Wrox Programmer to Programmer™

Design - Build - Run

Applied Practices and Principles for Production-Ready Software Development

Dave Ingram

Updates, source code, and Wrox technical support at **www.wrox.com**

2

The Quality Landscape

No software system is completely and utterly defect-free. The testing and reviewing that you perform will always highlight defects and deficiencies in your outputs. The level of testing and reviewing that you actually perform and when you perform it dictate how many defects can be highlighted and fixed before the system is put into live service or production. The preparation, execution, and delivery that you perform should really ensure that quality is maintained throughout the project. The software quality landscape encompasses the categories and measures for defining and maintaining software quality. The measures are intended to reduce the number of defects that are found out late in the process and to produce high-quality code and artifacts throughout the project. There's no doubt that quality has costs associated with it, but the extent to which it actually costs needs to be understood, controlled, and minimized without compromising the final result. I mentioned in the previous chapter that I consider quality to be a component of scope, and the software quality characteristics contain the high-level categories for each of the quality areas. The total number of defects and their scale should be reduced as much as possible during construction to avoid unnecessary delays, excessive numbers of defects during formal testing, and cost overruns. To support these principles, you need to implement and employ tools and processes that help to maintain quality throughout the project.

This chapter is organized into the following sections:

- **The Quality Characteristics —** Provides an overview of the individual quality characteristics, including correctness and completeness, usability, accessibility, reliability and stability, performance, efficiency, availability, integrity and security, operability and supportability, deployability, configurability, maintainability, readability, reusability, modularity, flexibility and extensibility, and testability.
- **Why Quality (and Scope) Matter —** Takes a look at a robust construction phase and the activities performed to get a realistic picture of what's involved in quality construction. This section also covers how budgets and timescales are affected if the appropriate scope isn't factored in accordingly. I then look at what potentially lies beyond the construction phase, including overlapping test phases, the profile of defects during testing, turning around defects, hot-fixing, technical tuning and re-factoring, and sweeping. I describe

Excerpted with permission of Wrox Press from Design - Build - Run

how some of these activities lead to a drop in the quality bar and outline a number of practices that you can employ to improve and maintain quality and productivity.

- **Quality Comes at a Price —** Provides an overview of the financial matters involved in quality construction, and a basis for understanding the financial implications to assist in the decision-making process. The section also looks at calculating the potential cost of defects and performing cost/benefit analyses, and, finally, looks at the implications of realistic estimating to ensure that the scope, budget, and timescales are set and agreed on.

In this chapter you'll get an overview of the quality characteristics and some of the activities you can perform to improve the quality of your software and projects as well as understanding the costs and benefits associated with them. As the book progresses, I'll cover some of these items in more detail.

Before diving in, I'd like to share a short story with you that is quite poignant at this point. It was a marvelous day when my editor told me that my proposal for this book had been accepted and that I should start work immediately. I asked him if the final contract had been sent and he told me that it had been. However, he said there was an issue — the courier's online tracking system was reporting an "Incorrect Address" error, although we confirmed that the contract had been addressed correctly. I took it upon myself to telephone the courier company to follow up with them. I gave the representative (rep) my tracking number and she replied, "Okay, I'm just waiting for the details to load on my screen." There was a lengthy pause. The rep then said, "I'm sorry about this, but my system is running really slow today." "No worries," I replied. After another lengthy pause, the representative said, "I'm really sorry. Do you mind if I transfer you to someone else as my computer just froze?"

With reference to the preceding short story, I actually wondered whether this sort of thing (the computers running slowly and freezing up) happened quite often or whether it was simply a one-time event. If the situation occurs on a regular basis, I can only imagine the frustration of the users and customers, and how it would erode confidence in the overall quality of the solution. There's nothing worse than having to use an application or service that you don't have confidence in. The quality characteristics and, more important, ensuring that the solution actually meets them, will help to instill confidence in the solution.

The Quality Characteristics

The quality characteristics are often born out of a set of guiding principles, which set the scene or vision for the solution. The guiding principles are a set of high-level statements that outline the intent of the solution. Typically, there are around ten or so guiding principles for any undertaking, although this varies greatly depending on what they refer to. Some organizations use guiding principles to set out what matters to them, their employees, and their customers. Projects generally use guiding principles to set out key capabilities and characteristics of the solution. If the project is implementing a new version of an existing application, the guiding principles will often include capabilities that are an improvement over the previous version. For example, if scalability is limited or non-existent in the existing system, one guiding principle for the new system may be "highly scalable." The principles do not describe the exact functionality; instead, they capture a high-level manifesto that underpins the vision and goals for the future state solution. The following are examples of guiding principles:

Excerpted with permission of Wrox Press from Design - Build - Run

- **Positive user experience —** Positively impact the user experience and satisfaction of the system while retaining and satisfying the business goals and requirements. Providing a rich and satisfying user experience is not just good for the customer — it's good for business and the company. With respect to websites and a global population, however, there are potentially millions of users, all of whom will have their own point of view on how user friendly the site is. The designer needs to come up with an easy-to-use interface while providing all the relevant functionality.
- **Flexible —** Support for a growing, changing, and adapting marketplace by providing the ability to add new functionality quickly and easily. The ability to react quickly to changes in the market and provide new functionality quickly and easily is a factor for success. The solution should be flexible enough without dramatically impacting costs and timescales when it comes to adding new functionality.
- **High performance —** Support for global transaction levels and volumes. With a worldwide population the site could be accessed by millions of users, so performance is a key principle that should underpin the design. The end-to-end transaction time is crucial to end users. It's very frustrating sitting around waiting for pages to refresh, especially when you have no idea of what is happening in the background.
- **Cost effective —** Efficient and cost effective to operate, support, maintain and enhance. The system shouldn't introduce an unnecessary burden on the support organization. The solution needs to be generally easy to operate. The system needs to be relatively easy to maintain. Additional features and enhancements will be added over time. During analysis and design a number of features will be deemed out of scope, all of which could be candidates for a future release.
- **Highly secure —** The system may capture personal information about customers. If this information were to get into the wrong hands, it could not only be newsworthy but it could seriously affect the customers, the organization's reputation, market share, and bottom-line figures. The system should implement highly secure protocols for data capture, viewing, extraction, and amendment. The system will maintain and protect customer and user privacy and information at all times.
- **New technologies —** The system should be built and tested using the latest generation of technologies.

The guiding principles are usually mapped to a set of business benefits and drivers that can be realized through the adoption of the solution. For instance, one business driver may be to reduce manual effort (and costs) by 20 percent. In such a case, automation would feature quite high on the software implementation agenda.

The overall solution doesn't just include code. Contrary to popular belief, software developers don't just develop code. They're responsible for many other tasks, including reviewing documentation, writing technical designs and other documentation, writing test scripts, testing software, preparing presentations, and showing results. They implement development processes and practices. They write code and scripts; develop test data; and design configuration files, components, and applications. They're also responsible for helping other people, handing over their solution and documentation as well as doing a whole bunch of other things. Improving the quality of our system means applying the same due diligence to everything you do and not just focusing on the quality of your code. If you take a step back and think about what you are doing and why you are doing it, you can not only improve the quality of your code but everything else around it. Beautifully crafted code can be let down by poor,

Excerpted with permission of Wrox Press from Design - Build - Run

inaccurate, or incomplete documentation or tools. In these days of agile, rapid application development and model-driven engineering techniques, there's still a reasonable proportion of the job that doesn't involve actual coding. As I mentioned in the previous chapter, the use of wikis can really help to reduce the amount of "formal" documentation that is produced. However, it still needs to be fit for purpose.

Continuous improvement is about ensuring that everything you do is high quality and displays a number of different quality characteristics. The following table briefly summarizes the essence of each quality characteristic. I've used the term *it* and not *application*, *software*, or *system* because I believe that these terms somewhat imply source code or source-related artifacts, and, as you've seen, production readiness applies to applications, environments, processes, and tools, which involve more than just source code — the quality characteristics can apply to everything you produce, although not every characteristic will apply to a particular deliverable.

Correctness and completeness	Correctness and completeness represent the extent to which it delivers what it should. Correctness and completeness are derived from the scope — that is, the requirements and constraints whether documented or otherwise.
Usability	Usability is the ease of which it can be used. This is not to say that all things will be easy to use, but "usability" refers to the overall ease of use from a variety of different user groups and perspectives.
Accessibility	Accessibility is the extent to which it can support a variety of different users. This doesn't just mean supporting users with disabilities. It includes a variety of subject areas, including alternative languages and users in different locations.
Reliability and stability	Reliability and stability represent the ability for it to perform its functions under normal (and adverse) conditions. This includes repeatability and predictability, in that it should produce the same results under the same conditions. This also includes all failure and recovery and disaster recovery situations.
Performance (Speed/Users)	Performance is the speed at which it performs its functions under normal and adverse conditions and load. In order to gauge true performance, the number of users, locations, and transactions also need to be considered. The term "performance" is often used to include other characteristics; however, I've chosen to separate the definition as I've included the other characteristics individually.
Efficiency	Efficiency is the extent to which it utilizes resources. Resources include system resources (such as CPU, memory, disk) as well as human resources and other resources (such as printers, paper, and the environment).
Availability	Availability is the extent to which it is available and ready for use. Different users have different expectations of availability that should be taken into account.

Excerpted with permission of Wrox Press from Design - Build - Run

Scalability	The ability to which it can scale to meet future demand or growth needs.
Integrity and security	Integrity is the extent to which it prevents unauthorized or improper use or distribution.
Operability and supportability	Operability is the extent to which it can be operated and kept up-and-running (functioning and in a healthy state). Supportability is the extent to which it can be effectively supported.
Deployability	Deployability is the extent to which it can be deployed. There are typically many users and environments involved in the project, and deployability is vital to getting the right artifacts out to the right people and places.
Configurability and adaptability	Configurability is the extent to which it can be configured or adapted for different scenarios and situations.
Maintainability	Maintainability is the extent to which it can be maintained and enhanced as the project progresses.
Readability	Readability is the ease of which it can be read or understood. There are many different users and groups of users, so readability is often seen from a number of different perspectives.
Reusability	Reusability is the extent to which it can be reused, in whole or in part, and for other purposes or in other areas.
Modularity	Modularity is the extent to which it is broken up into component parts or building blocks. Modularity often breeds re-use by providing smaller artifacts that can be pieced together into a larger solution.
Flexibility and extensibility	Flexibility and extensibility is the extent to which its usage can be changed or extended. Unlike maintainability and configurability, flexibility and extensibility deal with changing its usage and extending it beyond its original scope.
Testability	Testability is the extent to which it can be tested, proven, and quantified. If it can't be quantified, it can't be proven to work. Certain situations call for a pragmatic risk assessment based on skills and knowledge to avoid lengthy and costly testing that covers very extreme and unlikely circumstances.

The degree or extent to which each of these characteristics applies depends entirely on what they are being applied to. For instance, documentation needs to be correct and complete, readable, and usable, as well as reusable and maintainable. Documentation may also need to be integral and secure. This could be as simple as including a security classification on the document. Tools will generally need to be more configurable and extensible, given the number of potential uses, environments, and situations they will be used in. Architecture and framework components will generally be more reusable. Processes also need to be usable, efficient, and scalable to cope with future demand. A process that doesn't scale can

impact a project greatly. For example, assume that you have a single DBA who is responsible for developing all database artifacts. If there's an influx of database requirements, it's likely that one person could be overloaded, which can have an impact on timescales. If your project team is distributed across multiple locations, the applications, environments, processes, and tools would also need to accommodate this.

It would take a very large table to list all the individual items that we use and produce, along with their associated quality characteristics. It's worth thinking about each of these characteristics and how they could apply to what you're producing or implementing.

There are many quality characteristics, and a search on the Internet would return many results on the subject. I've included the preceding characteristics because they cover the entire system and they're the primary characteristics I'll focus on within this book.

Why Quality (and Scope) Matter

Studies have shown that the cost of fixing bugs later in the lifecycle is generally higher than finding and fixing them earlier on. In the previous chapter, you saw that projects can fail or be seen to be a failure because of poor quality and poor scope.

