SAMPLE CHAPTER

Re-Engineering LEGACY SOFTWARE

Chris Birchall

MANNING

Re-Engineering Legacy Software

by Chris Birchall

Chapter 1

brief contents

1 Understanding the challenges of legacy projects

This chapter covers

- What a legacy project is
- Examples of legacy code and legacy infrastructure
- Organizational factors that contribute to legacy projects
- A plan for improvement

Hands up if this scene sounds familiar: You arrive at work, grab a coffee, and decide to catch up on the latest tech blogs. You start to read about how the hippest young startup in Silicon Valley is combining fashionable programming language X with exciting NoSQL datastore Y and big data tool Z to change the world, and your heart sinks as you realize that you'll never find the time to even try any of these technologies in your own job, let alone use them to improve your product.

Why not? Because you're tasked with maintaining a few zillion lines of untested, undocumented, incomprehensible legacy code. This code has been in production since before you wrote your first Hello World and has seen dozens of developers

come and go. You spend half of your working day reviewing commits to make sure that they don't cause any regressions, and the other half fighting fires when a bug inevitably slips through the cracks. And the most depressing part of it is that as time goes by, and more code is added to the increasingly fragile codebase, the problem gets worse.

But don't despair! First of all, remember that you're not alone. The average developer spends much more time working with existing code than writing new code, and the vast majority of developers have to deal with legacy projects in some shape or form. Secondly, remember that there's always hope for revitalizing a legacy project, no matter how far gone it may first appear. The aim of this book is to do exactly that.

In this introductory chapter we'll look at examples of the types of problems we're trying to solve, and start to put together a plan for revitalization.

1.1 Definition of a legacy project

First of all, I want to make sure we're on the same page concerning what a legacy project is. I tend to use a very broad definition, labeling as *legacy* any existing project that's difficult to maintain or extend.

Note that we're talking about a project here, not just a codebase. As developers, we tend to focus on the code, but a project encompasses many other aspects, including

- Build tools and scripts
- Dependencies on other systems
- The infrastructure on which the software runs
- Project documentation
- Methods of communication, such as between developers, or between developers and stakeholders

Of course, the code itself is important, but all of these factors can contribute to the quality and maintainability of a project.

1.1.1 Characteristics of legacy projects

It's neither easy nor particularly useful to lay down a rule about what counts as a legacy project, but there are a few features that many legacy projects have in common.

OLD

Usually a project needs to exist for a few years before it gains enough entropy to become really difficult to maintain. In that time, it will also go through a number of generations of maintainers. With each of these handoffs, knowledge about the original design of the system and the intentions of the previous maintainer is also lost.

LARGE

It goes without saying that the larger the project is, the more difficult it is to maintain. There is more code to understand, a larger number of existing bugs (if we assume a constant defect rate in software, more code = more bugs), and a higher probability of a new change causing a regression, because there is more existing code that it can

potentially affect. The size of a project also affects decisions about how the project is maintained. Large projects are difficult and risky to replace, so they are more likely to live on and become legacy.

INHERITED

As is implied by the common meaning of the word legacy, these projects are usually inherited from a previous developer or team. In other words, the people who originally wrote the code and those who now maintain it are not the same individuals, and they may even be separated by several intermediate generations of developers. This means that the current maintainers have no way of knowing why the code works the way it does, and they're often forced to guess the intentions and tacit design assumptions of the people who wrote it.

POORLY DOCUMENTED

Given that the project spans multiple generations of developers, it would seem that keeping accurate and thorough documentation is essential to its long-term survival. Unfortunately, if there's one thing that developers enjoy less than writing documentation, it's keeping that documentation up to date. So any technical documents that do exist must invariably be taken with a pinch of salt.

