepared to update and refine some of these activities. Setting out how this will take place and what needs to be updated will greatly improve efficiency.

These activities should highlight any specific workstation requirements for particular development activities. They should also highlight areas where shared databases or other shared infrastructure components are required.

The list is not exhaustive, but it gives you an idea of what to prepare for. I’d like to re-iterate that these items do not need to be all encompassing or overwhelming. Simple things that don’t require much effort can save time in the future. However, performing a number of simple tasks can add up in time and effort. If you’re going to do them, they need to be factored into the scope to ensure that you have sufficient time and money to implement them. In fact, that advice applies to all the subjects in this chapter.

Defining the Best Practices and Patterns

To mobilize construction, you must define the best practices and patterns. For each component group, the best practices and patterns should be documented (ideally using implementation specific examples), including the following items:

- ❑ Design guidelines (layering, encapsulation, re-use, and so on)
- ❑ Coding standards (naming conventions, formatting, and architectural patterns and practices)
- ❑ Commenting standards
- ❑ Exception handling and logging patterns
- ❑ Diagnostic patterns
- ❑ Common service usage (Data access, configuration settings)
- ❑ Other specific or unique patterns and practices
- ❑ Best practice unit and integration testing
- ❑ Testing tools and their usage
- ❑ Code coverage tools and their usage

Part I: Production-Ready Software

- Best practice (static) analysis tools and usage guidelines
- Best practice (performance) analysis tools and usage guidelines
- Documentation-generation tools and usage guidelines

As you can imagine, some of the tasks in this activity can take a while to implement. Therefore, start them as early as possible to ensure a reasonable development buffer. Use comments to document the custom-built tools so that the usage documentation can be generated from the code, which saves valuable time and effort when updates are required. For example, simply update the comments and re-generate the documentation. However, it is not always possible to do this, so you may need additional time to produce “formal” documentation for the tools.

Defining the Core Activities

Defining the core activities involves setting out the tasks for each component group. Defining core activities is nothing more than defining the most basic high-level tasks that each member of the team needs to perform. These tasks include the following:

- Installing and configuring the development environment
- Getting the most recent or a specific version of the solution from the source repository
- Compiling and building the entire solution
- Running the unit tests and verifying the results
- Running the integration tests and verifying the results
- Running the profiling tools and verifying the results

The preceding activities ensure that every developer can obtain, compile, and verify the solution prior to starting his or her daily activities. The following sets out some basic tasks that the developers need to perform as part of their daily routine:

- Creating new components (the term “components” refers to any and all items in the solution):
 - Creating the new component in the configuration-management system
 - Copying and renaming the exemplar template files (exemplars are discussed shortly)
 - Adding the component to the overall solution
 - Completing all the TODO items
 - Plumbing or configuring the component into the overall solution
 - Updating existing scripts and productivity tools to include the new components
 - Submitting the component into the build and release processes
- Developing tests:
 - Defining unit test/integration test conditions and scenarios
 - Creating unit test/integration test data
 - Creating automated unit/integration tests and adding them to the solution

Chapter 3: Preparing for “Production”

- ❑ Updating components and tests:
 - ❑ Obtaining component (and test) artifacts from the source repository (checking out)
 - ❑ Commenting changes
 - ❑ Verifying changes
 - ❑ Updating the component and test artifacts in the source repository (checking in)
- ❑ Documenting the completion of activities. The completion report would provide evidence that all the activities had been completed appropriately. For example, a completion report could include the following:
 - ❑ Evidence that all code review comments have been addressed
 - ❑ Confirmation that all queries have been successfully clarified and incorporated
 - ❑ Documented test results (including code coverage and profiling outputs)
 - ❑ Outstanding activities that need to be carried through to later phases
 - ❑ The actual time spent completing all the activities, which can be used to calibrate future estimates
- ❑ Submitting for (and performing) review and acceptance:
 - ❑ Peer reviews
 - ❑ Technical/group reviews
 - ❑ External reviews
 - ❑ Sign offs (including who signs off what)
- ❑ Submitting components into a release

The preceding are all very basic tasks, but if someone is new to a project or organization these are just a few things they need to know. When the development team ramps up and your organization takes on new developers, it will save a significant amount of time if you can simply point them to a Wiki where they can get all the information. Of course, you’re always on hand to assist them if they get stuck.

The last part in defining the activities is to define the daily routine. The routine should capture the specific actions needed to ensure that the development effort is consistent and streamlined. I’ve listed a few possible actions here, but it will depend on the individual project:

1. At the start of the day developers will always get the “latest” view from the version-control system, compile it, and verify it before starting any new work. This ensures that at the start of the day every developer is working from a single, consistent baseline. There may be exceptions to this rule when a developer may require a specific version.
2. All code must be backed up overnight. This protects the project and the development team against loss or damage. The source code repository will invariably be backed up far more frequently than developer workstations. This means that each developer needs to check in (or shelve) his or her code at the end of the day. This also protects the development team from not being able to access vital files when the developer, who has them checked out, is not available. Tasks that spread over multiple days need to be managed. Although the code may not be “ready to build,” it needs to be safely stored overnight to protect against loss or damage. Losing work is

Part I: Production-Ready Software

costly to a project. Then again, it is costly to have team members sitting idle because they can't access the files they need. This problem needs to be managed, along with the next point, because the code must compile at all times. There's no perfect solution, but some version control systems support "parking" or "shelving" code and artifacts. This allows developers to shelve their work on the source control system and have it backed up without having to check it in.

3. All code is verified against the latest version prior to being checked in. This helps to protect the development team from getting code that does not compile or build. Checking in code or artifacts that do not work or do not compile may disrupt development, causing downtime. The development process and practices should include clear instructions on how to use the configuration management-system and what to do prior to checking in anything. Breaking the build is painful and costly, so all developers must do what they can to ensure this doesn't happen.

The basic idea is to try to document as much as possible to ensure that each and every developer has the necessary information to get on with the actual business of developing software. If the project is producing a formal pilot, this can be a great time to prove all these processes and the others in this chapter.

Defining the Templates and Exemplars

During mobilization it's important to ensure that each output artifact is outlined appropriately in the scope. For instance, if you're going to be producing technical designs, what sections should they include? Should they include class diagrams and complete sequence diagrams? Should they contain pseudo code?

An exemplar is a fully completed unit, whereas a template is a shell that needs to be completed. I find that defining initial templates, completing them, and going round the loop a couple of times naturally creates an exemplar and a template at the same time. Another difference between an exemplar and a template is that a template normally contains `TODO` statements, whereas an exemplar doesn't because exemplars include and follow all the best practices and patterns. The exemplars provide the basis for "real-world" references for the team instead of it being hypothetical. Reusing exemplars across projects can greatly improve productivity and development time.

Where possible, at least one of each of the items in the scope (where required) should be developed up front to provide the basis for the exemplar and template pattern. If the outputs are documentation-based, ideally there should be at least one exemplar or template document per group that contains all the relevant sections and instructions for how to complete them (if it's a template). This can then be agreed on by the consumer to avoid delays later in the project.

If the outputs are code-based, the exemplars and templates should ideally cover the following:

- One or more exemplar/template per component group
- One or more complete unit test exemplar/template per component group
- One or more complete integration test exemplar/template per component group
- One or more complete smoke test exemplars/templates

This output documents and indicates what the developer should do to complete his or her work.

Chapter 3: Preparing for “Production”

You should use comments to fully document the exemplars and templates, which will save time producing documentation. You can even generate all the usage guides from the exemplar and template code comments.

Producing the exemplars and even the templates will help to ratify the process end-to-end and will act as the foundation for the actual construction process. As you produce each of the exemplars and templates, you’ll also start to identify improvements and refinements to the inputs.

The development mobilization activities take each of the exemplars through the process to prove not only the process but the exemplar, as well, and to flesh out as much detail as possible. Not only is there a fully documented and functional development process at the end of it, there is a set of exemplars and templates that are used for reference purposes and that provide the basis for all new components. If any innovative approaches are identified, they should also be adopted to reduce overheads.

Wrapping It Up

Finally, wrapping up construction involves defining the exit criteria and the outputs from construction. As you’ve seen, development completion is a major milestone and all the artifacts need to be brought together so that you can show your results.

Closing off development starts by defining the contents of the completion report and matching the contents against the exit criteria. The process outlined in the previous chapter showed the individual developer completion summaries. The following provides a high-level list of items that can be included in the overall construction completion report:

- Input inventory
- Query log inventory
- Technical design inventory
- Component inventory
- Unit test inventory
- Integration test inventory
- Code coverage statistics
- Code profiling statistics
- Defect summary
- Financial summary (estimates vs. actual)

In addition to the preceding, there are, of course, the final releases themselves. You should keep the development completion reports and artifacts under version control so that you can refer to them at a later stage, if necessary. It’s important that everything is included in the report, including the release, so that a clean machine can be used to verify the contents and even compile, build, and run everything.

While all these activities are going on, you need to document any and all known issues or areas for improvement to move forward. If something adverse was seen during the mobilization activities and hasn’t been specifically fixed, it should be noted and assessed because the last thing you want is it to happen again when there’s potentially a number of developers impacted.



Part I: Production-Ready Software

All these activities are prime examples of where an agile or rapid development approach can be adopted. The activities should be done incrementally so that at every stage there is a working version that can be reviewed and taken forward as, and when, required.

Ideally, all this information would be available on the project portal so that every developer has easy access to the relevant documentation and associated process diagrams. The primary goal is to ensure that as many aspects as possible are covered. If there is a gap, plug it with a process and tool or guideline. The path of least resistance is generally the easiest, so let's concentrate on developing and not having to waste time with undefined or inefficient processes and tools. These activities not only help to build better quality software, but the documentation can be re-used again and again when handing over to live service and the application maintenance teams. This saves a huge amount of time and effort.

Thus far, this chapter has walked through development mobilization in quite a lot of detail. The remainder of this chapter is dedicated to the other key mobilization activities, including:

- Build, release, and regular integration
- Test
- Fix (defect management)
- Configuration management
- Change control

Build, Release, and Regular Integration Mobilization

Although build and release can be considered as two separate entities (that is, the tasks can be separated and executed in parallel), I've chosen to include them together. The purpose of build, release, and regular integration is to take all the development artifacts and compile them into a single release.

Releases will have different requirements, so they will have different packaging inputs. The build and release processes should take what has already been defined and execute all the best practice tools, as well as all the unit tests and integration tests, thereby providing the initial regression testing capability.

Builds should be done on a regular basis to bring everything together. The build cycle and inputs change as the project progresses. Initially, the latest version of the solution is taken into a build. As testing starts, a more structured approach to individual fixes or groups of fixes needs to be adopted. The build and regular integration processes should define the relevant approaches.

