The Addison-Wesley Signature Series

A Martin Fowler Signature Book

Continuous Delivery

Reliable Software Releases through Build, Test, and Deployment Automation

Jez Humble

David Farley

Foreword by Martin Fowler

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382–3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data:

Humble, Jez.
Continuous delivery : reliable software releases through build, test, and deployment automation / Jez Humble, David Farley.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-60191-9 (hardback : alk. paper) 1. Computer software--Development. 2. Computer software--Reliability. 3. Computer software--Testing. I. Farley, David, 1959- II. Title.
QA76.76.D47H843 2010
005.1--dc22
2010022186

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671 3447

ISBN-13: 978–0–321–60191–9
ISBN-10: 0–321–60191–2
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing August 2010

Contents

Foreword by Martin Fowler

In the late 90s, I paid a visit to Kent Beck, then working in Switzerland for an insurance company. He showed me around his project, and one of the interesting aspects of his highly disciplined team was the fact that they deployed their software into production every night. This regular deployment gave them many advantages: Written software wasn't waiting uselessly until it was deployed, they could respond quickly to problems and opportunities, and the rapid turnaround led to a much deeper relationship between them, their business customer, and their final customers.

In the last decade I've worked at ThoughtWorks, and a common theme of our projects has been reducing the cycle time between an idea and usable software. I see plenty of project stories, and almost all involve a determined shortening of that cycle. While we don't usually do daily deliveries into production, it's now common to see teams doing bi-weekly releases.

Dave and Jez have been part of that sea change, actively involved in projects that have built a culture of frequent, reliable deliveries. They and our colleagues have taken organizations that struggled to deploy software once a year into the world of Continuous Delivery, where releasing becomes routine.

The foundation for the approach, at least for the development team, is Continuous Integration (CI). CI keeps the entire development team in sync, removing the delays due to integration issues. A couple of years ago, Paul Duvall wrote a book on CI in this series. But CI is just the first step. Software that's been successfully integrated into a mainline code stream still isn't software that's out in production doing its job. Dave and Jez's book pick up the story from CI to deal with that "last mile," describing how to build the deployment pipeline that turns integrated code into production software.

This kind of delivery thinking has long been a forgotten corner of software development, falling into a hole between developers and operations teams. So it's no surprise that the techniques in this book rest upon bringing these teams together—a harbinger of the nascent but growing DevOps movement. This process also involves testers, as testing is a key element of ensuring error-free releases.

Threading through all this is a high degree of automation, so things can be done quickly and without error.

Getting all this working takes effort, but benefits are profound. Long, high-intensity releases become a thing of the past. Customers of software see ideas rapidly turn into working code that they can use every day. Perhaps most importantly, we remove one of the biggest sources of baleful stress in software development. Nobody likes those tense weekends trying to get a system upgrade released before Monday dawns.

It seems to me that a book that can show you how to deliver your software frequently and without the usual stresses is a no-brainer to read. For your team's sake, I hope you agree.

Preface

Introduction

Yesterday your boss asked you to demonstrate the great new features of your system to a customer, but you can't show them anything. All your developers are halfway through developing new features and none of them can run the application right now. You have code, it compiles, and all the unit tests pass on your continuous integration server, but it takes a couple of days to release the new version into the publicly accessible UAT environment. Isn't it unreasonable to expect the demo at such short notice?

You have a critical bug in production. It is losing money for your business every day. You know what the fix is: A one-liner in a library that is used in all three layers of your three-tier system, and a corresponding change to one database table. But the last time you released a new version of your software to production it took a weekend of working until 3 A.M., and the person who did the deployment quit in disgust shortly afterward. You know the next release is going to overrun the weekend, which means the application will be down for a period during the business week. If only the business understood our problems.

These problems, although all too common, are not an inevitable outcome of the software development process: They are an indication that something is wrong. Software release should be a fast, repeatable process. These days, many companies are putting out multiple releases in a *day*. This is possible even with large projects with complex codebases. In this book, we will show you how this is done.

Mary and Tom Poppendieck asked, "How long would it take your organization to deploy a change that involves just one single line of code? Do you do this on a repeatable, reliable basis?"[1] The time from deciding that you need to make a change to having it in production is known as the *cycle time*, and it is a vital metric for any project.

1. *Implementing Lean Software Development*, p. 59.

In many organizations, cycle time is measured in weeks or months, and the release process is certainly not repeatable or reliable. It is manual and often requires a team of people to deploy the software even into a testing or staging environment, let alone into production. However, we have come across equally complex projects which started out like this but where, after extensive reengineering, teams were able to achieve a cycle time of hours or even minutes for a critical fix. This was possible because a fully automated, repeatable, reliable process was created for taking changes through the various stages of the build, deploy, test, and release process. Automation is the key. It allows all of the common tasks involved in the creation and deployment of software to be performed by developers, testers, and operations personnel, at the push of a button.

This book describes how to revolutionize software delivery by making the path from idea to realized business value—the cycle time—shorter and safer.

Software delivers no revenue until it is in the hands of its users. This is obvious, but in most organizations the release of software into production is a manually intensive, error-prone, and risky process. While a cycle time measured in months is common, many companies do much worse than this: Release cycles of more than a year are not unknown. For large companies every week of delay between having an idea and releasing the code that implements it can represent millions of dollars in opportunity costs—and yet these are often the ones with the longest cycle times.

Despite all this, the mechanisms and processes that allow for low-risk delivery of software have not become part of the fabric in most of today's software development projects.

Our aim is to make the delivery of software from the hands of developers into production a reliable, predictable, visible, and largely automated process with well-understood, quantifiable risks. Using the approach that we describe in this book, it is possible to go from having an idea to delivering working code that implements it into production in a matter of minutes or hours, while at the same time improving the quality of the software thus delivered.

The vast majority of the cost associated with delivering successful software is incurred after the first release. This is the cost of support, maintenance, adding new features, and fixing defects. This is especially true of software delivered via iterative processes, where the first release contains the minimum amount of functionality providing value to the customer. Hence the title of this book, *Continuous Delivery*, which is taken from the first principle of the Agile Manifesto: "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software" [bibNp0]. This reflects the reality: For successful software, the first release is just the beginning of the delivery process.

All the techniques we describe in this book reduce the time and risks associated with delivering new versions of your software to users. They do this by increasing feedback and improving collaboration between the development, testing, and operations personnel responsible for delivery. These techniques ensure that when you need to modify applications, either to fix bugs or deliver new features, the time between making modifications and having the results deployed and in use is as low as possible, problems are found early when they are easy to fix, and associated risks are well understood.

Who Is This Book for, and What Does It Cover?

One of the major aims of this book is to improve collaboration between the people responsible for delivering software. In particular, we have in mind developers, testers, systems and database administrators, and managers.

We cover topics from traditional configuration management, source code control, release planning, auditing, compliance, and integration to the automation of your building, testing, and deployment processes. We also describe techniques such as automated acceptance testing, dependency management, database migration, and the creation and management of testing and production environments.

Many people involved in creating software consider these activities secondary to writing code. However, in our experience they take up a great deal of time and effort, and are critical to successful software delivery. When the risks surrounding these activities are not managed adequately, they can end up costing a lot of money, often more than the cost of building the software in the first place. This book provides the information that you need to understand these risks and, more importantly, describes strategies to mitigate them.

This is an ambitious aim, and of course we can't cover all these topics in detail in one book. Indeed we run the risk of alienating each of our target audiences: developers, by failing to treat topics such as architecture, behavior-driven development, and refactoring in depth; testers, by not spending sufficient time on exploratory testing and test management strategies; operations personnel, by not paying due attention to capacity planning, database migration, and production monitoring.

However, books exist that address each of these topics in detail. What we think is lacking in the literature is a book that discusses how all the moving parts fit together: configuration management, automated testing, continuous integration and deployment, data management, environment management, and release management. One of the things that the lean software development movement teaches is that it is important to optimize the whole. In order to do this, a holistic approach is necessary that ties together every part of the delivery process and everybody involved in it. Only when you have control over the progression of every change from introduction to release can you begin to optimize and improve the quality and speed of software delivery.

Our aim is to present a holistic approach, as well as the principles involved in this approach. We will provide you with the information that you will need to decide how to apply these practices in your own projects. We do not believe that there is a "one size fits all" approach to any aspect of software development, let alone a subject area as large as the configuration management and operational control of an enterprise system. However, the fundamentals that we describe in this book are widely applicable to all sorts of different software projects—big, small, highly technical or short sprints to early value.

As you begin to put these principles into practice, you will discover the areas where more detail is required for your particular situation. There is a bibliography at the end of this book, as well as pointers to other resources online where you can find more information on each of the topics that we cover.

This book consists of three parts. The first part presents the principles behind continuous delivery and the practices necessary to support it. Part two describes the central paradigm of the book—a pattern we call the deployment pipeline. The third part goes into more detail on the ecosystem that supports the deployment pipeline—techniques to enable incremental development; advanced version control patterns; infrastructure, environment and data management; and governance.

Many of these techniques may appear to apply only to large-scale applications. While it is true that much of our experience is with large applications, we believe that even the smallest projects can benefit from a thorough grounding in these techniques, for the simple reason that projects grow. The decisions that you make when starting a small project will have an inevitable impact on its evolution, and by starting off in the right way, you will save yourself (or those who come after you) a great deal of pain further down the line.

Your authors share a background in lean and iterative software development philosophies. By this we mean that we aim to deliver valuable, working software to users rapidly and iteratively, working continuously to remove waste from the delivery process. Many of the principles and techniques that we describe were first developed in the context of large agile projects. However, the techniques that we present in this book are of general applicability. Much of our focus is on improving collaboration through better visibility and faster feedback. This will have a positive impact on every project, whether or not it uses iterative software development processes.

We have tried to ensure that chapters and even sections can be read in isolation. At the very least, we hope that anything you need to know, as well as references to further information, are clearly sign-posted and accessible so that you can use this book as a reference.

We should mention that we don't aim for academic rigor in our treatment of the subjects covered. There are plenty of more theoretical books on the market, many of which provide interesting reading and insights. In particular, we will not spend much time on standards, concentrating instead on battle-tested skills and techniques every person working on a software project will find useful, and explaining them clearly and simply so that they can be used every day in the real world. Where appropriate, we will provide some war stories illustrating these techniques to help place them in context.

Conspectus

We recognize that not everyone will want to read this book from end to end. We have written it so that once you have covered the introduction, you can attack it in several different ways. This has involved a certain amount of repetition, but hopefully not at a level that becomes tedious if you do decide to read it cover-to-cover.

This book consists of three parts. The first part, Chapters 1 to 4, takes you through the basic principles of regular, repeatable, low-risk releases and the practices that support them. Part two, Chapters 5 through 10, describe the deployment pipeline. From Chapter 11 we dive into the ecosystem that supports continuous delivery.

We recommend that everybody read Chapter 1. We believe that people who are new to the process of releasing software, even experienced developers, will find plenty of material challenging their view of what it means to do professional software development. The rest of the book can be dipped into either at your leisure—or when in a panic.

Part I—Foundations

Part I describes the prerequisites for understanding the deployment pipeline. Each chapter builds upon the last.

Chapter 1, "The Problem of Delivering Software," starts by describing some common antipatterns that we see in many software development teams, and moves on to describe our goal and how to realize it. We conclude by setting out the principles of software delivery upon which the rest of the book is based.

Chapter 2, "Configuration Management," sets out how to manage everything required to build, deploy, test, and release your application, from source code and build scripts to your environment and application configuration.

Chapter 3, "Continuous Integration," covers the practice of building and running automated tests against every change you make to your application so you can ensure that your software is always in a working state.