I once worked on the software for a forum in which users could post messages in threads. The system had an API that allowed you to retrieve a list of the most popular threads, along with a few of the most recently posted messages in each of those threads. The API looked something like the following listing.

```
/**
 * Retrieve a list of summaries of the most popular threads.
 *
 * @param numThreads
 *       how many threads to retrieve
 * @param recentMessagesPerThread
 *       how many recent messages to include in thread summary
 *       (set this to 0 if you don't need recent messages)
 * @return thread summaries in decreasing order of popularity
 */
public List<ThreadSummary> getPopularThreads(
        int numThreads, int recentMessagesPerThread);
```

According to the documentation, if you only wanted a list of threads and you didn't need any messages, you should set `recentMessagesPerThread` to `0`. But at some point the behavior of the system changed, so that `0` now meant "include every single message in the thread." Given that this was a list of the most popular threads in the application, most of them contained many thousands of messages, so any API call that passed a `0` now resulted in a monster SQL query and an API response many MB in size!

1.1.2 Exceptions to the rule

Just because a project fulfills some of the preceding criteria doesn't necessarily mean it should be treated as a legacy project.

A perfect example of this is the Linux kernel. It's been in development since 1991, so it's definitely old, and it's also large. (The exact number of lines of code is difficult to determine, as it depends on how you count them, but it's said to be around 15 million at the time of writing.) Despite this, the Linux kernel has managed to maintain a very high level of quality. As evidence of this, in 2012 Coverity ran a static analysis scan on the kernel and found it to have a defect density of 0.66 defects/kloc, which is lower than many commercial projects of a comparable size. Coverity's report concluded that "Linux continues to be a 'model citizen' open source project for good software quality." (The Coverity report is available at http://wpcme.coverity.com/wp-content/uploads/2012-Coverity-Scan-Report.pdf.)

I think the primary reason for Linux's continued success as a software project is its culture of open and frank communication. All incoming changes are thoroughly reviewed, which increases information-sharing between developers, and Linus Torvalds' uniquely "dictatorial" communication style makes his intentions clear to everybody involved with the project.

The following quote from Andrew Morton, a Linux kernel maintainer, demonstrates the value that the Linux development community places on code review.

> Well, it finds bugs. It improves the quality of the code. Sometimes it prevents really really bad things from getting into the product. Such as rootholes in the core kernel. I've spotted a decent number of these at review time.
>
> It also increases the number of people who have an understanding of the new code—both the reviewer(s) and those who closely followed the review are now better able to support that code.
>
> Also, I expect that the prospect of receiving a close review will keep the originators on their toes—make them take more care over their work.
>
> —Andrew Morton,
> kernel maintainer, discussing the value of
> code review in an interview with LWN in 2008
> (https://lwn.net/Articles/285088/)

1.2 Legacy code

The most important part of any software project, especially for an engineer, is the code itself. In this section we'll look at a few common characteristics usually seen in legacy code. In chapter 4 we'll look at refactoring techniques that you can use to alleviate these problems, but for now I'll whet your appetite with a few examples and leave you to think about possible solutions.

1.2.1 Untested, untestable code

Given that technical documentation for software projects is usually either nonexistent or unreliable, tests are often the best place to look for clues about the system's behavior and design assumptions. A good test suite can function as the de facto documentation for a project. In fact, tests can even be more useful than documentation, because they're more likely to be kept in sync with the actual behavior of the system. A socially responsible developer will take care to fix any tests that were broken by their changes

to production code. (Any developer in my team who breaks this social contract is sent straight to the firing squad!)

Unfortunately, many legacy projects have almost no tests. Not only this, but the projects are usually written without testing in mind, so retroactively adding tests is extremely difficult. A code sample is worth a thousand words, so let's look at an example that illustrates this.