The activities that will be conducted are similar to those in the development mobilization phase, so I am not going to go into details that have been previously discussed, but instead I'll focus on the specifics that relate to build, release, and regular integration. The following describes the key activities involved:



- ❑ **Defining the process constraints** — The build process constraints should list all the tools and technologies that will be used to build and produce releases, as well as the packaging solution.
- ❑ **Defining the architecture** — The tools required to build, regression test, and release software often require a lot of configuration, so it’s worth starting this process very early. Furthermore, the custom tools that are needed to install certain components should be developed as early as possible. For instance, creating and installing the database might involve a custom tool to execute various scripts. The architecture should include everything that is required to get the artifacts out of the configuration-management system, compile them, package them into releases, test them, and, finally, deploy them to an environment. In some cases, this also involves running post-installation scripts that change various values in configuration files or the database (e.g. environment-specific values). The build and release architecture components need to support silent installation or configured installation to streamline deployment into large-scale environments. This saves you from having to install a release manually. Ideally, the high-level (automatic) actions that should be supported include the following:
 - ❑ Gathering all the artifacts from the source repository (that are Ready-to-Build or Ready-To-Release)
 - ❑ Compiling the solutions according to their dependencies and release configurations
 - ❑ Statically profiling the code and outputting the results
 - ❑ Generating the documentation from the comments
 - ❑ Executing all the unit tests and validating the results
 - ❑ Executing all the integration tests and validating the results
 - ❑ Executing any additional regression tests and validating the results
 - ❑ Measuring code coverage during testing and outputting the results
 - ❑ Dynamically profiling the code during testing and outputting the results
 - ❑ Packaging the components into their respective releases (Ready-to-Release)
 - ❑ Running as much in parallel as possible
- ❑ **Defining build scope and inputs** — The build scope defines the components and artifacts that are taken into the build process. The build process compiles all the necessary artifacts into binaries. The release will generally contain the resultant binaries, although some releases may include the original source code files (for example, if a code review might need to be performed). It is a good idea to identify the components that require compilation or other prerequisite steps prior to being included in a release. For example, stored procedures are usually stored as text files and are not compiled as part of the build process. However, they may be required to generate code, so a build database will need to be implemented first so that the code generator can be executed and the resultant code can be included in the build.
- ❑ **Defining release scope and inputs** — The release scope defines the components and artifacts that are taken into the packaging process. The release scope maps the items in the configuration-management system and maps the outputs from the build process to a particular release configuration. Releases contain a number of different items, as I’ve mentioned before, and the

Part I: Production-Ready Software

purpose of the release scope is to define and implement the packaging of these into their associated releases. The following provides a starting point for the different types of releases:

- Unit and integration test releases
- Functional test releases
- Technical test releases
- Acceptance test releases
- Review releases
- Production releases (including disaster recovery)

Although typically the items included with a release differ, each release contains the specific configuration and artifacts for its given purpose. The release should contain everything that goes along with it, including the following:

- Documentation (including generated documentation)
- Test scripts and test data
- Configuration files
- Database scripts
- Post-installation scripts and other productivity scripts
- Custom tools (environment-specific tools, test tools, and so on)
- Binaries and libraries
- Source code (only where appropriate or required)
- Release notes
- Defining ready-to-build/ready-to-release processes** — In terms of the build and release, the ready-to-build process dictates how files and artifacts should be labeled or marked in the configuration-management system so that they can be easily extracted by the build process and included in a build. Often the latest files are taken. If you are using continuous integration tools and techniques, the very act of checking in a file will trigger a build. These should be complemented with the developer procedures and practices to ensure that only pre-approved artifacts are submitted to a build or release. The ready-to-build process goes hand-in-hand with the ready-to-release process. Just because something is ready to build doesn't mean that it is ready to release. For example, a large development piece of work may contain many different elements, each of which will be ready to build at different times, although they should all be released at the same time. The high-level activities include:
 - Defining the base-line ready-to-build process
 - Defining the defect/change specific build process
 - Defining the base-line ready-to-release process
 - Defining the defect/change specific release process
 - Developing and configuring the build and release tools

The build and release process will be used throughout the lifetime of the project, so it’s worth getting it as good as possible. It will be handed over to support and maintenance, so a well-documented and easily understood process helps greatly.

Test Mobilization

The test mobilization first needs to focus on the unit and integration testing, which is followed by the initial regression testing (which encompasses the unit and integration tests). Then, it is followed by the additional test activities of the project (functional testing, technical testing, and so on). Smoke tests are often included in the regression tests, which mirror certain functional and technical tests. This allows the regression test capability to be enhanced throughout the project to include more tests and scenarios into the overall suite.

Again, the activities are very similar to those discussed previously, so I am only going to concentrate on the specifics around testing. The quality processes and review processes should be defined as well as the specific activities.

- ❑ **Defining process constraints** — The test process constraints should list the tools and technologies that will be used to test the application. Testing involves a number of different tools used for various test scenarios. Each testing activity uses a combination of test tools, which generally includes:
 - ❑ Unit testing tools
 - ❑ User interface testing tools
 - ❑ Load or stress testing tools
 - ❑ Custom test and verification tools
 - ❑ Actual results and an expected results comparison
 - ❑ Requirement/test condition and test script management tools
 - ❑ Test execution and progress reporting tools
 - ❑ Defect management tools

It is also important to understand, define, and document the testing activities that will use the defect-tracking system. Typically, unit testing doesn’t track the defects because the developer normally runs the tests and corrects the defects. It is more likely that test defects are tracked, starting with integration testing.

It is important to ensure that, wherever possible, the testing activities are performed with “verbose” logging turned on. This greatly enhances the fix analysis and turnaround times and it alleviates defects going backwards and forwards.

Part I: Production-Ready Software

- ❑ **Defining the architecture** — In addition to the overall testing architecture, there are four specific areas of interest:
 - ❑ **Regression testing** — The regression testing architecture should be capable of executing a variety of tests (and tools) so that as much hands-free testing as possible can be performed on a release prior to it actually being released. The regression architecture needs to be extensible so that additional tests can be added easily. This allows the regression capability to be extended, including more scope and project progresses.
 - ❑ **Test stubs and simulators** — In the testing architecture, specific stubs or simulators are required when actual components are not available or otherwise difficult to test. The stubs and simulators should mimic the real components as much as possible, as well as provide specific scenario-based functionality. For instance, it may be possible for the real component to raise an exception under certain circumstances. Thus, it is important for the stub to be configured by a test scenario to work the same way. However, a lot of time and effort can be spent investigating and correcting issues with complicated stubs and simulators, which are really only used to test the application. Your focus needs to be on testing the actual application and ensuring that it interacts with the real component. Keeping stubs and simulators fairly simple avoids issues with using them. Whenever third-party components or integration is required, it is vital that these touch points are tested as much as possible and stubs and simulators can sometimes help to focus the testing effort.
 - ❑ **Custom test tools and scripts** — Another important part of the testing architecture is the custom test tools. The custom test tools should integrate into the regression architecture and not require human interaction, where possible. It's often very easy to build custom test tools that have a user interface, but because this requires manual input, test cycles need to be executed manually. Scripts are often required to “wrap” and integrate with the test tools. For instance, after each test, you gather the log files, diagnostic outputs, and database updates, and then place them somewhere safe (so they're not overwritten by the next test run). Other scripts may be used to gather all the test results and store them in the repository.
 - ❑ **Actual and expected results comparison tools** — It is also very important to consider how actual results are gathered and compared to expected results, which often involves custom development or customization of third-party tools. Some actual results are much easier to extract than others. The extraction and comparison tools should also be fully automated. However, this adds additional complexity to the tools, which is another important reason for starting ahead of time and identifying them, purchasing them, or developing them in-house.
- ❑ **Defining the test scope** — The test scope is usually derived from the lifecycle and lists the types of testing to be undertaken. The following list reiterates the testing activities that have been discussed:
 - ❑ Unit testing
 - ❑ Integration testing
 - ❑ Regression testing (including Smoke testing)
 - ❑ Functional testing
 - ❑ Technical testing
 - ❑ Acceptance testing

Another important part of the test scope is to ensure that all the various component groups are included within the test scope. For example, if the application has batch components or reports, these need to be fully tested as well. You’ve already seen the component types and groups that will be developed. It is important that these are all mapped so that the scope of each set of tests is clearly defined. As a best practice, unit and integration testing should include a broad range of scenarios to identify defects early. For instance, performance testing a method gives an indicative timing that can be verified against the overall performance requirements to determine whether it is/or could be a potential bottleneck in the end-to-end process. If these tests are relatively straightforward to develop, there’s no reason why they shouldn’t be included. I’m not saying that they are the be-all and end-all, but they can help. Similarly, testing batch components with a realistic set of data gives an indication of whether the component needs to be re-factored or at least support configurable parallel processing. If running a job on a developer workstation takes an hour, something probably needs to be addressed.

- ❑ **Defining the test scenarios** — The test scenarios map to the test scope and the component types, defining the high-level test groups that will be conducted. The test scenarios map to the quality characteristics, including the following examples:
 - ❑ Correctness tests (both positive and negative)
 - ❑ Performance tests
 - ❑ Failure and recovery tests
 - ❑ Monitoring tests
 - ❑ Operability tests
 - ❑ Usability tests
 - ❑ Penetration tests

The test cycles determine the high-level grouping for the individual test scenarios and conditions.

- ❑ **Defining the test inputs** — The test inputs are similar to those listed in the development section, such as requirements, use-cases, and so forth. The inputs map to the test cycles and test scope. However, different test activities will require different test inputs, and listing these will help to define the input verification criteria as well as define the entry and exit criteria.
- ❑ **Defining entry and exit criteria** — The entry and exit criteria stipulate the conditions that need to be met before the activity begins and before the activity is complete. Entry and exit criteria are entirely dependent on the project and the activities that need to be performed. The following are some common exit criteria and the potential issues with using them:
 - ❑ **Percentage of code covered** — As mentioned previously, I don’t actually believe this to be a true measure of quality. If only combining functional and technical tests achieves 100 percent code coverage, then you have a winner. It means that your tests exercise the entire code-base and that’s just grand. Achieving 100 percent code coverage elsewhere only shows that you have a set of tests (and stubs) that exercise all the code, not that all the code is actually used in the final solution. However, it may very well be a constraint to achieve 100 percent code coverage in unit test.
 - ❑ **Number and complexity of scenarios and/or conditions successfully executed and passed (or outstanding)** — This stipulates the number or percentage of test scenarios that must be completed — for example, 50 percent complex, 75 percent medium, and 100

Part I: Production-Ready Software

percent simple. The numbers can equally be reversed. However, if reaching the exit criteria comes down to “pairing off” two complex scenarios, the easier of the two is the most likely place to focus all the test (and/or development) effort. Clearly, the remaining scenarios and conditions need to be addressed at some point.