Chapter 4, "Implementing a Testing Strategy," introduces the various kinds of manual and automated testing that form an integral part of every project, and discusses how to decide which strategy is appropriate for your project.

Part II—The Deployment Pipeline

The second part of the book covers the deployment pipeline in detail, including how to implement the various stages in the pipeline.

Chapter 5, "Anatomy of the Deployment Pipeline," discusses the pattern that forms the core of this book—an automated process for taking every change from check-in to release. We also discuss how to implement pipelines at both the team and organizational levels.

Chapter 6, “Build and Deployment Scripting,” discusses scripting technologies that can be used for creating automated build and deployment processes, and the best practices for using them.

Chapter 7, “The Commit Stage,” covers the first stage of the pipeline, a set of automated processes that should be triggered the moment any change is introduced into your application. We also discuss how to create a fast, effective commit test suite.

Chapter 8, “Automated Acceptance Testing,” presents automated acceptance testing, from analysis to implementation. We discuss why acceptance tests are essential to continuous delivery, and how to create a cost-effective acceptance test suite that will protect your application’s valuable functionality.

Chapter 9, “Testing Nonfunctional Requirements,” discusses nonfunctional requirements, with an emphasis on capacity testing. We describe how to create capacity tests, and how to set up a capacity testing environment.

Chapter 10, “Deploying and Releasing Applications,” covers what happens after automated testing: push-button promotion of release candidates to manual testing environments, UAT, staging, and finally release, taking in essential topics such as continuous deployment, roll backs, and zero-downtime releases.

Part III—The Delivery Ecosystem

The final part of the book discusses crosscutting practices and techniques that support the deployment pipeline.

Chapter 11, “Managing Infrastructure and Environments,” covers the automated creation, management, and monitoring of environments, including the use of virtualization and cloud computing.

Chapter 12, “Managing Data,” shows how to create and migrate testing and production data through the lifecycle of your application.

Chapter 13, “Managing Components and Dependencies,” starts with a discussion of how to keep your application in a releasable state at all times without branching. We then describe how to organize your application as a collection of components, and how to manage building and testing them.

Chapter 14, “Advanced Version Control,” gives an overview of the most popular tools, and goes into detail on the various patterns for using version control.

Chapter 15, “Managing Continuous Delivery,” sets out approaches to risk management and compliance, and provides a maturity model for configuration and release management. Along the way, we discuss the value of continuous delivery to the business, and the lifecycle of iterative projects that deliver incrementally.

Web Links in This Book

Rather than putting in complete links to external websites, we have shortened them and put in the key in this format: `[bibNp0]`. You can go to the link in one of two ways. Either use bit.ly, in which case the url for the example key would be `http://bit.ly/bibNp0`. Alternatively, you can use a url shortening service we've installed at http://continuousdelivery.com/go/ which uses the same keys—so the url for the example key is `http://continuousdelivery.com/go/bibNp0`. The idea is that if for some reason bit.ly goes under, the links are preserved. If the web pages change address, we'll try to keep the shortening service at http://continuousdelivery.com/go/ up-to-date, so try that if the links don't work at bit.ly.

About the Cover

All books in Martin Fowler's Signature Series have a bridge on the cover. We'd originally planned to use a photo of the Iron Bridge, but it had already been chosen for another book in the series. So instead, we chose another British bridge: the Forth Railway Bridge, captured here in a stunning photo by Stewart Hardy.

The Forth Railway Bridge was the first bridge in the UK constructed using steel, manufactured using the new Siemens-Martin open-hearth process, and delivered from two steel works in Scotland and one in Wales. The steel was delivered in the form of manufactured tubular trusses—the first time a bridge in the UK used mass-produced parts. Unlike earlier bridges, the designers, Sir John Fowler, Sir Benjamin Baker, and Allan Stewart, made calculations for incidence of erection stresses, provisions for reducing future maintenance costs, and calculations for wind pressures and the effect of temperature stresses on the structure—much like the functional and nonfunctional requirements we make in software. They also supervised the construction of the bridge to ensure these requirements were met.

The bridge's construction involved more than 4,600 workers, of whom tragically around one hundred died and hundreds more were crippled. However, the end result is one of the marvels of the industrial revolution: At the time of completion in 1890 it was the longest bridge in the world, and at the start of the 21st century it remains the world's second longest cantilever bridge. Like a long-lived software project, the bridge needs constant maintenance. This was planned for as part of the design, with ancillary works for the bridge including not only a maintenance workshop and yard but a railway "colony" of some fifty houses at Dalmeny Station. The remaining working life of the bridge is estimated at over 100 years.

Colophon

This book was written directly in DocBook. Dave edited the text in TextMate, and Jez used Aquamacs Emacs. The diagrams were created with OmniGraffle. Dave and Jez were usually not in the same part of the world, and collaborated by having everything checked in to Subversion. We also employed continuous integration, using a CruiseControl.rb server that ran dblatex to produce a PDF of the book every time one of us committed a change.

A month before the book went to print, Dmitry Kirsanov and Alina Kirsanova started the production work, collaborating with the authors through their Subversion repository, email, and a shared Google Docs table for coordination. Dmitry worked on copyediting of the DocBook source in XEmacs, and Alina did everything else: typesetting the pages using a custom XSLT stylesheet and an XSL-FO formatter, compiling and editing the Index from the author's indexing tags in the source, and final proofreading of the book.

Chapter 5

Anatomy of the Deployment Pipeline

Introduction

Continuous integration is an enormous step forward in productivity and quality for most projects that adopt it. It ensures that teams working together to create large and complex systems can do so with a higher level of confidence and control than is achievable without it. CI ensures that the code that we create, as a team, works by providing us with rapid feedback on any problems that we may introduce with the changes we commit. It is primarily focused on asserting that the code compiles successfully and passes a body of unit and acceptance tests. However, CI is not enough.

CI mainly focuses on development teams. The output of the CI system normally forms the input to the manual testing process and thence to the rest of the release process. Much of the waste in releasing software comes from the progress of software through testing and operations. For example, it is common to see

- Build and operations teams waiting for documentation or fixes
- Testers waiting for "good" builds of the software
- Development teams receiving bug reports weeks after the team has moved on to new functionality
- Discovering, towards the end of the development process, that the application's architecture will not support the system's nonfunctional requirements

This leads to software that is undeployable because it has taken so long to get it into a production-like environment, and buggy because the feedback cycle between the development team and the testing and operations team is so long.

There are various incremental improvements to the way software is delivered which will yield immediate benefits, such as teaching developers to write production-ready software, running CI on production-like systems, and instituting cross-functional teams. However, while practices like these will certainly improve matters, they still don't give you an insight into where the bottlenecks are in the delivery process or how to optimize for them.

The solution is to adopt a more holistic, end-to-end approach to delivering software. We have addressed the broader issues of configuration management and automating large swathes of our build, deploy, test, and release processes. We have taken this to the point where deploying our applications, even to production, is often done by a simple click of a button to select the build that we wish to deploy. This creates a powerful feedback loop: Since it's so simple to deploy your application to testing environments, your team gets rapid feedback on both the code and the deployment process. Since the deployment process (whether to a development machine or for final release) is automated, it gets run and therefore tested regularly, lowering the risk of a release and transferring knowledge of the deployment process to the development team.

What we end up with is (in lean parlance) a *pull system*. Testing teams deploy builds into testing environments themselves, at the push of a button. Operations can deploy builds into staging and production environments at the push of a button. Developers can see which builds have been through which stages in the release process, and what problems were found. Managers can watch such key metrics as cycle time, throughput, and code quality. As a result, everybody in the delivery process gets two things: access to the things they need when they need them, and visibility into the release process to improve feedback so that bottlenecks can be identified, optimized, and removed. This leads to a delivery process which is not only faster but also safer.

The implementation of end-to-end automation of our build, deploy, test, and release processes has had a number of knock-on effects, bringing some unexpected benefits. One such outcome is that over the course of many projects utilizing such techniques, we have identified much in common between the deployment pipeline systems that we have built. We believe that with the abstractions we have identified, some general patterns have, so far, fit all of the projects in which we have tried them. This understanding has allowed us to get fairly sophisticated build, test, and deployment systems up and running very quickly from the start of our projects. These end-to-end deployment pipeline systems have meant that we have experienced a degree of freedom and flexibility in our delivery projects that would have been hard to imagine a few years ago. We are convinced that this approach has allowed us to create, test, and deploy complex systems of higher quality and at significantly lower cost and risk than we could otherwise have done.

This is what the deployment pipeline is for.

What Is a Deployment Pipeline?

At an abstract level, a deployment pipeline is an automated manifestation of your process for getting software from version control into the hands of your users. Every change to your software goes through a complex process on its way to being released. That process involves building the software, followed by the progress of these builds through multiple stages of testing and deployment. This, in turn, requires collaboration between many individuals, and perhaps several teams. The deployment pipeline models this process, and its incarnation in a continuous integration and release management tool is what allows you to see and control the progress of each change as it moves from version control through various sets of tests and deployments to release to users.

Thus the process modeled by the deployment pipeline, the process of getting software from check-in to release, forms a part of the process of getting a feature from the mind of a customer or user into their hands. The entire process—from concept to cash—can be modeled as a value stream map. A high-level value stream map for the creation of a new product is shown in Figure 5.1.

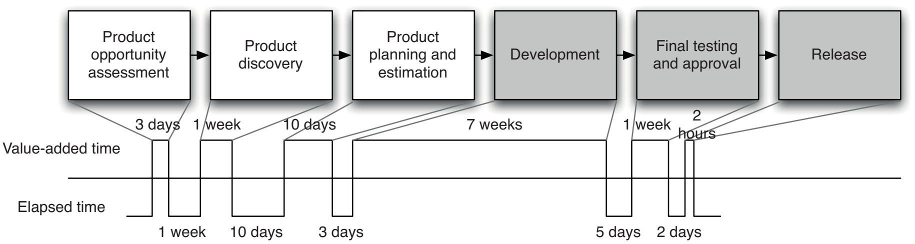

Figure 5.1 *A simple value stream map for a product*

This value stream map tells a story. The whole process takes about three and a half months. About two and a half months of that is actual work being done—there are waits between the various stages in the process of getting the software from concept to cash. For example, there is a five-day wait between the development team completing work on the first release and the start of the testing process. This might be due to the time it takes to deploy the application to a production-like environment, for example. As an aside, it has been left deliberately unclear in this diagram whether or not this product is being developed in an iterative way. In an iterative process, you'd expect to see the development process itself consist of several iterations which include testing and showcasing. The whole process from discovery to release would also be repeated many times[1]

Creating a value stream map can be a low-tech process. In Mary and Tom Poppendieck's classic, *Lean Software Development: An Agile Toolkit*, they describe it as follows.

1. The importance of iterative discovery based on customer feedback in the product development process is emphasized in books like *Inspired* by Marty Cagan and *The Four Steps to the Epiphany* by Steven Gary Blank.

> With a pencil and pad in hand, go to the place where a customer request comes into your organization. You goal is to draw a chart of the average customer request, from arrival to completion. Working with the people involved in each activity, you sketch all the process steps necessary to fill the request, as well as the average amount of time that a request spends in each step. At the bottom of the map, draw a timeline that shows how much time the request spends in value-adding activities and how much in waiting states and non-value-adding activities.

If you were interested in doing some organizational transformation work to improve the process, you would need to go into even more detail and describe who is responsible for which part of the process, what subprocesses occur in exceptional conditions, who approves the hand-offs, what resources are required, what the organizational reporting structures are, and so forth. However, that's not necessary for our discussion here. For more details on this, consult Mary and Tom Poppendieck's book *Implementing Lean Software Development: From Concept to Cash*.