Listing 1.1 Some untestable code

```
public class ImageResizer {
    /* Where to store resized images */
    public static final String CACHE_DIR = "/var/data";

    /* Maximum width of resized images */
    private final int maxWidth =
            Integer.parseInt(System.getProperty("Resizer.maxWidth", "1000"));

    /* Helper to download an image from a URL */
    private final Downloader downloader = new HttpDownloader();

    /* Cache in which to store resized images */
    private final ImageCache cache = new FileImageCache(CACHE_DIR);

    /**
     * Retrieve the image at the given URL
     * and resize it to the given dimensions.
     */
    public Image getImage(String url, int width, int height) {
        String cacheKey = url + "_" + width + "_" + height;

        // First look in the cache
        Image cached = cache.get(cacheKey);
        if (cached != null) {
            // Cache hit
            return cached;
        } else {
           // Cache miss. Download the image, resize it and cache the result.
            byte[] original = downloader.get(url);
            Image resized = resize(original, width, height);
            cache.put(cacheKey, resized);
            return resized;
        }
    }

    private Image resize(byte[] original, int width, int height) {
        ...
    }
}
```

The `ImageResizer`'s job is to retrieve an image from a given URL and resize it to a given height and width. It has a helper class to do the downloading and a cache to save the resized images. You'd like to write a unit test for `ImageResizer` so that you can verify your assumptions about how the image resizing logic works.

Unfortunately, this class is difficult to test because the implementations of its dependencies (the downloader and the cache) are hardcoded. Ideally you'd like to mock these in your tests, so that you can avoid actually downloading files from the internet or storing them on the filesystem. You could provide a mock downloader that simply returns some predefined data when asked to retrieve an image from http://example.com/foo.jpg. But the implementations are fixed and you have no way of overriding them for your tests.

You're stuck with using the file-based cache implementation, but can you at least set the cache's data directory so that tests use a different directory from production code? Nope, that's hardcoded as well. You'll have to use /var/data (or C:\\var\\data if you're on Windows).

At least the `maxWidth` field is set via system property rather than being hardcoded, so you can change the value of this field and test that the image-resizing logic correctly limits the width of images. But setting system properties for a test is extremely cumbersome. You have to

1. Save any existing value of the system property.
2. Set the system property to the value you want.
3. Run the test.
4. Restore the system property to the value that you saved. You must make sure to do this even if the test fails or throws an exception.

You also need to be careful when running tests in parallel, as changing a system property may affect the result of another test running simultaneously.

1.2.2 Inflexible code

A common problem with legacy code is that implementing new features or changes to existing behavior is inordinately difficult. What seems like a minor change can involve editing code in a lot of places. To make matters worse, each one of those edits also needs to be tested, often manually.

Imagine your application defines two types of users: Admins and Normal users. Admins are allowed to do anything, whereas the actions of Normal users are restricted. The authorization checks are implemented simply as `if` statements, spread throughout the codebase. It's a large and complex application, so there are a few hundred of these checks in total, each one looking like the following.

```
public void deleteWibble(Wibble wibble)
                  throws NotAuthorizedException {
    if (!loggedInUser.isAdmin()) {
        throw new NotAuthorizedException(
            "Only Admins are allowed to delete wibbles");
    }
        ...
}
```

One day you're asked to add a new user type called Power User. These users can do more than Normal users but are not as powerful as Admins. So for every action that Power Users are allowed to perform, you'll have to search through the codebase, find the corresponding `if` statement, and update it to look like this.

```
public void deleteWibble(Wibble wibble)
                  throws NotAuthorizedException {
    if (!(loggedInUser.isAdmin() || loggedInUser.isPowerUser()) {
        throw new NotAuthorizedException(
            "Only Admins and Power Users are allowed to delete wibbles");
    }
        ...
}
```

We'll come back to this example in chapter 4 and look at how you could refactor the application to be more amenable to change.

1.2.3 Code encumbered by technical debt

Every developer is occasionally guilty of writing code that they know isn't perfect, but is good enough for now. In fact, this is often the correct approach. As Voltaire wrote, *le mieux est l'ennemi du bien* (perfect is the enemy of good).

In other words, it's often more useful and appropriate to ship something that works than to spend excessive amounts of time striving for a paragon of algorithmic excellence.