I'm a firm believer that no system is completely defect-free and the short story I mentioned at the start of this chapter would go some way to support this statement. That's not to say that your processes shouldn't strive to achieve zero defects; it simply means that you're probably not going to achieve a truly perfect solution. In any case, you'd first need to define exactly what a "perfect solution" means. It will almost certainly mean different things to different people. If you can stand back and say, "It's my best work yet," then you're probably in a good place. That doesn't mean to say that you couldn't identify some areas of improvement or refinement. The problem is that you don't always get the time to revisit your work in the way that you would like. You test and review during the project to ensure that as many defects are captured and fixed as possible, although there's always the opportunity for some defects to slip through the net. In fact, software is very often shipped along with a set of "known issues." These known issues have been identified and are not seen as show-stoppers for the release. Minor defects are often carried over into future releases. The defects may affect a small area of the overall solution or occur only under very specific circumstances. There may be issues with documentation and other artifacts that are also not seen as critical.

Quality is in the eye of the beholder, and it really depends on what is perceived as quality and what is perceived as a defect. The word "defective" has many meanings, but the most applicable for this section are "lacking a part," "incomplete," and/or "faulty." If the scope isn't defined sufficiently, not only is the scope incomplete, it's extremely likely that the end result will be, too. Your primary goal is to ensure that your software and the associated artifacts are accepted by the recipient, whomever that may be and whatever it is you're delivering.

Quality (and scope) matter because *you* need to produce the final state solution. If the applications, environments, processes, and tools to achieve this don't enable you to do it effectively, you're in a bit of trouble before you've even started. A development environment that crashes every five minutes will have an impact on your ability to program. If the source control system is down, you can't access your artifacts. If it's not backed up, you run the risk of losing everything. Furthermore, if the scope isn't agreed to and understood, you have no idea what you're supposed to be producing, how you're supposed to be producing it, and what you're ultimately meant to be delivering.

Excerpted with permission of Wrox Press from Design - Build - Run

Quality (and scope) matter because whether it's source code, configuration files, database artifacts, test scripts, test data or documentation you're producing and delivering, it's typically subject to some form of inspection, review, and acceptance. This could be a peer review, a team lead review, or even an external review. It's no good writing huge specifications only to find that the requirements are wrong, or the specification is incomplete or there's not enough detail. Furthermore, let's assume that you're developing a solution for a third-party organization and you develop the components to adhere to your own internal standards and practices. If the solution is subsequently reviewed by the external party, all manner of issues could be found with it. In my experience, these types of issues can have a very dramatic effect on the overall outcome of the project and its perceived quality. It's no good finding out that all the documentation and code will need to be re-factored because they're not acceptable.

You need to estimate the effort involved in developing and testing the item to meet this acceptance. The level of testing will depend on the solution, but, ultimately, all the testing and acceptance is to ensure that the final product is fit for purpose. Again, it's no good finding out that the system should support automatic recovery when it hasn't been implemented. Although agile development techniques attempt to address this by ensuring that construction iterations produce a working solution, what's a working solution? End users may consider this as the solution incorporates all the functionality they require to perform their job. Support and operations staff may consider this as it meets all the necessary operability criteria. The application maintenance team may consider this as the software meets all of its maintenance criteria.

The most important thing in software development is to understand (and agree on) all the relevant quality characteristics and include them in the overall scope.

Construction Quality Echoes throughout the Project

Improving quality means improving everything you do from start to finish. This means ensuring that what comes out of the construction activities is fit for purpose. Initially, this means that it (the software) can be passed on to the test team for further testing and verification with the *minimum amount* of fuss, issues, and rework. You don't want your test activities to be *defect bound*, meaning that there are either too many issues to continue or there are show-stopping issues. However, you don't want to push the development out so far that it balloons the costs and timescales of the project.

The beginning of this chapter looked at some of the quality characteristics your systems should display. Now let's consider some of the processes and practices that you can employ during construction to improve the quality of your software and outputs, these activities include:

- Reviewing functional and technical designs and/or requirements
- Producing low-level technical designs and specifications for review and acceptance prior to coding
- Developing and reviewing components and documentation
- Developing and reviewing unit test plans, scripts, test data, and test harnesses
- Executing and reviewing unit tests and results
- Submitting components and artifacts for review and release
- Developing and reviewing integration test plans, scripts, data, and test harnesses
- Executing and reviewing integration tests and results
- Documenting completion reports

The exact order in which these activities are performed depends on your chosen development approach, and the degree to which any of these activities is performed determines the quality of the outputs from construction.

Figure 2-1 shows a hypothetical but structured construction process based on the preceding activities. The process outlined does not strictly follow any specific methodology; however, it covers the key activities listed as well as indicating an initial flow (and order) and incorporates review checkpoints.

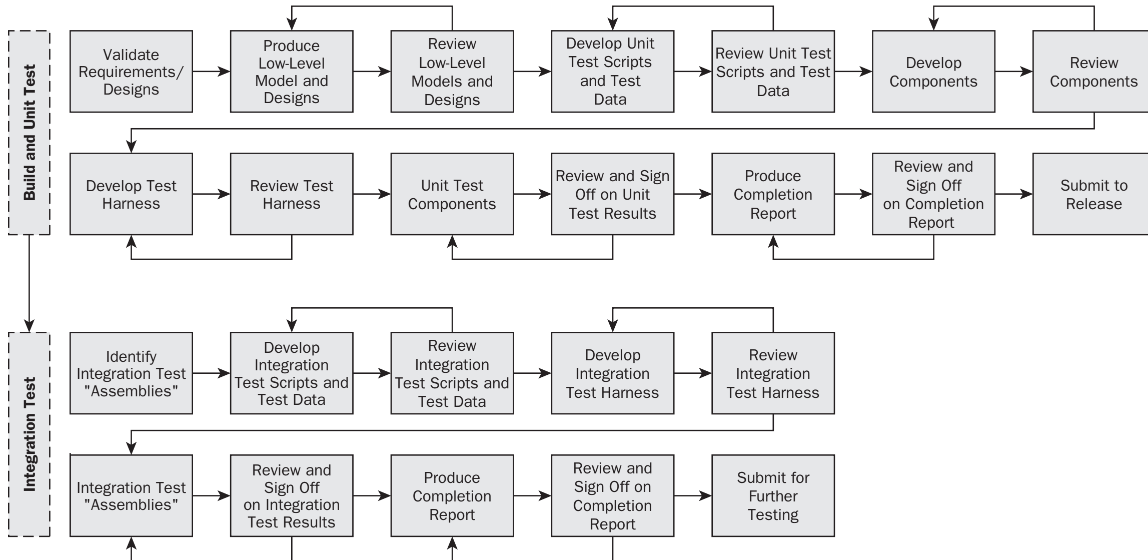

Figure 2-1

Although the process outlined might appear quite lengthy and regimented, it's really a question of how quickly each of the steps can be performed. You may already be performing these activities on your projects, although perhaps not in the same order as the plan shows or in a structured manner. For example, when you're developing your code, you typically review your code by reading through it. However, when you write down the individual tasks, it can really help to identify productivity enhancements and structured approaches. With the right tools and processes in place, there's no reason why the tasks and activities can't be executed in a relatively straightforward and expeditious manner. The process and the tools are all just another part of the overall scope of work.

As I mentioned earlier, the degree to which each of these activities is performed is truly where the quality bar lies. Setting the quality bar involves assessing the processes, practices, and tools, and evaluating these against costs and timescales. That said, a very high quality bar generally means a high quality solution.

The level and quality of testing and reviews you perform will ultimately determine the number of defects known or perhaps unknown in the construction outputs. Your mission, therefore, should be *to catch as many defects as early as possible and assess/fix them accordingly*. To achieve this goal, you will need to put some industrial-strength processes and tools in place, while balancing them against budgets and timescales.

The development methodology and approach will typically stipulate what the inputs and outputs of the activities are. In addition, the activities may not be performed in the order shown. Figure 2-1 depicts two high-level processes, one for "Build and Unit Test" and one for "Integration Test." These activities are

Excerpted with permission of Wrox Press from Design - Build - Run

described in the following sections and would generally apply to all components in the production-ready solution. This includes all framework and architecture components, application components, batch processes, reports, tools, scripts, and so forth.

The Build and Unit Test Process

The following describes the activities shown in Figure 2-1:

- **Validate requirements/designs.** First and foremost, you need to ensure that the requirements and designs meet the needs of the development team. That's not to say that if you think the design or requirements are poor or could be done better that you shouldn't flag it. You need to ensure that the inputs to the development phase are complete and unambiguous, and are sufficient for you to perform development, unit testing, and integration testing. During requirements and design validation, it's best to keep a log of queries, risks, and issues that may arise from the review, and to ensure that these are tracked and managed appropriately. These queries will need to be addressed before the construction can be fully closed off on the component. The requirements and designs should map back to the quality characteristics and include all the relevant items. The actual content of the requirements or design will depend entirely on the type of component that needs to be built and tested. However, the following provides a reasonable starting point:
  - **Functional design/requirements** — A detailed description of the component and what it is meant to do. This could consist of use cases, activity diagrams, flow diagrams, and textual descriptions to describe the component in detail. The functional aspects may also include various logging, usability, accessibility, and security information.
  - **Technical requirements and considerations** — A detailed description of the component's technical characteristics, such as failure and recovery scenarios, exception processing, and performance considerations. The technical requirements and considerations may also include areas covering configuration, scalability, resilience, and so forth. The requirements should map back to the relevant quality characteristics.
  - **Monitoring requirements and considerations** — A detailed description of the component's monitoring characteristics, such as instrumentation requirements and logging and tracing requirements.
  - **Operability considerations** — A detailed description of the component's operability considerations — for instance, whether the component is controlled by batch or how the component is started/stopped and managed.
- **Develop unit test scripts and test data.** The unit test plan contains the list of tests that are going to be carried out. Ideally, tests should be grouped into the following categories:
  - **Functional tests —** Covering the necessary functionality outlined in the functional requirements.
  - **Technical tests —** Covering the technical requirements including performance. Where it's not possible to conduct certain tests because of environment limitations, these need to be logged so that they can be carried out later.
  - **Monitoring tests** — Covering the monitoring requirements, including all instrumentation updates and logs.
  - **Operability tests** — Covering operability requirements. Again, limitations in the environment that don't support certain tests need to be flagged for execution in a later test activity.

Each group of tests should be further divided into successful scenarios and failure scenarios. Each test must contain a detailed description of the test being carried out and the relevant input data and expected results. The expected results should include the following (where appropriate):

- Return values and output values (including any external updates, such as file system, database, and so on)
- Instrumentation and diagnostic outputs (including events, tracing, performance counters, and so on)

There may be other outputs for the particular unit that should also be captured.

- **Review unit test scripts and test data.** The unit test plans and data are reviewed against the Develop Unit Test Scripts and Test Data Checklist. The checklist is essentially based on the preceding recommendations and practices — for example, ensuring that the unit tests are present and correct and contain all the necessary conditions and expected results, and that the test data being used is appropriate. The reviewer then provides comments back to the developer for further clarification and/or updates. Where necessary, the reviewer should work with the developer to ensure a full understanding of the comment and its impact. Communication is paramount in software development.
- **Develop components.** The components are developed according to the functional and technical requirements and specifications. While developing components, you should consider the items for the Develop Components Checklist:
    - All code must conform to the coding standards and guidelines.
    - All code is checked for performance and technical issues.
    - All comments adhere to the commenting standards and documentation-generation guidelines.
    - All functional and technical queries must be addressed.
    - All additional/modified exceptions and contextual information must be agreed on.
    - All modified input/output values must be agreed on.
    - All instrumentation and diagnostic updates must be agreed on.
    - All implementation updates and deviations must be agreed on.
- **Review components.** The components are reviewed according to the functional and technical requirements and specifications, as well as the Develop Components Checklist, and comments are provided for further updates. Again, the checklist is based on the outlined recommendations and practices. It is important to ensure that any modifications to the specification are agreed on by the relevant groups of people, such as end-users, business staff, support staff and so forth. It is equally important that this information is passed on to other teams, such as the testing team, to ensure that they are captured and incorporated appropriately.