The part of the value stream we discuss in this book is the one that goes from development through to release. These are the shaded boxes in the value stream in Figure 5.1. One key difference of this part of the value stream is that builds pass through it many times on their way to release. In fact, one way to understand the deployment pipeline and how changes move through it is to visualize it as a sequence diagram,[2] as shown in Figure 5.2.

Notice that the input to the pipeline is a particular revision in version control. Every change creates a build that will, rather like some mythical hero, pass through a sequence of tests of, and challenges to, its viability as a production release. This process of a sequence of test stages, each evaluating the build from a different perspective, is begun with every commit to the version control system, in the same way as the initiation of a continuous integration process.

As the build passes each test of its fitness, confidence in it increases. Therefore, the resources that we are willing to expend on it increase, which means that the environments the build passes through become progressively more production-like. The objective is to eliminate unfit release candidates as early in the process as we can and get feedback on the root cause of failure to the team as rapidly as possible. To this end, any build that fails a stage in the process will not generally be promoted to the next. These trade-offs are shown in Figure 5.3.

There are some important consequences of applying this pattern. First, you are effectively prevented from releasing into production builds that are not thoroughly tested and found to be fit for their intended purpose. Regression bugs are avoided, especially where urgent fixes need releasing into production (such fixes go through the same process as any other change). In our experience, it is also extremely common for newly released software to break down due to some unforeseen interaction between the components of the system and its environment, for example due to a new network topology or a slight difference in the

2. Chris Read came up with this idea [9EIHHS].

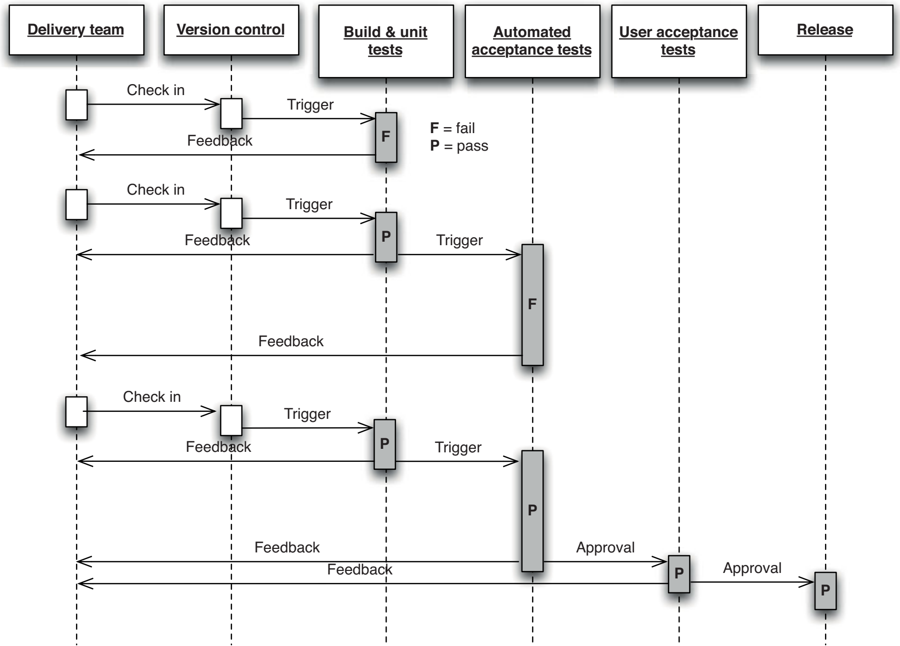

Figure 5.2 *Changes moving through the deployment pipeline*

configuration of a production server. The discipline of the deployment pipeline mitigates this.

Second, when deployment and production release themselves are automated, they are rapid, repeatable, and reliable. It is often so much easier to perform a release once the process is automated that they become "normal" events—meaning that, should you choose, you can perform releases more frequently. This is particularly the case where you are able to step back to an earlier version as well as move forward. When this capability is available, releases are essentially without risk. The worst that can happen is that you find that you have introduced a critical bug—at which point you revert to an earlier version that doesn't contain the bug while you fix the new release offline (see Chapter 10, "Deploying and Releasing Applications").

To achieve this enviable state, we must automate a suite of tests that prove that our release candidates are fit for their purpose. We must also automate deployment to testing, staging, and production environments to remove these manually intensive, error-prone steps. For many systems, other forms of testing and so other stages in the release process are also needed, but the subset that is common to all projects is as follows.

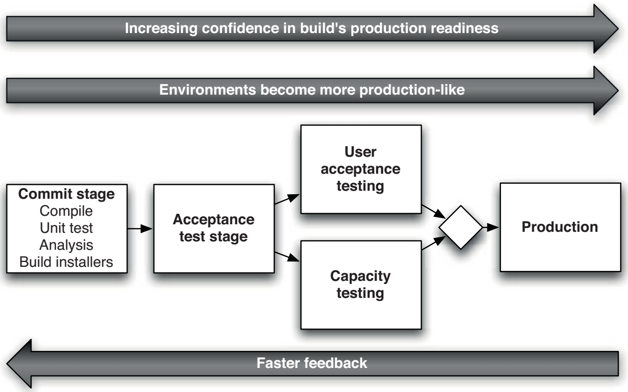

Figure 5.3 *Trade-offs in the deployment pipeline*

- *The commit stage* asserts that the system works at the technical level. It compiles, passes a suite of (primarily unit-level) automated tests, and runs code analysis.
- *Automated acceptance test stages* assert that the system works at the functional and nonfunctional level, that behaviorally it meets the needs of its users and the specifications of the customer.
- *Manual test stages* assert that the system is usable and fulfills its requirements, detect any defects not caught by automated tests, and verify that it provides value to its users. These stages might typically include exploratory testing environments, integration environments, and UAT (user acceptance testing).
- *Release stage* delivers the system to users, either as packaged software or by deploying it into a production or staging environment (a staging environment is a testing environment identical to the production environment).

We refer to these stages, and any additional ones that may be required to model your process for delivering software, as a *deployment pipeline*. It is also sometimes referred to as a continuous integration pipeline, a build pipeline, a deployment production line, or a living build. Whatever it is called, this is, fundamentally, an automated software delivery process. This is not intended to imply that there is no human interaction with the system through this release process; rather, it ensures that error-prone and complex steps are automated, reliable, and repeatable in execution. In fact, human interaction is increased: The ability to deploy the system at all stages of its development by pressing a button encourages its frequent use by testers, analysts, developers, and (most importantly) users.

A Basic Deployment Pipeline

Figure 5.4 shows a typical deployment pipeline and captures the essence of the approach. Of course, a real pipeline will reflect your project's actual process for delivering software.

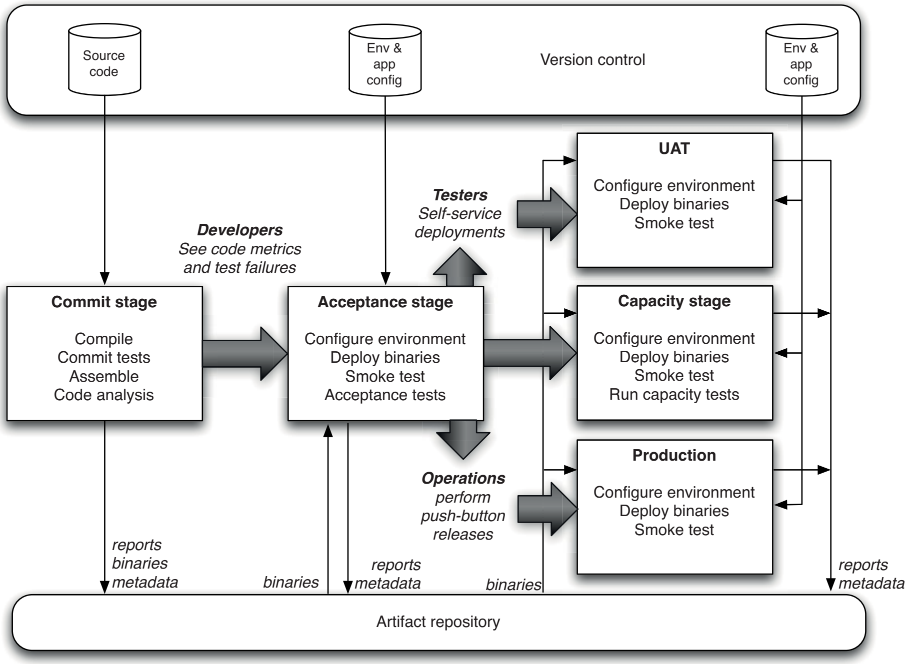

Figure 5.4 *Basic deployment pipeline*

The process starts with the developers committing changes into their version control system. At this point, the continuous integration management system responds to the commit by triggering a new instance of our pipeline. The first (commit) stage of the pipeline compiles the code, runs unit tests, performs code analysis, and creates installers. If the unit tests all pass and the code is up to scratch, we assemble the executable code into binaries and store them in an artifact repository. Modern CI servers provide a facility to store artifacts like these and make them easily accessible both to the users and to the later stages in your pipeline. Alternatively, there are plenty of tools like Nexus and Artifactory which help you manage artifacts. There are other tasks that you might also run as part of the commit stage of your pipeline, such as preparing a test database to use for your acceptance tests. Modern CI servers will let you execute these jobs in parallel on a build grid.

The second stage is typically composed of longer-running automated acceptance tests. Again, your CI server should let you split these tests into suites which can be executed in parallel to increase their speed and give you feedback faster—typically within an hour or two. This stage will be triggered automatically by the successful completion of the first stage in your pipeline.

At this point, the pipeline branches to enable independent deployment of your build to various environments—in this case, UAT (user acceptance testing), capacity testing, and production. Often, you won't want these stages to be automatically triggered by the successful completion of your acceptance test stage. Instead, you'll want your testers or operations team to be able to self-service builds into their environments manually. To facilitate this, you'll need an automated script that performs this deployment. Your testers should be able to see the release candidates available to them as well as their status—which of the previous two stages each build has passed, what were the check-in comments, and any other comments on those builds. They should then be able to press a button to deploy the selected build by running the deployment script in the relevant environment.

The same principle applies to further stages in the pipeline, except that, typically, the various environments you want to be able to deploy to will have different groups of users who "own" these environments and have the ability to self-service deployments to them. For example, your operations team will likely want to be the only one who can approve deployments to production.

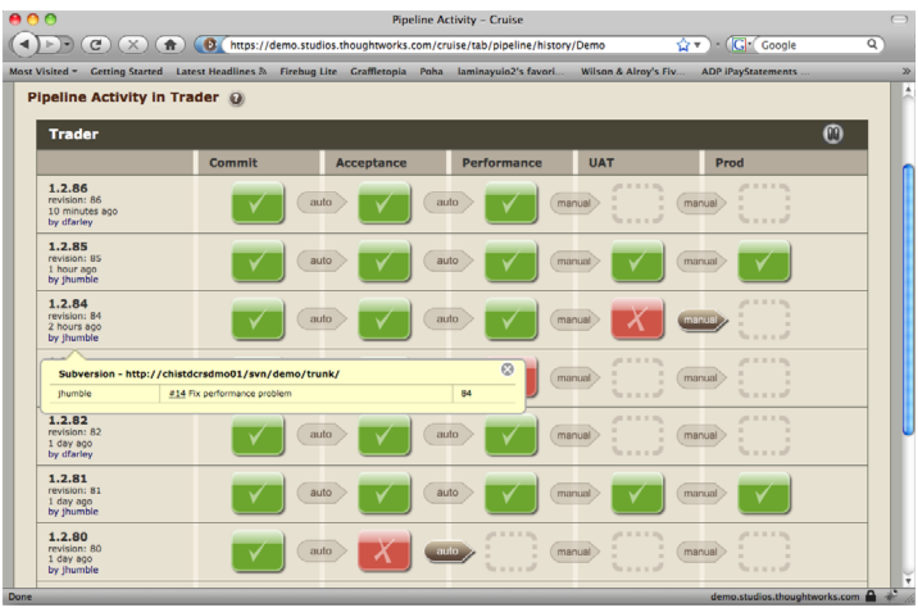

Figure 5.5 *Go showing which changes have passed which stages*

Finally, it's important to remember that the purpose of all this is to get feedback as fast as possible. To make the feedback cycle fast, you need to be able to see which build is deployed into which environment, and which stages in your pipeline each build has passed. Figure 5.5 is a screenshot from Go showing what this looks like in practice.