But every time you add one of these good enough solutions to your project, you should plan to revisit the code and clean it up when you have more time to spend on it. Every temporary or hacky solution reduces the overall quality of the project and makes future work more difficult. If you let too many of them accumulate, eventually progress on the project will grind to a halt.

Debt is often used as a metaphor for this accumulation of quality issues. Implementing a quick-fix solution is analogous to taking out a loan, at some point this loan must be paid back. Until you repay the loan by refactoring and cleaning up the code, you'll be burdened with interest payments, meaning a codebase that's more difficult to work with. If you take out too many loans without paying them back, eventually the interest payments will catch up with you, and useful work will grind to a halt.

For example, imagine your company runs InstaHedgehog.com, a social network in which users can upload pictures of their pet hedgehogs and send messages to each other about hedgehog maintenance. The original developers didn't have scalability in mind when they wrote the software, as they only expected to support a few thousand users. Specifically, the database in which users' messages are stored was designed to be easy to write queries against, rather than to achieve optimal performance.

At first, everything ran smoothly, but one day a celebrity hedgehog owner joined the site, and InstaHedgehog.com's popularity exploded! Within a few months, the site's userbase had grown from a few thousand users to almost a million. The DB, which wasn't designed for this kind of load, started to struggle and the site's performance

suffered. The developers knew that they needed to work on improving scalability, but achieving a truly scalable system would involve major architectural changes, including sharding the DB and perhaps even switching from the traditional relational DB to a NoSQL datastore.

In the meantime, all these new users brought with them new feature requests. The team decided to focus initially on adding new features, while also implementing a few stop-gap measures to improve performance. This included adding a couple of DB indexes, introducing ad hoc caching measures wherever possible, and throwing hardware at the problem by upgrading the DB server. Unfortunately, the new features vastly increased the complexity of the system, partially because their implementation involved working around the fundamental architectural problems with the DB. The caching systems also increased complexity, as anybody implementing a new feature now had to consider the effect on the various caches. This led to a variety of obscure bugs and memory leaks.

Fast forward a few years to the present day, and you're charged with maintaining this behemoth. The system is now so complex that it's pretty much impossible to add new features, and the caching systems are still regularly leaking memory. You've given up on trying to fix that, opting instead to restart the servers once a day. And it goes without saying that the re-architecting of the DB was never done, as the system became complex enough to render it impossible.

The moral of the story is, of course, that if the original developers had tackled their technical debt earlier, you wouldn't be in this mess. It's also interesting to note that debt begets debt. Because the original technical debt (the inadequate DB architecture) was not paid off, the implementation of new features became excessively complex. This extra complexity is itself technical debt, as it makes the maintenance of those features more difficult. Finally, as an ironic twist, the presence of this new debt makes it more difficult to pay off the original debt.

1.3 Legacy infrastructure

Although the quality of the code is a major factor in the maintainability of any legacy project, just looking at the code doesn't show you the whole picture. Most software depends on an assortment of tools and infrastructure in order to run, and the quality of these tools can also have a dramatic effect on a team's productivity. In part 3 we'll look at ways to make improvements in this area.

1.3.1 Development environment

Think about the last time you took an existing project and set it up on your development machine. Approximately how long did it take, from first checking the code out of version control to reaching a state where you could do the following?

- View and edit the code in your IDE
- Run the unit and integration tests
- Run the application on your local machine

If you were lucky, the project was using a modern, popular build tool and was structured in accordance with that build tool's conventions, so the setup process was complete in a matter of minutes. Or perhaps there were a few dependencies to set up, such as a database and a message queue, so it took a couple of hours of following the instructions in the README file before you could get everything working. When it comes to legacy projects, however, the time taken to set up the development environment can often be measured in days!