Excerpted with permission of Wrox Press from Design - Build - Run

- **Develop unit test harness.** The test harness is developed in accordance with the unit test plan. During development of the test harness, the following also needs to be ensured:
    - All code must conform to the coding standards and guidelines.
    - All code is checked for performance and technical issues.
    - All comments adhere to the commenting standards and documentation generation guidelines.
    - All unit test conditions have the necessary test classes and methods.
    - All input data matches the unit test plan.
    - All output data and expected results match the unit test plan.
    - All actual results are verified (where possible, this should not involve manual effort).
    - Any unexpected results or conditions cause the tests to fail.
- **Review unit test harness.** The test harness is reviewed according to the unit test plan and the Develop Test Harness Checklist, and comments are provided for further updates.
- **Unit test components.** The components are unit tested and verified. During this activity the following needs to be ensured:
    - All unit tests pass and provide all the relevant assertions.
    - All code is covered. Where code can't be tested for whatever reason, it must be flagged so that it can be tested later or removed if not required.
    - All changes to expected results or actual results are agreed on.
    - All changes to implementation are agreed on.
- **Review and sign off on unit test.** The unit test results are reviewed and signed off on according to the Unit Test Checklist, which is based on the preceding recommendations. Comments are provided for further updates.
- **Produce completion report.** The completion report is compiled and includes the following:
    - Updated log with all queries addressed where possible and all open queries or additional testing requirements documented.
    - Completed unit test plan and test data.
    - Unit test harness source code and artifacts. These also include the actual results extraction scripts and tools. You may be able to use these later.
    - Component source code and artifacts such as configuration files.
    - Source code compliance reports. The compliance reports are generated from the static code analysis tools and the performance analysis tools. The compliance reports cover all source code, including the unit test harness and the component source code.
    - Unit test results, including all instrumentation extracts and logs.
    - Code coverage report detailing what code has been covered. Where it's not possible to cover certain aspects, a detailed synopsis is provided.

    - Performance and technical reports detailing analysis of components and database elements.
    - Review comments showing where each comment has been addressed or a detailed synopsis of why it has not been addressed.
    - All documentation, whether generated automatically or otherwise.
- **Review and sign off on completion report.** The completion report is reviewed and signed off on according to the Completion Report Checklist, and comments are provided for further updates.
- **Submit to release.** Once the completion report has been reviewed and signed off on, everything is in place and can be submitted into a formal release according to the release process. Each release package will have different configurations and include different artifacts; as such, all artifacts included in the completion report should be included in the configuration management system and submitted to build (where appropriate).

The scope encompasses the activities, tasks, and outputs of the chosen construction process. The activities are intended to deliver a high-quality solution to the further test phases. Later in this chapter, you look at the costs associated with quality, and these can be applied more importantly to the costs associated with poor quality. Whether it's a tool, a core architecture component, or anything else for that matter, the construction quality needs to be met and maintained through the lifetime of the project or solution. Although I'm not dictating the actual approach, the essence of the tasks and activities should be considered carefully. This is just one area where quality and scope are interrelated. The more activities involved in construction, the longer it's likely to take and cost. However, to balance this out, savings can be made further down the line by reducing testing and fix effort as well as support and maintenance effort.

Code Quality 101

It can sometimes be very difficult to agree on the necessary quality characteristics because of different points of view. For instance, in the previous chapter I posed the question, "What defines code quality?" Code quality can again mean different things to different people. However, as far as I'm concerned the following are just a few of the key principles and practices of "code quality 101":

- Conforming to naming conventions, coding standards, and best practice architectural and language patterns
- Well-structured and commented code for maintainability and readability
- Employing layering, isolation, and encapsulation techniques to promote re-use and to reduce duplicated code sections
- Modular code that is not overly complex, not too difficult to understand and test, such as not including large classes, large methods, and multiple nested conditions
- Not including any redundant code, unused libraries, and/or parameters
- Using configuration values instead of hard-coded values and "magic numbers"
- Using interfaces, late instantiation techniques and "mock" objects or stubs/simulators to support thorough testing
- Including exception handling and resilience patterns

Excerpted with permission of Wrox Press from Design - Build - Run

- Including logging, tracing, and diagnostics for operability and supportability
- Efficient use of system resources and tuned for “production” performance

I’m using “code quality 101” to refer to the very basic principles and practices for quality code. “101” has long been used by teaching institutes and training vendors to identify the first and most basic course in a series.

It is entirely possible that the artifact could meet all its requirements without ever incorporating any of the above, especially if they’re not defined and included within the scope. For example, I could ask someone to write a simple tool for use during incident investigation. It definitely depends on the person as to whether all, some, or none of the preceding would be taken into account. More important, it really depends on how I set the scope for the task. In the software industry, we often take “best practice” for granted. We sometimes assume that because these practices are well known, every program will (and must) conform to them and that every developer will incorporate them. However, I may not actually require all the “best practice” for a particular task. There’s the infamous “throw-away code” situation. This is when something needs to be done very quickly and it’s only required for a very short-lived period. In this situation, performing “all” the best practices isn’t always necessary; however, the item still needs to be fit for purpose. The item is simply a means to an end. However, I often find that “throw-away code” isn’t just a means to an end, and if I need it now, then I need to really understand whether it’s needed in the future.

There can be very differing opinions on what “best practice” actually is and what it means. As the software industry progresses, new practices and patterns are always identified and become “the thing to do.” If the quality is included in the scope, then a system that meets its completion criteria will also meet all its true quality criteria. The following provides some additional context for some of the code quality items presented in the preceding list:

- **Standards and guidelines —** Development standards ensure that all developers are writing similar code and artifacts that adhere to a given set of standards. These need to include commenting standards, naming conventions, configuration, exception handling, instrumentation, and logging and architecture usage. The standards also need to cover the rudimentary practices for performance and other technical characteristics. These standards and procedures should cover all languages and technologies, such as database artifacts, source code, and scripting languages. In addition, the standards need to be followed for everything that this developed, including tools and productivity scripts.
- **Reusable components and layering —** During development, you will develop a number of different components — some for the core application, some for batch, some for reporting, and some for tools. Layering the architecture and components so that you can reuse as much as possible for many different purposes will reduce overhead and improve the overall quality of the solution. For instance, a logging component should be able to be used everywhere to ensure consistency throughout the system. Ensuring that architecture components are application independent greatly improves reusability. Often architecture components are built with the assumption that they are going to be used by the core application and, as such, are not so adaptable to other uses such as batch, reporting, and tools. Batch and reporting generally involve developing *generic services*. For example, a common batch component is one that can execute a stored procedure and write the results to a file. This functionality could be reused by your test tools to extract actual results from the database for comparison against expected results. Furthermore, these scripts could be used in live environments to extract data for issue investigation.

- **Configuration values and settings —** Your system is going to need to deal with a number of different environments and situations, and one size doesn't fit all. You need to have an appropriate level of configuration to support this. Your instrumentation and logging should be highly configurable to support the different levels required in the different environments. You should also ensure that database connectivity is highly configurable for the same reasons. Low-level configuration components can be used to standardize access to configuration values and settings, parts of which can often be generated from the configuration files and they can also be reused by tools and other components.
- **Instrumentation and diagnostics —** You need to ensure that the appropriate level of instrumentation and diagnostics is in place in your applications and tools so that when issues arise, they can be tracked down quickly and efficiently. Test tools are often thought of as a means to an end and don't often include logging or instrumentation. However, including logging and instrumentation in the tools themselves enables you to measure their own performance and effectively diagnose any bugs in them. Incorporating reusable instrumentation and logging classes within the architecture allows you to make use of them everywhere. You must also ensure that instrumentation and logging don't adversely affect performance. The instrumentation needs to be configurable so that it can be tuned for different environments. This is to ensure that you are in a position to react to issues and turn them around quickly. It is equally important that you test and validate the instrumentation and logging outputs to ensure that they are correct. When you get to the real issues with your system, the quality of your instrumentation and diagnostics will count more than anything else. Having productivity tools and scripts to gather and extract logs, instrumentation, and events allows you to verify their correctness during testing and can be reused during live service incident investigation. Instrumentation and logging should be highly configurable so that the most appropriate levels can be calibrated in each environment.

The following sections take a slightly closer look at some of the practices I've mentioned in the build and unit test process, namely:

- Code profiling and peer reviews
- Code coverage
- Unit testing
- Documentation generation

Code Profiling and Peer Reviews

There are many tools to assess code quality. These tools can automatically highlight various errors with the code. Some tools work by analyzing the code from a static point of view, whereas others examine the solution while it's actually running. However, it may not be possible for all your code to be automatically profiled. For instance, you might be using a custom package that doesn't support it. Therefore, documented standards and "Mark One Eyeball" (or manual) reviews are typically the only way of checking the quality. In fact, manual reviews and checks should still be performed to ensure complete quality.

As much as your processes and tools allow you to perform your own quality checks, a peer review will uncover many different issues. Peer reviews should be performed throughout the development process, and should focus on all areas, not just code. Peer reviews should cover unit test plans and data, integration test plans and data, source code, and other artifacts, as well as release readiness reports. A peer review checklist should be used and updated as they are conducted. Peer review comments should be documented thoroughly and managed appropriately. Where common issues are found, they

Excerpted with permission of Wrox Press from Design - Build - Run

should be filtered through to the productivity and process guides to provide more information. It often helps to have a Top 10 list of common issues to avoid future occurrences.

> Mark One Eyeball *is a basic military term that refers to visual reconnaissance instead of using any high-tech means when on maneuvers or out searching.*

Using code analysis and code profiling tools can help to ensure adherence to standards and highlight potential performance issues with the code. It also helps with peer reviews because the outputs from the profiler are the starting point for the review. If the report shows a large number of errors that can't be explained, it is a fairly reasonable indication that the code isn't ready.

Static code analysis generally checks the code against coding standards, abstraction and dependencies, configuration settings, and other statically identifiable issues. For example, the profiler may detect the keyword `new` inside a loop and raise a warning. Static code profilers can often find potential areas for bugs, memory leaks, and so on. They often point out many naming standard errors, unused libraries, methods and parameters, and hard-coded values.

One very important aspect of code profiling is the "cyclomatic complexity" reporting features. In short, cyclomatic complexity is a measurement of how complex the code is and the number of test cases required to achieve coverage. The measurement is based on the number of branches and paths in the code. The more branches and paths in the code, the more complex it is.

Dynamic code profiling takes place when the code is actually executing. The tools typically look at the resource utilization of the application, memory, CPU, database, and so on. They can often highlight actual memory leaks and inefficient code. The output reports are used to tune the application accordingly.

These tools often require up-front configuration and may not support all languages and technologies used in the project. As such, they should be complemented with peer reviews to ensure that the code adheres to the appropriate standards.

Code profiling should be used carefully, and often some components need to deviate from the standards for very good reasons. These components need to be noted, understood, and incorporated. In general, using these tools early on can provide some useful insights and help to improve code quality during construction. When the technical test fully ramps up and starts probing, you can bet your bottom dollar that it won't be long before a profiler is wheeled out to "inspect" the solution. Having them used up front will really help to avoid re-work later. The technical testing often encounters very convoluted issues that code profiling can often help to diagnose.

Code profiling should be automated and easy to execute. It's more likely to be used if it can be executed easily. The results need to be captured for analysis and possible correction. It's important that the team understand the outputs of the report in order to address the issues. Writing a guide for executing and analyzing code profiling is a good place to start.

In addition to code profiling tools, stored procedures and other database artifacts can be profiled to identify potential bottlenecks in the data layer. This is typically achieved by running an execution plan against the stored procedure for various scenarios. The report will show how efficient the procedure is and will often highlight areas for improvement.

It's important that the scope includes the use of these tools (where determined) and, more important, the details of the underlying rules. All too often there's a debate over which configuration is the right one and which rules should be switched on or off.