- ❑ **Number and severity of defects open** — This is often used when an activity can end with only a certain number of defects open — such as, no critical, no high, x medium and y low. Clearly, both x and y need to be addressed at some point.

This scenario is very similar to the previous one. To explain, let’s assume that during a functional test activity, a “low” defect can be a simple spelling mistake on a page. Let’s also assume that a “low” defect could also be one that only happens under very extreme circumstances. Now, assume that you can “exit” with 30 “low” defects. If there are 31 open at the time you need to exit, what’s going to happen? The most probable action is to fix the defect with the very least amount of development effort. In this instance, it’s going to be the spelling mistake. The fact remains that the other defect could have far greater consequences later on. This measure depends on how defects are classified — a subject discussed in greater detail shortly.

- ❑ **Specific transactions executed and passed** — This is a good measure when you want to ensure that particular functionality or routes through the system have been clearly exercised. The transactions may not include all the individual permutations, but they’ve tested the flow in a variety of scenarios. Regression “smoke” tests are one such activity where this criterion is applied.

As mentioned previously, no system is completely free of defects. In order to start a particular test phase, you want to be in a position for it to continue without being adversely impacted. Thus, the earlier you identify more defects the better. The chosen methodology determines the number of test activities and when they are performed in the lifecycle. Each test activity has its own agenda, and the entry and exit criteria needs to be carefully considered and defined. Putting the appropriate entry and exit criteria in place helps to define clear boundaries for each testing activity. That still doesn’t mean that the test phases won’t overrun and overlap with each other. The more that you can do up front to ensure quality outputs from the construction page will help to reduce the testing downtime.

- ❑ **Defining the test completion report** — This is similar to development completion because testing completion complements development. All the testing artifacts need to be brought together into a single package for review and sign-off. The following provides a simple list of what might be included in the test completion report:
 - ❑ Inputs inventory
 - ❑ Test condition and scenario inventory
 - ❑ Test execution statistics (tests executed, passed, and outstanding)
 - ❑ Defect summary
 - ❑ Financial summary (estimates vs. actual)
- ❑ **Defining the test exemplars and templates** — The test mobilization activities should implement a series of exemplars and templates that exercise all the testing capabilities and tools. The exemplars should be organized and grouped according to the test scope and activities. They should also contain multiple scenarios and conditions that exercise the various test data and the actual and expected results comparisons. This level of coverage will greatly improve the overall

efficiency when the actual testing begins. You should be able to reduce the number of issues encountered and get straight to the real bugs.

- ❑ **Defining the test data and management approach** — The test data and associated management is critical to all test activities. As testing progresses, the test data is added to and improved. Test data management should make this easy, as well as deployment and installation of test data in the environments. The test data starts getting defined right back at the start of the project. Capturing as much test and reference data as possible provides a great start, but managing the data for the different test activities and cycles is somewhat harder — the sheer volume and the different combinations can be unwieldy. There are many environments, each of which will probably involve multiple activities and scenarios. The tools and processes for managing the test data need to be piloted very early, which is why test mobilization starts way up front. That doesn’t mean it’s complete; it just means you start it very early to define practices. The following are the high-level items that you should consider:
 - ❑ Defining the test data management approach
 - ❑ Defining the base reference data
 - ❑ Defining the base test data
 - ❑ Developing test data installation tools and usage guides

The test data should be based around real data and transaction scenarios. You should also test using expected production volumes of transactions and data. For instance, if the production environment is going to be potentially processing millions of requests or transactions, the test data should contain a similar number (if not more to really stress the system). There are tools that can generate test data and the use of these should be considered. Using live-like test data right from the start (unit test) can really help to flush out issues. However, it does also mean that there could be very large data sets, which could be a problem in smaller environments. The key is to use as much data as possible, as this will stress the system and provide more indicative performance figures.

Common test data should be identified and used across multiple phases of testing to reduce the overhead of both installation and management. As the project progresses, the situation will only get worse, and having to make multiple changes and script updates will severely hinder progress.

- ❑ **Defining rapid turnaround approach** — The lack of rapid turn-around scripts is often something that gets in the way of testing. There is often a need to turn around an environment from one test activity to another. Regression testing will exercise the release using various test data sets and scenarios, including going from functional testing to technical testing. This is usually performed by installing a release with a different configuration and underlying test data. Because this can often be a time-consuming exercise, it is better to have a rapid turnaround approach that reduces the overall time between installations and configurations.

Taking a set of exemplar test scenarios and conditions end-to-end will define the best practices and uncover areas where further tooling may be required to streamline the activities even more. In the end, your results will be used by all the testers (including developers). In addition, you’ll create a series of exemplars and templates that can be used by all. These will also be handed over to support and maintenance, where they will be well-defined which will help ease the transition.

Defect Management (and Fix) Mobilization

Defect management (and fix) mobilization is going to be very important. As you've seen, once the formal test activities are initiated, you're going to see a number of defects raised. However, defects can be raised at any point during the project lifecycle. For example, designers can raise defects against the requirements; developers can raise defects against the design; and testers can raise defects against the solution. Of course, a lot of this depends on the methodology and approach.

As defects are encountered and raised, they need to be tracked, verified, and resolved. Defect management also encompasses management reporting. At various points throughout the project it's important to understand the number of "open" defects, their severity and priority, and so on. The reports help to shape how the project progresses. If entry and exit criteria are based on the number and severity of defects, the reports will help to compile the current picture and provide status reporting. It's important to understand the reporting requirements because they will help to shape the defect-tracking system and guide how it should be used. Each team may also have specific reporting requirements.

While individuals normally raise defects, it is possible that your tools can also raise defects automatically. For example, the regular integration system can raise a defect automatically to indicate a broken build, or the regression test tools can automatically raise a defect indicating the tests have failed. It's important to understand these integration points and the requirements of the defect-tracking system's Application Programming Interface (API). Custom reports may also need to make use of the API.

This section looks at the high-level defect management, fix mobilization activities, and some of the elements required to track and manage defects successfully.

Defining Process Constraints

The defect-management process constraints should list the tools and technologies that will be used for tracking and managing defects throughout the project. This will mainly consist of the defect-tracking tool as well as any custom or other tools that surround it. The defect-tracking tool may also need to integrate with other tools and applications, which need to be listed and accounted for.

The defect-tracking system will invariably require customization that is specific to the project. Using the out-of-the-box configuration can sometimes be like "trying to fit a square peg in a round hole." The process is defined by how the tool works, rather than how the process should work. Off-the-shelf tools typically allow customization of defect contents (the data elements) and workflow (state transitions). You need to ensure that the defect-tracking system is fit for purpose and that everyone understands how to use it properly.

The size and scale of the fix team depends entirely on the project and the technologies being used. I typically recommend that the fix team start at 100 percent of the development team size. However, if you've done everything you can to capture and fix many defects early, this percentage should be

greatly reduced, very quickly. As the project progresses, this can be further assessed and reduced, as appropriate. The fix team doesn't need to be standing idle waiting for defects to be raised, it can be working on future enhancements or releases. However, when defects are raised, their priorities over existing work needs to be managed accordingly. If an agile approach is taken, the development team and fix team is usually one and the same. It's important to ensure that the defect-tracking system supports all the required users.

Defining the Process

As the project progresses, the requirements on the defect-tracking system will change. In the early days, it's all about the tracking of defects raised by the design, development, and test teams. Later on, it's about which defects are in which releases. Then, finally, it's about providing project wide management defect information. In the run-up to go-live, stakeholders will be particularly interested in the number of “open” defects, what they are, and whether they affect the go-live date. They'll also be interested in ensuring that *all* other defects raised throughout the project have been closed and verified appropriately.

Defect management can be quite stressful at times, especially when testing is blocked and defects need to be turned around quickly. Having a clearly defined process in place will help to ensure that defects are tracked and managed efficiently. Defining the process involves the following high-level activities:

- Defining the workflow and state transitions
- Defining the contents of a defect
- Defining the management report requirements

As you progress through these sections, you'll get an idea of what's involved, as well as some of the best practices for defect management.

Defining the Workflow

The defect-management workflow must have the appropriate transitions and mandatory entries to ensure that the correct procedure is followed. You must also ensure that the transitions support the appropriate alternative flows to avoid unnecessary overhead and confusion.

The defect-management workflow follows a defect from the point at which it has been raised to the point at which it is resolved (or rejected). Because defects can be raised at any point in the project, the people involved in the activities can be members of any one of the teams in the project, and the defects can relate to any of the parts of the solution. For example, a technical designer might raise a defect against a functional design document. It's important to remember that defects aren't just raised against code. To help with defining the workflow, let's look at some high-level, “popular” usage scenarios:

- Starting with the most popular, testers raise defects during testing and verification. The system has been through construction (including unit testing, integration testing, and whatever else led to this), and it's being tested as part of a formal test activity.
- Developers may raise defects against the solution design or any of the inputs into the construction process. For instance, a use case may have a missing step, or in the worst case, the design may not be able to be implemented as stated. If the development team is using third-party

Part I: Production-Ready Software

components (for example, one they're not responsible for developing), then it's likely that they'll be raising defects when the components don't work correctly.

- ❑ The development team may raise defects against its own components. Formal defect tracking isn't generally used during unit testing. However, during integration testing there could be a problem with another component. Developers may even notice something in the solution that they think needs to be addressed. For example, they may come across a class that they think hasn't been implemented correctly or needs to be re-factored. This is a fairly common situation during development and fix. Furthermore, the defect may have nothing to do with what they are currently working on or even responsible for.

The actors work together in a customer- and supplier-style relationship. The customer raises the defect, while the supplier deals with it, although the customer and supplier can be the same team, as you've seen in the preceding scenarios. There are many other scenarios, but the focus here is primarily on development and fix. You can extrapolate from the examples and apply them to other areas of the lifecycle. The scenarios listed are very common and each has its own specific considerations for management.

Before looking at the workflow, let's look at a very simple scenario to set the scene:

1. You click on the Create New Account link.
2. The Create New Account page is displayed.
3. You verify the page layout and contents, and you notice a defect:
 - ❑ Defect 1: There is a missing "mandatory field indicator" next to "Security Question."
4. You start to complete the form and notice some more defects:
 - ❑ Defect 2: The tab order is all over the place.
 - ❑ Defect 3: The username field allows only 15 characters, instead of 16 as stated in the entry tip or hint.
5. You click the submit button and are presented with a technical error:
 - ❑ Defect 4: Technical error on submission.
6. You are now unable to continue with your script. Out of due diligence, you check the log files. You also try a couple of transactions on some other pages, and they all work fine. You head off to get a coffee while rebooting the entire system. You return and uninstall and re-install the software and run through the scenario again. The same thing happens again. You lean over to Sarah who is also testing and ask her to quickly run through the same scenario. Aha! It happens on her machine, too. This must be a defect!

Figure 3-4 shows a high-level sample workflow that highlights the common activities and transition points.