Setting up a legacy project often involves a combination of

- Downloading, installing, and learning how to run whatever arcane build tool the project uses
- Running the mysterious and unmaintained scripts you found in the project's /bin folder
- Taking a large number of manual steps, listed on an invariably out-of-date wiki page

You only have to run this gauntlet once, when you first join the project, so working to make the process easier and faster may not seem worth the effort. But there are good reasons to make the project setup procedure as smooth as possible. First, it's not just you who has to perform this setup. Every single developer on your team, now and in the future, will have to do it as well, and the cost of all those wasted days adds up.

Second, the easier it is to set up the development environment, the more people you can persuade to contribute to the project. When it comes to software quality, the more eyes you have on code the better. You want to avoid a situation where the only developers able to work on a project are those who have already bothered to set it up on their machines. Instead you should strive to create an environment in which any developer in the organization can make contributions to the project as easily as possible.

1.3.2 Outdated dependencies

Nearly any software project has some dependencies on third-party software. For example, if it's a Java servlet web application, it will depend on Java. It also needs to run inside a servlet container such as Tomcat, and may also use a web server such as Apache. Most likely, it will also use various Java libraries such as Apache Commons.

The rate at which these external dependencies change is outside of your control. Keeping up with the latest versions of all dependencies is a constant effort, but it's usually worthwhile. Upgrades often provide performance improvements and bug fixes, and sometimes critical security patches. While you shouldn't upgrade dependencies just for the sake of it, upgrading is usually a Good Thing.

Dependency upgrades are a bit like household chores. If you wash the dishes, vacuum the carpets, and give the house a quick once-over every few days, it's no big deal, but if you fall behind on your chores, it turns into a major mission when you finally get around to it. Keeping a project's dependencies up to date is just the same. Regularly upgrading to keep up with the latest minor version is a matter of a few minutes' work

per month, but if you get lazy and fall behind, before you know it you're one or more major versions out of date, and you're looking at a major investment of development and testing resources and a lot of risk when you finally get around to upgrading.

I once witnessed a development team spending months trying to upgrade their application from Java 6 to Java 7. For years they'd resisted upgrading any of their dependencies, mostly because they were afraid of breaking some obscure part of the application. It was a large, sprawling legacy application with few automated tests and no specifications, so nobody had a clear idea of how the application was currently behaving or how it was supposed to behave. But when Java 6's end of service life rolled around, the team decided it was time to bite the bullet. Unfortunately, upgrading to Java 7 meant that they also had to upgrade a whole host of other dependencies. This resulted in a slew of breaking API changes, as well as subtle and undocumented changes in behavior. This particular story doesn't have a happy ending—after a few weeks battling to work around an obscure change in the behavior of XML serialization, they gave up on the upgrade and rolled back their changes.

1.3.3 Heterogeneous environments

Most software will be run in a number of environments during its lifetime. The number and names of the environments are not set in stone, but the process usually goes something like this:

1. Developers run the software on their local machines.
2. They deploy it to a test environment for automatic and manual testing.
3. It is deployed to a staging environment that mimics production as closely as possible.
4. It is released and deployed to the production environment (or, in the case of packaged software, it's shipped to the customer).

The point of this multistage process is to verify that the software works correctly before releasing it to production, but the value of this verification is directly affected by the degree of parity you can achieve between the environments. Unless the staging environment is an exact replica (within reason) of the production environment, its value in showing you how the software will behave in production is severely diminished. Similarly for the development and test environments, the more faithfully they replicate production, the more useful they'll be, allowing you to spot environment-related issues with the software quickly, without having to deploy it to the staging environment. For example, using the same version of MySQL across all environments obviates any risk of a tiny change in behavior between MySQL versions causing the software to work correctly in one environment and fail in another.

However, keeping these multiple environments perfectly aligned is easier said than done, especially without the aid of automation. If they're being managed manually, they are pretty much guaranteed to diverge. This divergence can happen in a few different ways.