Excerpted with permission of Wrox Press from Design - Build - Run

Code Coverage

Using a code coverage tool effectively is a great way to determine how much of the source code is executed during testing. The outputs can be used to determine whether additional tests need to be developed to hit more code or, in some cases, the results can identify redundant code. The code coverage tool typically injects instrumentation into the code so that when a test is executed the tool can examine how much code has been executed. I typically refer to these as *covered builds*. The code is measured on a number of criteria that includes but is not limited to:

- **Function (methods)** — Measures the methods that are executed.
- **Statement** — Measures the statements in each method that are executed.
- **Condition** — Measures the conditions that are executed.
- **Path** — Measures the conditional paths that are executed.
- **Entry/exit** — Measures the number of permutations of entry and exit for the function (method).

It may not be possible to automatically check code coverage for your entire solution; therefore, the unit test approach needs to be documented and understood so that the development team produces and executes scripts that cover the required amount of code. The code review approach needs to be documented and understood so that Mark One Eyeball reviews confirm this has been achieved. Where possible, the unit test approach and review should be aligned to the rules outlined previously.

Mission-critical systems often require 100 percent of the code to be covered. If you commit to achieving 100 percent code coverage during unit testing, you'll need to include tests for every condition and every possible scenario. To ensure this happens, you may need many additional test classes, stubs and/or mock/simulator objects, as well as the conditions and scripts.

Developers often write defensive code — for instance, including `if` statements and `null` checks. While this may be seen to add to the robustness of the solution, you actually need to test that they work. If you have high code coverage criteria then you will need to devise tests and data in such a way that each scenario can be fully tested. I personally believe that every line of code you write should be tested thoroughly.

Achieving 100 percent code coverage during unit testing doesn't mean the application works the way it should or that it even displays all the necessary quality characteristics. It's just one part of the scope and quality. Maintaining unit tests and integration tests can be a costly business and, therefore, it's important that they do exactly what they should — to ensure that the system works correctly.

Unit Testing

Although the concept of unit testing has existed for a long time, there are differing views on what unit testing should actually test and how it should be performed. Traditionally, unit testing has been categorized as a "white-box test" that tests the smallest part in the solution, such as a single method. The term "white-box" is used because you can see the actual code that is going to be executed by the method. However, in test driven approaches, the tests are initially based on the designs (functional and technical) and the public interfaces exposed by classes because the actual code hasn't been written at the time the test cases are being devised. In either case, unit testing should be aligned with code coverage in that the tests should be devised to cover the following aspects of the unit:

Excerpted with permission of Wrox Press from Design - Build - Run

- Statements
- Conditions
- Paths
- Entry/exit permutations

Unit tests are independent of each other, that is, they shouldn't rely on the outcome or state from a previous test. Unit tests should support being executed in any order.

Unit tests aren't really supposed to span outside the boundaries of the actual class being tested — for instance, if the class being tested contains a method that makes a call to another referenced class, the unit test spills into the referenced class. To support testing the unit in complete isolation, an interface should be defined and a "mock" object should be developed and used during unit testing. This approach can improve the overall quality of unit testing because the mock object can be written to simulate various conditions.

Mock objects are often referred to as *stubs* or *simulators*. The complexity and features of the mock object will depend entirely on its purpose. In most cases, the mock object needs to contain conditions for each of the possible calls and permutations from the consumer. The development and test effort required for this needs to be captured in the overall budgets and timescales. Furthermore, a lot of time, effort, and money can be wasted trying to determine whether it is the code that doesn't work or the mock object that is not functioning correctly. It is not always possible to keep the mock object simple but the more you try to, the easier it is to test and fix.

The level and approach to unit testing is entirely dependent on the system being implemented. In some cases, unit testing is a form of integration testing in that some objects are not stubbed and as such they are called as part of the overall test. Although this can be seen as wrong in the purest view, it is important that the approach is agreed to by all the stakeholders, rather than having a debate about it afterwards. A classic example is instrumentation and diagnostics. It is perfectly possible to mock these objects and make gathering the output for completion much easier. However, using the real object early on (where possible) will ensure that the system works the way it should in production. This is nothing more than striking a balance between reality and theory. These decisions and the unit testing approach need to be documented and included in the scope, as they will have an effect on the budgets and timescales.

There are many tools that help to automate unit testing rather than you having to execute scripts manually. I'm definitely a fan of automated unit testing. The business of executing tests manually is simply a chore and it also means that an automated regression test capability can't be employed, which affects the regular integration approach and adds unnecessary delays on the project.

Documentation Generation

Using documentation generation tools helps with the production of system documentation for handover and future development purposes. Documentation is generated from the *comments* in the code and can be a lot easier than writing it by hand. The generated documentation typically includes:

- Namespaces and namespace hierarchy
- Class, interface, delegate, and enumeration lists and descriptions

- Public property, method, and parameter lists and descriptions
- Return value and exception lists and descriptions

The output is usually a compiled help file that contains links to make it easier to navigate. However, these tools do not support all languages and technologies and often require up-front configuration to be used effectively.

Your commenting standards need to stipulate what needs to be included in your code to support documentation generation, and your process guides need to contain step-by-step instructions on how to produce the relevant system documentation.

Many other practices can be employed during the construction phase and some of these are highlighted later. The practices listed here are simply a core set of practices that should be considered during build and unit test to ensure high-quality outputs. While the amount of work may have increased during construction, the savings further down the line will be truly beneficial.

The Integration Test Process

The integration test process and activities are very similar to their unit testing counterparts, so I won't go into the details again. Integration testing is generally considered *black-box testing*. The tests are designed to call the *assembly* through a *public* interface on the first class (or object) in the assembly, and assert the expected results. The tests are designed to cover the main integration scenarios between the individual components. Running integration tests and examining the code coverage often highlights areas where additional tests could be developed or where there's redundant code in the solution. Redundant code isn't always picked up by static code profilers because they typically check only to see if there are references (or calls) to classes and methods within the solution. The code coverage output shows exactly what was executed and called. However, this assumes that you have a very thorough set of integration tests.

Successful integration tests start with identifying the assemblies in the solution. It is important to get the right level of granularity when performing integration tests. Integration testing can be performed in many ways. However, in a lot of cases, integration tests start by testing the system with real components where stubs or mock objects were being used for unit testing. This has the effect of testing the flow through all the components below it. Each layer is tested in turn, moving up to the very top layer. This is typically referred to as a *bottom up* approach. However, in some cases, an assembly can be an entire vertical slice of an application — for example, the Create New Account function. It is important to identify and agree on the assemblies and the approach early so that they can be tested appropriately and avoid issues later in the lifecycle.

It's very important to remember that integration testing may also need to use mock, stub, or simulator components. Although this might seem to be a contradiction in terms, integration testing doesn't always test the entire solution. For example, suppose you're using a third-party component for username/password authentication. There may be constraints whereby you can't use the actual component during integration testing. For example, it might require some backend features that aren't available. Therefore, it will need to be "stubbed" or "simulated" during development and integration testing.

Excerpted with permission of Wrox Press from Design - Build - Run

The integration tests are again organized into the following various categories and test both success and failure scenarios:

- Functional tests
- Technical tests
- Monitoring tests
- Operability tests

It is important during integration testing that the entry/exit testing be thorough and that it exercise all the various permutations and possible outputs to ensure that the assembly is fit for purpose. Testing a single permutation will only go so far in assessing the quality of the build.

The construction process should focus on capturing and correcting as many defects as possible prior to formal testing. What comes out of construction will ultimately determine how straightforward the remainder of the project goes. When there's a mass of defects raised beyond construction, it is not long before someone says "How many of these could have been detected earlier?"

Defects Affect Testing and Ongoing Development

You saw in the previous chapter that you can potentially have multiple test activities and/or phases running in parallel. Each of the test teams will generally want their fixes as soon as possible. Functional testing may argue that the system needs to be functionally correct and performance doesn't matter at *this* point. Performance can always be enhanced and tweaked later, or so the argument goes. Technical testing, on the other hand, may argue that the system needs to be technically correct. Obviously, both sets of requirements need to be met to ensure that the system is production-ready. The fixes need to be managed, prioritized, and implemented according to their allotted priority. Furthermore, functional and technical test phases can often be broken into multiple streams of testing, with each stream concentrating on a particular area or part of the system. For instance, technical tests might split, with one stream concentrating on failure and recovery scenarios, while another focuses on monitoring and operability tests. Functional tests may be split into functional areas or application slices. For example, one area deals with creating new accounts while another area deals with online shopping or cart management.

Once you start testing (and acceptance activities), defects start getting raised and fix turnaround time becomes paramount. Fix turnaround time is the total time it takes to perform all the necessary activities to get a fix out to the required environment. There could be multiple teams of people that are potentially unproductive because of show-stopping defects. Testing and acceptance is basically halted until certain issues are resolved. This can impact not only deadlines, but also costs. If customers are involved in the testing and there are major defects, it can also cause further perception issues with respect to the product's quality. If the development team is impacted by a large number of defects that need to be fixed, it can also cause further delays and cost overruns to outstanding development efforts, such as the inclusion of additional functionality.

When formal testing begins, there are usually many issues that need to be addressed before testing is fully up and running. These early issues are generally not related to the actual application or its functionality. However, there is a fine line between the two.

Regardless of whether you are conducting unit, integration, or system testing, the profile of defects generally follows the path outlined in Figure 2-2.

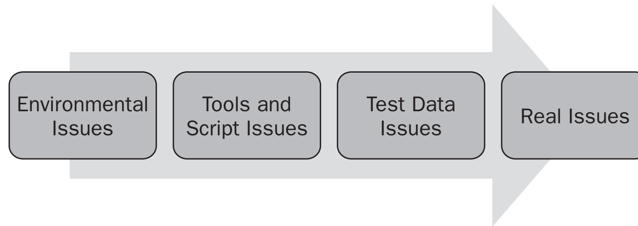

Figure 2-2

Each of these high-level defect categories is discussed in the list that follows:

- **Environmental issues —** These issues concern getting the test environment up-and-running. Assuming that the test hardware is in place, these issues are more concerned with access to the environment, deploying the application in the environment, ensuring the correct configuration settings are in place, installing the test data, and bringing the application into a testable state.
- **Tools and script issues —** These issues concern executing actual test scripts, getting the test tools up-and-running, injecting transactions manually or automatically from the tools and scripts, and determining the results. The scripts may have been developed separately from the test tools and this is the first time they've been put together. The tools themselves may be new, and "operator error" may come into the equation. Other issues can be encountered when trying to compare results — for example, actual results can't be extracted or compared properly.
- **Test data issues —** With the environment and script issues resolved to a degree, you typically start to experience issues with the test data. Test data issues can cover a wide variety of scenarios, including the following:
  - Not all the test data is installed.
  - There's too much test data to load in the test environment.
  - The data is actually incorrect or incomplete.
- **Real issues (bugs) —** Once the testing activities are truly up-and-running, you start to identify the real bugs in the system. That's not to say that you won't find anymore bugs in the environment, test tools, or test data, but you've gotten yourself into a position to be able to test and uncover the real issues that need to be dealt with.

Testing is the primary proving ground for the solution. No matter which approach is actually taken, the software must ultimately meet all its functional and technical requirements; therefore, the tests that are carried out need to ensure that this is achieved. The quality of the software and the testing is going to be the biggest area of concern when testing begins. Having fit-for-purpose and production-ready applications, environments, processes, and tools, as well as trained test (and fix) resources, will greatly improve the quality and performance of your testing.

Fixing Defects Quickly Can Reduce Quality

When testing halts because of show-stopping issues, the development or fix teams need to turn around defects fast. In situations like this, the quality can start to drop because you need to get fixes out to the test streams very quickly. All the good stuff you put in place during development typically gets put to

Excerpted with permission of Wrox Press from Design - Build - Run

the side in order to drop a fix to a test team quickly. There isn't enough time to update the design; have it reviewed and signed off on; update the code; update all the unit tests; update the integration tests; update the regression tests; run them all; ensure code coverage remains the same; profile the code for memory leaks, performance, and adherence to standards; generate the documentation; collate all the results; and have it all reviewed and signed off before delivering a fix into the build process; have it go round the entire build and regression cycle; and finally getting it installed in a test environment. That said, not all defects will need to go back as far as the requirements or design stages, and everything will depend on the criticality of the defect. In addition, the level of tooling that is used can dramatically reduce the overall time to deliver a fix while retaining the appropriate levels of quality. In most cases, the quality can drop when:

- Testing is blocked
- Tuning and re-factoring are required

These scenarios are discussed in the following sections.