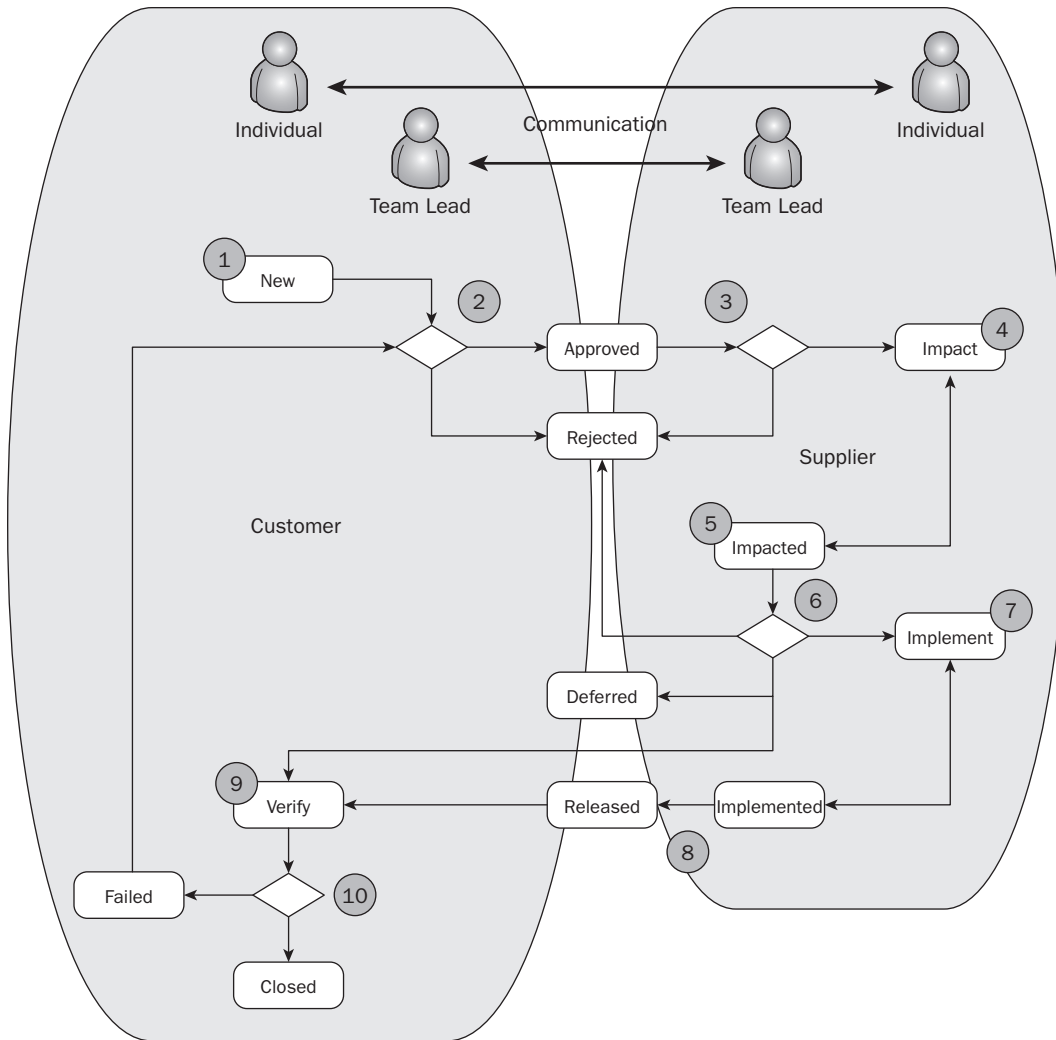


Figure 3-4

I’ve simplified the diagram so that it really only shows the actors and the main points in the workflow, rather than all the alternative flows that could be required. The workflow steps are as follows:

1. **New** — The defect is raised by an individual (or it could be a sub-system) and is given a basic state of *new* or something similar. When defects are raised, it is crucial that they contain all the relevant information so that you can estimate, locate, and fix the issue. The following is some of the key information that should be captured:
 - Exact steps to reproduce
 - Release and associated configuration installed

Part I: Production-Ready Software

- ❑ Test script, test data, and inputs
- ❑ Screenshots of the end-to-end process
- ❑ Logs (verbose, where possible), events, and other diagnostics
- ❑ Database extracts (including reference data and transactional data)

The information listed is generally related to code-based defects. Tools and scripts should be developed to assist with obtaining this information so that it can easily be added to the defect. These tools and scripts will also be useful during any incident investigation.

For management reporting and tracking purposes, when the defect is raised, it should include the *activity* in which it was detected — for example, “Integration Test,” “Functional Test,” “Technical Test,” or “Acceptance Test.” When these activities are broken down into separate streams, the stream should also be included — for instance, “Operability Test,” “Performance Test,” and so forth. These can easily be obtained from the approach and the plan.

Although it is very possible that the exact nature (root cause) is known, this should be managed carefully when raising defects. It’s easy to state something potentially incorrect or incomplete — for example, “The data model must have indexes.” Well, yes, it probably should, but which data model? The tools may have an underlying database. Does this mean that they also must have indexes? It would probably help. It’s therefore important when raising defects that specific *components* be identified (where possible). Generic defects can be useful, but they often require a lot of analysis. For example, consider the infamous “The logging is insufficient” or “The error handling isn’t good enough” generic defects. A large-scale solution could have hundreds of components, perhaps thousands. These kinds of defects linger; you can never really close them properly until the system’s gone live. Even then it could still be a problem in certain cases. Where possible, the defect should be descriptive and cite specific cases. Another common scenario is stating the resolution (or stating an incorrect resolution) — for example, “The *for* loop in *xyz* should be replaced with a *while* loop.” Although it may be true, there may be a very good reason for the way something has been done. If someone doesn’t know any better, he or she may simply implement this change and find that the solution no longer works the way it should. A detailed impact analysis *should* determine the resolution. The last one that really grinds my gears is asking questions (and having conversations) through the defect-management tool — for example, “Can the startup scripts be made easier?”

The defect should state what the problem is and how it was encountered. The impact analysis will state how to resolve it, the exact root cause, and the activity that introduced it (design, construction, and so on).

Guidelines should be developed to assist people when raising defects to avoid unnecessary cleansing and potentially sending them around the loop for no reason at all. These are all part of the defect management best practices. The approval process should ensure that all the necessary due diligence has been performed and the defect is good to go through the system. Defects are often visible to the entire project team (including stakeholders), and it’s important that people raise defects in a professional manner. Stating that “This system needs re-writing” isn’t professional. Defects are very different from changes, and proposed changes to functionality should be raised and discussed outside of the defect-tracking process. Throughout the project it is often necessary to control *who* can raise new defects.

Finally, when defects are raised, they’re typically assigned a severity. You must have very strict guidelines on how to assess and assign severities. Defects may also have a priority and again it’s

very important to have guidelines for assessing and assigning this, too. This should avoid the situation where everything’s a “Sev 1, P1.” Severity and Priority are discussed shortly.

2. **Approval** — All defects raised should be validated by an authorized approver. In most cases, this would be the individual’s team leader. The following represents a handful of tasks the team leader might perform:
 - Ensure that the defect conforms to the standards and guidelines.
 - Ensure that all the relevant information is specified, correct, and contains valid values.
 - Ensure that the defect has all the relevant attachments.
 - Validate that there’s sufficient evidence that points to a defect — for example, it’s not operator error, or the script, data, or environment. Furthermore, the team leader should do his or her best to ensure that it’s not a change to the requirements or design.
 - Verify that the “severity” (and possibly “priority”) is assigned appropriately, and if not change them as necessary (according to the appropriate process). The priority may be affected by other outstanding defects that are higher priority.
 - Reject the defect. There should be very strict guidelines for approving defects. Where a defect doesn’t meet the strict guidelines, it should be rejected. The guidelines should include all necessary steps, as well as any other specifics.
 - Approve the defect. The approval process should be clearly documented and understood to avoid debate. The defect should have all the necessary information and it’s important to correctly identify the phase or activity in which the defect is raised.
3. **Approved** — The “approved” defects are taken into the “supplier” workflow (for example, fix). The fix team leader reviews the defect. The review follows the same steps as outlined earlier. The customer and supplier can work together on this to ensure a consistent picture and avoid many “rejected” defects. The defect is assigned to an individual or team for *impact analysis*. This step may involve discussions with multiple teams to determine the best person or team to perform the impact analysis. At this point in the process, it’s good to get an idea of the *estimated time to impact*. Although this will be high-level, it provides a basis for planning. It will also help with the management reporting and financial summaries at completion, as well as continuous improvement.
4. **Impact** — The individual (or team) that the defect has been assigned to performs an impact assessment. The impact process involves reviewing all the information on the defect, re-creating the issue, and providing an assessment on how to resolve the issue. In many cases, the hardest part of impacting a defect is re-creating it. All too often there’s not enough detail included in the defect report, which is why you’re really trying to ensure that they’re properly explained. The productivity scripts will really help to ensure that all the information can be easily gathered and attached to the defect. Even with all this information, it might be necessary to work with the individual or team that raised the defect. This is indicated on the workflow diagram as “communication.” The fact is that sometimes people just don’t talk to each other, which can lead

Part I: Production-Ready Software

to one of my pet peeves, mentioned previously: communicating through the defect-tracking tool. For example:

```
[10:00][Developer] I can't reproduce it. Can you send me the logs?
[10:05][Tester] I've attached logs.
[10:40][Developer] I can't find the logs you're referring to.
[11:00][Tester] They're called "xyz.log".
[11:30][Developer] Oh yeah. Thanks.
[13:30][Developer] Can you try again with verbose logging on?
[15:30][Tester] How do I do that?
...
```

This wastes valuable time (as you can see from the sample timestamps) because the developer and tester are not constantly monitoring the defect-tracking tool, which leads to large gaps between questions and replies. Alternative communication methods should be used, such as “Pick up the phone and work through the problem.” The defect-tracking tool is not a discussion board; it should be used to describe the defect, its resolution, and key decisions. If you want to use the tool for this purpose, the defect description and root cause analysis (including key decision points) should be kept separate from any discussion threads. Otherwise, the conversational information might appear on management reports.

Assessing the impact of defects should be a very thorough process. It's important that the assessment is sufficiently detailed and I've listed just a handful of useful tips for assessing the impact:

- What's the problem? Provide additional context to the original defect description to describe exactly what the problem is.
- How did it come about? This simply adds context to how the defect came about. For example, the root cause analysis might be due to the design not being signed off and the development team proceeding at risk. The root cause analysis is not to point blame but to feed into the continuous improvement process to reduce the number of defects in the future.
- What's the solution? Provide a detailed analysis of the solution and what impact this may have on other areas of the system.
- Is it a big change? If so, what risks are associated with it?
- Will it require a change to any of the inputs into the construction phase? If so, which ones?
- Will it require a change to the test scripts/test data? If so, which ones?
- What needs to be updated? Provide a list of the things that will need updating (including documentation, classes, methods, unit tests, integration tests, scripts, data, tools, and so on).
- Will it require more than one person to work on it? If so, how many and from what departments?
- Will it require changing or updating any batch jobs or reports? If so, which ones?
- Will it require updates to support or operational procedures? If so, which ones?