- *Upgrades trickle down from production*—Say the ops team upgrades Tomcat in production in response to a zero-day exploit. A few weeks later, somebody notices that the upgrade hasn't been applied to the staging environment, so it's done there as well. A few months after that, somebody finally gets around to upgrading the test environment. The Tomcat on the developers' machines works fine, so they never bother upgrading.
- *Different tools in different environments*—You might use a lightweight database such as SQLite or H2 on developers' machines but a "proper" database in other environments.
- *Ad hoc changes*—Say you're prototyping an experimental new feature that depends on Redis, so you install Redis in the test environment. In the end, you decide not to go ahead with the new feature, so Redis is no longer needed, but you don't bother uninstalling it. A couple of years later, there's still a Redis instance running in the test environment, but nobody can remember why.

As this inter-environment divergence accumulates over time, eventually it's likely to lead to that most dreaded of software phenomena, the *production-only bug*. This is a bug caused by an interaction between the software and its surrounding environment that occurs only in production. No amount of testing in the other environments will help you to catch this bug because it never occurs in those environments, so they've become basically pointless.

In part 3 we'll look at how you can use automation to keep all your environments in sync and up to date, helping to make development and testing run more smoothly and avoid problems like this.

1.4 Legacy culture

The term *legacy culture* is perhaps a little contentious—nobody wants to think of themselves and their culture as legacy—but I've noticed that many software teams who spend too much time maintaining legacy projects share certain characteristics with respect to how they develop software and communicate among themselves.

1.4.1 Fear of change

Many legacy projects are so complex and poorly documented that even the teams charged with maintaining them don't understand everything about them. For example,

- Which features are no longer in use and are thus safe to delete?
- Which *bugs* are safe to fix? (Some users of the software may be depending on the bug and treating it as a feature.)
- Which users need to be consulted before making changes to behavior?

Because of this lack of information, many teams come to respect the status quo as the safest option, and become fearful of making any unnecessary changes to the software. Any change is seen purely as a risk, and the potential benefits of changes are ignored. Thus the project enters a kind of stasis, whereby the developers expend most of their

energy on maintaining the status quo and trying to protect the software from all externalities, like a mosquito trapped in a blob of amber.

The irony is that by being so risk-averse that they don't allow the software to evolve, they often leave their organization exposed to a very major risk, that of being left behind by its competitors. If your competitors can add new features faster than you can, which is quite likely if your project has entered stasis mode, then it's only a matter of time before they steal your customers and push you out of the market. This is, of course, a much larger risk than those the development team was worried about.

But it doesn't have to be this way. Although a gung-ho approach, whereby all risks are ignored and changes are rolled out willy-nilly, is certainly a recipe for disaster, a more balanced approach is feasible. If the team can maintain a sense of perspective, weighing each change's risks against its benefits, as well as actively seeking out the missing information that will help them to make such decisions, the software will be able to evolve and adapt to change.

A few examples of possible changes for a legacy project, with their associated risks and benefits, are shown in table 1.1. There's also a column suggesting how you could gather more information about the risks.

Table 1.1 Changes, benefits and risks for a legacy project

Changes	Benefits	Risks	Actions required
Removing an old feature	· Easier development · Better performance	· Somebody is still using the feature	· Check access logs · Ask users
Refactoring	· Easier development	· Accidental regression	· Code review · Testing
Upgrading a library	· Bug fixes · Performance improvements	· Regression due to a change in the library's behavior	· Read the changelog · Review library code · Manually test major features

1.4.2 Knowledge silos

The biggest problem encountered by developers when writing and maintaining software is often a lack of knowledge. This may include

- Domain information about the users' requirements and the functional specifications of the software
- Project-specific technical information about the software's design, architecture, and internals
- General technical knowledge such as efficient algorithms, advanced language features, handy coding tricks, and useful libraries

A benefit of working on a team is that if you're missing a particular piece of knowledge, a fellow team member might be able to provide it. For this to happen, either you need to ask them for the information, or, even better, they need to give it to you of their own accord.

This sounds obvious, but unfortunately on many teams this giving and receiving of knowledge doesn't happen. Unless you work hard to foster an environment of communication and information sharing, each individual developer becomes a metaphorical silo of information, with all their valuable knowledge sitting untapped inside their heads instead of being shared for the benefit of the whole team.