When Testing Is Blocked

When either show-stopping issues occur or a large number of fixes need to be delivered during testing, the development team generally turns into a rapid response unit and really needs to turn around fixes very quickly to keep the test teams up and running and on track. This is often achieved by short-cutting the quality process to get the fix out the door and sweeping up the other items later. Interestingly enough, it is exactly the same situation with issues that are encountered during live service, although these are generally tested more thoroughly prior to being deployed.

An example of this situation is the infamous "one-liner" — a very simple change needs to be made to one line of code that will only take two seconds to implement. The test manager is breathing down your neck about how many people are waiting for the fix, and the development manager is embarrassed at how such a simple issue wasn't found during build, unit, and/or integration testing.

In these types of situations, it's not very long before the decision is made to compile the solution and drop a few "hot-fixed" DLLs into the environment to keep everyone happy and everything running smoothly. Another very common example of hot-fixing is the even more infamous "don't know, can't reproduce" type of issue. You've looked at the code, the data, the scripts, the logs, and everything else, and you just don't have a clue what's going on. This invariably leads to adding more trace statements to the component in the hope that this will provide an insight into what is going on so that the defect can be found and fixed. This again is very often the case during technical testing and early live service because in general verbose logging and tracing is turned off to improve the overall performance of the system.

The rationale for hot-fixing when applied to a test phase goes along the lines of "If people can't do anything anyway, we can't really make the situation any worse, so let's give it a go." Issues encountered during live service may still shortcut the full quality process, but to a much lesser degree than during test. More often than not, the fix will be tested and proven, put into a formal release and regression tested prior to being deployed to the production environment.

A formal hot-fix or "patch" process should be identified early to ensure that it delivers quality artifacts and doesn't necessarily leave an abundance of tasks to be carried out later. It's equally important that all hot-fixes or patches are uniquely identified and have a release note, just like any other software release.

When Tuning and Re-Factoring Are Required

During development the full end-to-end product is not typically technically tested in its entirety. For instance, certain failure and recovery scenarios may need to be tested in a specific environment. Load testing may also require a specific environment. Only when the complete system is tested in a live (or live-like) environment, with live data and live situations, do you get a true indication of the system's actual technical characteristics, such as performance, stability, and recovery. This is also true of some functional characteristics, but these will generally be dealt with in the normal way, unless of course they happen to be show-stoppers. It's not often that components need to be completely re-factored as a result of functional inadequacies, although it has been known to happen. Core algorithms and calculations can be so badly written that they simply have to be rewritten.

When a component isn't functionally or technically satisfactory, it may be tuned or sometimes re-factored completely. This can often be to the detriment of its functionality but more so to the detriment of its quality. The component is a bottleneck to the continuation of technical testing and needs to be fixed and fixed quickly. The really technical folks know all about tuning, but explaining this in a defect report will take too long and the turnaround time is not quick enough. In some cases, this leads to a situation in which the technical team makes local updates to the component to continue testing.

Furthermore, this technical tuning and re-factoring exercise generally doesn't include updating the documentation, updating the unit tests, updating the integration tests, and so on, and once again the quality drops, leaving everything else to be swept up later.

A formal approach to tuning and re-factoring should be agreed on so that whatever happens, the components maintain their quality. Clearly, the more that you can do during the construction phase to ensure that the components are technically correct, the better. Not having very large and unwieldy components in the solution can help. The smaller the component, the less there is to change.

When Quality Drops, Sweeping Is Left

Sweeping is a slang term to describe tidying up the system and bringing it back up to its original quality bar. How high you set that bar will depend on what needs to be done. However, with respect to the construction process outlined earlier in this chapter, sweeping would actually involve all the following tasks:

- ❑ Updating designs and other documentation
- ❑ Sweeping the code and updating comments, updating logging and tracing, updating exception handling, and generally tidying up the code
- ❑ Updating and correcting unit tests and integration tests, as well as associated documentation, scripts, and test data
- ❑ Re-executing all the quality and performance tools to ensure that adherence to standards and profiles and code coverage is achieved

In fact, sweeping is everything you would do during the construction phase, albeit in a usually compressed timeframe (which, again, can affect the overall quality of the outputs).

The rationale behind sweeping can be summarized as follows:

Excerpted with permission of Wrox Press from Design - Build - Run

- When the level of defects drops, the fix team can split their time between fixing bugs and tidying up everything else.
- As long as the "smoke" tests that are included within the regression testing pass, the solution is "good to go" even if the unit tests and integration tests don't pass. The cost implications of sweeping are not always detrimental to the overall financial state of the project; however, depending on the size of the system and the level to which the quality has dropped, it can take a large effort to bring it back up again. Sweeping exercises really need to be planned and executed effectively; otherwise, they can introduce more defects and again reduce the overall quality of the solution.

There are two main reasons for bringing the quality bar back up. The first is that when the system goes into live running and maintenance, it needs to meet the original quality characteristics so that it can be supported and maintained efficiently. The second reason is that most projects have multiple releases, and when you pass all the artifacts to the next team, everything needs to meet the required quality so that they can use them to effectively design and develop the next release.

The bottom line: Wherever possible, quality should be maintained throughout the project and not left to a sweeping or cleanup operation.

More Tips for Improving and Maintaining Quality

In the previous section you saw the typical profile of defects once the software leaves construction, and their potential impact. You need to use this information to your advantage to improve the quality of your processes, tools, and applications. You've also seen some practices that you can employ during construction to ensure your systems meet the necessary quality characteristics. These included thorough unit testing and integration testing, as well as including instrumentation and diagnostics. The following are some additional tips for improving construction quality and the construction process:

- **Work with the test teams.** While you're in development, the test teams are busy working away on their own agenda and that's usually developing their own test plans, test scripts, and test data. First, you need to ensure that what you are doing is in line with their expectations and that you are working from the same sets of requirements and designs. If you are developing to version 1 of a document and the test teams are working off of version 2, what you will deliver is not going to match their expectations and you'll encounter issues (see the "Define releases and their content" bullet later in this list). Another benefit of this relationship is getting a bird's-eye view of the types of tests that will be executed and the input data and expected results. You should use this as much as possible during development (see the following bullet).
- **Use common test data.** Where possible, you should use a common set of test data during unit testing and integration testing to avoid issues later in the lifecycle. It is often not possible to replicate the exact quantity of the test data — for instance, technical test will generally use much larger sets of test data to fully stress the system, but you should look to use a reduced set of the same data. Identifying common configuration and transaction data greatly reduces test data issues during functional and technical tests, and simplifies test scripts and tools.
- **Separate data and databases.** There are a number of different environments and processes that you need to support and having effective data and database management in place will help. As mentioned previously, using common test data assists with this greatly, but you will still need to support different data and databases in different environments. Separating data and databases enables you to deploy only the required databases, artifacts, and data required. You don't want

test tables, test views, or test tool databases deployed along with the primary database into the production environment.

- **Use common test scripts and scenarios.** In the same way that you should try to use a common set of test data, you should also try to minimize the number of different test scripts that are used. This is especially true for integration testing when you are testing a set of components. Technical tests will usually isolate a single integrated set of components and put them through their paces. Where you can, you should align your tests to get early insights into the technical characteristics. Obviously, the tests are conducted in different environments and, as such, the expected results may need to be tweaked, but aligning yourself against what will come next can greatly reduce the number of issues you encounter. It's the same for functional tests; although functional tests will ripple through functionality in a more end-to-end fashion, you can still align yourself nicely to reduce the impact. Your unit tests and integration tests should be automated and exercise as much of the system as possible via the binaries and not rely on manual effort.

- **Use common test tools, stubs, and simulators.** Where possible, the test tools that are going to be used during functional and technical testing should be used during the development phase. This irons out a number of issues with the tools early on and eases the testing processes. Using off-the-shelf tools enables you to configure them appropriately for your testing needs. By developing your own test tools, you can ensure that you capture the requirements of the test teams to ensure that they are fit for purpose. In some situations, you might be using third-party or other applications that are not available during development, and you'll need to develop a stub or simulator to exercise certain functionality. You should try and use a common stub or simulator for all testing activities. Where possible, you should use the actual third-party components to avoid downstream issues.

- **Track releases and features.** Requirements and designs change during the project lifecycle, and often it's not possible to incorporate a change immediately. You need to keep track of what features are in each release (the Release Note) to ensure that test teams install the appropriate release for the tests that they are performing. If a tester is following a test script for something that's not included within the release, this script isn't going to pass and an issue will be raised against the software. Release planning also tracks which defects have been fixed in which release. Patches and hot-fixes should also be tracked, just like any other release.

- **Define releases and their content.** A release doesn't just include DLLs. It can include configuration files, database scripts, productivity scripts, data, test scripts (including expected results), and documentation. Everything that is required for a particular purpose needs to be included in a release that can be installed quickly and easily with all the right artifacts and configuration. Copying files from one place to another is often tedious and time consuming. The installation package should be complete and should not require access to network drives, source control, or any other repository. This allows the consumer to install the software and artifacts in a completely independent and isolated environment. Things move on and source control or network drives have the *latest* view. If the test scripts have been updated because of a change or enhancement, they may no longer work with a previous release and the latest release may not be at the correct stage to be deployed. The packaging solution should have different configurations for each purpose, such as developer testing, functional testing, and technical testing, as well as supporting custom configuration to allow picking and choosing of what is to be installed. Silent installation should also be supported to reduce manual intervention and support productivity tools and scripts (see the upcoming bullet "Develop productivity tools and scripts").

Excerpted with permission of Wrox Press from Design - Build - Run

- **Separate test matter from production.** Throughout development you may introduce different release types, such as Debug, Test, Final, and so on. This ensures that only the relevant components and artifacts are compiled and included in the resulting binaries. It avoids including test statements in a final release and allows you to have specific features in test releases, enabling you to better test some of the more complex functionality of the system. However, you need to ensure that switching between release types doesn't impact your ability to find, fix, and test. You don't want to be messing about too much installing different releases to ensure that the code works correctly. Functional and technical testing will generally use final releases to remove all ambiguity. However, there may be instances where functional testing can't use all the "live" components; therefore, stubs and simulators might need to be used. It's worth thinking about how the number of different code bases can be reduced, a topic I'll discuss further in Chapter 23.

- **Perform regular integration and automated builds.** Performing regular builds ensures that everything that is *ready to build* is included in a single build and doesn't affect anything else. The build process should be automated as much as possible as this will be used moving forward to deliver into the release process. It is important that every developer knows what ready to build means, what is required, and how to submit their artifacts into the process. This avoids unnecessary issues, including missing files and compilation errors, when everything is brought together. The regular build process should build all the various compiled release types as well as *covered builds* for each of them.

- **Fully regression test build and releases —** Once everything has been brought together into a single build, it should be *fully* regression tested. At first this will consist of executing all the unit tests. However, it could also include some rudimentary "smoke" tests. Moving forward, the regression pack would include and execute the integration tests, and, ultimately, it will be extended to cover a magnitude of tests, including functional tests and technical tests, collectively referred to as "smoke" tests. The regression test tools need to be extensible to be able to support different test scenarios. Unit tests and integration tests may need to run against a particular release configuration and its *covered* counterpart, whereas the functional and technical tests will need to run against a final release. The interesting thing here is that you can run all the functional tests and technical tests against a *covered* build and see how much code is exercised, as it often provides very useful results. Once the quality bar has dropped and the unit tests don't work or the integration tests don't work, the only tests left are the functional and technical regression tests, which can start to become the single measure of quality. The regression tests have usually been built up into a really wide reaching set, covering a wide variety of functionality and technical features, so they should be capable of validating more of the system from a live-like fashion. The tests are generally automated, so in some cases they won't cover the more manually intensive tests. However, they are a good indication of "functional" quality if all the tests pass. If the system passes these regression tests, it is generally good enough to be deployed, or that's the idea anyway. In many cases, this can be true — just because the unit tests pass doesn't mean that they are functionally or technically correct. Once the quality bar is dropped, there is no way of telling whether it's the code that doesn't work or the tests that are wrong. It's usually the tests that are broken, as the code has been "smoke" tested in the environment and proven to work. So, the broken unit or integration tests now need to match the code along with everything else that needs to be addressed. Misaligned code and unit/integration tests should be frowned upon. However, having a rich set of regression tests and an extensible regression capability will allow the product to be tested in a variety of ways, preferably in parallel to get a true measure of its "overall" quality.