Notice that you can see every check-in down the side of the page, every stage in the pipeline that each check-in has been through, and whether it passed or failed that stage. Being able to correlate a particular check-in, and hence build, to the stages in the pipeline it has passed through is crucial. It means that if you see a problem in the acceptance tests (for example), you can immediately find out which changes were checked into version control that resulted in the acceptance tests failing.

Deployment Pipeline Practices

Shortly, we'll go into some more detail on the stages in the deployment pipeline. But before we do so, in order to get the benefits of this approach, there are some practices you should follow.

Only Build Your Binaries Once

For convenience, we will refer to the collections of executable code as binaries, although if you don't need to compile your code these "binaries" may be just collections of source files. Jars, .NET assemblies, and .so files are all examples of binaries.

Many build systems use the source code held in the version control system as the canonical source for many steps. The code will be compiled repeatedly in different contexts: during the commit process, again at acceptance test time, again for capacity testing, and often once for each separate deployment target. Every time you compile the code, you run the risk of introducing some difference. The version of the compiler installed in the later stages may be different from the version that you used for your commit tests. You may pick up a different version of some third-party library that you didn't intend. Even the configuration of the compiler may change the behavior of the application. We have seen bugs from every one of these sources reaching production.

A related antipattern is to promote at the source-code level rather than at the binary level. For more information on this antipattern, see the "ClearCase and the Rebuilding-from-Source Antipattern" section on page 403.

This antipattern violates two important principles. The first is to keep the deployment pipeline efficient, so the team gets feedback as soon as possible. Recompiling violates this principle because it takes time, especially in large systems. The second principle is to always build upon foundations known to be sound. The binaries that get deployed into production should be exactly the same as those that went through the acceptance test process—and indeed in many pipeline implementations, this is checked by storing hashes of the binaries at the time they are created and verifying that the binary is identical at every subsequent stage in the process.

If we re-create binaries, we run the risk that some change will be introduced between the creation of the binaries and their release, such as a change in the toolchain between compilations, and that the binary we release will be different from the one we tested. For auditing purposes, it is essential to ensure that no changes have been introduced, either maliciously or by mistake, between creating the binaries and performing the release. Some organizations insist that compilation and assembly, or packaging in the case of interpreted languages, occurs in a special environment that cannot be accessed by anyone except senior personnel. Once we have created our binaries, we will reuse them without re-creating them at the point of use.

So, you should only build your binaries once, during the commit stage of the build. These binaries should be stored on a filesystem somewhere (not in version control, since they are derivatives of your baseline, not part of its definition) where it is easy to retrieve them for later stages in the pipeline. Most CI servers will handle this for you, and will also perform the crucial task of allowing you to trace back to the version control check-in which was used to create them. It isn't worth spending a lot of time and effort ensuring binaries are backed up—it should be possible to exactly re-create them by running your automated build process from the correct revision in version control.

If you take our advice, it will initially feel as though you have more work to do. You will need to establish some way of propagating your binaries to the later stages in the deployment pipeline, if your CI tool doesn't do this for you already. Some of the simplistic configuration management tools that come with popular development environments will be doing the wrong thing. A notable example of this is project templates that directly generate assemblies containing both code and configuration files, such as ear and war files, as a single step in the build process.

One important corollary of this principle is that it must be possible to deploy these binaries to every environment. This forces you to separate code, which remains the same between environments, and configuration, which differs between environments. This, in turn, will lead you to managing your configuration correctly, applying a gentle pressure towards better-structured build systems.

Why Binaries Should Not Be Environment-Specific

We consider it a very bad practice to create binary files intended to run in a single environment. This approach, while common, has several serious drawbacks that compromise the overall ease of deployment, flexibility, and maintainability of the system. Some tools even encourage this approach.

When build systems are organized in this way, they usually become very complex very quickly, spawning lots of special-case hacks to cope with the differences and the vagaries of various deployment environments. On one project that we worked on, the build system was so complex that it took a full-time team of five people to maintain it. Eventually, we relieved them of this unpopular job by reorganizing the build and separating the environment-specific configuration from the environment-agnostic binaries.

Such build systems make unnecessarily complex what should be trivial tasks, such as adding a new server to a cluster. This, in turn, forces us into fragile, expensive release processes. If your build creates binaries that only run on specific machines, start planning how to restructure them now!

This brings us neatly to the next practice.

Deploy the Same Way to Every Environment

It is essential to use the same process to deploy to every environment—whether a developer or analyst's workstation, a testing environment, or production—in order to ensure that the build and deployment process is tested effectively. Developers deploy all the time; testers and analysts, less often; and usually, you will deploy to production fairly infrequently. But this frequency of deployment is the inverse of the risk associated with each environment. The environment you deploy to least frequently (production) is the most important. Only after you have tested the deployment process hundreds of times on many environments can you eliminate the deployment script as a source of error.

Every environment is different in some way. If nothing else, it will have a unique IP address, but often there are other differences: operating system and middleware configuration settings, the location of databases and external services, and other configuration information that needs to be set at deployment time. This does not mean you should use a different deployment script for every environment. Instead, keep the settings that are unique for each environment separate. One way to do this is to use properties files to hold configuration information. You can have a separate properties file for each environment. These files should be checked in to version control, and the correct one selected either by looking at the hostname of the local server, or (in a multimachine environment) through the use of an environment variable supplied to the deployment script. Some other ways to supply deploy-time configuration include keeping it in a directory service (like LDAP or ActiveDirectory) or storing it in a database and accessing it through an application like ESCAPE [apvrEr]. There is more on managing software configuration in the "Managing Software Configuration" section on page 39.

It's important to use the same deploy-time configuration mechanism for each of your applications. This is especially true in a large company, or where many heterogeneous technologies are in play. Generally, we're against handing down edicts from on high—but we've seen too many organizations where it was impossibly arduous to work out, for a given application in a given environment, what configuration was actually supplied at deployment time. We know places where you have to email separate teams on separate continents to piece together this information. This becomes an enormous barrier to efficiency when you're trying to work out the root cause of some bug—and when you add together the delays it introduces into your value stream, it is incredibly costly.

It should be possible to consult one single source (a version control repository, a directory service, or a database) to find configuration settings for all your applications in all of your environments.

If you work in a company where production environments are managed by a team different from the team responsible for development and testing environments, both teams will need to work together to make sure the automated deployment process works effectively across all environments, including development environments. Using the same script to deploy to production that you use to deploy to development environments is a fantastic way to prevent the "it works on my machine" syndrome [c29ETR]. It also means that when you come to release, you will have tested your deployment process hundreds of times by deploying to all of your other environments. This is one of the best ways we know to mitigate the risk of releasing software.

We've assumed that you have an automated process for deploying your application—but, of course, many organizations still deploy manually. If you have a manual deployment process, you should start by ensuring that the process is the same for every environment and then begin to automate it bit by bit, with the goal of having it fully scripted. Ultimately, you should only need to specify the target environment and the version of the application to initiate a successful deployment. An automated, standardized deployment process will have a huge positive effect on your ability to release your application repeatably and reliably, and ensure that the process is completely documented and audited. We cover automating deployment in detail in the following chapter.

This principle is really another application of the rule that you should separate what changes from what doesn't. If your deployment script is different for different environments, you have no way of knowing that what you're testing will actually work when you go live. Instead, if you use the same process to deploy everywhere, when a deployment doesn't work to a particular environment you can narrow it down to one of three causes:

- A setting in your application's environment-specific configuration file
- A problem with your infrastructure or one of the services on which your application depends
- The configuration of your environment

Establishing which of these is the underlying cause is the subject of the next two practices.

Smoke-Test Your Deployments

When you deploy your application, you should have an automated script that does a smoke test to make sure that it is up and running. This could be as simple as launching the application and checking to make sure that the main screen comes up with the expected content. Your smoke test should also check that any services your application depends on are up and running—such as a database, messaging bus, or external service.

The smoke test, or deployment test, is probably the most important test to write once you have a unit test suite up and running—indeed, it's arguably even more important. It gives you the confidence that your application actually runs. If it doesn't run, your smoke test should be able to give you some basic diagnostics as to whether your application is down because something it depends on is not working.

Deploy into a Copy of Production

The other main problem many teams experience going live is that their production environment is significantly different from their testing and development environments. To get a good level of confidence that going live will actually work, you need to do your testing and continuous integration on environments that are as similar as possible to your production environment.

Ideally, if your production environment is simple or you have a sufficiently large budget, you can have exact copies of production to run your manual and automated tests on. Making sure that your environments are the same requires a certain amount of discipline to apply good configuration management practices. You need to ensure that:

- Your infrastructure, such as network topology and firewall configuration, is the same.
- Your operating system configuration, including patches, is the same.
- Your application stack is the same.
- Your application's data is in a known, valid state. Migrating data when performing upgrades can be a major source of pain in deployments. We deal more with this topic in Chapter 12, "Managing Data."

You can use such practices as disk imaging and virtualization, and tools like Puppet and InstallShield along with a version control repository, to manage your environments' configuration. We discuss this in detail in Chapter 11, "Managing Infrastructure and Environments."

Each Change Should Propagate through the Pipeline Instantly

Before continuous integration was introduced, many projects ran various parts of their process off a schedule—for example, builds might run hourly, acceptance tests nightly, and capacity tests over the weekend. The deployment pipeline takes a different approach: The first stage should be triggered upon every check-in, and each stage should trigger the next one immediately upon successful completion. Of course this is not always possible when developers (especially on large teams) are checking in very frequently, given that the stages in your process can take a not insignificant amount of time. The problem is shown in Figure 5.6.

In this example, somebody checks a change into version control, creating version 1. This, in turn, triggers the first stage in the pipeline (build and unit tests). This passes, and triggers the second stage: the automated acceptance tests. Somebody then checks in another change, creating version 2. This triggers the build and unit tests again. However, even though these have passed, they cannot trigger a new instance of the automated acceptance tests, since they are already running. In the meantime, two more check-ins have occurred in quick succession. However, the CI system should not attempt to build both of them—if it followed that rule, and developers continued to check in at the same rate, the builds would get further and further behind what the developers are currently doing.

Instead, once an instance of the build and unit tests has finished, the CI system checks to see if new changes are available, and if so, builds off the most recent set available—in this case, version 4. Suppose this breaks the build and unit tests stage. The build system doesn't know which commit, 3 or 4, caused the stage to break, but it is usually simple for the developers to work this out for themselves. Some CI systems will let you run specified versions out of order, in which case the developers could trigger the first stage off revision 3 to see if it passes or fails, and thus whether it was commit 3 or 4 that broke the build. Either way, the development team checks in version 5, which fixes the problem.