Factors contributing to a paucity of communication within a team can include

- *Lack of face-to-face communication*—I often see developers chatting on Skype or IRC even though they're sitting right next to each other. These tools have their uses, especially if your team includes remote workers or you want to send somebody a message without disturbing them, but if you want to talk in real time to somebody sitting three feet away from you, using IRC to do so just isn't healthy!
- *Code ego*—"If I let anybody figure out how my code works, they might criticize it." This is a common mindset among developers, and it can be a serious barrier to communication.
- *Busy-face*—Projecting an air of busyness is a common defense strategy used by developers to avoid being piled with unwelcome work. (I'm guilty of this, making sure to slip on my headphones and a stressed-looking expression whenever I see anybody in a suit approaching my desk.) But this can also make these people less approachable to other developers who want to ask them for advice.

There are a number of things you can try in order to increase communication within your team, including code review, pair programming, and hackathons. We'll discuss this further in the final chapter of the book.

1.5 Summary

After a whole chapter of moaning about the state of software development, it looks like we have our work cut out for us if we want to tackle all of these problems. But don't worry, we don't have to fix everything at once. In the rest of the book, we'll attempt to revitalize legacy projects one step at a time.

Here are some takeaway points from this chapter:

- Legacy software is often large, old, inherited from somebody else, and poorly documented. But there are exceptions to the rule: the Linux kernel largely fulfills all of those criteria, but its quality is high.
- Legacy software often lacks tests and is difficult to test. Low testability implies few tests, but the converse is also true. If a codebase currently has few tests, it probably has an untestable design and writing new tests for it is thus difficult.
- Legacy code is often inflexible, meaning that it takes a lot of work to make a simple change. Refactoring can improve the situation.
- Legacy software is encumbered by years' worth of accumulated technical debt.
- The infrastructure on which the code runs, all the way from the developer's machine through to production, deserves attention.
- The culture of the team that maintains the software can be an impediment to improvement.

SOFTWARE ENGINEERING

Re-Engineering Legacy Software

Chris Birchall

As a developer, you may inherit projects built on existing codebases with design patterns, usage assumptions, infrastructure, and tooling from another time and another team. Fortunately, there are ways to breathe new life into legacy projects so you can maintain, improve, and scale them without fighting their limitations.

Re-Engineering Legacy Software is an experience-driven guide to revitalizing inherited projects. It covers refactoring, quality metrics, toolchain and workflow, continuous integration, infrastructure automation, and organizational culture. You'll learn techniques for introducing dependency injection for code modularity, quantitatively measuring quality, and automating infrastructure. You'll also develop practical processes for deciding whether to rewrite or refactor, organizing teams, and convincing management that quality matters. Core topics include deciphering and modularizing awkward code structures, integrating and automating tests, replacing outdated build systems, and using tools like Vagrant and Ansible for infrastructure automation.

What's Inside

- Refactoring legacy codebases
- Continuous inspection and integration
- Automating legacy infrastructure
- New tests for old code
- Modularizing monolithic projects

This book is written for developers and team leads comfortable with an OO language like Java or C#.

Chris Birchall is a senior developer at the *Guardian* in London, working on the back-end services that power the website.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/re-engineering-legacy-software

"A comprehensive guide to turning legacy software into modern, up-to-date projects!"
—Jean-François Morin
Université Laval

"I learn something new every time I read it."
—Lorrie MacKinnon
Treasury Board Secretariat
Province of Ontario

"The book I wanted to read years ago."
—Ferdinando Santacroce, 7Pixel

"A very practical guide to one of the most difficult aspects of software development."
—William E. Wheeler
West Corporation

ISBN-13: 978-1-61729-250-7
ISBN 10: 1-61729-250-8
56499
9 781617 292507

MANNING $64.99 / Can $74.99 [INCLUDING eBOOK]