- **Define a ready-to-build and ready-to-release process.** Before anything is submitted to a build or a release, it must pass all the required quality checks. First, the process will need to include steps that ensure all the source code compiles and runs with the latest code base. Additional steps in the process will involve ensuring that all the relevant tests have been executed (and passed); all the required results are in place (including log files, instrumentation reports, and event logs); all the necessary documentation has been generated and is correct; all the test scripts, expected results, and actual results are in place and match; all the necessary profiling reports are in place and meet the required quality level; all requirements and design comments have been incorporated or addressed; and that all peer review comments have been incorporated or addressed.
- **Develop and use templates containing TODO statements.** Copying and pasting is very dangerous during development. Logging statements, instrumentation, and comments are not updated properly, leading to poor quality documentation and code. Providing a set of base templates with TODO notes helps to ensure that all your components follow an agreed pattern — for instance, `TODO – put your implementation here.` TODO notes need to be checked and then removed from the code prior to delivery to build to avoid any confusion later down the line. The templates also need to adhere to the standards and guidelines.
- **Develop productivity tools and scripts.** Do it regularly manually and it should be automated. Reducing the amount of manual effort by providing tools and scripts not only reduces the number of issues but increases the speed at which the task can be done. For instance, turning an environment around from development testing to functional testing can involve installing a different release with different test data and scripts. Gathering log files, databases extracts, and other artifacts supports many purposes and should be automated. For example, after running the unit tests, you want to capture all the log files, events logs, and performance counters to include in build completion. These tools and scripts can also be used during testing and live running to assist in issue identification and resolution.
- **Using code generation techniques —** Generating code through the use of code generators often speeds up development by effectively automating laborious tasks. Code generators are often used to generate data access components because they can be driven from the database schema. You need to ensure that not only the code generators themselves adhere to all the required quality characteristics, but that the code they produce also adheres to them. In the cases of Model Driven Engineering, code is generated from the design. The generated code needs to meet all the required quality characteristics. Generated code needs to be reviewed, profiled, and used in exactly the same way as if it were written by hand.
- **Continuously improve.** Finally, everything you do during the development phase must be reviewed regularly and streamlined. You must maintain the quality bar and not get into situations where you're lagging behind due to ineffective processes. Wherever possible, you need to improve the performance of your tools and processes so that they can be used throughout the lifecycle effectively. Running things in parallel helps to keep the total end-to-end time down, and reducing the number of dependencies between tasks, tests, and steps means that given a suitable environment, you can run multiple tests and processes in parallel.

In the next chapter, "Preparing for 'Production'", you'll examine some more of the activities that should be considered prior to launching into formal coding.

Excerpted with permission of Wrox Press from Design - Build - Run

Quality Comes at a Price

This section looks at some of the financial implications of quality and, more important, the costs associated with a lack of quality. Quality generally comes at a price, although an abundance of activities can be performed that are low cost and very high gain, assuming that they are put in place early. I'm not going to use a lot of statistics in this section because I think we are all generally aware of the cost implications of getting things wrong up front, and there are more than ample books and references that cover this subject. This section discusses some of the financial aspects that should be considered during the decision-making process. Once you understand all the processes and the financial implications, you can ensure that the Project Management Triangle is accurate. You'll use the construction process and some of the tips outlined earlier to assess the relevant costs and savings. The fact that you're a developer doesn't mean that you shouldn't be aware of the financial impact you can have on a project.

While you might not be a financier, the decisions that you make and the actions that you take affect the financial health of the project and the system as a whole. These decisions and actions need to be justified and cost effective for now and in the future.

Calculating the Potential Cost of Defects

Given that it's difficult to predict the future and the number of defects that you might encounter, a simple cost-benefit analysis needs to take best-guess and real-world estimates into account. For example, it would be extremely naive to assume that there would be no defects in the system following the construction phase. It would also be naive to assume that the system will go into production completely defect-free. However, there are usually many test phases or activities between construction and production that will result in the production release having far fewer defects than the first release that came out of construction. Therefore, it's prudent to assume a certain level of defects following initial construction. Many studies have been conducted into this subject, although I am not going to go into these here. Suffice it to say that defects will be present in the solution. However, the number and complexity of defects can't be determined up front. The further the project gets through testing, the more subtle the issues can become and the more thought they require on how to resolve them.

Let's look at a very simple defect model to try and calculate the cost of defects and use this information to determine whether "code quality 101" could be cost effective. To keep the math simple, assume that the cost of each developer is $10 per hour, and that each developer works a standard 8-hour day. If adhering to "code quality 101" were to cost $160 (that is, 2 days), it would need to save at least $160 further down the line to be cost effective. This figure does not include the initial set-up and implementation costs. This is where the educated guesswork comes in and defect modeling is one way of achieving a possible figure.

The following table shows some high-level defect categories and hypothetical effort/costs associated with them:

Defect Category	Fix Effort / Cost	Brief Explanation
Very Simple	.5 hour / $5	Assume a very simple change to a class that doesn't require any test script updates, such as updating or correcting the comments or tidying up the code. The estimate includes the time it takes to check out the code, make the change, execute all the quality checks and processes, check in the change, and submit the change into a release.
Simple	2 hours / $20	Assume a simple change to a class that requires one test script update and expected results change. Assume that this change also needs to be factored into further testing.
Medium	4 hours / $40	Assume a reasonable defect with multiple test changes and conditions that need to be factored throughout.
Complex	40 hours / $400	Assume a fairly sizable re-factoring exercise of a reasonably sized component.
Very Complex	160 hours / $1600	Assume a rewrite of a fairly complex component.

No two applications are the same and as such each application will generally have it's own specific effort estimates. However, by using the categories in the preceding table, you can put together a sliding scale or model of defect totals and associated costs that can show various positions throughout the project lifetime.

Figure 2-3 shows a hypothetical defect model based on the preceding inputs.

Defect Model	Very Simple		Simple		Medium		Complex		Very Complex		Total Cost per Defect Count
$ Cost per Defect	$5.00		$20.00		$40.00		$400.00		$1,600.00		
Percentage of Overall Defects	50%		30%		15%		3%		2%		
30	15	$75.00	9	$180.00	4.5	$180.00	0.9	$360.00	0.6	$960.00	**$1,755.00**
100	50	$250.00	30	$600.00	15	$600.00	3	$1,200.00	2	$3,200.00	**$5,850.00**
500	250	$1,250.00	150	$3,000.00	75	$3,000.00	15	$6,000.00	10	$16,000.00	**$29,250.00**
1,000	500	$2,500.00	300	$6,000.00	150	$6,000.00	30	$12,000.00	20	$32,000.00	**$58,500.00**
5,000	2,500	$12,500.00	1,500	$30,000.00	750	$30,000.00	150	$60,000.00	100	$160,000.00	**$292,500.00**

Figure 2-3

The defect model shown is used to highlight the basic analysis. It does not take into account any other resource downtime and it does not take into account individual component complexities or developer skills. As such, it provides a very static average used for example purposes only.

Excerpted with permission of Wrox Press from Design - Build - Run

The defect model in Figure 2-3 has two main rows, *$ Cost per Defect* and *Percentage of Overall Defects*, which can apply to a single class, component, or assembly, or it provides an average across all the components and assemblies.

- **$ Cost per Defect** — Contains a cost for each high-level defect category, as outlined in the preceding table. The table sets the scene for the defect model by examining the different categories and associating a baseline cost with each one.
- **Percentage of Overall Defects —** Contains a figure that represents the percentage of defects assumed in this category. For example, this model is estimating that 50 percent of the overall defects will be in the category Very Simple. It estimates that 2 percent will be in the Very Complex category.

The two rows provide the basis of the remaining calculations in the model and as such should be based on educated best guesses or real-world estimates from previous calibration exercises.

The remaining rows in the model show a total number of defects in the first column, and then each category column shows how may defects in the category it represents and the total cost for this category. For example, the row estimating a total of 30 overall defects calculates the following statistics:

- 15 very simple defects, at a total cost of $75
- 9 simple defects, at a total cost of $180
- 4.5 medium defects, at a total cost of $180
- 0.9 complex defects, at a total cost of $360
- 0.6 very complex defects, at a total cost of $960

The statistics total up to $1,755, which could be spent fixing 30 defects according to the various percentage splits. It's clearly not possible to actually have 0.9 or 0.6 defects, so these will probably roll up to whole units and increase the costs again. These figures are clearly only representative of the overall percentage within the defect category.

So, assuming the defect model is somewhat realistic and based on some real-world examples, it would show that the "code quality 101," which was estimated to cost $160, needs to potentially capture and fix the equivalent of eight simple defects to make it cost-effective. I guess you need to ask yourself "How many simple defects would the result have if I didn't adhere to any code quality at all?" Remember that defects are not just functional or execution issues. A review of the code could have highlighted thirty two very simple defects that would need to be addressed. When it comes to maintaining the system and adding new functionality, it could take someone a long while to "get their head around the code," which would also increase the costs. Although it was possible to arrive at this conclusion based solely on the information in the preceding table, it is a useful exercise to produce a basic model because it can be used to ratify the overall categories and percentages. It is also very useful at the end of each phase to examine how close the estimates were to the actual figures and update them accordingly.

Using a defect model such as this or any other model really helps to determine the foundation of the cost-benefit analysis. Cost-benefit analysis is discussed in more detail shortly. It is important that the model be based on real-world findings or estimates to ensure that the figures are as accurate as possible.

It is actually astonishing when you plug in real-world figures and see just how much you can potentially save by performing a few rudimentary activities up front. It is also astonishing to see just how much some defects can really cost further down the line.

This is just one simple example of calculating a cost-benefit figure that is related to potential defects and fix effort. There are other situations where defect modeling could be inappropriate and another model is required. For example, when choosing to use an existing component instead of custom component, building a solution will involve determining the amount of effort required to design, build, and implement the custom solution, as well as balancing these against the costs associated with product selection, procurement, licensing, implementation, and usage of the existing component.

Basic Financial Analysis

To meet the ever-increasing challenge of production-ready development, it's clear that some pretty industrialized tools, processes, and practices need to be put in place, and there are costs associated with doing this. Having a basic understanding of some of the financial implications helps to bolster the decision-making process. Financial discussions should always be held up front to avoid budget increases and to avoid unnecessary disputes later.

Let's look at two financial measures to bear in mind during development: the total cost of ownership (TCO) and the cost of poor quality (COPQ).

- Total cost of ownership (TCO) is a financial statement that covers the costs associated with the entire system, from its initial development and implementation to its final decommission. The following list is a representative view of what is generally included within a TCO statement:
    - Costs associated with initial development and implementation
    - Costs associated with running the system (infrastructure, electricity, floor space, and so on)
    - Costs associated with the system's usage, support, and maintenance
    - Costs associated with training (including project staff, users, and support staff)
    - Costs associated with failures and outages (planned and unplanned)
    - Costs associated with performance and response time issues (degradation)
    - Costs associated with reputation loss and recovery
    - Costs associated with decommission
- The cost of poor quality (COPQ) is the sum of the costs associated with producing defective material, including but not limited to:

- Costs associated with finding and fixing the defect
- Costs associated with lost opportunities
- Costs associated with loss of resources due to fixing the defect

There are many other financial controls and disciplines that should be carefully considered when setting the quality bar for a project. However, the preceding financial elements cover what you need to demonstrate best practice in the quality landscape.

Excerpted with permission of Wrox Press from Design - Build - Run

Any practice that is used during the project increases the *costs associated with initial development and implementation* in the TCO statement. However, the additional costs should be met or bettered in savings or potential savings in the other areas. For instance, the following is a very simple example:

- If the cost of integration testing (and fixing) an assembly or sub-assembly is $500, then it must save at least $500 or have the potential to save at least $500 in other areas further down the line to make it a worthwhile practice.

