1 Introduction

> If you try to make the software foolproof,
> they will just invent a better fool!
>
> **Dorothy Graham**

1.1 Why Agile?

In today's highly competitive IT business, companies experience massive pressures to be as effective and efficient as possible in developing and delivering successful software solutions. If you don't find strategies to reduce the cost of software development, your competitors will, allowing them to undercut your prices, to offer to develop and deliver products faster, and ultimately to steal business from you.

Often in the past, testing was an afterthought; now it is increasingly seen as the essential activity in software development and delivery. However, poor or ineffective testing can be just as bad as no testing and may cost significant time, effort, and money, but ultimately fail to improve software quality, with the result that your customers are the ones who find and report the defects in your software!

If testing is the right thing to do, how can you ensure that you are doing testing right?

If you ask managers involved in producing software whether they follow industry best practices in their development and testing activities, almost all of them will confidently assure you that they do. The reality is often far less clear; even where a large formal process documenting best development and testing practice has been introduced into an organization, it is very likely that different members of the team will apply their own testing techniques, employ a variety of different documentation (such as their own copies of test plans and test scripts), and use different approaches for assessing and reporting testing progress on different projects. Even the language is likely to be different, with staff using a variety of terms for the same thing, as well as using the same terms for different things!

Just how much time, effort, and money does this testing chaos cost your organization? Can you estimate just how much risk a project carries in terms of late delivery, with poor testing resulting in the release of poor-quality software? To put this in perspective, the U.S. National Institute of Standards and Technology recently reported that, for every $1 million spent on software implementations, businesses typically incur more than $210,000 (or between a fifth and a quarter of the overall budget) of

additional costs caused by problems associated with impact of postimplementation faults [1].

The most common reason that companies put up with this situation is that they take a short-term view of the projects they run; it is much better to just get on with it and "make progress" than to take a more enlightened, but longer-term, view to actually address and fix the problems.

Many organizations are now adopting some form of formal test process as the solution to these problems. In this context, a process provides a means of documenting and delivering industry best practice in software development and testing to all of the staff in the organization. The process defines who should do what and when, with standard roles and responsibilities for project staff, and guidance on the correct way of completing their tasks. The process also provides standard reusable templates for things like test plans, test scripts, and testing summary reports and may even address issues of process improvement [2].

Although there have been numerous attempts to produce an "industry standard" software testing process (e.g., the Software Process Engineering Metamodel [3]), many practitioners and organizations express concerns about the complexity of such processes. Typical objections include:

- "The process is too big" – there is just too much information involved and it takes too long to rollout, adopt, and maintain.
- "That's not the way we do things here" – every organization is different and there is no one-size-fits-all process.
- "The process is too prescriptive" – a formal process stifles the creativity and intuition of bright and imaginative developers and testers.
- "The process is too expensive" – if we are trying to reduce the cost of software development, why would we spend lots of money on somebody else's best practices?

Interestingly, even where individuals and organizations say they have no process, this is unlikely to be true – testers may invent it on the fly each morning when they start work, but each tester will follow some consistent approach to how he or she performs their testing. It is possible for this "approach" to be successful if you are one of those talented supertesters or you work in an organization that only hires "miracle QA" staff. For the rest of us, we need to rely on documented best practices to provide guidance on the who, the what, and the when of testing, and to provide reusable templates for the things we create, use, or deliver as part of our testing activities.

So, here is the challenge: how is it possible to produce good-quality software, on time and to budget, without forcing a large, unwieldy, and complex process on the developers and testers, but still providing them with sufficient guidance and best practices to enable them to be effective and efficient at their jobs? To restate this question, what is the minimum subset of industry best practice that can be used while still delivering quality software?

This book provides practical guidance to answer this question by means of real-world case studies, and will help you to select, adopt, and use a personally customized set of agile best practices that will enable you and your colleagues to deliver quality testing in as effective and efficient a manner as possible.

1.2 Suggestions on How to Read This Book

This book is divided into three main sections (plus the appendices), each of which are closely linked, but each of which can be read and applied separately.

Part 1 of the book provides a review of both the traditional or "classic" view of software testing process and examples of agile approaches:

- If you are interested in reviewing the early history of software development and testing process, Chapter 2 (Old-School Development and Testing) begins by reviewing the traditional or "classic" view of process. This chapter explores the good and the bad aspects of classic test process, and provides a useful baseline for the rest of the book to build on.
- If you are interested in understanding the development of agile approaches to software development and testing, Chapter 3 (Agile Development and Testing) provides an overview of the principal agile approaches that have been used to develop software, with particular emphasis on the testing aspects of the method described.
- Although Chapter 3 provides a high-level overview of the principal agile approaches, if you require a deeper understanding of these methods then refer to Appendices A through D. You may find this to be of particular benefit in preparation for reading the agile case studies in Part 2 of the book.