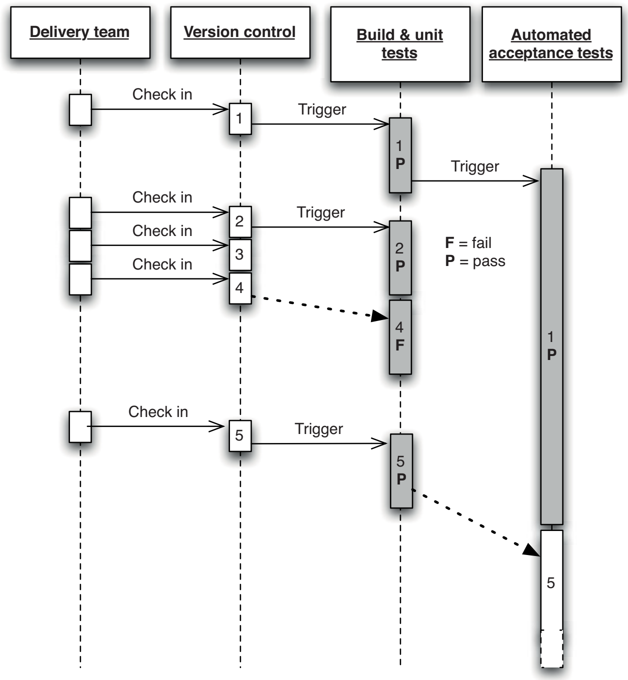

Figure 5.6 *Scheduling stages in a pipeline*

When the acceptance tests finally finish, the CI system's scheduler notices that new changes are available, and triggers a new run of the acceptance tests against version 5.

This intelligent scheduling is crucial to implementing a deployment pipeline. Make sure your CI server supports this kind of scheduling workflow—many do—and ensure that changes propagate immediately so that you don't have to run stages off a fixed schedule.

This only applies to stages that are fully automated, such as those containing automated tests. The later stages in the pipeline that perform deployments to manual testing environments need to be activated on demand, which we describe in a later section in this chapter.

If Any Part of the Pipeline Fails, Stop the Line

As we said in the "Implementing Continuous Integration" section on page 56, the most important step in achieving the goals of this book—rapid, repeatable, reliable releases—is for your team to accept that every time they check code into version control, it will successfully build and pass every test. This applies to the entire deployment pipeline. If a deployment to an environment fails, the whole team owns that failure. They should stop and fix it before doing anything else.

The Commit Stage

A new instance of your deployment pipeline is created upon every check-in and, if the first stage passes, results in the creation of a release candidate. The aim of the first stage in the pipeline is to eliminate builds that are unfit for production and signal the team that the application is broken as quickly as possible. We want to expend a minimum of time and effort on a version of the application that is obviously broken. So, when a developer commits a change to the version control system, we want to evaluate the latest version of the application quickly. The developer who checked in then waits for the results before moving on to the next task.

There are a few things we want to do as part of our commit stage. Typically, these tasks are run as a set of jobs on a build grid (a facility provided by most CI servers) so the stage completes in a reasonable length of time. The commit stage should ideally take less than five minutes to run, and certainly no more than ten minutes. The commit stage typically includes the following steps:

- Compile the code (if necessary).
- Run a set of commit tests.
- Create binaries for use by later stages.
- Perform analysis of the code to check its health.
- Prepare artifacts, such as test databases, for use by later stages.

The first step is to compile the latest version of the source code and notify the developers who committed changes since the last successful check-in if there is an error in compilation. If this step fails, we can fail the commit stage immediately and eliminate this instance of the pipeline from further consideration.

Next, a suite of tests is run, optimized to execute very quickly. We refer to this suite of tests as commit stage tests rather than unit tests because, although the vast majority of them are indeed unit tests, it is useful to include a small selection of tests of other types at this stage in order to get a higher level of confidence that the application is really working if the commit stage passes. These are the same tests that developers run before they check in their code (or, if they have the facility to do so, through a pretested commit on the build grid).

Begin the design of your commit test suite by running all unit tests. Later, as you learn more about what types of failure are common in acceptance test runs and other later stages in the pipeline, you should add specific tests to your commit test suite to try and find them early on. This is an ongoing process optimization that is important if you are to avoid the higher costs of finding and fixing bugs in later pipeline stages.

Establishing that your code compiles and passes tests is great, but it doesn't tell you a lot about the nonfunctional characteristics of your application. Testing nonfunctional characteristics such as capacity can be hard, but you can run analysis tools giving you feedback on such characteristics of your code base as test coverage, maintainability, and security breaches. Failure of your code to meet preset thresholds for these metrics should fail the commit stage the same way that a failing test does. Useful metrics include:

- Test coverage (if your commit tests only cover 5% of your codebase, they're pretty useless)
- Amount of duplicated code
- Cyclomatic complexity
- Afferent and efferent coupling
- Number of warnings
- Code style

The final step in the commit stage, following successful execution of everything up to this point, is the creation of a deployable assembly of your code ready for deployment into any subsequent environment. This, too, must succeed for the commit stage to be considered a success as a whole. Treating the creation of the executable code as a success criteria in its own right is a simple way of ensuring that our build process itself is also under constant evaluation and review by our continuous integration system.

Commit Stage Best Practices

Most of the practices described in Chapter 3, "Continuous Integration," apply to the commit stage. Developers are expected to wait until the commit stage of the deployment pipeline succeeds. If it fails, they should either quickly fix the problem, or back their changes out from version control. In the ideal world—a world of infinite processor power and unlimited network bandwidth—we would like our developers to wait for all tests to pass, even the manual ones, so that they could fix any problem immediately. In reality, this is not practical, as the later stages in the deployment pipeline (automated acceptance testing, capacity testing, and manual acceptance testing) are lengthy activities. This is the reason for pipelining your test process—it's important to get feedback as quickly as possible, when problems are cheap to fix, but not at the expense of getting more comprehensive feedback when it becomes available.

The Origin of the Term "Deployment Pipeline"

When we first used this idea, we named it a pipeline not because it was like a liquid flowing through a pipe; rather, for the hardcore geeks amongst us, it reminded us of the way processors "pipeline" their instruction execution in order to get a degree of parallelism. Processor chips can execute instructions in parallel. But how do you take a stream of machine instructions intended to be executed serially and divide them up into parallel streams that make sense? The way processors do this is very clever and quite complex, but in essence they often come to points where they effectively "guess" the result of an operation in a separate execution pipeline and start executing on the assumption of that guess. If the guess is later found to be wrong, the results of the stream that was based on it are simply dumped. There has been no gain—but no loss either. However, if the guess was good, the processor has just done twice as much work in the time it would take a single stream of execution—so for that spell, it was running twice as fast.

Our deployment pipeline concept works in the same way. We design our commit stage so that it will catch the majority of problems, while running very quickly. As a result, we make a "guess" that all of our subsequent test stages will pass, so we resume work on new features, preparing for the next commit and the initiation of the next release candidate. Meanwhile, our pipeline optimistically works on our assumption of success, in parallel to our development of new features.

Passing the commit stage is an important milestone in the journey of a release candidate. It is a gate in our process that, once passed, frees developers to move on to their next task. However, they retain a responsibility to monitor the progress of the later stages too. Fixing broken builds remains the top priority for the development team even when those breakages occur in the later stages of the pipeline. We are gambling on success—but are ready to pay our technical debts should our gamble fail.

If you only implement a commit stage in your development process, it usually represents an enormous step forward in the reliability and quality of the output of your teams. However, there are several more stages necessary to complete what we consider to be a minimal deployment pipeline.

The Automated Acceptance Test Gate

A comprehensive commit test suite is an excellent litmus test for many classes of errors, but there is much that it won't catch. Unit tests, which comprise the vast majority of the commit tests, are so coupled to the low-level API that it is often hard for the developers to avoid the trap of proving that the solution works in a particular way, rather than asserting that is solves a particular problem.

Why Unit Tests Aren't Enough

We once worked on a large project with around 80 developers. The system was developed using continuous integration at the heart of our development process. As a team, our build discipline was pretty good; we needed it to be with a team of this size.

One day we deployed the latest build that had passed our unit tests into a test environment. This was a lengthy but controlled approach to deployment that our environment specialists carried out. However, the system didn't seem to work. We spent a lot of time trying to find what was wrong with the configuration of the environment, but we couldn't find the problem. Then one of our senior developers tried the application on his development machine. It didn't work there either.

He stepped back through earlier and earlier versions, until he found that the system had actually stopped working three weeks earlier. A tiny, obscure bug had prevented the system from starting correctly.

This project had good unit test coverage, with the average for all modules around 90%. Despite this, 80 developers, who usually only ran the tests rather than the application itself, did not see the problem for three weeks.

We fixed the bug and introduced a couple of simple, automated smoke tests that proved that the application ran and could perform its most fundamental function as part of our continuous integration process.

We learned a lot of lessons from this and many other experiences on this big complex project. But the most fundamental one was that unit tests only test a developer's perspective of the solution to a problem. They have only a limited ability to prove that the application does what it is supposed to from a users perspective. If we want to be sure that the application provides to its users the value that we hope it will, we will need another form of test. Our developers could have achieved this by running the application more frequently themselves and interacting with it. This would have solved the specific problem that we described above, but it is not a very effective approach for a big complex application.

This story also points to another common failing in the development process that we were using. Our first assumption was that there was a problem with our deployment—that we had somehow misconfigured the system when we deployed it to our test environment. This was a fair assumption, because that sort of failure was quite common. Deploying the application was a complex, manually intensive process that was quite prone to error.

So, although we had a sophisticated, well-managed, disciplined continuous integration process in place, we still could not be confident that we could identify real functional problems. Nor could we be sure that, when it came time to deploy the system, further errors would not be introduced. Furthermore, since deployments took so long, it was often the case that the process for deployment would change every time the deployment happened. This meant that every attempt at deployment was a new experiment—a manual, error-prone process. This created a vicious circle which meant very high-risk releases.

Commit tests that run against every check-in provide us with timely feedback on problems with the latest build and on bugs in our application in the small. But without running acceptance tests in a production-like environment, we know nothing about whether the application meets the customer's specifications, nor whether it can be deployed and survive in the real world. If we want timely feedback on these topics, we must extend the range of our continuous integration process to test and rehearse these aspects of our system too.

The relationship of the automated acceptance test stage of our deployment pipeline to functional acceptance testing is similar to that of the commit stage to unit testing. The majority of tests running during the acceptance test stage are functional acceptance tests, but not all.

The goal of the acceptance test stage is to assert that the system delivers the value the customer is expecting and that it meets the acceptance criteria. The acceptance test stage also serves as a regression test suite, verifying that no bugs are introduced into existing behavior by new changes. As we describe in Chapter 8, "Automated Acceptance Testing," the process of creating and maintaining automated acceptance tests is not carried out by separate teams but is brought into the heart of the development process and carried out by cross-functional delivery teams. Developers, testers, and customers work together to create these tests alongside the unit tests and the code they write as part of their normal development process.

Crucially, the development team must respond immediately to acceptance test breakages that occur as part of the normal development process. They must decide if the breakage is a result of a regression that has been introduced, an intentional change in the behavior of the application, or a problem with the test. Then they must take the appropriate action to get the automated acceptance test suite passing again.