When possible, scripts and tools should be developed or used to help provide a more detailed impact assessment. For example, tools that walk the code tree can find areas where methods are called and this can provide information on which test cases might need to change. During an impact assessment it's very common for the following exception scenarios to arise:

- There's not enough detail on the defect (despite going through *two* approvals).
- You can't reproduce the defect.

- ❑ It’s a problem with the test script or test data.
- ❑ It’s not a defect; it is expected functionality.
- ❑ It’s a defect relating to another part of the solution that is not managed by *this* team (which can also include environment and tooling issues).
- ❑ It’s a defect that relates to both *this* part of the solution and other parts of the solution managed by other teams (which can also include environment and tooling issues).
- ❑ It has already been fixed.

It is important that the person who assesses the impact record the amount of time that he or she spent investigating and assessing the impact of the defect. When the defect is fully understood, it’s equally important for the impact assessor to provide an estimated time to fix (and test) it. This information will help with planning, management reporting and financial summaries at completion, as well as continuous improvement.

5. **Impacted** — Once the defect has been fully impacted, it can be reviewed, checked, and moved through the workflow. It depends on the outcome of the impact assessment as to how the defect will be moved on. The diagram shows a few of the alternative transitions, including:
 - ❑ **Impact** — If the assessment is not satisfactory, it may be necessary to re-impact the defect. Furthermore, it is important that the defect be impacted by all the relevant teams. For instance, during an initial assessment it might be noted that a database change needs to be made. This change may need to be impacted by many people, depending on how far-reaching the change is. For example, it might involve changing multiple test scripts and test data.
 - ❑ **Verify** — The defect may no longer be present in the latest release. Therefore, the defect needs to be assigned for retesting against the latest release. Alternatively, the defect may be related to an existing defect, and, as such, has been fixed with the original. The defect needs to be marked as such and associated with the original for tracking.
 - ❑ **Rejected** — After a detailed impact analysis, the defect may be rejected. It is possible that it’s rejected because it isn’t actually a defect. In some cases, defects are rejected because they can’t be reproduced. Strict guidelines should be developed around rejecting impacted defects, and the parties involved should agree that the defect is actually a candidate for rejection. This avoids having to revisit rejected defects later on.
 - ❑ **Deferred** — Based on the impact analysis, there’s a possibility that the fix to a defect may be deferred to later in the project lifecycle. High-risk, low-gain defects typically fall into this category.
6. **Assignment** — The defect is assigned out to the various parties for implementation. This could be a single individual, but it could be a number of different teams based on the impact. One common scenario is to send a defect for implementation, and at some point during the process it’s noted that the impact analysis wasn’t done correctly. Then another team needs to impact it or perform a greater impact analysis. The defect-tracking tool (or process) should be capable of supporting this multiple distribution/multiple state scenario.
7. **Implement** — The implementation should start by validating the defect, its impact, and its proposed resolution. Taking a step back will help to ensure that you’re not about to do something that you shouldn’t be doing. During this review and validation exercise, an alternative resolution might be identified to the one originally specified and agreed. This is a very common scenario because things may have moved on in development since the original

Part I: Production-Ready Software

assessment was performed. Furthermore, the initial assessment may have missed something that is uncovered during the detailed implementation. Changes to the originally proposed implementation need to be approved and documented in the tracking tool, along with the updated estimates.

Once the defect has been implemented, the actual time taken to implement should be recorded for reporting purposes. It provides realistic figures for the amount of time spent fixing defects. This can be compared against the estimates to provide a basis for improvement. In addition, the “fixed in activity” should be recorded to support management reporting.

8. **Release** — When all the implementation activities are complete, they can be reviewed, agreed on, and submitted into a release. This will follow all the same practices and processes for ready-to-build and ready-to-release. The defect-tracking tool should keep track of the release that a defect has been incorporated in. This avoids wasted effort retesting and verifying defects in the wrong release.
9. **Verify** — The defect is retested against the particular release that includes its resolution. It should have been noted during the impact analysis whether the test scripts and test data needed to be updated, and these should be verified. If necessary, the teams and individuals should work together to ensure that everything is present and correct. This will avoid defects going around the loop again when they may not need to — for instance, failing the implementation when the test script wasn’t updated correctly or the wrong release has been installed.

Depending on the outcome of the verification the defect can take one of two routes:

- ❑ **Failed** — If the resolution hasn’t successfully resolved the issue, the defect needs to be marked as failed. Particularly difficult defects may go around this loop quite a few times, and it’s important that all additional or new information is added to the defect each time.

It is very important to document the defect-failure process to avoid a defect being failed when, in fact, it should be a completely new defect. This happens a lot, too, and it’s sometimes unavoidable. It happens primarily because the individual doesn’t know the exact nature of the failure or doesn’t look into it. As far as he or she is concerned, the transaction they are performing still doesn’t work. In the scene at the start of this section, you experienced a “technical error” on submission of a new account. If you still get a technical error on submission, you’re likely to say that the defect hasn’t been fixed, despite the fact that an entirely new problem is causing the technical error this time. This links right back to the start of the process and actually re-enforces my previous messages. I mentioned that you might have different releases for different purposes. This is a prime example. The “technical error” page in production will probably show a generic message and not contain error information. However, in testing it would be a good idea for it to show a proper error message. That way, instead of a defect being raised with “Technical Error on Submission of Create New Account,” you can have something more distinguishable, such as “Connection String Error on submission of Create New Account.” You now stand a much better chance of closing this defect and opening a new “Account Table Not Found Error on submission of Create New Account.” This can also be achieved if the test team reviews the events and logs to obtain the exact error, and includes a reasonable headline for the defect. However, this would require detailed guidelines and instructions for the test team to follow.

- ❑ **Closed** — Once the defect has been successfully retested and verified, it can be closed. When a defect is closed, the activity in which it was closed should be recorded. Some defects are put on hold due to their severity or nature, and, therefore, might not be resolved in the same activity that they were raised in. This information also helps with management reporting.

There’s one other very important thing to consider when looking at defect management. You might have multiple releases running in parallel on the project. This was highlighted previously. It’s possible that a defect needs to be impacted across multiple teams and releases. In addition, you will need to ensure that the defect-management system caters for multiple releases (or branches) to avoid the defects getting muddled up.

This completes the walk-through of the process. Quite a lot is involved, so it’s well worth getting a foundation process in place early. Identifying the roles and responsibilities will also help when defining the security model and checks required throughout the process.

Defining the Severity and Priority of Defects

This is probably the most controversial area in defect management. The fact is that defects need to be fixed because they indicate that the system doesn’t “work” as it should. The problem is classifying defects in terms of their *impact* and the *order* in which they need to be fixed. These are traditionally referred to as the defect “severity” and “priority,” respectively.

Severity is usually expressed in terms of Critical, High, Medium, and Low. Severity can also be expressed numerically — for example, 1, 2, 3, and 4. Similarly, priority is usually expressed in the same way — that is, High, Medium, or Low, or numerically.

How do I classify my defects? Well, first there needs to be very clear guidelines on how to assess and assign the classification. Second, it depends entirely on the approach to classifying defects. The following table shows some high-level severity classifications from the testing activity perspective and from the “system” perspective:

Severity	Classification From the “Testing” Perspective	Classification From the “System” Perspective
Critical	The defect blocks the closure of a complete test phase.	The defect causes the entire system to be unusable.
High	The defect blocks the closure of a complete test suite.	The defect causes part of the system to be unusable.
Medium	The defect blocks the closure of a complete test scenario.	The defect affects the usability or user experience.
Low	The defect isn’t currently blocking the continuation of execution.	The defect doesn’t affect usability or transaction processing.

And the following table shows a “lowest impact” classification of the defects raised in our little scenario using these definitions:

Part I: Production-Ready Software

Defect	Classification From the "Testing" Perspective	Classification From the "System" Perspective
Defect 1: There is a missing "mandatory field indicator" next to "Security Question".	Low	Medium
Defect 2: The tab order is all over the place.	Low	Medium
Defect 3: The username field only allows 15 characters, instead of 16 as stated in the entry tip or hint.	Low	Medium
Defect 4: Technical error on submission.	Medium	High

From the system perspective, none of the defects cause the entire system to be unusable; therefore, none of them are critical. A very severe technical error could cause the entire system to crash, and this would be classified as "Critical" (using these definitions).

So, what does this tell you? It definitely tells you that Defect 4 is affecting a test case or scenario. It's also telling you that Defect 4 is causing a part of the system to be unusable. However, when looking at the other rows in the table, the testing side is telling you, "There's not much to worry about here," and the system side is telling you, "We've got some usability issues." In such a case, you need to ask yourself "Do you want to go-live (on the Internet) with defects 1, 2, or 3 outstanding?"

I'd prefer to see them all fully addressed before go-live. If you look at the defects, it doesn't appear that there will be much effort involved in fixing them. Or will there? You could have generated all the screens from configuration automatically, or you could have created them manually. Either way, you need to understand exactly what's involved in fixing them.

From a testing perspective, the preceding tables show that execution can actually continue because the defects 1, 2, and 3 don't really have an effect on execution. However, they do (and will) have an effect on closure of the scenario. Defect 3 will have an effect on your ability to confirm the boundaries. You can also classify them all as "Medium" in the first instance. However, you need to distinguish the order in which they should be prioritized. In this example, clearly defect 4 is what you want to continue testing. The priority can change, but this is your starting position. The following table shows a potential starting point to ascertain the severity and set priority from the testing perspective:

Defect	Severity	Priority
Defect 1: There is a missing "mandatory field indicator" next to "Security Question".	Medium	Low
Defect 2: The tab order is all over the place.	Medium	Low
Defect 3: The username field only allows 15 characters, instead of 16 as stated in the entry tip or hint.	Medium	Medium
Defect 4: Technical error on submission.	Medium	High

Chapter 3: Preparing for “Production”

Why classify from the *testing* perspective? Your solution is made up of many applications, processes, and tools. Let’s assume you’re using a tool that injects transactions into the system. The tool and the application open just fine. You load your transaction set and click “Inject.” Nothing happens. What’s the problem?

- Is it a defect in the tool?
- Is it a defect in the application?
- Is it a defect in the transaction set/script?
- Is it a defect in the set-up (including environment, configuration, and test data)?