Part 2 of the book contains twenty case studies, which provide real-world examples of how different organizations and individual practitioners have worked in an agile development and testing framework or have implemented their own agile testing approaches. Each chapter reviews the specific testing requirements faced by the testers, provides a summary of the agile solution they adopted, describes the overall success of the approach, and provides a discussion of which specific aspects of the approach worked well, and which aspects might be improved or omitted in future testing projects.

Part 3 of this book provides an analysis of the agile case studies presented in Part 2 and draws upon the material from Part 1 to make a series of proposals about what components might be used to generate your own successful agile testing process:

- If you would like some guidance on agile best practices from a practitioner perspective, Chapter 24 (Analysis of the Case Studies) examines in detail the

agile case studies presented in Part 2, identifying particularly successful agile techniques, common themes (such as successful reusable templates), as well as those testing approaches that were not successful and which may need to be treated with caution.

- If you are interested in guidance on how to set up your own agile development and testing process, Chapter 25 (My Agile Process) draws on the information provided in the case studies and their analysis to make a series of proposals for how you might set up and run a practical, effective, and efficient agile testing process.
- If you would like some guidance on how to introduce your agile testing method into your own organization, Chapter 26 (The Roll-out and Adoption of My Agile Process) provides a series of tried and tested best practices describing how you can roll out the process and drive its successful use and adoption.

The Appendices

If you would like to find more detail on the agile methods described briefly in Chapter 3, Appendices A through D provide further description of each of the key agile approaches covered in Chapter 3, with particular emphasis on the software quality aspects of each approach. You may find value in reading these appendices in preparation for reading the case studies presented in Part 2 of this book.

Appendices E through G provide a set of reusable testing templates that can be used as a resource to be reused in your own agile process (these templates are also available in electronic format from the Cambridge University Press Web site at http://www.cup.agiletemplates.com), including

- an agile test script template,
- an agile test result record form template, and
- an agile test summary report template.

Appendix H contains a checklist of agile best practices that shows which practices are particularly appropriate for the different styles and sizes of agile project described in Chapter 25. This checklist can be used as a summary of the practices and as an *aide memoire* to assist you in populating your own agile process.

References cited in the text are fully expanded in the References section at the back of the book.

PART ONE

REVIEW OF OLD-SCHOOL AND AGILE APPROACHES

> Fact of the matter is, there is no hip world, there is no straight world. There's a world, you see, which has people in it who believe in a variety of different things. Everybody believes in something and everybody, by virtue of the fact that they believe in something, use that something to support their own existence.
>
> **Frank Zappa**

This section of the book provides a review of both the traditional or "classic" view of software testing process and agile approaches.

The chapters in this section are:

- Chapter 2 – Old-School Development and Testing, which begins by reviewing the traditional or "classic" view of software testing process. This chapter will explore the good and bad aspects of classic test process, and provides a useful baseline for the rest of the book to build on
- Chapter 3 – Agile Development and Testing, which provides a review of the most prominent agile approaches that have been used to develop software, with particular emphasis on the testing aspects of the method described. If additional information on a particular approach is needed, more complete details of each method are provided in Appendices A to D.

2 Old-School Development and Testing

> Testing is never completed, it's simply abandoned!
>
> **Simon Mills**

2.1 Introduction

This chapter discusses what software development and testing process is, reviews the historical development of process, and concludes by providing a review of the elements of a traditional or "classic" software testing process, providing a useful baseline for the rest of the book to build on.

2.2 So, What Is Process?

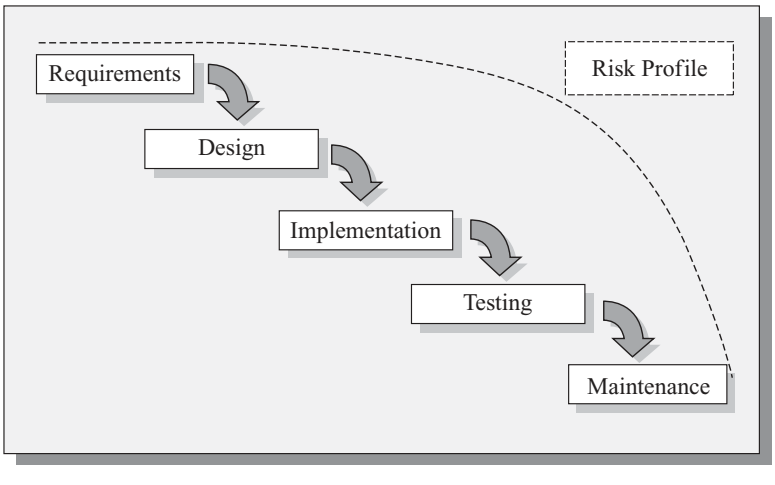
A process seeks to identify and reuse common elements of some particular approach to achieving a task, and to apply those common elements to other, related tasks. Without these common reusable elements, a process will struggle to provide an effective and efficient means of achieving those tasks, and find it difficult to achieve acceptance and use by other practitioners working in that field.