The automated acceptance test gate is the second significant milestone in the lifecycle of a release candidate. The deployment pipeline will only allow the later stages, such as manually requested deployments, to access builds that have successfully overcome the hurdle of automated acceptance testing. While it is possible to try and subvert the system, this is so time-consuming and expensive that the effort is much better spent on fixing the problem that the deployment pipeline has identified and deploying in the controlled and repeatable manner it supports. The deployment pipeline makes it easier to do the right thing than to do the wrong thing, so teams do the right thing.

Thus a release candidate that does not meet all of its acceptance criteria will never get released to users.

Automated Acceptance Test Best Practices

It is important to consider the environments that your application will encounter in production. If you're only deploying to a single production environment under your control, you're lucky. Simply run your acceptance tests on a copy of this environment. If the production environment is complex or expensive, you can use a scaled-down version of it, perhaps using a couple of middleware servers while there might be many of them in production. If your application depends on external services, you can use test doubles for any external infrastructure that you depend on. We go into more detail on these approaches in Chapter 8, "Automated Acceptance Testing."

If you have to target many different environments, for example if you're developing software that has to be installed on a user's computer, you will need to run acceptance tests on a selection of likely target environments. This is most easily accomplished with a build grid. Set up a selection of test environments, at least one for each target test environment, and run acceptance tests in parallel on all of them.

In many organizations where automated functional testing is done at all, a common practice is to have a separate team dedicated to the production and maintenance of the test suite. As described at length in Chapter 4, "Implementing a Testing Strategy," this is a bad idea. The most problematic outcome is that the developers don't feel as if they own the acceptance tests. As a result, they tend not to pay attention to the failure of this stage of the deployment pipeline, which leads to it being broken for long periods of time. Acceptance tests written without developer involvement also tend to be tightly coupled to the UI and thus brittle and badly factored, because the testers don't have any insight into the UI's underlying design and lack the skills to create abstraction layers or run acceptance tests against a public API.

The reality is that *the whole team owns the acceptance tests*, in the same way as the whole team owns every stage of the pipeline. If the acceptance tests fail, the whole team should stop and fix them immediately.

One important corollary of this practice is that developers must be able to run automated acceptance tests on their development environments. It should be easy for a developer who finds an acceptance test failure to fix it easily on their own machine and verify the fix by running that acceptance test locally. The most common obstacles to this are insufficient licenses for the testing software being used and an application architecture that prevents the system from being deployed on a development environment so that the acceptance tests can be run against it. If your automated acceptance testing strategy is to succeed in the long term, these kinds of obstacles need to be removed.

It can be easy for acceptance tests to become too tightly coupled to a particular solution in the application rather than asserting the business value of the system. When this happens, more and more time is spent maintaining the acceptance tests as small changes in the behavior of the system invalidate tests. Acceptance tests should be expressed in the language of the business (what Eric Evans calls the "ubiquitous language"[3]), not in the language of the technology of the application. By this we mean that while it is fine to write the acceptance tests in the

3. Evans, 2004.

same programming language that your team uses for development, the abstraction should work at the level of business behavior—"place order" rather than "click order button," "confirm fund transfer" rather than "check fund_table has results," and so on.

While acceptance tests are extremely valuable, they can also be expensive to create and maintain. It is thus essential to bear in mind that automated acceptance tests are also regression tests. Don't follow a naive process of taking your acceptance criteria and blindly automating every one.

We have worked on several projects that found, as a result of following some of the bad practices described above, that the automated functional tests were not delivering enough value. They were costing far too much to maintain, and so automated functional testing was stopped. This is the right decision if the tests cost more effort than they save, but changing the way the creation and maintenance of the tests are managed can dramatically reduce the effort expended and change the cost-benefit equation significantly. Doing acceptance testing right is the main subject of Chapter 8, "Automated Acceptance Testing."

Subsequent Test Stages

The acceptance test stage is a significant milestone in the lifecycle of a release candidate. Once this stage has been completed, a successful release candidate has moved on from something that is largely the domain of the development team to something of wider interest and use.

For the simplest deployment pipelines, a build that has passed acceptance testing is ready for release to users, at least as far as the automated testing of the system is concerned. If the candidate fails this stage, it by definition is not fit to be released.

The progression of the release candidate to this point has been automatic, with successful candidates being automatically promoted to the next stage. If you are delivering software incrementally, it is possible to have an automated deployment to production, as described in Timothy Fitz' blog entry, "Continuous Deployment" [dbnlG8]. But for many systems, some form of manual testing is desirable before release, even when you have a comprehensive set of automated tests. Many projects have environments for testing integration with other systems, environments for testing capacity, exploratory testing environments, and staging and production environments. Each of these environments can be more or less production-like and have their own unique configuration.

The deployment pipeline takes care of deployments to testing environments too. Release management systems such as AntHill Pro and Go provide the ability to see what is currently deployed into each environment and to perform a push-button deployment into that environment. Of course behind the scenes, these simply run the deployment scripts you have written to perform the deployment.

It is also possible to build your own system to do this, based on open source tools such as Hudson or the CruiseControl family, although commercial tools provide visualizations, reporting, and fine-grained authorization of deployments out of the box. If you create your own system, the key requirements are to be able to see a list of release candidates that have passed the acceptance test stage, have a button to deploy the version of your choice into the environment of your choice, see which release candidate is currently deployed in each environment and which version in version control it came from. Figure 5.7 shows a home-brewed system that performs these functions.

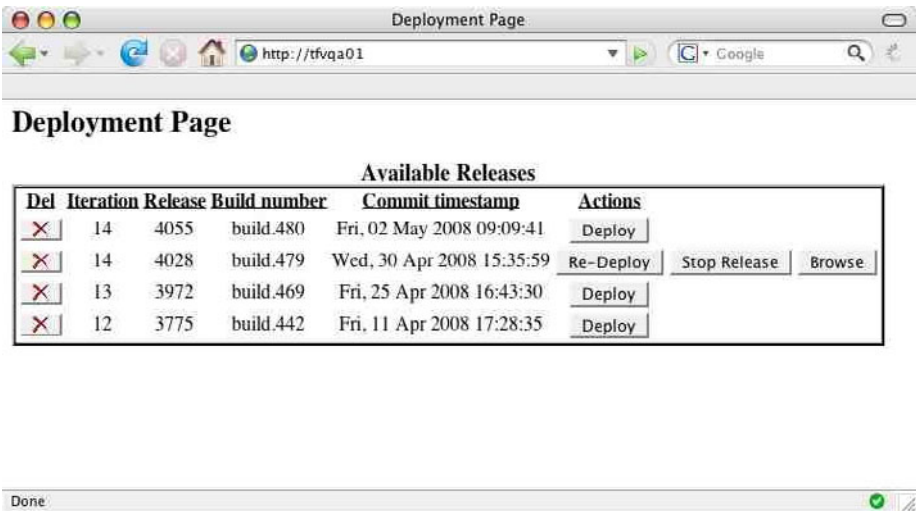

Figure 5.7 *Example deployment page*

Deployments to these environments may be executed in sequence, each one depending on the successful outcome of the one before, so that you can only deploy to production once you have deployed to UAT and staging. They could also occur in parallel, or be offered as optional stages that are manually selected.

Crucially, the deployment pipeline allows testers to deploy any build to their testing environments on demand. This replaces the concept of the "nightly build." In the deployment pipeline, instead of testers being given a build based on an arbitrary revision (the last change committed before everybody went home), testers can see which builds passed the automated tests, which changes were made to the application, and choose the build they want. If the build turns out to be unsatisfactory in some way—perhaps it does not include the correct change, or contains some bug which makes it unsuitable for testing—the testers can redeploy any other build.

Manual Testing

In iterative processes, acceptance testing is always followed by some manual testing in the form of exploratory testing, usability testing, and showcases. Before this point, developers may have demonstrated features of the application to analysts and testers, but neither of these roles will have wasted time on a build that is not known to have passed automated acceptance testing. A tester's role in this process should not be to regression-test the system, but first of all to ensure that the acceptance tests genuinely validate the behavior of the system by manually proving that the acceptance criteria are met.

After that, testers focus on the sort of testing that human beings excel at but automated tests are poor at. They do exploratory testing, perform user testing of the application's usability, check the look and feel on various platforms, and carry out pathological worst-case tests. Automated acceptance testing is what frees up time for testers so they can concentrate on these high-value activities, instead of being human test-script execution machines.

Nonfunctional Testing

Every system has many nonfunctional requirements. For example, almost every system has some kind of requirements on capacity and security, or the service-level agreements it must conform to. It usually makes sense to run automated tests to measure how well the system adheres to these requirements. For more details on how to achieve this, see Chapter 9, "Testing Nonfunctional Requirements." For other systems, testing of nonfunctional requirements need not be done on a continuous basis. Where it is required, in our experience it is still valuable to create a stage in your pipeline for running these automated tests.

Whether the results of the capacity test stage form a gate or simply inform human decision-making is one of the criteria that determine the organization of the deployment pipeline. For very high-performance applications, it makes sense to run capacity testing as a wholly automated outcome of a release candidate successfully passing the acceptance test stage. If the candidate fails capacity testing, it is not usually deemed to be deployable.

For many applications, though, the judgment of what is deemed acceptable is more subjective than that. It makes more sense to present the results at the conclusion of the capacity test stage and allow a human being to decide whether the release candidate should be promoted or not.

Preparing to Release

There is a business risk associated with every release of a production system. At best, if there is a serious problem at the point of release, it may delay the introduction of valuable new capabilities. At worst, if there is no sensible back-out plan in place, it may leave the business without mission-critical resources because they had to be decommissioned as part of the release of the new system.

The mitigation of these problems is very simple when we view the release step as a natural outcome of our deployment pipeline. Fundamentally, we want to

- Have a release plan that is created and maintained by everybody involved in delivering the software, including developers and testers, as well as operations, infrastructure, and support personnel
- Minimize the effect of people making mistakes by automating as much of the process as possible, starting with the most error-prone stages
- Rehearse the procedure often in production-like environments, so you can debug the process and the technology supporting it
- Have the ability to back out a release if things don't go according to plan
- Have a strategy for migrating configuration and production data as part of the upgrade and rollback processes

Our goal is a completely automated release process. Releasing should be as simple as choosing a version of the application to release and pressing a button. Backing out should be just as simple. There is a great deal more information on these topics in Chapter 10, "Deploying and Releasing Applications."

Automating Deployment and Release

The less control we have over the environment in which our code executes, the more potential there is for unexpected behaviors. Thus, whenever we release a software system, we want to be in control of every single bit that is deployed. There are two factors that may work against this ideal. The first is that for many applications, you simply don't have full control of the operational environment of the software that you create. This is especially true of products and applications that are installed by users, such as games or office applications. This problem is generally mitigated by selecting a representative sample of target environments and running your automated acceptance test suite on each of these sample environments in parallel. You can then mine the data produced to work out which tests fail on which platforms.

The second constraint is that the cost of establishing that degree of control is usually assumed to outweigh the benefits. However, usually the converse is true: Most problems with production environments are caused by insufficient control. As we describe in Chapter 11, production environments should be completely locked down—changes to them should only be made through automated processes. That includes not only deployment of your application, but also changes to their configuration, software stack, network topology, and state. Only in this way is it possible to reliably audit them, diagnose problems, and repair them in a predictable time. As the complexity of the system increases, so does the number of different types of servers, and the higher the level of performance required, the more vital this level of control becomes.

The process for managing your production environment should be used for your other testing environments such as staging, integration, and so forth. In this way you can use your automated change management system to create a perfectly tuned configuration in your manual testing environments. These can be tuned to perfection, perhaps using feedback from capacity testing to evaluate the configuration changes that you make. When you are happy with the result, you can replicate it to every server that needs that configuration, including production, in a predictable, reliable way. All aspects of the environment should be managed in this way, including middleware (databases, web servers, message brokers, and application servers). Each can be tuned and tweaked, with the optimal settings added to your configuration baseline.