The problem is that you really don’t know without performing some potentially detailed analysis. It depends how the test team is structured, but you can’t expect the test team to perform a detailed and thorough investigation before raising the defect. For example, the test team might simply follow test scripts and instructions and not have the experience and knowledge to investigate issues. There’s definitely a case for providing them with some “Top 10 Tips” that they can try in any situation. However, the root cause may still not be highlighted. Now you’re a bit stuck when it comes to classifying the defect from the *system* perspective. This is precisely why defects are often classified from the *testing* perspective, in the first instance. It’s just like saying, “I don’t know what the problem is! It just doesn’t work the way it’s expected to. It’s stopping me from completing this activity, so I need it fixed ASAP!”

Some test teams, such as the technical test team, are used to conducting detailed analysis, which can highlight the defect’s impact to the system as well as their own testing activities. If the defect relates to a test tool, a test script, or test data, then the defect impacts their testing activities and not the final state solution. It’s not uncommon for the technical test team to provide detailed impact assessments for defects that relate to the final state solution. However, the assessments still need to be ratified and agreed to.

This situation highlights that, although the “impact” to the tester can be assessed immediately, it is not always possible to determine the impact on the actual system — at least until the detailed impact analysis has been performed.

As a developer, you’re used to incident investigation, especially on your own systems. However, if you had a problem with a third-party product, how would you classify a defect on the vendor’s tracking system? You would typically classify it from your own perspective, not theirs, although it depends on their classifications, too. It generally means specifying the severity it has on you and the priority you think it should be given by them.

Let’s examine another scenario. Suppose that you try to start a test tool and it simply doesn’t start. You now have a very clear “problem definition” and can attribute a fault specifically to the test tool. However, it still doesn’t change the impact it has on the activity. Using the testing perspective, “severity” and “priority,” is a great way to manage defects according to test activities, but it’s not very good for managing defects according to whether the system can go-live without them being addressed or for managing defects according to the actual severity of the defect.

People usually want to know the number and severity of defects related to the final state solution, as well as the defects that potentially impact the project’s timelines and budgets. Therefore, it is prudent to ensure that *all* situations are provided for. As defects are fully “impacted,” you can keep track of whether they actually relate to the final solution, whether they are required for go-live, and their impact on the solution. Using this approach you’re starting off with the following end goals in mind:

Part I: Production-Ready Software

- ❑ All critical go-live issues must be addressed.
- ❑ All “customer-facing” issues must be assessed as to when and how they can occur and what procedures/corrective actions need to be in place to deal with them.
- ❑ All issues relating to business processes and/or back-office processes need to be assessed, and suitable procedures/corrective actions need to be put in place.

You still need to define the “go-live” severity classifications. This can be used in your management reports. At any point, you can run your reports and determine what needs to be fixed to allow the current activities to continue unhindered, as well as what you need to ensure is fixed for go-live.

Because defects can be raised at almost any point in the lifecycle and against any part of the solution, you can generalize the sample classifications so that they are not specific to testing. The following table lists the sample “activity severity” classifications:

Activity Severity	Classification
Critical	The defect will block the closure of an entire activity.
High	The defect will block the closure of multiple tasks in an activity.
Medium	The defect will block the closure of an individual task in an activity.
Low	The defect isn't currently blocking closure.

There's one last scenario that I'd like to go through. Suppose that you're implementing a fix and notice that there's a problem with another class or method that's not part of the work you're doing. How do you raise and classify what you've found? Ultimately, it's “Low” severity and “Low” priority because it neither has an impact on what you're currently doing, nor is it a part of the scope you're working on. This means that you can at least capture defects that are not specific to the tasks you are performing, irrespective of its potential impact on the solution. If you're using defect severities for exit criteria, these might be included in the criteria. However, this must not discourage (authorized) people from “raising” what they find according to the proper guidelines. I always find it best to have a discussion before raising a formal defect in the tool even though it's reasonable to assume that the defect will be uncovered sometime in the future. At the very least, you always have a list of defects that were caught outside of any formal activities that may need to be assessed.

Whatever the severity and/or priority classifications are, there should be very strict guidelines on how to assess and assign them. Ensuring that the defect severities and priorities are appropriately classified, understood, and followed will help to ensure the fix process is consistent and efficient.

There are many ways and views of categorizing defects, and there may be strict constraints that need to be followed. The key is to ensure that the chosen severities are well-defined, documented, and followed accordingly.

Defining the Contents of Defect

The contents of a defect will differ from project to project. The mandatory fields are stipulated to ensure that defects contain all the relevant information. The documentation and usage guides should clearly state what fields are required and how to appropriately select or complete them at each stage. It also breeds best practice and avoids defects with no information or limited context. The key fields should be mandatory and used for management reporting. The following are some sample high-level field groups based on the previous discussion:

- ❑ **State** — Indicates the state of the defect within the overall process (New, Approved, Impact, Impacted, Implement, Implemented, Failed, Closed, and so on).
- ❑ **Severity and/or Priority** — Indicates the defect’s severity and priority. This could be from the testing perspective initially and then changed to the system perspective once a detailed analysis had been performed. Alternatively, both perspectives could be tracked against the defect, which would ease tracking and management.
- ❑ **Headline** — The headline should capture the “essence” of the defect, such as “Account Table Not Found Error on submission from Create New Account.”
- ❑ **Overview and Description (and associated information)** — This provides the detailed description of the defect along with the steps to re-create it and all other information discussed earlier, including the components affected (if known).
- ❑ **Detected in Release** — This should contain the release or version number that the defect was detected in. The fix team would use the same release when attempting to re-create the issue. Furthermore, if this value indicated a much older release, the fix team may simply attempt to re-create the issue in the latest release.
- ❑ **Raised in Activity/Stream** — Identifies the activity where the defect was found (Design, Development, Functional Testing, Technical Testing, and so on). The values could be broken down to provide for individual streams such as Operability Testing, Failure and Recovery Testing, and so on.
- ❑ **Raised Against Activity/Stream** — This would indicate the activity or stream responsible for introducing the defect and thus the team that the defect should be assigned to for impact assessment. It’s not always possible to identify the root cause until a thorough impact has been performed. For instance, when a defect is raised by a test team, this value might indicate “Online Application” or “Functional Design.” The activities and streams would be dictated by the development approach and structure.
- ❑ **Impact Analysis and Root Cause Analysis** — This provides all the information about the defect’s root cause and its resolution.
- ❑ **Deferred to Activity/Stream** — This is very useful when defects are being discussed. Not everything is fixed within the activity the defects are raised in. Often defects are deferred to another release or phase.
- ❑ **Resolved in Activity/Stream** — This would indicate the activity or stream where the defect was resolved. For example, if the functional test team actually fixed the issue because it was a script issue, this value might be “Functional Test.” However, if it was an application issue that was resolved during technical testing, this value might contain “Technical Testing” or “Operability Testing.” Again, the development approach would dictate the activities and streams in the overall project.

Part I: Production-Ready Software

- ❑ **Resolved in Release** — This would contain the release or version number where the defect has been resolved. This value would be assigned when the fix (or fixes) was included in a formal release as part of the build and release processes.
- ❑ **Estimated and Actual Times** — It is important that the estimated and actual times be properly used and completed with realistic figures. All too often these fields are not mandatory or are completed with unrealistic figures that result in a planning nightmare. This is especially true when there are large numbers of defects or limited resources. The defect description and associated information should be detailed enough so that a realistic estimate can be arrived at. The estimated and actual figures are used for management reporting and continuous improvement.

The defect would also include all the necessary dates and times associated with the activities — for example, date raised, date impacted, and so on.

This is not an exhaustive list of fields and elements, but it captures the highlights. During the defect management and fix mobilization activities, the defect modeling can be performed. Having all the right information included within the defects will help to refine your management reporting and help you to continually improve the project.

Defining the Management Reports

The management reports will be used for both defect scheduling and post activity analysis. When an activity finishes, the reports can be used to determine the number of defects that were raised, closed, failed, and so on. A number of reports will be required during the project, and it is worth trying to identify them early on. This will ensure that the tracking tool has the correct fields and that the process can ensure they are captured. The following are a few high-level report types:

- ❑ Defects by Activity (Raised In, Resolved In, Root Cause Activity)
- ❑ Defects by Release (Raised Against Release, Resolved In Release)
- ❑ Defects by State
- ❑ Defects by Severity
- ❑ Defects by Deferred To
- ❑ Defects by Components Affected (Raised Against, Resolved In, and so on)
- ❑ Actual and Estimated Time (Impact, Implementation)

The reports will generally be used for daily and weekly meetings, as well as activity completion analysis. As with everything in the book, reports such as these will be required, so it's best to try and define as much up front as possible, while at the same time defining the process and best practice.

The defect-tracking process will generally last a long time, during which it will be refined. It may also be handed over to the support and maintenance team. All the documentation and configuration will assist with the handover activities. The defect-tracking tools can be experimented with while the development, test, and build mobilization exemplars are being developed.

Configuration Management Mobilization

Configuration management is critical in software development. It is used to store, manage, and version control everything produced throughout the project lifecycle. The project will create and use thousands of artifacts. In the mobilization activities, you’ll have created a whole series. Some will be required in all environments, and some will be different in other environments. All these need to be kept safe, version controlled, and easily accessible and deployable.

Defining Process Constraints

The configuration-management process constraints should list the tools and technologies that will be used for storing, managing, and version controlling artifacts throughout the project. This will mainly consist of the configuration-management tool, as well as any custom or other tools that surround it. The configuration-management tool may also need to integrate with other tools and applications, and these need to be listed and accounted for.

The development tools often tightly integrate with the configuration-management system. This makes it easy for developers to check in and check out directly from the development IDE. You might be using tools to manage your requirements, test cases, test scripts, and test results, and these tools might integrate with the configuration-management system or implement their own version tracking. The defect and change tracking tools may also integrate with the configuration-management system. It’s also possible that the project management tools also integrate with the configuration-management system.

The configuration-management system requires configuration, the amount of which depends entirely on the sophistication of the tool. In some cases, this can mean far tighter control over the changes and files that can be accessed. For instance, some configuration-management systems require the definition of activities and tasks before anything can be added, checked in, or checked out. You might have a “Build Create New Account Component” activity assigned to a particular developer. The developer will use this activity when he or she is checking their code in or out. All changes are then tracked under this activity so that you can see the files that have changed. The project management tool may integrate with the configuration-management system and automatically create a series of “activities” based on the project plan. This can be very useful for defect management, too. The defect-tracking tool may integrate with the configuration-management system and support the automatic creation of work items or “activities.” That way, you can see all the changes associated with a single defect. If you want to make use of these kinds of features, you should think about them up front and plan them into the process accordingly.

The configuration-management system retains a history of changes to the artifacts, and traditionally “configuration management” is associated with “change control” or “change management.” For the purposes of this book and this section, “configuration management” refers to the actual configuration-management system itself, rather than to the change-control process (which is discussed later).