Test process is no different; we have many different tasks that need to be achieved to deliver effective and efficient testing, and at a variety of different levels of testing from component/unit/developer testing, through integration/module testing, on into systems testing, and through to acceptance testing [4].

Even before testing process was "invented", good testers have done things in a particular way to achieve good results – such as the best way to find the most defects, to complete testing more quickly or more cheaply, to save time by reusing things they had produced in earlier testing projects (such as a template for a test plan or a test script), or to ensure consistent nomenclature (such as common terms for testing phases).

Such enlightened practitioners were even known to share such best practices with their colleagues, passing on or swapping reusable templates, publishing papers on testing techniques, or mentoring other staff on test management approaches, for example.

As the IT industry matured, with customers demanding increasingly complex systems, of ever higher quality, in shorter timescales and with lower cost, the



2.1 The Waterfall Phases and Risk Profile (dotted line).

resulting commercial pressures forced those organizations developing software to seek methods to ensure their software development was as effective and efficient as possible. If they did not find the means to deliver software faster, cheaper, and with better quality, their competitors would.

Successive waves of new technologies, such as procedural programming, fourth-generation languages, and object orientation, all promised to ensure reductions in the occurrence of defects, to accelerate development times, and to reduce the cost of development. Interestingly, it was observed that it was still possible to write poor-quality software that failed to achieve its purpose and performed poorly or included defects, no matter what technologies were used!

As with so many instances of a new technology failing to solve a particular problem, the issue actually turns out to be a people problem. Human beings need guidance, they need to build upon the knowledge and experiences of others, they need to understand what works and what doesn't work, and they need to avoid wasting time reinventing things that other practitioners have already successfully produced and used. Project chaos, where each project and practitioner uses different techniques, employs different terminology, or uses (or worse, reinvents from scratch) different documentation, was increasingly considered to be unacceptable.

The following sections review a number of the early approaches to software development and testing that sought to avoid such project chaos.

2.3 Waterfall

One of the earliest approaches to software development is the waterfall approach. A paper published by Winston W. Royce in the 1970s [5] described a sequential software development model containing a number of phases, each of which must be completed before the next begins. Figure 2.1 shows the classic interpretation of the phases in a waterfall project.

From a quality perspective, the waterfall approach has been often criticized because testing begins late in the project; as a consequence, a high degree of project risk (that is, failure of the software to meet customer expectations, to be delivered with acceptable levels of defects, or to perform adequately) is retained until late into the project. With the resultant reworking and retesting caused by the late detection of defects, waterfall projects were also likely to incur additional effort, miss their delivery dates, and exceed their budgets.

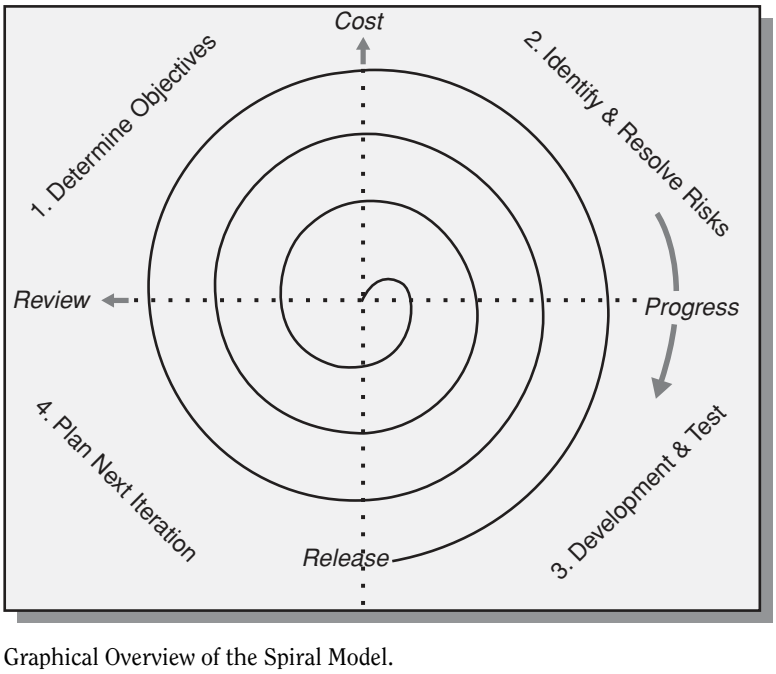
The waterfall approach has also been criticized for its lack of responsiveness to customer requests for changes to the system being developed. Historically, it was typical for all of the requirements to be captured at the start of the project and to be set in stone throughout the rest of the development. A frequent result of this approach was that by the time the software had been delivered (sometimes months or even years later), it no longer matched the needs of the customer, which had almost certainly changed by then.