The costs of automating the provision and maintenance of environments can be lowered significantly by using automated provisioning and management of environments, good configuration management practices, and (if appropriate) virtualization.

Once the environment's configuration is managed correctly, the application can be deployed. The details of this vary widely depending on the technologies employed in the system, but the steps are always very similar. We exploit this similarity in our approach to the creation of build and deployment scripts, discussed in Chapter 6, "Build and Deployment Scripting," and in the way in which we monitor our process.

With automated deployment and release, the process of delivery becomes democratized. Developers, testers, and operations teams no longer need to rely on ticketing systems and email threads to get builds deployed so they can gather feedback on the production readiness of the system. Testers can decide which version of the system they want in their test environment without needing to be technical experts themselves, nor relying on the availability of such expertise to make the deployment. Since deployment is simple, they can change the build under test more often, perhaps returning to an earlier version of the system to compare its behavior with that of the latest version when they find a particularly interesting bug. Sales people can access the latest version of the application with the killer feature that will swing the deal with a client. There are more subtle changes too. In our experience, people begin to relax a little. They perceive the project as a whole as less risky—mainly because it *is* less risky.

An important reason for the reduction in risk is the degree to which the process of release itself is rehearsed, tested, and perfected. Since you use the same process to deploy your system to each of your environments and to release it, the deployment process is tested very frequently—perhaps many times a day. After you have deployed a complex system for the fiftieth or hundredth time without a hitch, you don't think about it as a big event any more. Our goal is to get to that stage as quickly as possible. If we want to be wholly confident in the release process and the technology, we must use it and prove it to be good on a regular basis, just like any other aspect of our system. It should be possible to deploy a single change to production through the deployment pipeline with the minimum possible time and ceremony. The release process should be continuously evaluated and improved, identifying any problems as close to the point at which they were introduced as possible.

Many businesses require the ability to release new versions of their software several times a day. Even product companies often need to make new versions of their software available to users quickly, in case critical defects or security holes are found. The deployment pipeline and the associated practices in this book are what makes it possible to do this safely and reliably. Although many agile development processes thrive on frequent release into production—a process we recommend very strongly when it is applicable—it doesn't always make sense to do so. Sometimes we have to do a lot of work before we are in a position to release a set of features that makes sense to our users as a whole, particularly in the realm of product development. However, even if you don't need to release your software several times a day, the process of implementing a deployment pipeline will still make an enormous positive impact on your organization's ability to deliver software rapidly and reliably.

Backing Out Changes

There are two reasons why release days are traditionally feared. The first one is the fear of introducing a problem because somebody might make a hard-to-detect mistake while going through the manual steps of a software release, or because there is a mistake in the instructions. The second fear is that, should the release fail, either because of a problem in the release process or a defect in the new version of the software, you are committed. In either case, the only hope is that you will be clever enough to solve the problem very quickly.

The first problem we mitigate by essentially rehearsing the release many times a day, proving that our automated deployment system works. The second fear is mitigated by providing a back-out strategy. In the worst case, you can then get back to where you were before you began the release, which allows you to take time to evaluate the problem and find a sensible solution.

In general, the best back-out strategy is to keep the previous version of your application available while the new version is being released—and for some time afterwards. This is the basis for some of the deployment patterns we discuss in Chapter 10, "Deploying and Releasing Applications." In a very simple application, this can be achieved (ignoring data and configuration migrations) by having each release in a directory and using a symlink to point to the current version. Usually, the most complex problem associated with both deploying and rolling back is migrating the production data. This is discussed at length in Chapter 12, "Managing Data."

The next best option is to redeploy the previous good version of your appliation from scratch. To this end, you should have the ability to click a button to release any version of your application that has passed all stages of testing, just as you can with other environments under the control of the deployment pipeline. This idealistic position is fully achievable for some systems, even for systems with significant amounts of data associated with them. However, for some systems, even for individual changes, the cost of providing a full, version-neutral back-out may be excessive in time, if not money. Nevertheless, the ideal is useful, because it sets a target which every project should strive to achieve. Even if it falls somewhat short in some respects, the closer you approach this ideal position the easier your deployment becomes.

On no account should you have a different process for backing out than you do for deploying, or perform incremental deployments or rollbacks. These processes will be rarely tested and therefore unreliable. They will also not start from a known-good baseline, and therefore will be brittle. Always roll back either by keeping an old version of the application deployed or by completely redeploying a previous known-good version.

Building on Success

By the time a release candidate is available for deployment into production, we will know with certainty that the following assertions about it are true:

- The code can compile.
- The code does what our developers think it should because it passed its unit tests.
- The system does what our analysts or users think it should because it passed all of the acceptance tests.
- Configuration of infrastructure and baseline environments is managed appropriately, because the application has been tested in an analog of production.
- The code has all of the right components in place because it was deployable.
- The deployment system works because, at a minimum, it will have been used on this release candidate at least once in a development environment, once in the acceptance test stage, and once in a testing environment before the candidate could have been promoted to this stage.
- The version control system holds everything we need to deploy, without the need for manual intervention, because we have already deployed the system several times.

This "building upon success" approach, allied with our mantra of failing the process or any part of it as quickly as possible, works at every level.

Implementing a Deployment Pipeline

Whether you're starting a new project from scratch or trying to create an automated pipeline for an existing system, you should generally take an incremental approach to implementing a deployment pipeline. In this section we'll set out a strategy for going from nothing to a complete pipeline. In general, the steps look like this:

1. Model your value stream and create a walking skeleton.
2. Automate the build and deployment process.
3. Automate unit tests and code analysis.
4. Automate acceptance tests.
5. Automate releases.

Modeling Your Value Stream and Creating a Walking Skeleton

As described at the beginning of this chapter, the first step is to map out the part of your value stream that goes from check-in to release. If your project is already up and running, you can do this in about half an hour using pencil and paper. Go and speak to everybody involved in this process, and write down the steps. Include best guesses for elapsed time and value-added time. If you're working on a new project, you will have to come up with an appropriate value stream. One way to do this is to look at another project within the same organization that has characteristics similar to yours. Alternatively, you could start with a bare minimum: a commit stage to build your application and run basic metrics and unit tests, a stage to run acceptance tests, and a third stage to deploy your application to a production-like environment so you can demo it.

Once you have a value stream map, you can go ahead and model your process in your continuous integration and release management tool. If your tool doesn't allow you to model your value stream directly, you can simulate it by using dependencies between projects. Each of these projects should do nothing at first—they are just placeholders that you can trigger in turn. Using our "bare minimum" example, the commit stage should be run every time somebody checks in to version control. The stage that runs the acceptance tests should trigger automatically when the commit stage passes, using the same binary created in the commit stage. Any stages that deploy the binaries to a production-like environment for manual testing or release purposes should require you to press a button in order to select the version to deploy, and this capability will usually require authorization.

You can then make these placeholders actually do something. If your project is already well under way, that means plugging in your existing build, test, and deploy scripts. If not, your aim is to create a "walking skeleton" [bEUuac], which means doing the smallest possible amount of work to get all the key elements in place. First of all, get the commit stage working. If you don't have any code or unit tests yet, just create the simplest possible "Hello world" example or, for a web application, a single HTML page, and put a single unit test in place that asserts true. Then you can do the deployment—perhaps setting up a virtual directory on IIS and putting your web page into it. Finally, you can do the acceptance test—you need to do this after you've done the deployment, since you need your application deployed in order to run acceptance tests against it. Your acceptance test can crank up WebDriver or Sahi and verify that the web page contains the text "Hello world."

On a new project, all this should be done before work starts on development—as part of iteration zero, if you're using an iterative development process. Your organization's system administrators or operations personnel should be involved in setting up a production-like environment to run demos from and developing the scripts to deploy your application to it. In the following sections, there's more detail on how to create the walking skeleton and develop it as your project grows.

Automating the Build and Deployment Process

The first step in implementing a pipeline is to automate the build and deployment process. The build process takes source code as its input and produces binaries as output. "Binaries" is a deliberately vague word, since what your build process produces will depend on what technology you're using. The key characteristic of binaries is that you should be able to copy them onto a new machine and, given an appropriately configured environment and the correct configuration for the application in that environment, start your application—without relying on any part of your development toolchain being installed on that machine.

The build process should be performed every time someone checks in by your continuous integration server software. Use one of the many tools listed in the "Implementing Continuous Integration" section on page 56. Your CI server should be configured to watch your version control system, check out or update your source code every time a change is made to it, run the automated build process, and store the binaries on the filesystem where they are accessible to the whole team via the CI server's user interface.

Once you have a continuous build process up and running, the next step is automating deployment. First of all, you need to get a machine to deploy your application on. For a new project, this can be the machine your continuous integration server is on. For a project that is more mature, you may need to find several machines. Depending on your organization's conventions, this environment can be called the staging or user acceptance testing (UAT) environment. Either way, this environment should be somewhat production-like, as described in Chapter 10, "Deploying and Releasing Applications," and its provisioning and maintenance should be a fully automated process, as described in Chapter 11, "Managing Infrastructure and Environments."

Several common approaches to deployment automation are discussed in Chapter 6, "Build and Deployment Scripting." Deployment may involve packaging your application first, perhaps into several separate packages if different parts of the application need to be installed on separate machines. Next, the process of installing and configuring your application should be automated. Finally, you should write some form of automated deployment test that verifies that the application has been successfully deployed. It is important that the deployment process is reliable, as it is also used as a prerequisite for automated acceptance testing.

Once your application's deployment process is automated, the next step is to be able to perform push-button deployments to your UAT environment. Configure your CI server so that you can choose any build of your application and click a button to trigger a process that takes the binaries produced by that build, runs the script that deploys the build, and runs the deployment test. Make sure that when developing your build and deployment system you make use of the principles we describe, such as building your binaries only once and separating configuration from binaries, so that the same binaries may be used in every environment. This will ensure that the configuration management for your project is put on a sound footing.

Except for user-installed software, the release process should be the same process you use to deploy to a testing environment. The only technical differences should be in the configuration of the environment.

Automating the Unit Tests and Code Analysis

The next step in developing your deployment pipeline is implementing a full commit stage. This means running unit tests, code analysis, and ultimately a selection of acceptance and integration tests on every check-in. Running unit tests should not require any complex setup, because unit tests by definition don't rely on your application running. Instead, they can be run by one of the many xUnit-style frameworks against your binaries.

Since unit tests do not touch the filesystem or database (or they'd be component tests), they should also be fast to run. This is why you should start running your unit tests directly after building your application. You can also then run static analysis tools against your application to report useful diagnostic data such as coding style, code coverage, cyclomatic complexity, coupling, and so forth.

As your application gets more complex, you will need to write a large number of unit tests and a set of component tests as well. These should all go into the commit stage. Once the commit stage gets over five minutes, it makes sense to split it into suites that run in parallel. In order to do this, you'll need to get several machines (or one machine with plenty of RAM and a few CPUs) and use a CI server that supports splitting up work and running it in parallel.

Automating Acceptance Tests

The acceptance test phase of your pipeline can reuse the script you use to deploy to your testing environment. The only difference is that after the smoke tests are run, the acceptance test framework needs to be started up, and the reports it generates should be collected at the end of the test run for analysis. It also makes sense to store the logs created by your application. If your application has a GUI, you can also use a tool like Vnc2swf to create a screen recording as the acceptance tests are running to help you debug problems.