Depending on the size of the functional and technical test teams, this cost could be easily realized by reducing the amount of time and effort spent idle as a result of defects. This is especially true if the assembly is architectural in nature and resides lower down in the stack, affecting a number of components higher up.

Cost-Benefit Analysis

It is often prudent to perform a rudimentary cost-benefit analysis to determine whether a process or practice should be implemented. The primary purpose of the cost-benefit analysis is to calculate the difference between what the solution will cost to put in versus the amount of money it will save by implementing it. Any practice increases costs associated with development and implementation, so you want to concentrate on where these can reduce additional costs.

The following table shows a very high-level mapping. To keep this section relatively brief, I have chosen to map a handful of best practices that are close to my heart, but you can easily see the purpose of the exercise. The table is only partly completed and as we progress throughout this book there are many other practices that can be included that help to reduce costs. For instance, a fault-tolerant design would be included with *costs associated with failures and outages*.

Total Cost of Ownership	Best Practice	Brief Explanation
Costs associated with the system's usage, support, and maintenance	Instrumentation and diagnostics Standards and guidelines Process and productivity guides (including generated documentation) Productivity tools and scripts Templates and TODO statements Configuration Build and regression testing	In addition to all the project documentation, the best practices listed improve the overall support and maintenance staff's productivity and knowledge of the system and how to support, maintain, and deploy it.
Costs associated with training (including project staff, users, and support staff)	Process and productivity guides (including documentation generation)	The documentation produced will help each individual user group understand the system and the processes and tools surrounding it.

(continued)

(continued)

Total Cost of Ownership	Best Practice	Brief Explanation
Costs associated with failures and outages (planned and unplanned)	Common test foundation (data, scripts, regression) Unit and integration testing Static Code Profiling and Peer review	Thorough testing at an early stage with common data and scenarios will help to reduce the number of potential defects. Static code profiling and peer reviews will ensure that the components are thoroughly reviewed prior to release.
Costs associated with performance and response time issues (degradation)	Performance profiling Unit and integration testing (performance cycles)	The cursory code profiling and performance cycles will help to identify potential performance bottlenecks and issues early.

This type of exercise should be conducted in full against any other financial measures that are in place. As mentioned at the start of this chapter, the TCO and COPQ measures provide a good basis for this sort of analysis. COPQ is not mapped in this section, although it would be quite simple to produce.

The simple mapping provides two important purposes:

- It shows how the initiative can be used to reduce costs.
- It highlights any gaps that might need to be plugged by introducing a particular initiative or practice to reduce costs further.

Once the basic mapping has been done, additional cost-benefit analysis can be performed to further bolster the information presented. The important thing to remember is that everything you do during the project and especially the construction phase is to try and reduce costs (and defects) further down the line.

Best Practice Analysis

This section simply bolsters the previous one by examining some of the best practices and the tools that are associated with them. Providing a simple set of pros and cons is very useful when determining where to set the quality bar for construction. It is important to note that best practices aren't without drawbacks. Bearing in mind these drawbacks and taking effective action and putting the appropriate controls in place are vital to a successful construction process.

The following C# code snippet would be very easy to write and manually test:

```
public void OutputMessage( string message )
{
    Console.WriteLine("OutputMessage: {0}", message);
}
```


Excerpted with permission of Wrox Press from Design - Build - Run

If some basic exception handling and logging is added, the code might look like something like this:

```
public void OutputMessage( string message )
{
    try
    {
        Console.WriteLine("OutputMessage: {0}", message);
    }
    catch( Exception e )
    {
        Console.WriteLine("OutputMessageException: {0}", e );
    }
}
```

In this very simple example, the code is now harder to fully test because the exception handling and logging section also needs to be tested. In this example, handling the exception is nothing more than catching it, and logging is simply outputting the error message to the console (without any contextual information, e.g. the message being passed in).

A very simple solution to testing the exception handling is to introduce a special test `message` argument and a compiler directive such as `TEST`. A compiler directive is essentially a command used by the source code compiler. In this case, the command is a conditional directive to determine whether the condition evaluates to true. If it does, the code will be compiled and included in the compiled version. If the condition evaluates to false, the code will not be compiled and included.

```
public void OutputMessage( string message )
{
    try
    {
        #if TEST

        if( message == "EXCEPTION_TEST" )
        {
            throw new Exception( message );
        }

        #endif

        Console.WriteLine("OutputMessage: {0}", message);

    }
    catch( Exception e )
    {

        Console.WriteLine("OutputMessageException: {0}", e );
    }
}
```

Using these conditional directives would allow two different versions of the code to be compiled — one for normal testing and one for exception testing. Furthermore, including the special test value allows a single version for testing, which can be built upon. There are many different ways of dealing with this type of problem, including the use of interfaces to swap in special test components. When a development team ramps up, a common approach needs to be in place to avoid multiple ways of doing the same

thing. Designing for testing is covered later in this book, so I'm not going to go into the details and alternatives right now.

In this simple example, there is nothing really special happening in the exception handling section, so a simple test is probably good enough to test it. However, this is a prime example of the 80/20 rule: 80 percent of the time is spent proving 20 percent of the functionality. The 80/20 rule also applies to other development practices, such as coding standards and commenting. Once these are applied, the sample source code might look something like this:

```
<summary>
    The OutputMessage method is used to display a message on the console
</summary>
<arguments>
    <argument name="message">Message to be displayed. In test mode, when the
    input message contains 'EXCEPTION_TEST' an internal exception will be
    raised.</argument>
</arguments>
public void OutputMessage( string message )
{
    try
    {
        #region TEST_CODE

        #if TEST

        // check for test mode message
        if( message == "EXCEPTION_TEST" )
        {

            // throw a new exception based on the incoming message
            throw new Exception( message );
        }

        #end if

        #endregion

        // Functional Requirement 101 – Output Message to Console
        Console.WriteLine("OutputMessage: {0}", message);

    }
    catch( Exception e )
    {

        // Technical Requirement 101 – Output Exception to Console
        Console.WriteLine("OutputMessageException: {0}", e );

    }
}
```

Although this is an extremely basic example, it highlights some of the important factors that need to be taken into account when defining the construction process and setting the quality bar. Of course, the preceding code snippet probably wouldn't ever be used in a real-world scenario.

Excerpted with permission of Wrox Press from Design - Build - Run

The following table lists some pros and cons with a few of the practices I've discussed and includes some basic high-level actions associated with them. As has been mentioned, in general terms, any additional practice that is introduced will increase development effort to some degree. However, I've chosen not to include *increases development effort* in the cons because the drawbacks of not including the best practices far outweigh including them.

Best Practice	Pros	Cons	Actions
Coding standards (including naming conventions and coding conventions)	Makes the code easier to read and follow. Provides a consistent basis for all coding and scripting.	All developers need to understand and follow the guidelines. Code needs to be checked for adherence and non-compliance. Needs to be updated and maintained as new practices are introduced.	Must have: A coding standards and guidelines document and induction guide. Tools and guidelines for checking adherence and non-conformance. A process whereby new practices can be introduced and re-factoring can be taken into account.
Commenting	Makes the code easier to understand, follow, and maintain.	Needs to be updated when the code changes. Needs to be reviewed for correctness and meaningfulness.	Must have: Clear guidelines that when code is updated, comments are updated accordingly. Review checklist that includes commenting checks.
Exception handling, including defensive coding	Protects the system against unknown or invalid circumstances and situations.	Needs to be tested and asserted. Needs to be reviewed for compliance to standards.	Must have: Guidelines and templates for exception handling coding. A development and test framework for testing exceptions. Review checklist that includes exception-handling checks.
Event logging and tracing	Helps with issue investigation and resolution. Helps with monitoring and alerting.	Can affect performance if not implemented efficiently. Needs to be tested and asserted. Needs to be reviewed for correctness and completeness.	Must have: Guidelines and templates for logging and tracing usage. A development and test framework for testing logging and tracing. Review checklist that includes logging and tracing checks.

(continued)

(continued)

Best Practice	Pros	Cons	Actions
Instrumentation	Helps with support monitoring and alerting.	Can affect performance if not implemented efficiently. Needs to be tested and asserted. Needs to be reviewed for correctness and completeness.	Must have: Guidelines and templates for instrumentation implementation. A development and test framework for testing instrumentation. Review checklist that includes instrumentation checks.

You need to fully understand the implications in terms of cost (and timescales) of the practices being proposed or introduced. It's all too easy to jump on to the latest thinking or a cool tool that's been announced. Doing the homework and some background analysis will clarify specific benefits and what else needs to be implemented to support the practice's usage. The following table lists some of the pros and cons of the tools associated with these practices. I'll leave it to you to determine which actions you would put in place to counter the cons, although some of the manual processes were touched on earlier.

Tool	Pros	Cons
Static code analysis	Automates the process of checking code against the coding standards. Allows developers to check work and correct issues prior to formal review, saving valuable review time. Reviewers can re-execute the tool to ensure conformance and validate exceptions.	Specific coding standards need to be configured unless the out-of-the-box configuration is adequate (in most cases, it isn't). Needs to be updated and maintained as new practices are introduced. Static code analysis does not remove the need for formal reviews.
Code profiling	Helps to identify potential performance and technical issues prior to formal review or build (including database element profiling). This reduces the amount of review time and potential defects.	Specific profiling needs to be configured unless the out-of-the-box configuration is adequate (in most cases, it only goes so far). Code and database profiling does not remove the need for formal reviews.
Test coverage analysis	Code coverage identifies areas of code that have not been tested. This analysis can be used to develop further tests or remove areas of redundant code.	Striving to meet 100 percent coverage can increase development and test times if the appropriate practices are not already in place. An appropriate benchmark needs to be established. The 80/20 rule applies here in that 80 percent of the time can be spent trying to cover 20 percent of the code. Some tests may need to be run against different configuration settings to achieve a true representation of code coverage.

Excerpted with permission of Wrox Press from Design - Build - Run

Tool	Pros	Cons
Documentation generation	Generating the documentation from the code saves you from having to write it manually and avoids rework, and keeps the code and documentation in-line.	The generated documentation often needs to be updated with class diagrams and interaction diagrams generated from other tools, which need to be carefully understood and configured. This can require manual effort in documentation production.
Automated tests	Manual testing is often a laborious task and mistakes can be made. Once a series of tests has been automated, it reduces manual effort and provides a solid foundation for regression testing.	The tests and expected results need to be maintained throughout to ensure changes and updates are reflected correctly. This is especially true of user interface testing. As soon as fields move or additional fields are added, you can sometimes see a dramatic effect on the user interface test scripts. The tools often require complex configuration, which needs to be managed and maintained.

Some of these tools can be purchased and some can be developed in-house. In either case, they need to be configured appropriately and managed as a part of the overall solution and justified accordingly. That said, I'm a firm believer in using these types of tools for any and all development projects.

Estimates and Estimating

One final quality input to the planning process is the estimates. Build and unit test estimates should cover the resources and time required to build a component and unit test it. The estimates are highly dependent on the level of quality, the effectiveness and efficiency of the development and test processes, procedures and tools, and the skill level of the developer. Estimating is a true discipline and getting ready for development as early as possible helps to ensure better estimates.

Estimating is essentially answering the question "How long will it take?" and its counterpart question "How much will it cost?" For the purpose of this exercise, I am going to use a very simple case to demonstrate the value of realistic estimating. The challenge question is as follows:

- ❑ How long will it take to produce a simple console application in C# that takes a single string argument and displays the message on the screen?

You may be thinking of a figure right now, based on the preceding example code snippets. The answer to this question is that it really depends. However, for the purposes of this exercise, I'm going to put a stake in the ground and say 15 minutes for a very simple solution with manual testing and minimum best practice. 15 minutes is a realistic estimate to perform the following tasks:

1. Open Visual Studio.
2. Create a new console application.
3. Add a `Console.WriteLine` statement that outputs the argument.
4. Compile the solution and generate an EXE file.