The high-level activities, aside from defining the constraints, include:

- Defining the contents and individual repositories, which detail what you’re going to store under configuration management and whether the contents will be separated using multiple repositories
- Defining the structure of each of the individual repositories

Defining the Contents and Repositories

Every application and every environment has its own configuration-management requirements. For example, the core operating system will reside on every computer, although it may be a different version in certain environments and may require different configuration for different purposes. The tools and applications that are used may only be required in certain environments, but again they may have a different configuration depending on what they are being used for. The test data can be different across the environments, and the database installation and configuration may be different, too. The configuration-management solution is used to store and manage these items.

You need to determine which files you're going to store and how you're going to store them within the configuration-management solution. Ideally, the configuration-management system would be used to store everything. However, some projects and companies also favor the use of network drives and other storage areas for artifacts that don't change very frequently — for instance, if the operating system doesn't change. That is, you might be upgrading to a new version at some point in the project, but in general you're not making changes to the operating system code or documentation. Therefore, do you really want to store the operating system's DVD image (ISO files, and so on) under version control? This question also goes for other third-party product media.

The configuration-management system retains a history of the changes to a file. Each time a file is checked in, the previous version on the file is kept. The fact that you're not frequently changing the files means they take relatively the same amount of disk space as they would on a network drive. In addition, it's often practical to have a server or base image. These images can be used to install and set up a server very quickly. Traditionally the software was installed from the ground up (usually manually) and could take a long time, depending on the number of applications to install and the configuration tasks. Disk images are essentially a backup or clone of the entire machine. If you're introducing a new server or need to restore an existing server, you can use an image to rapidly deploy everything. At lot of organizations and projects choose this approach to reduce manual errors and improve turnaround time. It's typically the production hardware that's imaged, although some companies use it for any critical servers or environments. For example, software houses will typically use this approach for development and test machines and environments. The time to build a new development machine is greatly reduced. There's no reason why you can't store the images within your configuration-management system.

You are making changes to the operating system and third-party product's configuration, so these files should definitely be kept under version control because the changes are very specific to *this* project. If applications do not easily support configuration management — for example, it's hard to determine which files contain the “variable” settings — the scripted installation or installation and configuration documentation should be kept under configuration management.

Furthermore, you'll be making many releases of your software throughout the project. Depending on the size of the project, there could be thousands of builds and hundreds of releases. Do you really want to store all these packages under configuration management? You need to keep all of them for historical purposes. Again, a lot of companies or projects choose to store these within the configuration-management system. The releases need to be kept neat and tidy so that obtaining and installing them is straightforward. One way of doing this is to keep them available via the project portal. They can be organized in such a way that they don't present a 1,000-item list with all the releases from Release 1 through to Release 1,000. The important thing is that they are backed up and kept safe. If you think about it, the project portal resides on the file system, and the files that are uploaded to it are placed on shared disk. So, as long as the shared file system is backed up, you can always restore your releases.

Using the configuration-management system to store items that don’t really change is an alternative approach to using network drives. The configuration-management system will provide a consistent and central location for *all* your artifacts. You can distribute items from the configuration management system to the project portal.

Configuration-management systems support multiple repositories, with each repository essentially being a separate “database.” So, you have the choice of using a single repository or multiple repositories. Using multiple repositories can help to separate artifacts, such as those discussed, as well as provide individual teams with their own repository. For example, technical testing may have enormous datasets that need to be version controlled. These could be stored in a “technical test” repository, for instance.

The following are a few examples of potential configuration-management repositories:

- ❑ **Infrastructure, Builds, and Releases** — This repository can be used to store all infrastructure-related items, including disk images, ISO files, and so on. It can also be used to store all the builds and releases.
- ❑ **Development** — This repository can be used to store all items for the development team.
- ❑ **Configuration** — This repository can be used to store all the configuration items across the project.

The remainder of this section takes a quick look at the following:

- ❑ Defining the structure
- ❑ Defining the branching and merging approach

Defining the Structure

Typically, the core development artifacts would include the following:

- ❑ **Database scripts** — The database scripts would include `CREATE DATABASE`, `CREATE TABLE`, `CREATE VIEW`, and `CREATE PROCEDURE`, for instance.
- ❑ **Source code files** — The source code files include all files associated with the solution, including `.sql`, `.cs`, `.aspx`, and so on.
- ❑ **Configuration files** — Configuration files include all `.config` files and other custom configuration files.
- ❑ **Productivity scripts** — Productivity scripts include all environment set-up scripts, instrumentation scripts, and log file-gathering scripts.
- ❑ **Test scripts and test data** — Test scripts and test data include all test configuration scripts, set-up and teardown scripts, data installation scripts, execution scripts, data extraction scripts, and comparison scripts.
- ❑ **Documentation** — The documentation includes all process guides, productivity guides, test plans, query logs, comment sheets, and other documentation.

Part I: Production-Ready Software

It's generally good practice to separate out items that require different versions across environments. It should be relatively straightforward to pull out of this list what is initially most likely to require different versions. Starting with the most obvious, these are:

- ❑ **Configuration files** — The configuration files are most likely to require different versions for different purposes and environments. For instance, the `web.config` used for development may be very different from the one used in technical testing and production. Furthermore, technical testing may have different versions for different test activities.
- ❑ **Database scripts** — Certain database scripts, such as those containing `CREATE DATABASE` statements, will be different because technical test and production databases often make use of multiple disks and file systems. The databases will be installed differently across environments. There may also be other database scripts that are different across activities and environments.
- ❑ **Productivity scripts** — Certain productivity scripts will be different across environments. For example, technical testing is likely to have more servers, and the productivity scripts may need to support this. For example, in development and testing, your basic defect information gathering tool (or script) may only use a single server. However, this will need to be enhanced (or wrapped) for use in the technical test and production environments, where multiple servers could be used.
- ❑ **Testing scripts and test data** — Certain test scripts will be different. For instance, functional testing may only start certain components in the solution (therefore, the environment start-up will be different), whereas technical testing will probably start up everything. In addition, the number of servers and deployment of components might be different. The test data is also likely to be different. In development and functional testing, a core set of data may be used, whereas technical testing may have multiple sets of data for different scenarios. Technical testing is also likely to have extremely large data sets for load testing and soak testing. Soak testing is typically performed using a significant volume of data injected over an extended period of time. For example, the system may perform exactly as expected when tested with a few thousand transactions injected over an hour or so; however, when tested with millions of transactions injected at expected load over a few days, the application may display abnormal behaviors or characteristics that need to be investigated and corrected.

The number of environments will differ from project to project, but as long as there is more than one, the key is to identify what is most likely to be different across them. These “different” files need to be managed and tracked appropriately.

For the moment, let's look at the `web.config` files. When you add new configuration values or make changes to these files, it's possible that you'll need to update a number of different versions. You need to determine how you are going to manage this and where these “different” files are stored.

The configuration-management structure is really nothing more than a simple folder structure under which everything is stored. Organizing the folder structure is a personal choice and depends on the project. As the project progresses, the structure may change, which is why most structures start out with the basics required for development and are subsequently modified throughout the remainder of the project. There are often tools and scripts created to extract artifacts from the configuration-management system, and as such these are closely tied to the structure. These can be used to extract everything for a

Chapter 3: Preparing for “Production”

particular release or everything for a particular environment. The tools may look for certain versions or may just get the latest version. If the structure of the configuration-management system changes, these tools also need to be updated. It’s a good idea to try to start off with a reasonable structure to streamline this process.

You have a couple of choices for where you could store the different `web.config` files:

- You could store copies of them in their respective repositories. For example, technical tests have their own copies, functional tests have their own copies, and so forth.
- You could store all the individual copies in a single “Configuration” repository.

Alternatively, you could choose to not store copies of them at all but instead implement some other mechanism for maintaining the individual configuration. It’s possible in certain configuration files to have separate sections for different environments and purposes, which can sometimes make things easier, although it can also complicate the configuration file and lead to time being wasted changing values in the wrong place. This needs to be balanced against multiple configuration files and the associated maintenance.

Anyway, let’s say that during development (or fix) you add a new configuration value to this file. This change *may* need to be propagated through each of these copies. I use the term *may* because the system may provide default behaviors if a particular configuration value is found. (This subject is discussed later in this book.) With respect to the updates to the configuration files, you have some choices at this point:

- The development team updates all the respective copies.
- The individual teams are responsible for the updates to their own copies.

If the individual teams are responsible for their own updates, this needs to be managed via the process. If the development team is responsible for the updates, it needs access to each file and/or repository.

Storing the files in a central repository can often make updates much easier, enabling you to note the differences between the files. Configuration-management tools generally support “diff” capabilities. A “diff” usually highlights the differences between two files side-by-side. If all the files are within the same repository, it’s much easier to “diff” the files between development and technical testing, for example.

If the files are stored in a central repository, you have a lot of choices for how you can organize them within a folder structure. The following are just two sample structures:

- Filename\Activity\Purpose
- Activity\Purpose\Filename

(Note that if the files are not stored centrally, you still need to determine the overall structure.) Figure 3-5 illustrates the preceding sample organizational structures.

Part I: Production-Ready Software

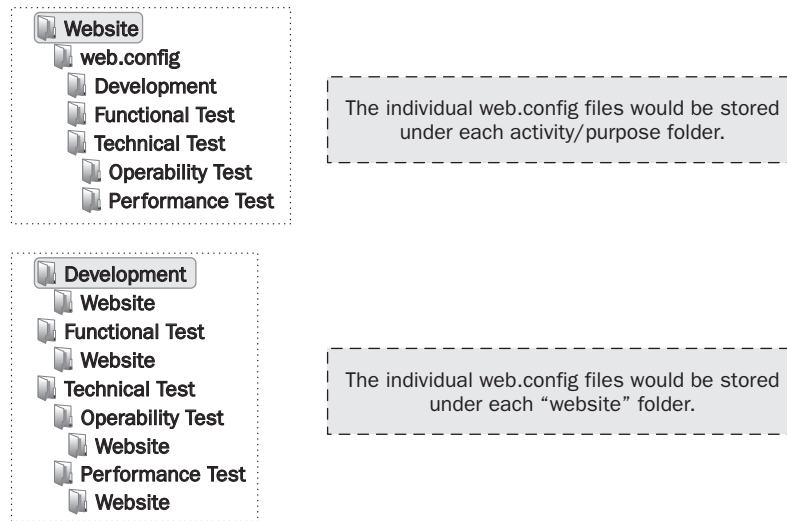


Figure 3-5

In the preceding examples, the basic structure could be extended to include other configuration files, database scripts, tools, test scripts, test data, and so on. When you're making changes to the database schema, it can often help to have things in a central repository because schema changes can affect all the test data (depending on how you're managing test data, of course).