Acceptance tests fall into two types: functional and nonfunctional. It is essential to start testing nonfunctional parameters such as capacity and scaling characteristics from early on in any project, so that you have some idea of whether your application will meet its nonfunctional requirements. In terms of setup and deployment, this stage can work exactly the same way as the functional acceptance testing stage. However, the tests of course will differ (see Chapter 9, "Testing Nonfunctional Requirements," for more on creating such tests). When you start off, it is perfectly possible to run acceptance tests and performance tests back-to-back as part of a single stage. You can then separate them in order to be able to distinguish easily which set of tests failed. A good set of automated acceptance tests will help you track down intermittent and hard-to-reproduce problems such as race conditions, deadlocks, and resource contention that will be a good deal harder to discover and debug once your application is released.

The varieties of tests you create as part of the acceptance test and commit test stages of your pipeline will of course be determined by your testing strategy (see Chapter 4, "Implementing a Testing Strategy"). However, you should try and get at least one or two of each type of test you need to run automated early on in your project's life, and incorporate them into your deployment pipeline. Thus you will have a framework that makes it easy to add tests as your project grows.

Evolving Your Pipeline

The steps we describe above are found in pretty much every value stream, and hence pipeline, that we have seen. They are usually the first targets for automation. As your project gets more complex, your value stream will evolve. There are two other common potential extensions to the pipeline: components and branches. Large applications are best built as a set of components which are assembled together. In such projects, it may make sense to have a minipipeline for each component, and then a pipeline that assembles all the components and puts the entire application through acceptance tests, nonfunctional tests, and then deployment to testing, staging, and production environments. This topic is dealt with at length in Chapter 13, “Managing Components and Dependencies.” Managing branches is discussed in Chapter 14, “Advanced Version Control.”

The implementation of the pipeline will vary enormously between projects, but the tasks themselves are consistent for most projects. Using them as a pattern can speed up the creation of the build and deployment process for any project. However, ultimately, the point of the pipeline is to model your process for building, deploying, testing, and releasing your application. The pipeline then ensures that each change can pass through this process independently in as automated a fashion as possible.

As you implement the pipeline, you will find that the conversations you have with the people involved and the gains in efficiency you realize will, in turn, have an effect on your process. Thus it is important to remember three things.

First of all, the whole pipeline does not need to be implemented at once. It should be implemented incrementally. If there is a part of your process that is currently manual, create a placeholder for it in your workflow. Ensure your implementation records when this manual process is started and when it completes. This allows you to see how much time is spent on each manual process, and thus estimate to what extent it is a bottleneck.

Second, your pipeline is a rich source of data on the efficiency of your process for building, deploying, testing, and releasing applications. The deployment pipeline implementation you create should record every time a process starts and finishes, and what the exact changes were that went through each stage of your process. This data, in turn, allows you to measure the cycle time from committing a change to having it deployed into production, and the time spent on each stage in the process (some of the commercial tools on the market will do this for you). Thus it becomes possible to see exactly what your process’ bottlenecks are and attack them in order of priority.

Finally, your deployment pipeline is a living system. As you work continuously to improve your delivery process, you should continue to take care of your deployment pipeline, working to improve and refactor it the same way you work on the applications you are using it to deliver.

Metrics

Feedback is at the heart of any software delivery process. The best way to improve feedback is to make the feedback cycles short and the results visible. You should measure continually and broadcast the results of the measurements in some hard-to-avoid manner, such as on a very visible poster on the wall, or on a computer display dedicated to showing bold, big results. Such devices are known as information radiators.

The important question, though, is: What should you measure? What you choose to measure will have an enormous influence on the behavior of your team (this is known as the Hawthorne effect). Measure the lines of code, and developers will write many short lines of code. Measure the number of defects fixed, and testers will log bugs that could be fixed by a quick discussion with a developer.

According to the lean philosophy, it is essential to optimize globally, not locally. If you spend a lot of time removing a bottleneck that is not actually the one constraining your delivery process, you will make no difference to the delivery process. So it is important to have a global metric that can be used to determine if the delivery process as a whole has a problem.

For the software delivery process, the most important global metric is cycle time. This is the time between deciding that a feature needs to be implemented and having that feature released to users. As Mary Poppendieck asks, "How long would it take your organization to deploy a change that involves just one single line of code? Do you do this on a repeatable, reliable basis?"[4] This metric is hard to measure because it covers many parts of the software delivery process—from analysis, through development, to release. However, it tells you more about your process than any other metric.

Many projects, incorrectly, choose other measures as their primary metrics. Projects concerned with the quality of their software often choose to measure the number of defects. However, this is a secondary measure. If a team using this measure discovers a defect, but it takes six months to release a fix for it, knowing that the defect exists is not very useful. Focusing on the reduction of cycle time encourages the practices that increase quality, such as the use of a comprehensive automated suite of tests that is run as a result of every check-in.

A proper implementation of a deployment pipeline should make it simple to calculate the part of the cycle time corresponding to the part of the value stream from check-in to release. It should also let you see the lead time from the check-in to each stage of your process, so you can discover your bottlenecks.

Once you know the cycle time for your application, you can work out how best to reduce it. You can use the Theory of Constraints to do this by applying the following process.

1. Identify the limiting constraint on your system. This is the part of your build, test, deploy, and release process that is the bottleneck. To pick an example at random, perhaps it's the manual testing process.

2. Exploit the constraint. This means ensuring that you should maximize the throughput of that part of the process. In our example (manual testing), you would make sure that there is always a buffer of stories waiting to be manually tested, and ensure that the resources involved in manual testing don't get used for anything else.

3. Subordinate all other processes to the constraint. This implies that other resources will not work at 100%—for example, if your developers work developing stories at full capacity, the backlog of stories waiting to be tested would

4. *Implementing Lean Software Development*, p. 59.

keep on growing. Instead, have your developers work just hard enough to keep the backlog constant and spend the rest of their time writing automated tests to catch bugs so that less time needs to be spent testing manually.

4. Elevate the constraint. If your cycle time is still too long (in other words, steps 2 and 3 haven't helped enough), you need to increase the resources available—hire more testers, or perhaps invest more effort in automated testing.
5. Rinse and repeat. Find the next constraint on your system and go back to step 1.

While cycle time is the most important metric in software delivery, there are a number of other *diagnostics* that can warn you of problems. These include

- Automated test coverage
- Properties of the codebase such as the amount of duplication, cyclomatic complexity, efferent and afferent coupling, style problems, and so on
- Number of defects
- Velocity, the rate at which your team delivers working, tested, ready for use code
- Number of commits to the version control system per day
- Number of builds per day
- Number of build failures per day
- Duration of build, including automated tests

It is worth considering how these metrics are presented. The reports described above produce a huge amount of data, and interpreting this data is an art. Program managers, for example, might expect to see this data analyzed and aggregated into a single "health" metric that is represented in the form of a traffic light that shows red, amber, or green. A team's technical lead will want much more detail, but even they will not want to wade through pages and pages of reports. Our colleague, Julias Shaw, created a project called Panopticode that runs a series of these reports against Java code and produces rich, dense visualizations (such as Figure 5.8) that let you see at a glance whether there is a problem with your codebase and where it lies. The key is to create visualizations that aggregate the data and present them in such a form that the human brain can use its unparalleled pattern-matching skills most effectively to identify problems with your process or codebase.

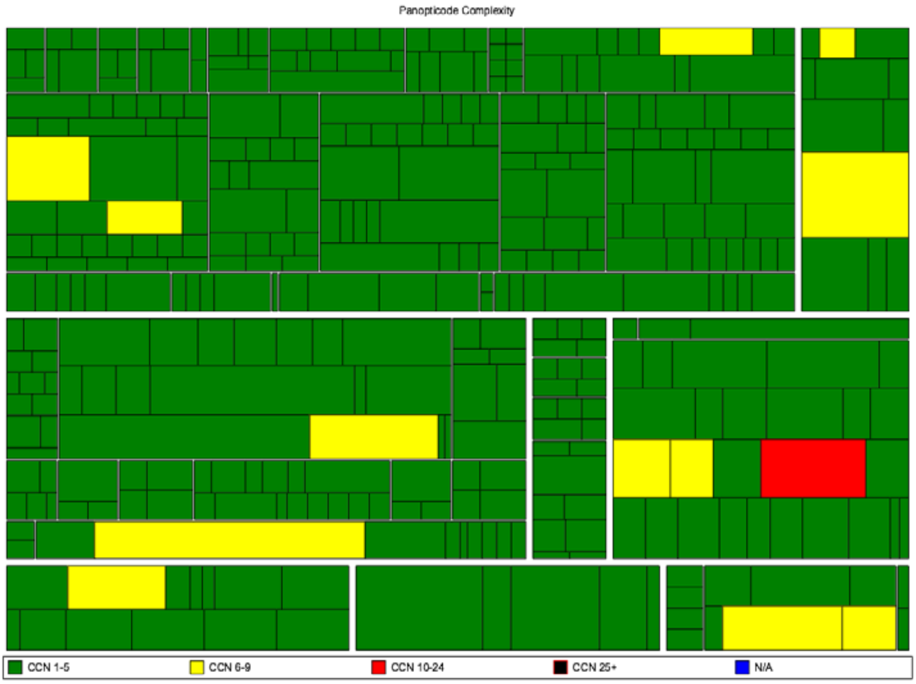

Figure 5.8 *A tree map generated by Panopticode showing cyclomatic complexity for a Java codebase*

Each team's continuous integration server should generate these reports and visualizations on each check-in, and store the reports in your artifact repository. You should then collate the results in a database, and track them across every team. These results should be published on an internal website—have a page for each project. Finally, aggregate them together so that they can be monitored across all of the projects in your development program, or even your whole organization.

Summary

The purpose of the deployment pipeline is to give everyone involved in delivering software visibility into the progress of builds from check-in to release. It should be possible to see which changes have broken the application and which resulted in release candidates suitable for manual testing or release. Your implementation should make it possible to perform push-button deployments into manual testing environments, and to see which release candidates are in those environments. Choosing to release a particular version of your application should also be a push-button task that can be performed with full knowledge that the release candidate being deployed has passed the entire pipeline successfully, and hence has had a battery of automated and manual tests performed on it in a production-like environment.

Once you have a deployment pipeline implemented, inefficiencies in your release process will become obvious. All kinds of useful information can be derived from a working deployment pipeline, such as how long it takes a release candidate to get through the various manual testing stages, the average cycle time from check-in to release, and how many defects are discovered at which stages in your process. Once you have this information, you can work to optimize your process for building and releasing software.

There is no one-size-fits-all solution to the complex problem of implementing a deployment pipeline. The crucial point is to create a system of record that manages each change from check-in to release, providing the information you need to discover problems as early as possible in the process. Having an implementation of the deployment pipeline can then be used to drive out inefficiencies in your process so you can make your feedback cycle faster and more powerful, perhaps by adding more automated acceptance tests and parallelizing them more aggressively, or by making your testing environments more production-like, or by implementing better configuration management processes.

A deployment pipeline, in turn, depends on having some foundations in place: good configuration management, automated scripts for building and deploying your application, and automated tests to prove that your application will deliver value to its users. It also requires discipline, such as ensuring that only changes that have passed through the automated build, test, and deployment system get released. We discuss these prerequisites and the necessary disciplines in Chapter 15, "Managing Continuous Delivery," which includes a maturity model for continuous integration, testing, data management, and so forth.

The following chapters of the book dive into considerably more detail on implementing deployment pipelines, exploring some of the common issues that may arise and discussing techniques that can be adopted within the context of the full lifecycle deployment pipelines described here.

Index

B

D

E

F

G

S

T

W

X

Y

Z