5. Open a command window.
6. Change the directory to the location of the generated EXE.
7. Run the EXE, passing an argument on the command line.
8. Check that the correct argument is displayed in the output.

There's nothing special about this and it's a viable solution to the problem and one that would probably be used in a C# training course. You saw an example of this earlier, albeit not as a form console application. In this instance, the estimate takes into account only a basic implementation and covers coding and very minimal testing. This example is used only to bolster the importance of understanding all the processes and practices I've covered so far and including them in the estimating process.

Once the quality bar is in place and you've done the up-front work, you can put some realistic estimates in place by walking through the process. The process and tools dictate the *minimum development time* and the component's complexity, and developer skill dictates the *maximum development time*. The more efficient the processes and tools are, the lower the minimum, and the less complex the components are and the better and faster developers are, the lower the maximum. The *mean development time* is middle ground between the most experienced developer and the least experienced developer. For instance, if it takes a highly skilled developer one day to complete a task and it takes two days for a less skilled developer, the *mean development time* would be approximately 1.5 days, the difference between the two. Over time, the actual development times can be recorded to improve estimating, although the estimates still need to factor into the *minimum development time* for the process, as it may have changed.

Figure 2-4 shows a mapping between developer skill and component complexity.

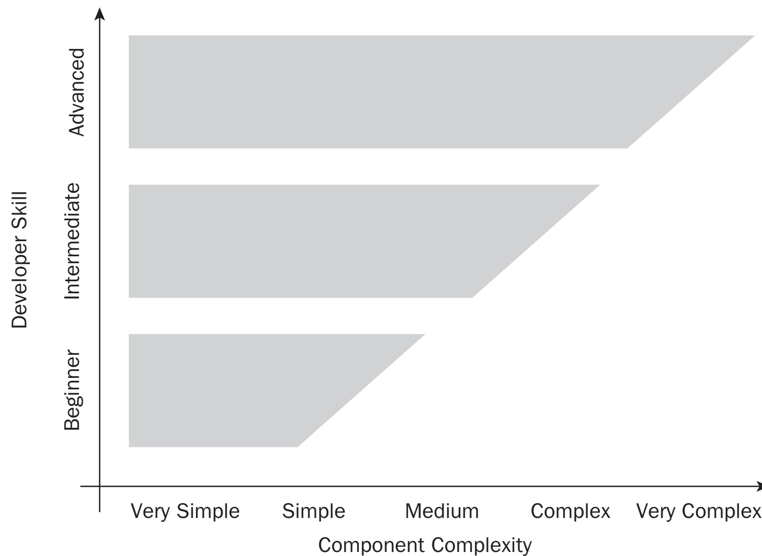

Figure 2-4

Minimizing the number of complex or very complex components allows for less highly skilled developers to work on them. The ideal solution is to keep all components within the range of Very Simple to Medium, allowing the maximum number of developers to work on them, although this needs

Excerpted with permission of Wrox Press from Design - Build - Run

to be balanced with development progression. Advanced developers want to work on complex programs, junior developers want to advance to intermediate programs, and so on.

If there are complex or very complex components in the solution, advanced developers are required on the team. The project plan will determine how many developers are required and the appropriate level of skill. It is generally easier to get beginner and intermediate developers than it is to get advanced developers. Keeping the solution simple means more people can work on it, and having the right processes and practices in place helps to keep everything on track and consistent.

In my experience, nothing takes less than the minimum development time individually. Volumes of scale need to be applied to achieve this. For instance, a rules engine may involve hundreds of rules components. If there are 100 simple rules, components that are around one or two lines of core code each and the appropriate templates are used. The time per component may dip below the minimum because of the volumes and parallelism involved. This, however, should not necessarily be relied on when estimating, as it can often cause a development bottleneck that needs to be reviewed carefully.

To arrive at a true estimate, you need to fill in the gaps in the process. For example, if you look at steps of developing test scripts and testing data from the development process, the process might look something like this (I've simplified the steps for this example):

1. Copy the unit test template to the appropriate component folder under unit test conditions.
2. Fill in the component name, developer, team, and reviewer fields.
3. Fill in the test conditions according to the test condition checklist.
4. Save the unit test conditions.
5. Conduct a formal review according to test condition checklist.

Steps 3 and 5 are the hardest to estimate. Step 3 is difficult because you need to know how many conditions there are, and Step 5 is based on the number of conditions. In general, conditions that can be met by input values and output values are far easier to test than internal conditions. Complexity is typically based on the number of conditions, which also include the input values and the different combinations, the number of branches in the component, the nesting of the branches, and the outputs or expected results.

The following pseudo code contains one input, `AccountType`, which has two possible values, `Administrator` and `User`. The code has two branches, one for valid account types and one invalid account types.

```
FUNCTION VALIDATE_ACCOUNT_TYPE( AccountType )

    VALID_ACCOUNT_TYPE = FALSE

    IF AccountType = "Administrator" OR AccountType = "User" THEN

        SET VALID_ACCOUNT_TYPE = TRUE

    ELSE

        RAISE BUSINESS EVENT: INVALID_ACCOUNT_TYPE + AccountType
```

```
            SET VALID_ACCOUNT_TYPE = FALSE

        END IF

        RETURN VALID_ACCOUNT_TYPE

    END FUNCTION
```

The pseudo code doesn't contain any error handling or other outputs, so it acts as a very simple case. The functional test conditions would include those listed in the following table:

Condition	Description	Input Value	Expected Results
1	Valid administrator account type	Administrator	Return value = TRUE.
2	Valid user account type	User	Return value = TRUE.
3	Invalid account type	XXX	Return value = FALSE. INVALID_ACCOUNT_TYPE event raised with XXX

The basic test conditions in the preceding table would obtain 100 percent code coverage. They also take into account the event being raised and the contextual information. The `INVALID_ACCOUNT_TYPE` event could be verified manually; however, it's still an expected result of the condition and not just that the method returns false.

When estimating how long it will take to build and test a component, you should take into account the number of conditions, input values, and expected results, as described in the following table. Functional designs sometimes don't take into account exception handling and logging (unless there are very specific requirements), as these are thought of as technical characteristics that should be documented in the technical or detailed design documents.

Condition	Category	Description	Input Value	Expected Results
1	Functional	Valid administrator account type	Administrator	Return value = TRUE.
2	Functional	Valid user account type	User	Return value = TRUE.
3	Functional	Invalid account type	XXX	Return value = FALSE. INVALID_ACCOUNT_TYPE event raised with XXX.

Excerpted with permission of Wrox Press from Design - Build - Run

Condition	Category	Description	Input Value	Expected Results
4	Performance	10,000 * valid administrator account type	Administrator	Return value = TRUE. < 5ms response time
5	Performance	10,000 * valid user account type	User	Return value = TRUE. < 5ms response time
6	Performance	10,000 * mixed valid administrator and user account types	2,000 * administrator 8,000 * user	Return value = TRUE. < 5ms response time
7	Monitoring / Incident Investigation	Invalid account type YYY	YYY	Return value = FALSE. INVALID_ACCOUNT_TYPE event raised with YYY.
8	Monitoring / Incident Investigation	Invalid account type ZZZ	ZZZ	Return value = FALSE. INVALID_ACCOUNT_TYPE event raised with ZZZ.
9	Monitoring / Incident Investigation Performance	10,000 * mixed invalid YYY and ZZZ account types	2,000 * YYY 8,000 * ZZZ	Return value = FALSE. 2,000 * INVALID_ACCOUNT_TYPE event raised with YYY. 8,000 * INVALID_ACCOUNT_TYPE event raised with ZZZ. < 5ms response time.

This may seem like an over-the-top set of test conditions for such a small component. However, they could be extended even further to take into account logging and other technical features.

The key message is to ensure that the level of testing is included in the scope and that the testing stresses the component appropriately, whether at the unit level or integration level, prior to it leaving construction. It should be firmly understood that not all issues will be resolved during construction, but the level of build quality and testing should underpin the quality bar for progressing further. The level of testing and the criteria should be taken into account when estimating.

Summary

This chapter covered the quality characteristics you need to bear in mind with everything that you do and implement. You've also seen what is involved in construction quality and the processes and practices you can employ to ensure your outputs are of a high quality. You've seen how you can better prepare yourself for testing and issue resolution. You might not be able to totally eradicate these situations, but you should do whatever you can up front to minimize them and keep costs and timescales under control. Quality needs to be factored into the scope so that the budgets and timescales can be realistically set and agreed on.

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

- **The quality characteristics apply to everything you do.** Quality is not just about code quality; it applies to all the artifacts that you produce and deliver. You should think about how each of the individual characteristics could apply to the particular item or artifact you are producing.
- **Construction quality echoes throughout the project.** You need to ensure that your construction processes and practices are tuned to produce high-quality outputs and deliverables. Some key activities and practices include:
    - Validating requirements and designs and documenting queries and questions that need to be addressed before the component can be closed off completely
    - Producing and reviewing low-level models and designs
    - Developing and reviewing unit test scripts and test data to ensure breadth and depth of test coverage
    - Developing and reviewing components (application, architecture and framework, batch, reporting, and so on)
    - Developing and reviewing test harnesses, including mock objects, test stubs, and simulators
    - Executing thorough unit tests and ensuring that all the relevant outputs are captured and verified
    - Identifying assemblies (collections of related components) and ensuring the appropriate level of granularity
    - Executing thorough integration tests and ensuring that all the relevant outputs are captured and verified
    - Compiling completion reports that document the evidence and outcomes of the activities carried out
    - Submitting artifacts into a release and ensuring that all the relevant artifacts are included in a release
    - Performing peer reviews and quality checks throughout the process
- **Include quality characteristics in the overall scope.** It is important to agree on the quality characteristics up front and include them in the overall scope. It's particularly important to ensure that all quality characteristics are captured and the processes reflect them. In addition,

Excerpted with permission of Wrox Press from Design - Build - Run

where tools are used, the configuration should be agreed on to avoid any disputes. The best practices include:

- Ensuring code quality by defining standards, guidelines, and templates
- Promoting re-use and layering to support testing and improve component re-use throughout the solution
- Using code generation techniques and ensuring that the resulting code meets all the coding standards and is profiled and reviewed as if it were crafted by hand
- Using automated static and dynamic code profiling to ensure code quality and to identify potential issues early
- Including instrumentation and diagnostics in all your components
- Using automated code coverage tools to identify the amount of code covered during testing
- Automatically generating documentation from code comments

- **Avoid an influx of issues during testing.** It is important to ensure that your test tools, test environments, and processes are fit for purpose and ready when you need them. You need to try to avoid an influx of:
    - Environment issues
    - Tool and script issues
    - Test data issues
    - Real bugs
- **Turning around defects quickly can affect quality.** You need to incorporate processes and practices that allow sustained quality and support rapid turnaround when testing is blocked. The processes that can potentially reduce quality include:
    - Hot-fixing or patching
    - Technical tuning and re-factoring
- **Improve and maintain quality throughout.** You can improve the overall quality of the system in a number of ways. The following lists the key activities discussed in addition to those already mentioned:
    - Work with the test teams.
    - Re-use common test data, scripts, and scenarios.
    - Re-use common test tools, stubs, and simulators.
    - Reduce the number of release configurations to avoid delays and installing and re-installing different releases for different testing activities.
    - Automate as much as possible. Do it twice manually, and it should be automated.
    - Review processes and practices and continuously improve them where possible.
- **Quality comes at a price.** Nothing is for free. You need to understand the implications in terms of cost (and timescales) of the choices that you make. The estimates need to be based on realistic

figures and the processes will actually determine the average amount of time required. However, proven and well-implemented processes and practices can help to reduce the costs. The costs need to be understood, agreed to, and included in the overall scope to ensure that the Project Management Triangle is set.

- **Identify and correct as many defects as early as possible.** The testing and verification that you perform during the construction phase should try to catch as many defects as possible. This will avoid costly and time-consuming "wash-up" sessions, whereby defects are scrutinized to determine whether they could have been detected during construction or earlier.

The following chapter examines some of the processes that should be considered and put in place early to ensure that you are fully prepared for production, e.g. the development and implementation of quality software products.

Excerpted with permission of Wrox Press from Design - Build - Run