An environment can be as small as a single computer for a specific purpose. Most development machines are a single computer, as are most test environments. Not all development machines are used for the same purpose, and neither are all test environments. One might be used to develop batch components, whereas another might be used to develop architecture components. If different configuration or files are required, ideally they should be separated in the configuration-management system.

There are many ways to slice-and-dice the configuration-management structure. If possible, the number of different versions of artifacts should be kept to a minimum to avoid maintenance overheads, and, if possible, development and testing should be conducted on a live-like configuration.

In addition to the core development artifacts, all the other components and files associated with the system need to be under configuration management, including the following:

- Construction tools, configuration, and scripts
- Build projects, configuration, and scripts
- Packaging projects, configuration, and scripts
- Deployment projects, configuration, and scripts
- Test tools, configuration, and scripts
- Development exemplars and templates
- Test exemplars and templates

- ❑ Batch configuration, schedules, and scripts
- ❑ Reports and data extracts
- ❑ Reference data and test data

The preceding is not an exhaustive list, but it gives you an idea of what to consider and how it can be best organized for all purposes. What you store and manage under configuration management depends entirely on your project and what you (or the project) develop/change.

Defining the Branching and Merging Approach

To complicate matters, I mentioned previously that systems development often involves multiple releases, which is something else you need to consider when looking at configuration management. When Release 2 comes along and is (potentially) running in parallel with Release 1, things start to get a little complicated. It’s at this point that “branching and merging” often occurs. Figure 3-6 shows an example of a branching and merging timeline.

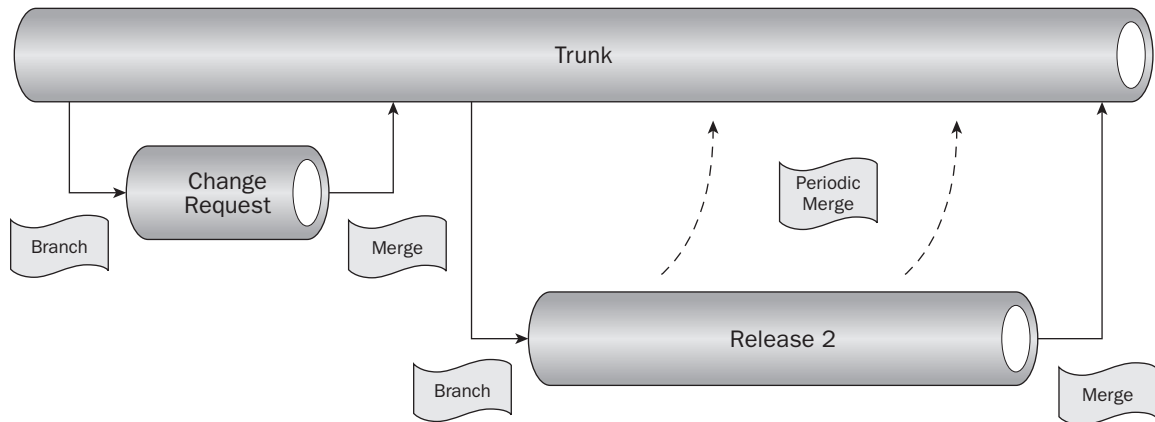


Figure 3-6

Essentially, the main branch (or “trunk”) stores all the project artifacts. It is typically created at the start of the project with Release 1. The figure shows a change request coming in at some point during the development of Release 1. It is sometimes required to implement change requests off the main branch to isolate the changes and maintain consistency. The situation can be even more complicated if the change request were to overlap with the development of Release 2. Changes are made in the individual branches and subsequently merged back into the main trunk. Longer running branches may perform periodic merges, as appropriate.

You should remember the following:

- ❑ When a release goes into production, the source code and all other artifacts associated with it should be kept separate from any other development activities. The reason for this is to ensure that there is a “fix-on-fail” branch of the system. If there is an issue in live service that needs to be addressed, you need to be able to fix it on an exact copy of the production system. You must

Part I: Production-Ready Software

ensure that you don't muddle up changes and potentially deploy part of Release 2 into production.

- ❑ When defects are encountered during testing, they might need to be impact assessed for each release and branch. For example, if Release 1 testing raises a defect, it will be assessed for its impact on the Release 1 branch. It may also need to be assessed for its impact on the Release 2 branch (and others) to determine if it's a problem there, too. It's possible that there are multiple (and differing) impacts for a single defect. For example, Release 2 may have changed the code in such a way that a different impact and resolution is required than that for Release 1. Furthermore, issues that are detected in live service will almost certainly require impact assessment against any other development branches and releases. Figure 3-6 shows the fix-on-fail branch potentially merging back into the main trunk. However, this would typically require a detailed impact analysis to ensure that the files changed were identical (prior to the change) across all releases and branches.

Branches can usually be taken from any point in tree, although some projects favor branching and merging the entire project — for example, if Release 2 is a complete copy of Release 1. Alternatively, branching can be done at lower levels within the structure. It's entirely dependent on the project, the purpose of the branch, and the duration.

Release 2 will be updating in a separate location files that have been branched from a specific version of the Release 1 files. Updates may still be being made to the Release 1 files, and these changes need to be merged into the Release 2 versions where appropriate.

The individual teams need to work together to determine which changes should be merged. This is another important reason for using a configuration-management tool, as doing this manually and tracking all changed files would be an arduous task.

It's a good idea to understand this early on. If you're using a single repository, keeping everything under a single, top-level release folder (for example, `Within-a-Click\Release 1`) makes things a lot easier because a new folder for release 2 can be added, and so on.

Change Control Mobilization

That things are going to change is a fact and part of the nature of software development. New ideas and new ways of thinking will constantly come about. When users are testing, they'll often identify areas where improvements can be made. It's vital to a project that an appropriate change-control process is in place and that everyone understands and adheres to it.

I've discussed the differences between defects and changes. Changes follow a very similar workflow to defects, although changes usually involve financial discussions. I mentioned in Chapter 1 that it depends on the methodology but changes are often considered "a change to the signed-off requirements" and as such have more impact on budgets and timescales. When there are changes in "scope," these need to be impacted and estimated accordingly.

It is important that everyone understand and follow the change-control process. It is all too easy during development to start changing things without going through the proper channels. The change-control process is there to ensure that all changes go through the relevant process and are either approved or rejected. When approved, the changes can be scheduled into the development lifecycle and released accordingly.

Where possible, tools can and should be used to assist in the impact and estimating of changes in the same way that tools are used to impact defects. It is rare that a change will require a completely new application or architecture stack, so the existing estimating models will also provide a good starting point. If the change is something completely new, it may require a full bottom up/top down design and impact.

You saw in the previous section that a change could be implemented on its own branch to ensure that it doesn't conflict with mainstream development. After the change has been completed, it can be merged into the main trunk, as required.

The high-level steps for change control mobilization include:

- ❑ **Defining the process constraints** — As with all mobilization activities, this involves listing the tools and technologies (as well as any other constraints). The tools used for change control are often very similar to those used in defect management. They can sometimes integrate with the configuration-management system to ensure “activities” and work items are defined, as well.
- ❑ **Defining the process** — This involves defining the workflow, processes, and activities that need to be performed when dealing with changes. The workflow typically follows a similar path to defect management. The “impact” activity can include activities such as requirements analysis, design, and commercial discussions. The impact will also need to determine the impact to the overall project timeline and the associated risks.

I'm not going to go into these activities in detail because, as you can see, they are very similar to those already discussed. The important thing is to ensure that change is accounted for in the overall project.

Summary

This chapter examined some of the mobilization activities that can improve your readiness for full-scale implementation. The important thing to remember is that these activities will usually happen naturally throughout the project. It's better to have a handle on them early to avoid unnecessary delays and ensure that everything fits together nicely. As you've seen, a lot of the activities, processes, and tools are very closely coupled. Thinking about how your production line is going to operate will help you to ensure it operates smoothly and the results are of high quality. If you were building the entire system (including the production line) for someone else, it's likely that all these outputs and processes would be handed over to them.

The samples and suggestions outlined in this chapter should be used for reference purposes and not necessarily followed verbatim. Every project is different and will have its own specific requirements and best fit.

Part I: Production-Ready Software

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

- ❑ **Mobilize early** — The best time to mobilize an activity is before it actually starts full-scale. Major changes to the processes, tools, and activities will have a far greater impact if they are already being used by large numbers of people. Mobilizing early ensures that everyone has a clearly defined process to follow and the appropriate tools to use.
- ❑ **Capture the constraints** — The most important part of mobilization is to ensure that all the relevant constraints are captured. You don't want to go off down your own path only to find that there's an existing process that you *must* follow. The constraints and categories discussed in this chapter included:
 - ❑ Process and methodology
 - ❑ Tools and technology
 - ❑ Resources
 - ❑ Infrastructure
 - ❑ Communication
 - ❑ Location (and security)
 - ❑ Environment
- ❑ **Define the processes and tools** — You need to think about what you're doing, whom you're doing it for, and when it will be required. The key to effective mobilization is to cover as much as possible. Thinking about the individual processes and tools, how they interact with each other, and how they will be used will ensure that you have a good foundation for success. During your mobilization activities, you can create a number of exemplars and templates that can be used throughout the project. The mobilization activities discussed in this chapter included the following:
 - ❑ **Development** — Development mobilization needs to capture the entire scope of the development activities — the inputs and the outputs. The inputs to development will be discussed in more detail in the next chapter and throughout this book.
 - ❑ **Build and regular integration** — You need to ensure that you can build and release effectively. There could be many different releases, containing many different things, for many different purposes.
 - ❑ **Test** — The testing activities will prove your system, so it's important to ensure that you've defined the right test activities, their scope, and their outputs.
 - ❑ **Defect management** — Defects are going to be raised throughout the lifecycle, and you need to be ready to track and manage these appropriately. You need to ensure that you have the right workflow in place and that the defects contain the right level of information. You can categorize defects in many ways depending on the requirements of the project/activity.
 - ❑ **Configuration management** — The configuration-management system is used to store all your project artifacts. You need to determine what you're going to store within it and how it's going to be structured in terms of repositories and trees. You may also need to take into account branching and merging, and ensure that you cater for this.

Chapter 3: Preparing for “Production”

- ❑ **Change control** — You need to ensure that you have the appropriate tools and processes in place to ensure changes are tracked and managed appropriately.

The number of applications, processes, and tools will depend entirely on the size and scale of the project. Setting up and configuring the applications and tools requires time and effort, which needs to be factored into the scope.

- ❑ **Continuously improve** — Just because you’ve mobilized the activity doesn’t mean it will be perfect. It may need the odd tweak here and there. You need to regularly review the processes and tools and make improvements, as necessary.

The following chapter discusses the last subject in the overall production-readiness landscape — the inputs to the development activities to ensure the build team has everything it needs to produce quality software products.