Because of increasing dissatisfaction with the rigid structure of waterfall projects, other solutions were investigated that would be more flexible in terms of addressing changing requirements.

2.4 Spiral

Many attempts were made to address the shortcomings of the waterfall approach, such as the spiral model of software development defined by Barry Boehm in 1988 [6]. Intended for use in large, complex, and costly projects, and intended to address the issues of meeting customer requirements, this incremental development process relied heavily on the development and testing of a series of software prototypes of the final system. The typical steps involved in a spiral model–driven project are as follows:

1. In discussion with the customer, the requirements for the system are defined and documented in as much detail as possible.
2. An initial design is created based on the requirements.
3. A sequence of increasingly complete prototypes are constructed from the design in order to
   - ▷ test the strengths and weaknesses of the prototypes, and to highlight any risks;
   - ▷ assist in refining the requirements by obtaining customer feedback; and
   - ▷ assist in refining the planning and design.
4. The risks identified by testing the prototypes are reviewed with the customer, who can make a decision whether to halt or continue the project.
5. Steps 2 through 4 are repeated until the customer is satisfied that the refined prototype reflects the functionality of the desired system, and the final system is then developed on this basis.
6. The completed system is thoroughly tested (including formal acceptance testing) and delivered to the customer.



2.2 Graphical Overview of the Spiral Model.

7. Where appropriate, ongoing maintenance and test are performed to prevent potential failures and to maximize system availability.

Figure 2.2 provides a graphical overview of a typical interpretation of the spiral model.

Although considered to be an improvement over the waterfall approach in terms of delivering systems that more closely match the customer's requirements, and for delivering higher-quality software (achieved in large part by the spiral model, which encourages early and continued testing of the prototypes), issues existed regarding the difficulty of estimating effort, timescales, and cost of delivery; the nondeterministic nature of the cycle of prototype development and testing meant that it was difficult to bound the duration and effort involved in delivering the final product.

2.5 Iterative

Iterative models of software development evolved to address issues raised by both waterfall and spiral approaches, with the goal of breaking large monolithic development projects into smaller, more easily managed iterations. Each iteration would produce a tangible deliverable (typically some executable element of the system under development).

The Objectory method [7] provides a good example of such an approach. In 1987, while assisting telecommunications company Ericsson AB with its software

development efforts, and concerned with the shortcomings of earlier methods, Ivar Jacobson brought together a number of the development concepts he had been thinking about, such as use cases [8], object-oriented design [9], and iterative development, to create a new approach to developing successful object-oriented applications.

The Objectory method supported innovative techniques for requirements analysis, visual modeling of the domain, and an iterative approach to managing the execution of the project. In essence, Objectory would break down a project that might have been run in a large and inflexible waterfall manner into smaller, more easily understood, implemented, and tested iterations.

Such an approach brought a number of important benefits for software quality:

- Testing could begin much earlier in the project (from the first iteration), enabling defects to be identified and fixed in a timely manner, with the result that timescales were not impacted, and that the effort and cost of fixing and retesting defects were kept low.[1]
- Testing would continue throughout the project (within each iteration), ensuring that new defects were found and fixed in a timely manner, that newly added system functionality did not adversely affect the existing software quality, and verifying that defects found in earlier iterations did not reappear in the most recent iteration.
- The valuable visual metaphor provided by use cases and visual modeling enabled the developers and the customer to more easily understand the intention of the system functionality – in effect the customer, analyst, designer, and tester share a common language and understanding of the system.
- Testers discovered that the scenarios described by the use cases could very easily be used to design and implement effective test cases[2] – the actors (people or other systems) identified in the use cases and their interactions with the system under development, mapped easily onto the steps and verifications needed to develop the test scripts.

The Objectory process was organized around three phases:

1. The *requirements phase* – which involves the construction of three models that describe in a simple, natural-language manner what the system should do:
   - The *use case model* – which documents the interactions between the actors and the system, which are typically captured using use case diagrams and natural-language descriptions.
   - The *domain model* – which documents the entities within the system, their properties, and their relationships.

[1] It is generally considered that for each phase of the project during which you fail to find a bug, the cost of fixing it increases by a factor of 10 [4]. In practice, my experience is that this is a conservative estimate.

[2] A test case represents the design for a test. A test case is implemented by means of a test script.